An Expressive Formal Model of the Web Infrastructure

Von der Fakultät 5 (Informatik, Elektrotechnik und Informationstechnik) der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Daniel Fett
aus Mons, Belgien

Hauptberichter:	Prof. Dr. Ralf Küsters
Mitberichter:	Dr. Karthikeyan Bhargavan
Tag der mündlichen Prüfung:	19.10.2018

Institut für Informationssicherheit (SEC) der Universität Stuttgart

2018

Acknowledgements

I would like to express my deepest gratitude to my advisor Ralf Küsters for his continuous motivation, support, and patience. His excellent advice and keen attention to detail were invaluable for my work. I could not have imagined having a better advisor and mentor.

I would like to thank my friends and colleagues who accompanied me during my PhD studies in Trier and Stuttgart, and who were always in for a game of darts. I thank Guido, with whom I devised and developed not only many parts of this work, but also *Pi and More*, for his friendship and help, and many fun days spent travelling.

Thanks are also due to the *Studienstiftung des Deutschen Volkes* (German National Academic Foundation), the *Stipendienstiftung Rheinland-Pfalz*, and the *Deutsche Forschungsgemeinschaft* (Grant KU 1434/10-1) for their financial support.

Finally, I am indebted to Janett and my family who never stopped encouraging me. This dissertation would not have been possible without their endless support and love.

Contents

Acknowledgements 3

List of Figures 11

List of Tables 13

List of Algorithms 15

List of Abbreviations and Glossary 17

Abstract 21

Kurzzusammenfassung 23

1 Introduction 25

1.1 Contributions of this Thesis . . . 28

1.1.1 The Web Infrastructure Model . . . 28

1.1.2 Formal Analysis of OAuth 2.0 . . . 29

1.1.3 Formal Analysis of OpenID Connect . . . 31

1.2 Structure of this Thesis . . . 32

1.3 Publications . . . 32

1.4 Related Work . . . 34

1.4.1 Formal Web Security Analysis . . . 34

1.4.2 Security Analysis of OAuth 2.0 . . . 35

1.4.3 Security Analysis of OpenID Connect . . . 36

2 The Web Infrastructure Model 37

2.1 Building a Model of the Web Infrastructure . . . 37

2.2 Architecture . . . 39

2.3 Terms, Messages, and Events . . . 40

2.4 Dolev-Yao Processes . . . 41

2.5 Attackers . . . 42

2.6 Scripts . . . 42

2.7 Systems and Web Systems . . . 44

2.8 HTTP and HTTPS Messages . . . 44

2.9 Name Resolution 46
2.10 Web Browsers 47
2.10.1 Browsers and Users 48
2.10.2 Two Types of Corruption 48
2.10.3 Windows and Documents 49
2.10.4 Cookies and Web Storage 50
2.10.5 HTTP(S) Message Dispatching 52
2.10.6 WebSockets 53
2.10.7 Strict Transport Security 54
2.10.8 WebMessaging 55
2.10.9 Message Processing 55
2.10.10 Executing a Script 59
2.11 Generic HTTPS Servers 61
2.12 Extension: WebRTC 62
2.12.1 WebRTC and the WebRTC Model 62
2.12.2 New Script Commands for WebRTC 65

3 Analysis of OAuth 2.0 **67**
3.1 OAuth 2.0 Basic Concepts 67
3.1.1 Token Types 68
3.1.2 Endpoints 68
3.1.3 Client Registration at the OAP 69
3.1.4 Login Sessions and State 69
3.1.5 Tracking User Intention 70
3.1.6 Further Recommendations and Options 70
3.2 OAuth 2.0 Grant Types 70
3.2.1 Authorization Code Grant 71
3.2.2 Implicit Grant 73
3.2.3 Resource Owner Password Credentials Grant 74
3.2.4 Client Credentials Grant 75
3.3 New Attacks on OAuth 76
3.3.1 307 Redirect Attack 76
3.3.2 AS Mix-Up Attack 77
3.3.3 State Leak Attack 84
3.3.4 Naïve Client Session Integrity Attack 85
3.3.5 Across-AS State Reuse Attack 85
3.4 Other Attacks on OAuth 86
3.4.1 Code/Token/State Leakage 86
3.4.2 CSRF Protection 87
3.4.3 Third-Party Resources 87

3.4.4 Open Redirectors . . . 87
3.4.5 Session Handling . . . 88
3.4.6 Access Token Introspection Client ID . . . 88
3.5 Formal Analysis of OAuth 2.0 . . . 88
3.5.1 Model: Design, Concepts, Limitations . . . 89
3.5.2 Model: Web Systems . . . 90
3.5.3 Security Properties . . . 92
3.5.4 The OAuth Security Theorem . . . 93
3.5.5 Proof of the OAuth Security Theorem: Outline . . . 93
3.5.6 Discussion of Results . . . 94

4 Analysis of OpenID Connect 97
4.1 OpenID Connect Basic Concepts . . . 97
4.1.1 Relationship to OAuth 2.0 . . . 97
4.1.2 Authentication, ID Tokens, and Issuer Identifiers . . . 98
4.2 Discovery and Dynamic Registration Extensions . . . 98
4.2.1 OpenID Connect Discovery . . . 99
4.2.2 OpenID Connect Dynamic Client Registration . . . 100
4.3 OpenID Connect Flows . . . 101
4.3.1 Authorization Code Flow . . . 101
4.3.2 Implicit Flow . . . 102
4.3.3 Hybrid Flow . . . 103
4.4 Attacks on OpenID Connect . . . 104
4.4.1 AS Mix-Up Attacks . . . 105
4.4.2 Attacks on the State Parameter . . . 109
4.4.3 307 Redirect Attack . . . 109
4.4.4 Server-Side Request Forgery . . . 109
4.4.5 CSRF Attacks and Third-Party Login Initiation . . . 110
4.5 Formal Analysis of OpenID Connect . . . 111
4.5.1 Model . . . 111
4.5.2 Main Security Properties . . . 112
4.5.3 Secondary Security Properties . . . 116
4.5.4 The OpenID Connect Security Theorem . . . 118
4.5.5 Proof of the OpenID Connect Security Theorem: Outline . . . 118
4.5.6 Discussion of Results . . . 119

5 Impact 121
5.1 Verification . . . 121
5.2 Disclosure . . . 121
5.3 Follow-Up . . . 122

6 Conclusion and Future Work 123

A The Web Infrastructure Model 125
A.1 Communication Model . . . 125
A.1.1 Terms, Messages and Events . . . 125
A.1.2 Notations . . . 126
A.1.3 Atomic Processes, Systems and Runs . . . 128
A.1.4 Atomic Dolev-Yao Processes . . . 130
A.2 Scripts . . . 131
A.3 Web System . . . 131
A.4 Message and Data Formats . . . 132
A.4.1 URLs . . . 132
A.4.2 Origins . . . 133
A.4.3 Cookies . . . 133
A.4.4 HTTP Messages . . . 133
A.4.5 DNS Messages . . . 135
A.4.6 WebSocket Messages . . . 135
A.4.7 WebRTC Messages . . . 136
A.5 DNS Server Model . . . 136
A.6 Web Browser Model . . . 136
A.6.1 Windows, Documents, and Related Notations . . . 136
A.6.2 Web Browser States $Z_{\text{webbrowser}}$ . . . 137
A.6.3 Web Browser Relation $R_{\text{webbrowser}}$ . . . 139
A.6.4 Definition of Web Browsers . . . 148
A.7 Generic HTTPS Server Model . . . 152

B Analysis of OAuth 2.0 155
B.1 Formal Model of OAuth with a Network Attacker . . . 155
B.1.1 Outline . . . 155
B.1.2 Addresses and Domain Names . . . 157
B.1.3 Keys and Secrets . . . 157
B.1.4 Identities, Passwords, and Protected Resources . . . 157
B.1.5 Corruption . . . 159
B.1.6 Processes in $\mathcal{W}$ (Overview) . . . 159
B.1.7 Network Attacker . . . 160
B.1.8 Browsers . . . 160
B.1.9 Clients . . . 160
B.1.10 OAuth Providers . . . 164
B.2 Formal Security Properties . . . 173
B.2.1 Authorization . . . 173

B.2.2 Authentication 173
B.2.3 Session Integrity for Authorization and Authentication 173
B.3 Proof of the OAuth Security Theorem 176
B.3.1 Properties of $\mathcal{OAuthWS}^n$ 177
B.3.2 Proof of Authentication 179
B.3.3 Proof of Authorization 186
B.3.4 Proof of Session Integrity 187

C Analysis of OpenID Connect 197
C.1 Formal Model of OpenID Connect with a Network Attacker 197
C.1.1 Outline 197
C.1.2 Addresses and Domain Names 198
C.1.3 Keys and Secrets 199
C.1.4 Identities and Passwords 199
C.1.5 Corruption 199
C.1.6 Network Attacker 199
C.1.7 Browsers 199
C.1.8 Relying Parties 200
C.1.9 Identity Providers 200
C.2 Formal Security Properties 209
C.2.1 Authentication 209
C.2.2 Authorization 209
C.2.3 Session Integrity for Authentication and Authorization 209
C.3 Proof of the OpenID Connect Security Theorem 212
C.3.1 Proof of Authentication 212
C.3.2 Proof of Authorization 217
C.3.3 Proof of Session Integrity 220
C.3.4 Proof of Theorem 2 225

Bibliography 227

Academic Curriculum and Publications 239

List of Figures

2.1 Illustration: Web systems with typical attacker setups. 40
2.2 Example: HTTP GET request. 44
2.3 Example: HTTP response. 45
2.4 The basic structure of the web browser relation. 56
2.5 Simple example flow of WebRTC without peer authentication. 63
2.6 WebRTC example flow with authentication. 65

3.1 OAuth 2.0 authorization code grant. 72
3.2 OAuth 2.0 implicit grant. 73
3.3 OAuth 2.0 resource owner password credentials grant. 74
3.4 OAuth 2.0 client credentials grant. 75
3.5 Start of the AS Mix-Up Attack on OAuth 2.0 authorization code grant. 79
3.6 AS Mix-Up Attack on OAuth 2.0: Breaking authorization without code injection. . 80
3.7 AS Mix-Up Attack on OAuth 2.0: Using code injection to break authorization. . . . 81
3.8 AS Mix-Up Attack on OAuth 2.0: Using code injection to break authentication. . . 82

4.1 OpenID Connect Discovery extension protocol flow. 99
4.2 OpenID Connect Dynamic Registration extension protocol flow. 100
4.3 OpenID Connect authorization code flow. 101
4.4 OpenID Connect implicit flow. 103
4.5 OpenID Connect hybrid flow. 104
4.6 Start of the AS Mix-Up Attack on the OpenID Connect hybrid flow. 106
4.7 AS Mix-Up Attack on OIDC: Breaking authentication with code injection. 108
4.8 AS Mix-Up Attack on OIDC: Breaking authentication without code injection. . . . 108
4.9 AS Mix-Up Attack on OIDC: Using a mock access token. 108
4.10 Server-Side Request Forgery in OIDC. 110

A.1 Equational theory for Σ. 126
A.2 Dictionary operators with $1 \leq i \leq n$. 127

B.1 Events as described in Lemma 18. 189
B.2 Structure of run from start to redirection endpoint. 193

List of Tables

3.1 Overview of attacks on OAuth 2.0. 76

4.1 Overview of attacks on OpenID Connect. 105

A.1 List of placeholders used in browser algorithms. 141

B.1 List of scripts in $\mathcal{S}$, their respective string representations, and their definitions. . . . 155
B.2 List of placeholders used in the client algorithm. 165

C.1 List of scripts in $\mathcal{S}$, their respective string representations, and their definitions. . . . 197
C.2 List of placeholders used in the relying party algorithm. 200
C.3 List of placeholders used in the identity provider algorithm. 201

List of Algorithms

2.1 Relation of a DNS server R^d. 47

A.1 Web Browser Model: Determine window for navigation. 142
A.2 Web Browser Model: Determine same-origin window. 142
A.3 Web Browser Model: Cancel pending requests for given window. 142
A.4 Web Browser Model: Prepare headers, do DNS resolution, save message. 143
A.5 Web Browser Model: Navigate a window backward. 143
A.6 Web Browser Model: Navigate a window forward. 143
A.7 Web Browser Model: Check remote server's WebRTC identity assertion. 144
A.8 Web Browser Model: Execute a script. 144
A.9 Web Browser Model: Deliver a message to the script in a document. 147
A.10 Web Browser Model: Process an HTTP response. 148
A.11 Web Browser Model: Main Algorithm. 150
A.12 Generic HTTPS Server Model: Sending a DNS message. 153
A.13 Generic HTTPS Server Model: Default HTTPS response handler. 153
A.14 Generic HTTPS Server Model: Default trigger event handler. 153
A.15 Generic HTTPS Server Model: Default HTTPS request handler. 153
A.16 Generic HTTPS Server Model: Default handler for other messages. 153
A.17 Generic HTTPS Server Model: Main relation of a generic HTTPS server. 154

B.1 Relation of *script_client_index*. 156
B.2 Relation of *script_client_implicit*. 156
B.3 Relation of *script_oap_form*. 156
B.4 Relation of a Client R^r. 167
B.5 Relation of an OAP R^i. 171

C.1 Relation of *script_rp_index*. 198
C.2 Relation of *script_rp_get_fragment*. 198
C.3 Relation of *script_op_form*. 198
C.4 Relation of a Relying Party R^r: Processing HTTPS Responses. 201
C.5 Relation of a Relying Party R^r: Processing HTTPS Requests. 202
C.6 Relation of a Relying Party R^r: Request to token endpoint. 203
C.7 Relation of a Relying Party R^r: Using the access token (no response expected). . . 203
C.8 Relation of a Relying Party R^r: Check ID token. 204

C.9 Relation of a Relying Party R^r: Continuing in the login flow. 205
C.10 Relation of an OP R^i: Processing HTTPS Requests. 206
C.11 Relation of an OP R^i: Processing other messages. 208

List of Abbreviations and Glossary

API Application Programming Interface

ARP Spoofing Spoofing messages of the Address Resolution Protocol (ARP) to intercept network messages.

AS Authorization Server

CORS Cross-Origin Resource Sharing

Cross-Origin Resource Sharing Standard to enable HTTP requests, for example using XMLHttpRequest, from one origin to another.

Cross-Site Request Forgery An attack in which an adversary causes a browser to create a request to some server on the adversary's behalf. The request uses the browser's resources (in particular, cookies) and can therefore cause unwanted behavior on the server.

CSRF Cross-Site Request Forgery

DNS Domain Name System

Fetch Defines the process of fetching web resources from the network including the security requirements for this process [Fetch] . Our model supports the `Origin` header defined in Fetch, which enables web servers to see whether a particular request was performed cross-origin.

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IA Identity Assertion

IdP Identity Provider

IETF Internet Engineering Task Force

IP Internet Protocol

JSON JavaScript Object Notation

OAP OAuth Provider (operating an OAuth AS and RS)

OIDC OpenID Connect

OP OpenID Provider (operating an OpenID Connect AS and RS)

Origin An origin is a tuple of a protocol, a host name, and a port number, usually written as a URL (e.g., `https://example.com`). Two origins are considered to be the *same* if and only if they are equal in all three components. The *same-origin policy* in browsers isolates web contents from different origins in various ways.

`Origin` **header** Header added to certain HTTP(S) requests by browsers to declare the origin of the document that caused the request. For example, when a user submits a form loaded from the URI `http://a/form` and this form is sent to `http://b/path` then the browser will add the origin header `http://a` in the request to `b`. Most modern browsers support origin headers. See [Ann14] for details.

postMessage Web Messaging API function to send messages from a browser window to another window in the same browser.

`Referer` **header** Header added by browsers to declare the URL of the document that caused the request. See [RFC7231] for details.

Referrer Policies Enable a web server to instruct a web browser to (partially or completely) suppress the Referer header.

RFC Request for Comments

RS Resource Server

SDK Software Development Kit

SSO Single sign-on

SSRF Server-Side Request Forgery

Strict Transport Security Using the `Strict-Transport-Security` header, web servers can recommend that all resources from their domain should be retrieved over secure connections (HTTPS) only. Browsers, when receiving such a header in an HTTPS response, add the respective domain to their internal list of Strict Transport Security domains. Requests to domains in this list will automatically be rewritten by the browser from HTTP to HTTPS. The list can be pre-populated by the browser vendor (usually from the Chrome HSTS Preload List, see [STSPre]).

STS Strict Transport Security

Subresource Integrity Enables websites to instruct web browsers to reject third-party content if this content does not match a specific hash [Akh+16].

TCP Transmission Control Protocol

TLS Transport Layer Security

TLS Stripping Attack on TLS in which an attacker intercepts network messages of his victim and removes all redirections to encrypted protocols in this traffic.

UDP User Datagram Protocol

URL Uniform Resource Locator

URI Uniform Resource Indicator

W3C World Wide Web Consortium

Web Messaging With Web Messaging [Hic15], two documents inside the same web browser can communicate with each other using an API called postMessage (even if the two documents have different origins).

WebRTC WebRTC, defined by the W3C and the IETF [Ber+18; IET18], is a technology for real-time peer-to-peer communication between browsers with an emphasis on the efficient and secure transport of media streams.

WebSocket Defined by the WHATWG and IETF [RFC6455; WS], the WebSocket API enables bi-directional data channels between browsers and servers. The main difference to HTTP is that with WebSockets, messages can be "pushed" from servers to browsers and are sent with a minimal overhead.

Web Storage Web Storage [Ian16] provides a mechanism for scripts to store data in the browser either permanently (localStorage) or temporarily (sessionStorage). See Section 2.10.4 for details.

WHATWG Web Hypertext Application Technology Working Group

XMLHttpRequest XMLHttpRequest is an API for scripts to send HTTP(S) requests to web servers without navigating the current page by activating a link or sending a form.

Abstract

The World Wide Web is arguably the most important medium of our time. Billions of users rely on the security of the web each day for tasks such as banking, shopping, and business and private communication.

The web is a heterogeneous infrastructure developing at a high pace. The question of whether the web infrastructure or certain web applications are secure is not easy to answer. Standards and applications today are reviewed by experts before they are deployed, but all too often even serious security vulnerabilities are simply overlooked.

In this thesis, we propose a formal model for the web infrastructure which enables a rigorous formal analysis of security and privacy in the web. Our model is the most comprehensive and expressive model of the web infrastructure to date. It facilitates accurate security and privacy analyses of current web standards and applications, and can serve as a reference for web security researchers, developers of new technologies and standards, and for teaching web security concepts.

As a case study we analyze the security of two important standards for federated authorization and authentication, OAuth 2.0 and OpenID Connect. Standardized by the IETF and OpenID Foundation, respectively, they are among the most widely deployed single sign-on systems in the web.

For our analysis, we develop detailed formal models for both systems based on our model of the web infrastructure. These models then allow us to precisely define the security goals of authentication, authorization and session integrity.

While proving security with respect to these goals, we found a total of five new attacks on the two single sign-on systems, breaking all of the security goals. In particular OAuth 2.0 had been analyzed many times before; the fact that we were able to find new attacks in OAuth 2.0 demonstrates the potential of rigorous analyses in our web infrastructure model.

We develop fixes against the underlying vulnerabilities and are then able to prove the security of OAuth 2.0 and OpenID Connect. Since our results are based on a comprehensive model, our proofs can exclude large classes of attacks against OAuth and OpenID Connect, including yet unknown attack vectors. Our attacks and fixes led to the development of new security recommendations by the standardization organizations.

Kurzzusammenfassung

Das World Wide Web ist wohl das wichtigste Medium unserer Zeit. Milliarden von Nutzern verlassen sich täglich für Banking, Shopping sowie geschäftliche und private Kommunikation auf die Sicherheit des Internets.

Das Web ist eine heterogene Infrastruktur, die sich in hohem Tempo entwickelt. Die Frage, ob die Web-Infrastruktur oder bestimmte Web-Anwendungen sicher sind, ist nicht einfach zu beantworten. Standards und Anwendungen werden heute von Experten überprüft, bevor sie eingesetzt werden, aber allzu oft werden auch schwerwiegende Sicherheitslücken einfach übersehen.

In dieser Arbeit schlagen wir ein formales Modell für die Web-Infrastruktur vor, das eine rigorose formale Analyse der Sicherheit und Privacy im Web ermöglicht. Unser Modell ist das bisher umfassendste und ausdrucksstärkste Modell für die Web-Infrastruktur. Es ermöglicht genaue Sicherheits- und Privacyanalysen aktueller Webstandards und Anwendungen und kann als Referenz für Websicherheitsforscher, Entwickler neuer Technologien und Standards sowie für die Vermittlung von Websicherheitskonzepten dienen.

Als Fallstudie analysieren wir die Sicherheit zweier wichtiger Standards für föderierte Autorisierung und Authentifizierung, OAuth 2.0 und OpenID Connect. Von der IETF bzw. OpenID Foundation standardisiert, gehören sie zu den am weitesten verbreiteten Single-Sign-On-Systemen im Web.

Für unsere Analyse entwickeln wir detaillierte formale Modelle für beide Systeme auf Basis unseres Modells der Web-Infrastruktur. Diese Modelle erlauben es uns dann, die Sicherheitsziele Authentifizierung, Autorisierung und Sitzungsintegrität genau zu definieren.

Während des Beweises der Sicherheit in Bezug auf diese Ziele fanden wir insgesamt fünf neue Angriffe auf die beiden Single-Sign-On-Systeme, die alle Sicherheitsziele verletzen. Insbesondere OAuth 2.0 wurde zuvor bereits vielfach analysiert; die Tatsache, dass wir noch neue Angriffe finden konnten, zeigt das Potenzial rigoroser Analysen in unserem Web-Infrastrukturmodell auf.

Wir entwickeln Schutzmaßnahmen gegen die zugrunde liegenden Schwachstellen und sind dann in der Lage, die Sicherheit von OAuth 2.0 und OpenID Connect zu beweisen. Da unsere Ergebnisse auf einem umfassenden Modell basieren, können unsere Beweise große Klassen von Angriffen gegen OAuth und OpenID Connect ausschließen, einschließlich noch unbekannter Angriffsvektoren. Unsere Angriffe und Schutzmaßnahmen führten zur Entwicklung neuer Sicherheitsempfehlungen durch die Standardisierungsorganisationen.

1. Introduction

"Storage of ASCII text, and display on 24x80 screens, is in the short term sufficient, and essential. Addition of graphics would be an optional extra with very much less penetration for the moment."

— Tim Berners-Lee, *Information Management: A Proposal*, March 1989 [Ber89]

In 1991, when the first web page was created, it was meant as a way to share text and simple graphics. Since then, the World Wide Web has grown remarkably: Today, it consists of hundreds of billions of web pages [GGS] and can be accessed by almost anybody via a myriad of devices ranging from smart phones to refrigerators. Initially limited to reading and writing mostly text-based information, today's web is a driver for rich, dynamic, and complex applications. Many million times each day, tasks such as banking, shopping, management and remote control of devices and infrastructure are performed over the web. Billions of private users and businesses rely on the security and privacy of the web.

The web, however, is a complex, heterogeneous infrastructure: Different kinds of entities, such as web browsers, web servers, and DNS servers interact using diverse technologies. Web browsers, in particular, have evolved from information viewers to runtime environments for highly interactive distributed applications. In one and the same browser, users routinely visit highly sensitive web services, like online banking, alongside arbitrary web sites of low trustworthiness.

Web browsers, the web infrastructure, and web applications therefore have to protect their users from a variety of attacks exploiting missing protections, logic flaws, the incorrect use of cryptographic primitives, and historic shortcomings in web technologies. Today, the state of the art to ensure that new standards and applications are secure is expert review: Designs are drafted and thoroughly reviewed by groups of experts. Afterwards, concrete implementations of standards and applications are often tested using penetration testing (pentesting), which, by definition, can only find previously known types of attacks. The hope is that the combination of these two methods can detect all critical flaws before they can do any harm. However, as illustrated by numerous attacks (e.g., [Akh+10; Arm+13; Ban+13; BBM12; IET09; Kar+07; Pel+16; SB12; Wan+11; Wan+13; Zhe+15]), this is not the case: Experts and penetration testers can easily overlook attacks, in particular those that do not follow known patterns. With the ever-growing and rapidly increasing complexity of web technologies, methods for a rigorous security and privacy analysis of the web infrastructure, web standards, and web applications are urgently needed.

First steps in this direction were made by Akhawe et al. [Akh+10] and Bansal et al. [Ban+13; BBM12]. In both lines of work, new methods were developed that enable the formal definition and validation of security properties. These methods, based on automatic analysis, uncover previously unknown web vulnerabilities in web applications and web standards. Constraints inherent to the tools used and their modeling languages, however, limit the expressiveness and comprehensiveness of the models and the results (see Section 1.4 for a more detailed discussion).

In this thesis, one main goal is to develop an all-new, expressive and comprehensive formal model of the web infrastructure. Our approach does not aim at automation or tool support (see future work, Section 6). Instead, our first priority is to precisely capture core security aspects of web applications and the web infrastructure. We aim at staying as close to the standards as possible while providing a level of abstraction that is suitable for a manual formal analysis.

Our model, the *Web Infrastructure Model* (WIM), constitutes a solid formal foundation for the modeling of a broad range of complex web applications and standards, for a precise definition of security and privacy properties, and for rigorous, model-based security analyses. It is much more detailed and comprehensive than earlier models.

Our model also serves an additional purpose: The standards and specifications that define the web are spread across many documents published by different organizations. To name just a few relevant examples, the still most widely used version of the Hypertext Transfer Protocol, HTTP/1.1 is defined by the Internet Engineering Task Force (IETF) in [RFC7230; RFC7231; RFC7232; RFC7233; RFC7234; RFC7235]; cookies are defined in [RFC6265]; Strict Transport Security (STS) is defined in [RFC6797]; the origin concept is defined in [RFC6454]; the Web Hypertext Application Technology Working Group (WHATWG) defines the Fetch standard [Fetch]; the World Wide Web Consortium (W3C) defines HTML5 [Ber+17] with many related specifications such as Web Storage [Ian16] and Cross-Origin Resource Sharing [Ann14]. Specifications for the Domain Name System (DNS) and communication protocols such as the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are relevant as well. The documents often build upon each other, replace older versions or other documents, and sometimes different versions coexist. Some details and behaviors are not specified at all and are only documented in the form of the source code of web browsers. Browser vendors occasionally implement different interpretations of the same standard.[1]

An accurate formal model like the one proposed in this thesis summarizes and condenses important specifications otherwise spread across different documents. As such, it is an important contribution to the discourse on web security and can serve as a reference for tool-supported analysis, web security researchers, developers of web technologies and standards, and for teaching web security concepts.

[1] For example, traditionally there have been differences in the implementation of the `Referer` [sic] header as defined in [RFC7231]: When a web site A links to a web site B that redirects to a web site C, some browsers used to send a `Referer` header to web site C containing the URL of web site A, while some others used to send the URL of B. Nowadays, most browsers send the URL of web site A.

Another main goal of this thesis is to demonstrate that the Web Infrastructure Model can be used to analyze the security of complex web applications and standards. To this end, we focus on a critical building block of almost any web application: user authentication and authorization.

Traditionally, user authentication in the web is based on a set of credentials, often an identifier (username, email address, etc.) and a secret (password). For each service a user intends to use, she has to set up a new set of credentials and verify her identity (typically by clicking on a link received via email). In this setup, the user has to select and memorize many different passwords, making signing up to a new service cumbersome.

To ease this burden on the user, another form of authentication has found widespread adoption on the web in recent years: *Web Single Sign-On* (SSO). SSO here means that a user can use her identity from one web site to log in at other web sites without establishing new credentials with these web sites. The verification of her identity is performed by exchanging messages between the web sites. In this scenario, different entities are involved in the login process in different roles, hence the term *federated authentication*.

A closely related concept is *federated authorization*, where a user grants a web site A access to her resources at web site B. For example, a user allows a photo printing service to access her photos on a social network.

The most popular framework for federated authorization is OAuth 2.0, in the following often simply called *OAuth*. OAuth 2.0 was released in 2012 as [RFC6749] and [RFC6750] by the IETF Web Authorization Protocol (OAuth) Working Group.[2] The OAuth standard defines a web-based protocol that allows a user to grant a so-called *client* web site access to her resources (data or services) at a so-called *resource server* (RS). (Here, the *client* is not to be confused with a browser.) In the protocol, the user is temporarily redirected to a so-called *authorization server* (AS), which, in many deployments, is the same entity as the RS. In practice, OAuth is often used for federated authentication as well, although it was originally only designed for authorization. In this case, the client is also called the *relying party* (RP), and the authorization server and resource server form what is called the *identity provider* (IdP).

OAuth is used by companies such as Amazon, Facebook, Google, Microsoft, Yahoo, GitHub, LinkedIn, StackExchange, and Dropbox, enabling billions of users [Sim] to share their data and resources. This makes OAuth probably the most widely used method for federated authorization/authentication on the web.

OAuth is diverse: The IETF Web Authorization Working Group regards OAuth as a *framework* for protocols. It supports four different modes of operation that utilize different kinds of tokens with different security requirements. Many details of these flows are left to the developers of a concrete implementation of the protocol. For example, it is left unspecified how redirections or state management techniques are realized.[3] It will become evident, however, that details such

[2] In this work we only consider OAuth 2.0, which uses a different approach and architecture than its predecessor, OAuth 1.0(a). For example, OAuth 1.0(a) required cryptographic operations even for simple setups, defined different roles (without a separation between the authorization server and the resource server), and had no easy method built in to revoke tokens. [OAuthD]

[3] Even [RFC6749] states that "as a rich and highly extensible framework with many optional components, on its

as these are critical for the secure operation of OAuth-based authorization and authentication solutions.

OAuth is also the foundation for the protocol OpenID Connect (OIDC), which aims at defining a new standard for federated authentication within the OAuth framework. The OpenID Connect standard is managed by the OpenID Foundation and already enjoys broad and growing industry support. OpenID Connect is currently in use and actively supported by PayPal, Google, Oracle, and Microsoft, among many others.[4] OpenID Connect extends OAuth by clearly defined interfaces for user authentication and additional (optional) features, such as dynamic identity provider discovery and relying party registration, signing and encryption of messages, and logout. OpenID Connect uses two of the four grants of OAuth to define three so-called *flows*. It operates within the boundaries defined by the OAuth framework, but adds many specific details that change fundamental properties of the protocol (e.g., dynamic registration, encryption/signing, a new token type, etc.).

In this thesis, we use our generic formal model to provide the most detailed security analysis of OAuth 2.0 and OpenID Connect. This analysis is based on the respective standards themselves rather than individual implementations: Security problems in the specifications usually lead to vulnerabilities in many concrete implementations, whereas errors found in implementation typically only affect a limited number of deployments. To acquire precise results, we analyze both systems separately. Although in particular OAuth had been analyzed many times before, our analysis reveals previously unknown attacks on both SSO systems. We develop fixes against these attacks and prove the security for the (fixed) systems. Since our results are based on a comprehensive model, our proofs can exclude large classes of attacks against OAuth and OpenID Connect, including yet unknown attack vectors.

1.1. Contributions of this Thesis

In detail, our contributions can be outlined as follows:

1.1.1. The Web Infrastructure Model

Our first contribution, a generic formal model for the web infrastructure, is intended to capture important classes of attacks on web applications and the web standards. To this end, we model properties of the network level (most importantly, TCP/IP and UDP messages), transmission protocols (DNS, HTTP and HTTPS, WebRTC and WebSocket messages), and applications (web browsers, web servers, and DNS servers).

For web browsers and servers, we model handling of HTTP and HTTPS messages including cookies, the `Referer` and `Origin` headers, redirections and Strict Transport Security (STS). The model of web browsers captures the concepts of windows, documents, and iframes as well as new

own, this specification is likely to produce a wide range of non-interoperable implementations".

[4] The OpenID Connect protocol is very different to its predecessor, OpenID. OpenID is not widely used any longer on the web. We therefore only consider OpenID Connect in the following.

technologies, such as Web Storage, Cross-Document Messaging, WebSockets and WebRTC. We take into account the complex security restrictions that are applied when accessing or navigating other windows, including the same-origin policy. JavaScript is modeled in an abstract way by what we call scripts. Scripts can be sent around and can, for example, create iframes, send and receive HTTP(S) messages using the XMLHttpRequest API, and initiate WebRTC connections.

Our model is based on a general Dolev-Yao-style communication model [AF01; DY83], in which processes have addresses (modeling IP addresses) and messages are modeled as formal terms, with properties of cryptographic primitives, such as encryption and digital signatures, expressed as equational theories on terms.

We define two different kinds of attackers, web attackers and network attackers, and consider two ways of dynamically corrupting browsers.

Altogether, the model is the most comprehensive and detailed model for the web infrastructure to date. It is defined independently of any web application and as such can be used to analyze the security and privacy of any web application which uses a subset of the features supported by the model.

While the case studies in this thesis focus on security aspects, our model has been used successfully to analyze privacy as well [FKS15a; FKS15b].

1.1.2. Formal Analysis of OAuth 2.0

To prove that the web model is useful in analyzing security properties of real-world web applications, and as a contribution in its own right, we perform a detailed and comprehensive analysis of federated authentication and authorization with OAuth.

Model of OAuth 2.0

Using our generic web model, we build a formal model of OAuth, closely following the OAuth 2.0 standard [RFC6749]. We additionally use the OAuth security recommendations [RFC6819], supplementary RFCs and OAuth Working Group drafts (e.g., [RFC7662], [BLZ18]), and current web best practices (e.g., regarding session handling). This helps us to fill gaps where [RFC6749] does not fix certain aspects of the protocol (while making as few assumptions as possible), to avoid any known attacks on OAuth, and to create a model with state-of-the-art security features in place. Our model includes clients and AS/RS that (simultaneously) support all four modes of operation available in OAuth and can be dynamically corrupted by the adversary. Also, we model all configuration options of OAuth.

Formalization of Security Properties

Based on this model of OAuth, we provide three central security properties of OAuth: authorization, authentication, and session integrity, where session integrity in turn is concerned with both authorization and authentication.

Session integrity here means that (concerning authorization) an attacker should be unable to force a client, during its interaction with an honest user, to access the services of the attacker instead of the user's resources at the RS and that (concerning authentication) the attacker should be unable to force an honest user's browser to be logged in at a client under the attacker's account. Attacks on session integrity are often referred to as session swapping, session fixation, or Login Cross-Site Request Forgery (CSRF).

Attacks on OAuth 2.0 and Fixes

While trying to prove these properties, we discovered five new attacks on OAuth:

– The first attack, the *307 Redirect Attack*, breaks the authorization and authentication properties. In this attack, authorization servers inadvertently forward user credentials (i.e., username and password) to the client or to the attacker.

– In the second attack, dubbed the *AS Mix-Up Attack*, a network attacker playing the role of an AS can impersonate any victim. This severe attack, which again breaks the authorization and authentication properties, is caused by a logical flaw in the OAuth 2.0 protocol.

– Three further attacks (*State Leak Attack*, *Naïve Client Session Integrity Attack*, and *Across-AS State Reuse Attack*) allow an attacker to force a browser to be logged in under the attacker's name at a client or force a client to use a resource of the attacker instead of a resource of the user, breaking the session integrity property.

We present our attacks on OAuth in detail in Section 3.3, also showing how the attacks can be fixed by changes that are easy to implement in new and existing deployments of OAuth and OpenID Connect. We have verified all five attacks on actual implementations of OAuth and OpenID Connect.

To ensure a better understanding of the relevance of our attacks and to give an overview of the attack mitigations needed for our proof (see below), we also give a summary of previously known attacks on OAuth.

We notified the respective working groups of our findings. They confirmed the attacks and we are now working with them to update the standards and recommendations (see Chapter 5 for details).

Proof of Security for OAuth 2.0

Finally, we use our model to show that OAuth, when fixes according to our recommendations, meets strong authorization, authentication, and session integrity properties. The assumptions required for this proof are realistic and can be met in real-world deployments. This is the first proof of the security of OAuth in a comprehensive formal model of the web.

1.1.3. Formal Analysis of OpenID Connect

Akin to our analysis of OAuth, we also provide the first in-depth formal security analysis of OpenID Connect. We use our formal web model and strong attacker models to analyze and prove the security of all flows available in the OIDC standard. We include many of the optional features and in particular the *Discovery* and *Dynamic Client Registration* extensions. The Discovery extension [Sak+14b] can be used by an OpenID Connect RP to find the IdP responsible for a certain identity. Using Dynamic Client Registration [SBJ14], the RP can then register itself at the IdP (for details, see Section 4.2). Both extensions add additional steps and interfaces to the protocol and thus create new attack surfaces.

In detail, our contributions are as follows.

Attacks on OpenID Connect and Security Guidelines

We first show that most of the attacks against OAuth (discovered by us or known previously) also apply to OpenID Connect. Even though OIDC is based on OAuth, we show that additional security features of OIDC have to be circumvented to carry out the attacks. Also, some attacks work somewhat differently in OIDC. In fact, certain features of OIDC enable attacks not applicable to OAuth. We briefly present these attacks, including some previously undocumented variants.

As before, we derive security recommendations from all of these attacks, which are then incorporated into our model of OIDC. Our formal security analysis (see below), demonstrates that these defenses are in fact effective and sufficient.

Formal Model of OpenID Connect

Our formal analysis of OIDC is based on our generic model of the web infrastructure. We build our formal model of OIDC by closely following the standard, employing the defenses and mitigations discussed earlier in order to create a model with state-of-the-art security features in place. Our model includes RPs and OPs that (simultaneously) support all modes of OIDC and can be corrupted dynamically by the adversary.

Formalization of Security Properties

Based on this model of OIDC, we formalize four main security properties of OIDC: authentication, authorization, session integrity for authentication, and session integrity for authorization. We also formalize further OIDC specific properties.

Proof of Security for OpenID Connect

Using the model and the formalized security properties, we subsequently show that OIDC in fact satisfies the security properties. This constitutes the first proof of the security of OIDC and for the Discovery and Dynamic Registration extensions.

1.2. Structure of this Thesis

This thesis is structured as follows: In Section 1.3, we give an overview of the papers that were published during the research project. In Section 1.4, we present related work. Afterwards, in Chapter 2, we describe the web model, with all definitions and technical details presented in Appendix A. Chapter 3 covers the security analysis of OAuth 2.0, with formalizations and proofs provided in Appendix B. Likewise, in Chapter 4, we present the security analysis of OpenID Connect, again with formalizations and proofs presented in Appendix C. We discuss the impact of our work in Chapter 5. Finally, we discuss future work and conclude in Chapter 6.

1.3. Publications

This thesis is based on five previous scientific papers that were published in international security conferences. An overview of these publications is given below.

The first main contribution presented in this thesis, the generic web model, was introduced in the first publication [FKS14] and extended throughout the subsequent publications. In the thesis at hand, we provide a new in-depth explanation of the web model. The second main contribution, the case studies on OAuth 2.0 and OpenID Connect, was developed in the last two publications [FKS16; FKS17]. The case studies on BrowserID and SPRESSO (see below) are not part of this thesis.

This thesis improves on these publications in two aspects: First, we extend the Web Infrastructure Model by adding models for WebRTC and WebSockets. Second, we show how code injection can be used in the AS Mix-Up Attack to break authorization in the OAuth authorization code grant even if client secrets are used. Additionally, the Across-AS State Reuse Attack is now featured as a separate attack, and the terminology throughout the thesis was adapted to follow the official terminologies in the OAuth and OpenID Connect specifications more closely.

An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System [FKS14]

In this paper, we presented the **first version of our formal web model**. We employed the model to study the security of the single sign-on system *Mozilla BrowserID* (also known as *Mozilla Persona*) which was the first web SSO to claim a certain kind of privacy property: IdPs do not learn at which RPs their users log in.[5]

During our **security analysis of the secondary IdP mode of BrowserID**, we discovered a number of severe attacks on BrowserID. For example, we found attacks that allowed an attacker to authenticate as any Google Mail or Yahoo user. We notified Mozilla of our findings and were awarded a Mozilla Security Bug Bounty.

[5]Since the secondary IdP mode of BrowserID, by design, does not provide privacy, we did not study the privacy properties of BrowserID in this paper.

With defenses against these attacks in place, we were able to prove that BrowserID provides authentication and session integrity for authentication. This marks the first time that these properties were proven for an SSO system of the complexity of BrowserID in a comprehensive web model.

Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web [FKS15a]

In the second publication, we complemented our prior analysis of BrowserID by studying the **security of the primary IdP mode** and we analyzed the **privacy properties of BrowserID**.

Using our model, we found a new attack on session integrity and developed a fix against this attack. We were then able to show authentication and session integrity properties for the fixed protocol. For the analysis, we extended the formal web model by adding the sessionStorage API and user identities (see Section 2.10).

Regarding privacy, we found a set of attacks that break the privacy promise of BrowserID completely. We were able to trace the roots of the problem to an oversight by the developers: The mere presence of a certain iframe enabled us to detect whether a user is logged in at a specific RP or not. Since there was no way to fix the problem without a major redesign of the whole system, we decided to address this in our next publication.

SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web [FKS15b]

To demonstrate that the privacy property of BrowserID can actually be fulfilled, we **designed SPRESSO** as a decentralized and federated SSO system where identity providers cannot learn at which web sites their users log in. Using our model, we first completed the design of SPRESSO on paper and proved its privacy, authentication, and session integrity properties before actually implementing the system.

For the **privacy analysis**, we had to modify parts of our formal model in order to be able to show indistinguishability properties between traces. In particular, we removed instances of nondeterminism in certain places, e.g., regarding the order of handling of network messages. For an accurate privacy analysis, we added the `Referer` header to the browser model.

A Comprehensive Formal Security Analysis of OAuth 2.0 [FKS16]

In the next step, we studied the **security of OAuth 2.0**. This is the first paper in which we not only analyzed authentication properties, but also authorization properties (both including session integrity).

Although OAuth 2.0 had been analyzed numerous times before (see Section 1.4), we found **new attacks on OAuth**, as described above. The OAuth Working Group acknowledged the

attacks and we are actively involved in standardization efforts for respective mitigations (see Chapter 5 for details).

In order to prove the effectiveness of our mitigations against the leakage of sensitive tokens, we added Referrer Policies to the model (see Section 3.3.3 for details).

The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines [FKS17]

After our successful analysis of OAuth 2.0, we turned our attention to OpenID Connect. As outlined above, OpenID Connect is based on OAuth 2.0, but introduces a well-defined scheme for authentication and new features like discovery and registration.

We demonstrated that the attacks found in the previous publication also apply to OpenID Connect. With fixes against these attacks in place, we were able to **prove authentication, authorization, and session integrity properties** for OpenID Connect.

We also introduced generic models for HTTPS web servers which helped to keep the models for OpenID Connect simpler and clearer.

1.4. Related Work

In the following, we give an overview of existing research in the area of formal web security analysis, security analysis of OAuth 2.0 and security analysis of OpenID Connect.[6]

1.4.1. Formal Web Security Analysis

Early research in the direction of formal web security analysis includes work by Kerschbaum [Ker07], in which a Cross-Site Request Forgery protection proposal was formally analyzed using a very simple model of browsers, scripts, and web pages expressed using Alloy, a finite-state model checker [Jac02].

In their seminal work, Akhawe et al. [Akh+10] initiated a more general formal treatment of web security. Again the model was provided in the Alloy modeling language. In five case studies, Akhawe et al. showed that their model can be used to identify security problems in standards and web applications. The model by Akhawe et al. has been analyzed extensively in the diploma thesis of the author [Fet11].

Kumar et al. [Kum12; Kum14; Pai+11] combined an Alloy model with BAN logic to analyze and automatically find attacks in web protocols. They applied their approach to the Security Assertion Markup Language (SAML) and OAuth (see below).

Bansal et al. [Ban+13; Ban+14; BBM12] built the WebSpi model for the web infrastructure, which is encoded in the modeling language of ProVerif. ProVerif is a specialized tool for cryptographic protocol analysis [Bla01] based on a variant of the applied pi-calculus [AF01].

[6] OAuth 2.0 is very different to its predecessor, OAuth 1.0(a). Likewise, OpenID Connect is very different to its predecessor, OpenID. We therefore only list research on OAuth 2.0 or OpenID Connect in the following.

The WebSpi library models several important features of the web infrastructure, such as cookies, origins, local storage, and CORS. As such, at the time of writing, it is the most comprehensive web model that is amenable to tool-based analysis (cf. Section 6 for recent developments). The WebSpi model has been applied successfully to find attacks in encrypted web storage services and deployments of OAuth 2.0 (for more details, see the next subsection).

While the models above support (partially or fully) automated analysis, they are necessarily tailored to and limited by constraints of the respective tools. For example, models for Alloy are finite-state. Terms (messages) need to be encoded in some way as they are not directly supported. Due to the analysis method employed in ProVerif, the WebSpi model is of a monotonic nature. For instance, cookies and localStorage entries can only be added, but not deleted or modified. Also, the number of cookies per request is limited, and several important features are missing (e.g., cross-document messaging, different redirection codes, and a precise structure of windows, documents, and iframes). These automated approaches therefore may miss important problems. Our model of the web is much more comprehensive and accurate, but requires manual proofs, at least for now (see the discussion on future work in Chapter 6).

Bai et al. [Bai+13] developed the AuthScan tool which is capable of extracting an authentication protocol specification from the protocol implementation. The extracted protocol specifications were then fed into ProVerif for a security analysis. Armando et al. [Arm+08; Arm+13] performed analyses of SSO (SAML and OpenID) based on custom-built models in the High-Level Protocol Specification Language (HLPSL++). Compared to our work, the models by Bai et al. and Armando et al. are not very detailed, since they focus mainly on the logic of the protocols. They do not consider a comprehensive model of the web infrastructure.

Bohannon and Pierce [BP10] proposed a formal model of a web browser core as a basis for experiments with security policies and mechanisms. There are several approaches towards tracking and controlling information flow inside web browsers [Bau+15; Gro+12; Guh+11; HBS16; Yos+09]. Börger et al. [BCG12] presented an approach for the analysis of web application frameworks, focusing on the server. None of these works include a model for the web infrastructure.

1.4.2. Security Analysis of OAuth 2.0

The work closest to ours is the already mentioned research by Bansal et al. [Ban+14; BBM12]. They analyzed the security of OAuth using their WebSpi library and ProVerif. Bansal et al. modeled various settings of OAuth 2.0, often assuming the presence of common web implementation flaws, for example, CSRF and open redirectors in RPs and IdPs.

They identified previously unknown attacks on the OAuth implementations of Facebook, Yahoo, Twitter, and many other websites. As pointed out by Bansal et al., the main focus of their work was to discover attacks on OAuth rather than proving security. They have some positive results, which, however, are based on their more limited model. In addition, in order to prove these results, further restrictions were required, e.g., they considered only one authorization server per client and all authorization servers were assumed to be honest.

Wang et al. [Wan+13] presented a systematic approach to finding implicit assumptions in software development kits (SDKs) used for authentication and authorization. Their case studies include the Facebook PHP SDK and other SDKs implementing OAuth 2.0.

Pai et al. [Pai+11] analyzed the security of OAuth in a very limited model that does not incorporate generic web features. They showed that through their approach, based on the Alloy finite-state model checker, known weaknesses can be found.

Chari, Jutla, and Roy [CJR11] analyzed the security of the authorization code grant in the Universal Composability (UC) model, again without considering web features, such as semantics of HTTP status codes, details of cookies, or window structures inside a browser.

Besides these formal approaches, empirical studies were conducted on deployments of OAuth. In [SB12], Sun and Beznosov analyzed the security of three IdPs and 96 RPs. In [LM14], Li and Mitchell studied the security of ten IdPs and 60 RPs based in China. In [Yan+16], Yang et al. performed an automated analysis of four OAuth IdPs and 500 RPs. Shernan et al. [She+15] evaluated the lack of CSRF protection in various OAuth deployments. In [Che+14; SM14], practical evaluations on the security of OAuth implementations of mobile apps were performed.

Many of the works listed here led to improved security recommendations for OAuth as documented in [RFC6749] and [RFC6819]. These are already taken into account in our model and analysis of OAuth.

1.4.3. Security Analysis of OpenID Connect

As mentioned in the introduction, the only previous works on the security of OIDC are [LM16; Mla+16]. In [LM16], the authors found implementation errors in deployments of Google Sign-In, which is based on OIDC. In [Mla+16], the authors described a specific variant of the AS Mix-Up Attack (see Section 4.4) and highlighted the possibility of Server-Side Request Forgery attacks at RPs in the OIDC standard (see Section 3.4). In contrast to our work, neither [LM16] nor [Mla+16] are based on formal analysis or establish security guarantees for the OIDC standard.

2. The Web Infrastructure Model

In this chapter, we present our model of the web infrastructure. We start with an overview of our design goals, before elaborating on the scope of the model, and eventually outlining our sources and approach. Subsequently, we outline the architecture before delineating the building blocks of the model. At the end of this section we demonstrate that our our model can be extended easily, using WebRTC as an example.

We introduce notations, formalisms, and definitions only as far as they are needed here. Some definitions in this section are simplified for presentation, in particular regarding the handling of nonces. All details, including the full definitions, can be found in Appendix A.

2.1. Building a Model of the Web Infrastructure

The design goal was to create a precise and rather comprehensive model of the web infrastructure. To this end, we decided to create a "pencil-and-paper model", one that is not encoded in the language of an automated analysis tool, and as such, not constrained by the tool's abilities. As we discuss in Chapter 6, our model can serve as a basis for future efforts towards a tool-supported analysis of web standards and applications.

The scope of the model is chosen in a way that it covers meaningful classes of attacks on several layers of web applications, standards, and the web infrastructure:

- Attacks targeting **interactions between web applications and browsers**, for example, missing checks for the origin of messages sent between documents (e.g., [BJM08b; SS13]), the 307 Redirect Attack (cf. Section 3.3.1), or leakage of OAuth tokens from URI fragments (as described in [RFC6819]).
- Attacks targeting the **behavior of web browsers**, for example, DNS rebinding [Jac+07], and attacks on the integrity of cookies [Zhe+15].
- Attacks targeting **web application code on servers**, for example, lack of user authentication or cross-site request forgery protections.
- Attacks targeting the **network layer**, for example, forgery of DNS responses and TLS stripping.

Conversely, certain aspects of the web infrastructure are considered out of the scope of our model. We do not model the following aspects:

- **Language details:** We employ an abstract model of JavaScript, i.e., we cannot model misuse of specific language features, timing attacks and race conditions in scripts, and do not track information flows inside scripts. We do, however, have an accurate model of the input and output behavior of JavaScript and our model precisely captures the effects that JavaScript, including malicious JavaScript, can have on other documents, network traffic, and the browser.
- **Typing, corruption, and system details:** We neither model buffer overflows or similar attacks in browsers, rendering or scripting engines, nor capture memory corruption attacks that can be triggered from JavaScript [GMM16]. We can, however, model the effects of browsers compromised by an attacker through such attacks.
- **User interface details:** We do not model a user interface, e.g., security indicators, overlapping frames, style sheets, etc. This prevents us from modeling, for example, browser fingerprinting [Eck10] or clickjacking attacks [Akh+14; Hua+12]. Again we can, however, capture the effects of such attacks. We also assume that a user can always distinguish between HTTPS and HTTP sites and that the user does not "click through" (ignore) browser warnings.
- **Attacks on cryptography and TLS:** As is typical for Dolev-Yao models, we assume that cryptographic primitives cannot be broken. For example, we assume that attackers cannot eavesdrop on the plain text exchanged in TLS connections without knowledge of the keys in use.
- **Proprietary and deprecated technologies:** Our model neither supports proprietary browser extensions such as ActiveX nor browser plugins like Flash or Java that use the Netscape Plugin API (NPAPI). While the NPAPI was once widely used, it has now been deprecated by browser vendors [Sch14; Sme15], first and foremost for its negative impact on browser security and user privacy [Sol+10].

To build our model, we studied the standards that define the web infrastructure and translated relevant parts into formalisms and notations of the model. As we describe more precisely in the following subsection, we provide, with a varying degree of abstraction, models for

- networking (IP [RFC791], TCP [RFC793], UDP [RFC768], DNS [RFC1034; RFC1035]),
- URIs [RFC3986] and HTTP/1.1 [RFC7230; RFC7231; RFC7232; RFC7233; RFC7234; RFC7235],
- HTML5 [Ber+17],
- Cookies [RFC6265],
- Web Storage [Ian16],

– Web Messaging [Hic15],

– the web origin concept [RFC6454],

– Fetch [Fetch],

– WebSockets [RFC6455],

– WebRTC [Ber+18]), and

– security technologies (HTTP Basic Authorization [RFC7617], Strict Transport Security [RFC6797], Referrer Policy [ES17]).

The initial version of the model, presented in [FKS14], contained only a subset of these technologies. As outlined in Section 1.3, more features were added over time.

To fill gaps where standards lacked details and to check the standards against the real-world implementations, we ran experiments for certain aspects of the modeling in the most popular browsers (Microsoft Internet Explorer and Microsoft Edge, Google Chrome, Mozilla Firefox).[1]

2.2. Architecture

We now give a high-level overview of the architecture of the WIM before going into the details.

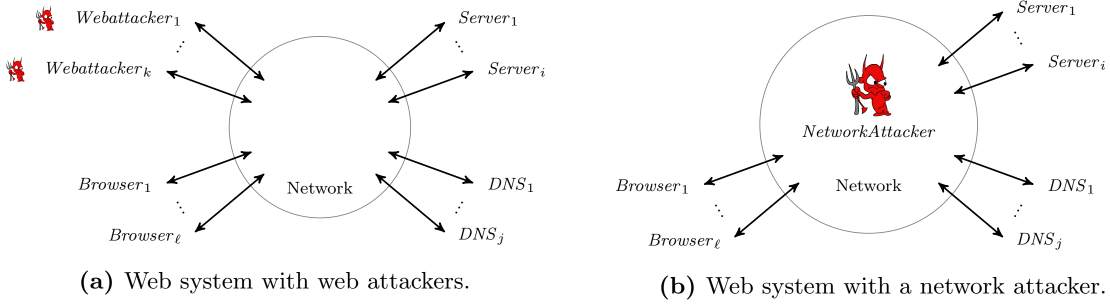
The Web Infrastructure Model defines a generic communication model, and, based on it, web systems consisting of web browsers, DNS servers, web servers, and web and network attackers.

All parties in a web system are formalized by **(Dolev-Yao) processes**. A (Dolev-Yao) process consists of a set of addresses the process listens to, a set of states (represented as formal terms, see below), an initial state, and a relation that takes an event (network message) and a state as input and (nondeterministically) returns a new state and a sequence of output events. The relation models a computation step of the process. It is required that the output can be computed (formally, derived in the usual Dolev-Yao style) from the input event and the state.

Processes communicate via **events**, which consist of a message (formal term) plus a receiver and a sender address. In every step of a **run** of a web system, one event is chosen nondeterministically from a pool of waiting events and is delivered to one of the processes that listens to the event's receiver address. The process can then handle the event and output new events, which are added to the pool of events, and so on.

A **web system** (as illustrated in Figure 2.1) formalizes the web infrastructure and web applications. It contains honest and attacker processes. Honest processes can be web browsers, web servers, or DNS servers.

[1] In particular, we ran experiments in browsers to check corner cases that are not defined in standards, for example: Which `Referer` header is set if a browser is redirected multiple times? How do browsers handle the "secure" flag for cookies transferred from non-secure origins? Can cookies with the secure flag be overwritten by non-secure cookies?



(a) Web system with web attackers.

(b) Web system with a network attacker.

Figure 2.1. Illustration: Web systems with typical attacker setups.

A **browser** is thought to be used by one honest user, who is modeled as part of the browser. User actions, such as following a link or entering credentials, are modeled as nondeterministic actions of the web browser. We provide a detailed model of web browsers, a simplified model for DNS servers and a generic framework to create customized servers (since the inner workings of servers depend heavily on the application that is to be analyzed). Typically (and included in our formalizations of browsers and servers), honest parties can become corrupted by an attacker.

Attackers can be either **web attackers** (who can listen to and send messages from their own addresses only) or **network attackers** (who may listen to and spoof all addresses and therefore are the most powerful attackers). Typically, one would consider either a single network attacker (which then also subsumes the DNS server, see below) or a set of web attackers, see Figure 2.1. We provide a full model covering both types of attackers.

2.3. Terms, Messages, and Events

As usual in Dolev-Yao models (see [AF01; DY83]), messages are expressed as formal *terms* over a signature Σ. The signature contains constants (for (IP) addresses, strings) as well as sequence, projection, and function symbols (for encryption/decryption and signatures). We also define a set of nonces $\mathcal{N}$, disjoint from Σ. Nonces can be used to model, among others, symmetric and asymmetric encryption keys.

Terms are defined inductively: All constants and nonces are terms. Additionally, if $f \in \Sigma$ is an n-ary function symbol for some $n \geq 0$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, f_n)$ is a term. For example, a sequence of two strings and a secret $x \in \mathcal{N}$ asymmetrically encrypted under the public key belonging to the private key $k \in \mathcal{N}$ could be

$$\langle \mathtt{abc}, \mathtt{def}, \mathsf{enc_a}(x, \mathsf{pub}(k)) \rangle \,.$$

The *equational theory* associated with Σ is defined as usual in Dolev-Yao models. For instance, the equation in the equational theory which captures asymmetric decryption is

$$\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x \,.$$

The theory induces a congruence relation $\equiv$ on terms, capturing the meaning of the function symbols in Σ. Let, for example, req, k', and $k_{\text{example.com}}$ be some terms. We then have that

$$\mathsf{dec_a}(\mathsf{enc_a}(\langle req, k' \rangle, \mathsf{pub}(k_{\text{example.com}})), k_{\text{example.com}}) \equiv \langle req, k' \rangle ,$$

i.e., these two terms are equivalent w.r.t. the equational theory.

We define *events* to capture network messages. Events are terms of the form $\langle a, f, m \rangle$, where a and f are interpreted to be the receiver and sender IP addresses, respectively, and m is the message (e.g., an HTTP request, see below).

We also introduce a notation for *mappings* (dictionaries):

$$[\mathtt{user}{:}\mathtt{alice}, \mathtt{password}{:}x] = \langle \langle \mathtt{user}, \mathtt{alice} \rangle, \langle \mathtt{password}, x \rangle \rangle .$$

2.4. Dolev-Yao Processes

Parties in a web system are formalized as *Dolev-Yao processes.* A Dolev-Yao process p is formalized as a tuple

$$p = (I^p, Z^p, R^p, s_0^p) .$$

The components are defined as follows:

- I^p is the set of IP addresses for which the process receives messages.
- Z^p is the set of states of the process. States of processes are encoded as terms.
- R^p is the relation defining the process' behavior. The relation maps an input event and a state to a sequence of output events and a new state. Algorithm 2.1 in Section 2.9 shows a simple example for a relation of a Dolev-Yao process.
- s_0^p is the initial state of the process.

We require that all output events and the new state must be *derivable* from the process' old state and the input event. For a set of terms M, the set of derivable terms $d(M)$ can be inductively defined as follows:

- For every constant c in Σ we have that $c \in d(M)$.
- For every term $m \in M$ we have that $m \in d(M)$, as well as all terms congruent to m.
- If $f \in \Sigma$ is an n-ary function symbol for some $n \geq 0$ and $t_1, \dots, t_n \in d(M)$, then $f(t_1, \dots, t_n) \in d(M)$.

For example, we have that $a \in d(\{\mathsf{enc_a}(\langle a, b, c \rangle, \mathsf{pub}(k)), k\})$. For a Dolev-Yao process that receives an input event e_{in} in a state s and outputs the new state s' and the events $E_{\text{out}} = \langle \langle a_1, f_1, m_1 \rangle, \dots, \langle a_n, f_n, m_n \rangle \rangle$ for $n \geq 0$, we require that $m_1, \dots, m_n \in d(\{e_{\text{in}}\} \cup \{s\})$ and $s' \in d(\{e_{\text{in}}\} \cup \{s\})$.

2.5. Attackers

We consider two types of attackers in this thesis: web attackers and network attackers.

Web attackers are our model for adversaries on an end-user internet connection, i.e., they can send and receive messages just like a normal user can, but do not have access to the network infrastructure. They can therefore neither intercept messages that are not sent directly to them nor spoof the sender addresses of their messages. In the internet, these attacks are prevented by the routing in the network and the handshake used in most protocols (e.g., TCP).

Network attackers, however, have full control over the network: They can intercept all messages and spoof all sender addresses. In real networks, such attacks can be performed, for example, by adversaries using ARP-spoofing attacks in local networks,[2] or nation-state sponsored adversaries with access to telecommunication infrastructure.

In the WIM, the so-called *attacker process* is a nondeterministic Dolev-Yao process that records all messages it receives and outputs all events it can possibly derive from its recorded messages. More formally, an attacker process (I, Z, R, s_0) that receives an input event e_{in} in a state s outputs the new state $s' = \langle e_{\text{in}}, E_{\text{out}}, s\rangle$ and the events $E_{\text{out}} = \langle\langle a_1, f_1, m_1\rangle, \ldots, \langle a_n, f_n, m_n\rangle\rangle$ for some $n \geq 0$, where the sender addresses $f_1, \ldots, f_n$ are chosen nondeterministically from I, the receiver addresses $a_1, \ldots, a_n$ are chosen nondeterministically from all IP addresses, and the messages $m_1, \ldots, m_n$ are chosen nondeterministically from $d(\{e_{\text{in}}\} \cup \{s\})$.

Hence, an attacker process carries out all attacks any Dolev-Yao process could possibly perform, but it cannot break cryptography. Whether an attacker process is a network attacker or a web attacker is defined by the set of IP addresses I the attacker has access to: The network attacker has access to all IP addresses, i.e., can receive messages intended for other parties and spoof their sender addresses, whereas the web attacker has a separate set of IP addresses that do not overlap with the IP addresses of honest processes.

As already mentioned above, attackers can corrupt other parties. This is modeled by sending a special message, $m = \texttt{CORRUPT}$, to the respective process. The process from then on acts exactly like an attacker process with access to the process' last state. Details of corruption depend on the concrete model of the process: as we will see below, two variants of corruption are considered for browsers.

2.6. Scripts

In real-world web browsers, HTML documents are used to present static information as well as links, forms, and other interactive elements to users. Documents can also embed other documents using iframes (see Section 2.10.3). Today, HTML documents are often accompanied by one or more JavaScript code parts. The JavaScript code can, among others, manipulate the document, fill and send forms, follow links, open new windows, and store data in the user's

[2] In this case, the adversary would only have unrestricted access to the local network, not to communication outside of the local network.

browser (using cookies, localStorage, or sessionStorage, see below). JavaScript can also use the XMLHttpRequest API to send HTTP(S) requests to other servers and read information from the corresponding responses. Two JavaScripts in different documents in the same browser can use the postMessage API to communicate with each other (see Section 2.10.8).

In our model, we conflate an HTML document and all JavaScript code parts within the document or included in the document from external sources into what is called a *script*. Formally, a script is defined as a relation mapping an input term *in* to an output term *out*, where the output term must be derivable from the input term, i.e., $out \in d(in)$. On a high level (see Section 2.10.10 for more details), the input term *in* contains the *script state* (an arbitrary term which is persisted across script executions by the browser), information about documents and windows in the browser, persisted user data (from cookies, localStorage, and session Storage), and input messages (like XMLHttpRequest responses and postMessages). The output term *out* contains a new script state, new values for cookies, localStorage, and sessionStorage, and a command to be interpreted by the browser. The command can instruct the browser, for instance, to follow a link, fill and send a form, create an iframe, or use the XMLHttpRequest and postMessage API for communication.

It is important to note that a script can, in particular, represent a plain HTML document without JavaScript, for example, one that consists merely of links: when called by the browser, the script would nondeterministically choose a link and output it to the browser, which would then load the corresponding web page.

Akin to an attacker process, the so-called *attacker script* outputs everything that is derivable from the input, i.e., carries out all attacks an adversary, having control over a script, could possibly perform. It is also the representation for a script which includes script code from an untrusted third party, since in this case, the adversary can gain full control over the embedding script.

Since scripts are relations, they cannot directly be encoded in terms, such as HTTP responses. We therefore assign a unique name (a string) to each script. This is captured by the injective mapping script from the set of scripts to their names. Browsers, when processing an HTTP(S) response (see below), expect the response body to be of the form

$$\langle script, scriptstate \rangle\,,$$

consisting of the name of a script (*script*) and the script's initial state (*scriptstate*). When receiving the HTTP(S) response, the browser first stores both components in the term representing the document (see below). Then, when executing the script (see Section 2.10.10), the browser uses the relation $\mathsf{script}^{-1}(script)$ that defines the script's behavior. Any script can be delivered to a browser by any server, i.e., scripts are not inherently bound to any origin. For an example for a simple script that only issues a single command, refer to Algorithm C.2 in the appendix.

```
GET /show?page=42 HTTP/1.1
Host: example.com
Connection: keep-alive
Accept-Encoding: gzip, deflate, br
Accept: text/html
```

Figure 2.2. Example: HTTP GET request.

2.7. Systems and Web Systems

A *system* $\mathcal{W}$ is a set of processes. A *configuration* (S, E, N) of $\mathcal{W}$ consists of the states of all processes in the system S, the pool of waiting events E, and a sequence of unused nonces N. Systems induce *runs*, i.e., sequences of configurations, where each configuration is obtained by delivering one of the waiting events of the preceding configuration to a process, which then performs a computation and outputs new events. The process can change its own state during the computation and consume nonces. The transition from one configuration to the next configuration in a run is called a *processing step*. We write, for example,

$$Q = (S, E, N) \rightarrow (S', E', N')$$

to denote the transition from the configuration (S, E, N) to the configuration (S', E', N').

A *web system* captures the web infrastructure and web applications. It is formalized as a tuple

$$\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)\,.$$

It contains a system $\mathcal{W}$ consisting of honest and attacker processes, a set of scripts $\mathcal{S}$ (comprising honest scripts and the attacker script) with the mapping $\mathsf{scripts}$ from scripts to their respective names, and an initial pool of waiting events E^0. Typically, E^0 initially only contains an infinite number of *trigger messages* (the string `TRIGGER`) for each process such that any process can get triggered at any time.

Web systems are at the core of each analysis in the WIM. For the analysis of authorization in OAuth, for example, we use a web system comprising honest servers, browsers, and scripts, and the attacker process and the attacker script.

2.8. HTTP and HTTPS Messages

The protocols HTTP and HTTPS form the backbone of the web. In a real-world web browser, when a user visits the URL `http://example.com/show?page=42`, the browser first opens a TCP connection to `example.com`. For this connection, the browser and the server use a hard-to-predict *TCP sequence number* that impedes an adversary from spoofing messages for the TCP connection unless he has observed one or more messages already.

After establishing the TCP connection, the browser sends an HTTP GET request over this

```
HTTP/1.1 200 OK
Cache-Control: private, no-store
Cache-Control: max-age=0
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html;charset=utf-8
Set-Cookie: SID=c29MT05HYW5kVEhBTktzNFBISVNI

<html>
<form action="/login">
<input type="hidden" name="csrf" value="6803993619024429">
<input type="text" name="username">
<input type="password" name="password">
</form>
</html>
```

Figure 2.3. Example: HTTP response.

connection, as shown in Figure 2.2. The request, in our example, contains the method `GET`, the path `/show`, the parameter `page` and its value `42`, the host name `example.com`, and a number of other headers. A POST request, or requests using other methods, may also contain a message body after the last header, for example to transmit form data to the server.

In the WIM, an HTTP request is represented by a term containing the string `HTTPReq`, a nonce (representing the TCP sequence number), an HTTP method, a host name, a path, URI parameters, request headers, and a message body. The HTTP request above is represented as

$$\mathit{req} := \langle \mathtt{HTTPReq}, n_1, \mathtt{GET}, \mathtt{example.com}, \mathtt{/show}, [\mathtt{page}{:}42], \langle\rangle, \langle\rangle\rangle\,.$$

The last two (empty) sequences in this term represent the list of message headers and the message body. The list of message headers is empty in this example since only certain headers are of interest in the WIM, for example cookie headers.[3]

An example for an HTTP response to the above GET request from a real server is shown in Figure 2.3. The status code `200` indicates a normal, successful response. The response contains, among others, a `Set-Cookie` header. After the headers, there is a message body containing a login form.

The response shown in Figure 2.3 could be formalized in the WIM as

$$\mathit{resp} := \langle \mathtt{HTTPResp}, n_1, \mathtt{200}, [\mathtt{Set\text{-}Cookie}{:}\langle \mathtt{SID}, \langle n_2, \bot, \top, \bot\rangle\rangle], \langle \mathtt{SCRIPT_LOGIN}, \langle \mathtt{csrf}, n_3\rangle\rangle\rangle\,.$$

The term *resp* contains, in the headers section, a cookie with the name `SID` and the value

[3]Obviously, any header can be encoded in terms representing HTTP messages, but only certain headers currently carry semantics in our model. For a list of these headers refer to Appendix A.4.4. The model can easily be extended to support other headers .

n_2 (see Section 2.10.4 for details on cookies), and, in the body section, the name of a script ($\mathtt{SCRIPT_LOGIN}$) and the script's initial state, here $\langle \mathtt{csrf}, n_3 \rangle$, carrying the nonce used in the form. We here assume that $\mathsf{script}^{-1}(\mathtt{SCRIPT_LOGIN})$ is a script that represents the login form in Figure 2.3, i.e., can instruct the browser to send a form equivalent to the login form. Recall that script states can be arbitrary terms and the script's exact use of, in this example, $\langle \mathtt{csrf}, n_3 \rangle$ depends on the definition of the script.

HTTPS protects HTTP messages by sending them over TLS connections. In the WIM, HTTPS messages are modeled by encrypting HTTP messages. The browser or server sending an HTTPS request encrypts the request asymmetrically using the public key of the intended receiver. Instead of using a public-key infrastructure with certificate authorities, we store a mapping from domains to public keys for all HTTPS-enabled domains in the initial state of each process that can send HTTPS requests. This way, different processes can use different asymmetric keys for encryption. Typically, all processes are configured to use a key for a specific domain that is only known to the (honest) server for that domain. To model compromised TLS connections, a domain can be mapped to a public key known to the attacker.

An HTTPS GET request for the URL `https://example.com/show?page=42` is formalized as

$$\mathit{httpsreq} := \mathsf{enc_a}(\langle \mathit{req}, k' \rangle, \mathsf{pub}(k_{\text{example.com}}))\,,$$

where k' is a fresh symmetric key (a nonce) generated by the sender of the request (typically a browser); the web server is supposed to use this key to encrypt the response.

HTTPS responses are encrypted symmetrically using the symmetric key contained in the corresponding HTTPS request. For example. an HTTPS response to *httpsreq* could be represented as

$$\mathit{httpsresp} := \mathsf{enc_s}(\mathit{resp}, k')\,.$$

2.9. Name Resolution

In the internet, the domain name system serves to resolve domains—like `example.com`—into IP addresses that can be used to establish network connections. In the WIM, we consider a flat DNS model in which DNS queries are answered directly by one DNS server and always with the same address for a domain. A full (hierarchical) DNS system with recursive DNS resolution, DNS caches, etc. could also be modeled to cover certain attacks involving details of the DNS system. Since DNS messages are always unencrypted in our model, a network attacker usually also takes the role of the DNS server (as illustrated in Figure 2.1). In our model, before sending out an HTTP(S) request, browsers always perform a DNS request to resolve the domain name to an IP address.

Formally, a DNS request for `example.com` is represented as

$$\langle \mathtt{DNSResolve}, \mathtt{example.com}, n_4 \rangle\,,$$

with a possible response being

$$\langle \texttt{DNSResolved}, \texttt{example.com}, \mathit{addr}, n_4 \rangle ,$$

where addr is an IP address. Just as in HTTP messages, a nonce (here n_4) is used to map the response to the request and emulates the (hard-to-guess) DNS query id and randomized port number in real-world DNS messages.

A *DNS server* d is modeled in a straightforward way as an atomic Dolev-Yao process $(I^d, \{s_0^d\}, R^d, s_0^d)$. Its initial (and only) state s_0^d encodes a mapping from domain names to IP addresses of the following form:

$$s_0^d = [\texttt{domain}_1{:}a_1, \texttt{domain}_2{:}a_2, \ldots] .$$

DNS queries are answered according to this table. If the requested DNS name cannot be found in the table, the request is ignored. Algorithm 2.1 shows the relation R^d implementing this behavior.

Algorithm 2.1 Relation of a DNS server R^d.

Input: $\langle a, f, m \rangle, s_0^d$
1: **let** domain, n **such that** $\langle \texttt{DNSResolve}, \mathit{domain}, n \rangle \equiv m$ **if possible; otherwise stop** $\langle \rangle, s_0^d$
2: **if** $\mathit{domain} \in s_0^d$ **then**
3:     **let** $\mathit{addr} := s_0^d[\mathit{domain}]$
4:     **let** $m' := \langle \texttt{DNSResolved}, \mathit{domain}, \mathit{addr}, n \rangle$
5:     **stop** $\langle \langle f, a, m' \rangle \rangle, s_0^d$
6: **stop** $\langle \rangle$, s_0^d

2.10. Web Browsers

Web browsers are the most complex entity defined in the WIM. The detailed model captures many important parts of the behavior of real-world web browsers.

A web browser b is an atomic Dolev-Yao processes $(I^b, Z_{\text{webbrowser}}, R_{\text{webbrowser}}, {s_0}^b)$. The relation $R_{\text{webbrowser}}$ modeling the behavior of browsers is defined by Algorithms A.1–A.11 in the appendix. The states $Z_{\text{webbrowser}}$ are of the form

$$\langle \mathit{windows}, \mathit{ids}, \mathit{secrets}, \mathit{cookies}, \mathit{localStorage}, \mathit{sessionStorage}, \mathit{keyMapping}, \mathit{sts}, \mathit{DNSaddress}, \\ \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{wsConnections}, \mathit{rtcConnections}, \mathit{isCorrupted} \rangle .$$

We will introduce the individual subterms of browser states over the next sections and give examples for concrete values.

2.10.1. Browsers and Users

We think of a browser to be used by one honest user—as long as the browser has not been corrupted by an attacker, see below. The user is a part of the browser model. Actions a user takes are modeled as nondeterministic actions of the web browser. For example, the web browser can spontaneously open new windows (tabs) and access random URIs. A browser randomly follows links in a document—here modeled as nondeterministic actions of the script that represents the document.

A browser's initial state contains, in the subterm *ids*, user secrets and user identities. A browser can have multiple *user identities*, public information such as email addresses that is accessible to any script. For example, if a browser is set up with the two identities

$$ids = \langle \texttt{alice@example.com}, \texttt{carla@example.com} \rangle\,,$$

then a login script can use either of these identities to log the user into some web site.

User secrets, such as passwords, are stored in the browser state subterm *secrets* indexed by origins and are only released to documents (scripts) with the corresponding origins.[4] For example, if the initial state of a browser contains a mapping of the form

$$secrets = [\langle \texttt{example.com}, \texttt{S} \rangle{:}n_5, \langle \texttt{example.com}, \texttt{P} \rangle{:}n_6]$$

then scripts loaded from the origin `https://example.com` (`S` for secure) would have access to the nonce (secret) n_5, and scripts loaded from the origin `http://example.com` (`P` for plain) would have access to n_6.

2.10.2. Two Types of Corruption

We also allow browsers to be taken over by attackers (dynamic corruption). In the real world, an attacker can exploit buffer overflows in web browsers, compromise operating systems, and physically take control over shared terminals. We model two types of corruption of browsers, namely *full corruption* and *limited corruption*, which are triggered by special network messages.

Full corruption models an attacker that gained full control over a web browser and its user. More precisely, the attacker gains access to all data, including secrets, stored in the browser, and can use the browser's IP address to send and receive arbitrary messages. Besides modeling a compromised operating system, full corruption can also serve as a vehicle for the attacker to participate in a protocol using secrets of honest browsers: Typically, in a concrete analysis, an attacker starts with no user secrets in its knowledge, but may fully corrupt any number of browsers and by this impersonate browsers/users.

Limited corruption models a browser that is taken over by the attacker after a user finished

[4] Since user secrets are restricted to certain origins, user identities and user secrets are not stored in the same data structure in the browser, i.e., there is no mapping from identities to secrets. A script can, however, non-deterministically select the *correct* secret for any identity.

her browsing session, i.e., after closing all windows of the browser. This form of corruption is relevant in situations where one browser can be used by many people, e.g., in an Internet café. Limited corruption removes from the browser's state all open documents, pending requests, user secrets, session storage, and session cookies before the attacker takes control over the browser. Non-session cookies and localStorage information (see Section 2.10.4) is left intact and can be misused by the attacker.

Whether or not a browser is corrupted is stored in the browser state subterm *isCorrupted*. By default, a browser is honest, i.e., we have $\mathit{isCorrupted} = \bot$. Otherwise, we have that $\mathit{isCorrupted} = \mathtt{FULLCORRUPT}$ or $\mathit{isCorrupted} = \mathtt{LIMITEDCORRUPT}$. In Section 2.10.9 we describe in more detail how a browser becomes corrupted and how a corrupted browser behaves. A browser, once corrupted, can never become honest again.

2.10.3. Windows and Documents

A browser may have a number of windows open at any time (representing the tabs in a real browser). The browser state subterm *windows* contains a list of windows opened in a web browser. Roughly speaking, documents in these windows and iframes are represented as subterms of windows.

More precisely, each window contains a list of documents of which one is "active". Being active means that this document is currently presented to the user and is available for interaction, similarly to the definition of active documents in the HTML5 specification [Ber+17]. The document list of a window represents the history of visited web pages in that window. A window may be navigated forward and backward (modeling forward and back buttons). This deactivates one document and activates its successor or predecessor.

Documents may contain subwindows, which correspond to iframes in real-world browsers, that may again contain other documents, and so on, effectively creating a forest of windows and trees. This structure induces a notion of a *top-level window* (a window which is not a subwindow), *parent window* (the window of which the current window is a direct subwindow) and *ancestor window* (some window of which the current window is a not necessarily direct subwindow) to describe the relationships in a tree of windows and documents.

A document inside a window is specified by a term which essentially contains the name of a script, the current state of the script, the input that the script obtained so far (from XML-HttpRequests, postMessages, WebRTC, and WebSocket messages), the origin of the document, and a list of subwindows.

A term describing a window or a document also contains a unique nonce, which we refer to by *document/window reference*. These references are used to match HTTP responses to the corresponding windows and documents from which they originate.[5]

Top-level windows may have been opened by another window. In this case, the term of

[5] We use references in several places. Some references are visible outside of the process using it, like the references in HTTP messages described above, while others stay internal to a process, like document references.

the opened window contains a reference to the window by which it was opened (the *opener*). Following the HTML5 standard, we call such a window an *auxiliary window*. Auxiliary windows are always top-level windows.

We call a window *active* if it is a top-level window or if it is a subwindow of an active document in an active window. The active documents in all active windows are exactly those documents a user can currently see/interact with in the browser.

The following is a slightly simplified[6] example of a list of windows containing a single window term carrying the window reference n_7, two documents identified by their document references n_8 and n_9, and a window reference to an opener (n_{10}):

$$\begin{aligned} \mathit{windows} = \langle\langle n_7, &\langle\langle n_8, \langle \mathtt{example.com}, \mathtt{P}\rangle, \mathtt{SCRIPT_LOGIN}, \langle \mathtt{csrf}, n_3\rangle, \langle\rangle, \langle\rangle, \bot\rangle, \\ &\langle n_9, \langle \mathtt{example.com}, \mathtt{S}\rangle, \mathtt{SCRIPT_HOME}, \langle\rangle, \langle\rangle, \langle\rangle, \top\rangle\rangle, n_{10}\rangle\rangle \ . \end{aligned}$$

The first document (reference n_8) was loaded from the origin $\langle \mathtt{example.com}, \mathtt{P}\rangle$, which translates into `http://example.com`. Its script is `SCRIPT_LOGIN`, the script state is $\langle \mathtt{csrf}, n_3\rangle$, and the input history of this script is empty. The document does not have subwindows and is inactive ($\bot$). The second document (reference n_9) was loaded from `https://example.com`, the script is `SCRIPT_HOME`, the script state is empty, there are no subwindows, and the document is active ($\top$).

2.10.4. Cookies and Web Storage

The browser model supports two mechanisms that allow web sites to persist data in the user's browser: Cookies and Web Storage.

Cookies

Cookies are name-value pairs that can be set either by scripts, or by web servers using the `Set-Cookie` header in a response. Once a browser has received a cookie, it will automatically add the cookie (in a `Cookie` header) to HTTP(S) requests to the domain from which it received the cookie.

Browsers store cookies per-domain and together with their attributes `secure`, `httpOnly`, and an expiration time. If `secure` is set, the cookie is only delivered to HTTPS origins. If `httpOnly` is set, the cookie cannot be accessed by JavaScript. The expiration time is optional: If it is not set, the cookie should be deleted by the browser when the user session ends, i.e., when the browser is closed. Otherwise, the cookie is kept in the browser until the expiration time is reached.[7]

[6] For brevity of presentation we omitted from the document term the full URL and the headers. See Definition 41 for details.

[7] Real-world browsers actually know two different attributes to determine the expiration time: "max-age" defines the maximum lifetime of the cookie in seconds, "expires" defines a concrete date and time on which the cookie will be deleted.

A cookie is, in our model, represented as a term of the form

$$\langle name, \langle value, secure, session, httpOnly \rangle \rangle \,.$$

The attributes *secure*, *session*, and *httpOnly* can either be $\top$ (true) or $\bot$ (false). If the *session* attribute is $\top$, the cookie is deleted when the browser is closed (this is only relevant for corruption, see Section 2.10.9). Otherwise, since the model does not have a notion of time, cookies are kept forever or until they are overwritten by a cookie of the same name.

An example for a server setting a cookie can be seen in the response *resp* in Section 2.8 above. Browsers store cookies in the subterm *cookies* of their state. The browser receiving *resp* would store the following subterm:

$$cookies = [\texttt{example.com}{:}[\texttt{SID}{:}\langle n_2, \bot, \top, \bot \rangle]] = \langle \langle \texttt{example.com}, \langle \langle \texttt{SID}, \langle n_2, \bot, \top, \bot \rangle \rangle \rangle \rangle \rangle \,.$$

In a request, say, req' that follows *resp*, the browser would then include the cookie's name and value (the attributes are not transferred to the server):

$$req' := \langle \texttt{HTTPReq}, n_{11}, \texttt{GET}, \texttt{example.com}, \texttt{/anything}, \langle \rangle, [\texttt{Cookie}{:}[\texttt{SID}{:}n_2]], \langle \rangle \rangle \,.$$

Web Storage

Web Storage is an API with a similar purpose as cookies, but with two major differences: First, WebStorage can only be accessed by JavaScript. Second, WebStorage is designed to handle much more data than cookies, which traditionally are limited to about 4 kilobytes of data. Web Storage offers two different kinds of storage: localStorage and sessionStorage. LocalStorage data is separated by origins, i.e., any script loaded from the same origin shares the same localStorage data set, but scripts cannot access other origin's localStorage data. SessionStorage is additionally indexed by top-level windows, i.e., two windows need to share the same origin and same top-level window in order to access the same sessionStorage data set.

Our implementation of Web Storage is straightforward: Scripts receive from the browser the localStorage and sessionStorage data sets accessible to them. This data is represented as arbitrary terms. The scripts can alter both terms and then output the new values for localStorage and sessionStorage. Browsers store the data, indexed by origin or by origin and top-level window nonce, in their state's subterm *localStorage* and *sessionStorage*, respectively.

For example, assume that a script from `example.com` stored strings in each localStorage and sessionStorage and the top-level window of the window in which the script runs has the window nonce n_7. Then, the two subterms of the browser state would contain the following data:

$$localStorage = [\texttt{example.com}{:}\texttt{foo}]$$
$$sessionStorage = [\langle \texttt{example.com}, n_7 \rangle{:}\texttt{bar}]$$

2.10.5. HTTP(S) Message Dispatching

We now take a closer look at the two steps needed for browsers to send an HTTP request. In Section 2.10.9, we will then see how browsers handle incoming messages.

First, recall that every HTTP(S) request contains a nonce created by the browser. A server is supposed to include this nonce into its HTTP response. By this, the browser can match the response to the request (a real web browser would use the TCP sequence number and port number for this purpose).

Browsers can send HTTP(S) requests

- "spontaneously" by nondeterministically reloading some document or by navigating to some URI (both of which can happen at any time),
- triggered by a `Location` header redirection (i.e., in response to an incoming HTTP response),
- or triggered by a script that follows a link, opens a new iframe, submits a form, sends an XMLHttpRequest, or tries to establish a WebSocket connection (one of the script commands `HREF`, `IFRAME`, `FORM`, `XMLHTTPREQUEST`, or `WS_OPEN`, described in more detail in Section 2.10.9).[8]

In all of these cases, the browser first resolves the domain name to an IP address.

Step 1: DNS Resolution

To this end, the browser first records the HTTP request in a subterm of its state called *pendingDNS* along with the original URL for the request and a reference:

- In the case of HTTP(S) requests, the reference is the constant string `REQ` plus the window reference of the window from which the request originated.
- In the case of XMLHttpRequests, the reference is a sequence of three elements, the constant string `XHR`, the document reference of the document from which the request originated, and a term (usually a fresh nonce) that was chosen by the script that issued the XMLHttpRequest. This enables the script to have multiple XMLHttpRequests running in parallel.
- In the case of WebSocket connections, the reference is of the same form as in the previous case, except the first element is now the constant string `WS`.

For example, let req' be an HTTP request as above, url be the URL[9] of req', n_7 be the window from which the request originated, and n_4 the nonce chosen by the browser for the DNS request

[8] An HTTP request can also be triggered when a WebRTC proxy script is loaded, see Section 2.12.
[9] The exact encoding of URLs is not important here. See Definition 28 in the appendix for details.

(as in the example in Section 2.9). Until the DNS resolution for req' finishes, *pendingDNS* would be of the form

$$\mathit{pendingDNS} = [n_4{:}\langle\langle \texttt{REQ}, n_7\rangle, \mathit{req}', \mathit{url}\rangle]\,.$$

The browser then sends the DNS request required to resolve the domain name into an IP address.

Step 2: Dispatching the HTTP Request

After receiving the corresponding DNS response, the browser sends the HTTP request and stores it (along with the window/document reference, the original url, and the resolved IP address) in *pendingRequests*. Before sending the HTTP request, the cookies stored in the browser for the domain of the request are added as `Cookie` headers to the request. (Cookies with attribute `secure` are only added for HTTPS requests.) The browser also checks if the domain of the request is contained in the browser's list of strict transport security domains, and, if that is the case, rewrites the request from HTTP to HTTPS (see below).

In our example, assuming that `example.com` was resolved into the IP address *addr*, the browser state subterm *pendingDNS* would then be of the following form (see below for the meaning of $\bot$ in this term):

$$\mathit{pendingRequests} = \langle\langle\langle \texttt{REQ}, n_7\rangle, \mathit{req}', \mathit{url}, \bot, \mathit{addr}\rangle\rangle\,.$$

As we see below in more detail, when an HTTP response arrives, the browser uses the nonce in this response to match it with the corresponding HTTP request (if any is recorded) and checks whether the address of the sender is as expected. The reference recorded along with the request then determines to which window/document the response belongs.

HTTPS Requests

For HTTPS requests, a fresh symmetric key (a nonce) is generated and added to the request by the browser before the request is sent. The resulting message is then encrypted using the public key corresponding to the domain in the request. The symmetric key is recorded along with the request in *pendingRequests*. If, for example, the key n_{12} was chosen, *pendingRequests* would look like this:

$$\mathit{pendingRequests} = \langle\langle\langle \texttt{REQ}, n_7\rangle, \mathit{req}', \mathit{url}, n_{12}, \mathit{addr}\rangle\rangle\,.$$

The response is, as mentioned above, supposed to be encrypted with the symmetric key n_{12}.

2.10.6. WebSockets

WebSockets enable real-time bidirectional communication between JavaScript (in a browser) and a server. On a high level, WebSockets work as follows: First, the WebSocket protocol, which is always initiated by the client-side JavaScript, starts as a regular HTTP(S) request in

which the browser indicates that it wants to create a WebSocket connection. The server, if it supports WebSockets, then agrees to this upgrade in the HTTP(S) response. From then on, server and browser can, at any time, send WebSocket messages containing arbitrary data over the TCP connection used for the HTTP(S) request and response, which is kept open. When a browser receives a WebSocket message, its contents are delivered to the script that started the connection.

The WebSocket protocol is defined in [RFC6455], but we do not need to introduce all details here. The important details of the protocols are captured in our model of WebSockets, which we delineate in the following.

In the WIM, scripts in a browser can issue the command `WS_OPEN` to create a new WebSocket connection to some URL. Just as in the model for XMLHttpRequests, the script identifies this connection using a freely chosen reference, the WebSocket reference.

To create the requested WebSocket connection, the browser first (just as in a real-world browser) sends a regular HTTP(S) request. In this request, the browser sends an `Upgrade` header to inform the server of its intent to switch to the WebSocket protocol:

$$\langle \texttt{HTTPReq}, n, \texttt{GET}, \mathit{host}, \mathit{path}, \mathit{parameters}, \langle\langle \texttt{Upgrade}, \texttt{websocket}\rangle\rangle, \langle\rangle\rangle\ .$$

This request can additionally be encrypted if the script requested a WebSocket connection over TLS.

The server is then expected to respond with an HTTP(S) response with status code `101` and the same `Upgrade` header.[10] Upon receiving this response, the connection is considered "open" and the browser stores connection information (WebSocket reference, HTTP request nonce, symmetric encryption key) in the list of open WebSocket connections (browser state subterm *wsConnections*).

Now, server and browser can (at any time) send data to each other using network messages of the form

$$\langle \texttt{WS_MSG}, \mathit{nonce}, \mathit{data}\rangle$$

(possibly encrypted using the symmetric key from the TLS connection as above). In the browser, a script can send such a message by calling the command `WS_SEND` and providing, as a parameter to the command, the WebSocket reference created when opening the WebSocket connection. Incoming WebSocket messages are appended to the script's inputs (see Section 2.10.9 for more details).

2.10.7. Strict Transport Security

With the security mechanism *Strict Transport Security* (STS or HSTS), defined in [RFC6797], servers can send a special header, `Strict-Transport-Security` in HTTPS responses. When a

[10] In the real-world handshake, the browser also has to send a nonce in a special header which must then be hashed, together with a static string, by the server. This is intended to ensure that the server actually understands the WebSocket protocol. We overapproximate in the WIM and do not include this part of the handshake.

supporting browser encounters this header, it will add the domain from which it received the header to an internal list of STS domains. The browser will not allow any HTTP connections to domains in this list, but instead automatically change all such requests to use HTTPS. This ensures that a browser always uses a TLS connection to a server and can help to protect against TLS stripping attacks.[11]

Our browser model supports STS and maintains the list of STS domains in the browser state subterm *sts*:

$$sts = \langle \texttt{example.com}, \texttt{foo.example}, \texttt{bar.example}, \ldots \rangle \,.$$

2.10.8. WebMessaging

WebMessaging [Hic15] defines methods for communication between documents/scripts inside the same web browser. The most commonly used WebMessaging API is postMessage. Using the postMessage API, JavaScript can dispatch messages to other windows or receive messages from other windows. The target window always has to be specified explicitly. PostMessage implements two important security mechanisms:

- The sender of a message can define the origins which are allowed to receive the message or use the wildcard `*` to allow all target origins. For example, if a script defines `https://example.com` as the target origin, but the document loaded inside the target window was loaded from `http://example.com`, the browser will not deliver the message.
- The receiver of a message learns the origin of the sender of the messages and a reference to the sender's window. The browser ensures that this information is trustworthy, i.e., scripts that run in an uncompromised browser can rely on this information.

In the WIM, scripts can use the `POSTMESSAGE` to send data via postMessage to scripts in other windows. Scripts receive postMessages as part of their script inputs. See Section 2.10.10 for details.

2.10.9. Message Processing

As usual in our model, all actions of the web browser atomic Dolev-Yao process need to be triggered by an incoming message. The only "entry point" into the web browser is the algorithm outlined in the following (see Algorithm A.11 for details). Figure 2.4 provides an overview of the structure of this algorithm.

If the browser is corrupted, i.e., $isCorrupted \neq \bot$, it acts as an attacker process: it simply adds the input message m to its current state (i.e., knowledge) and then nondeterministically outputs an event derivable from its state. More formally, when a corrupted browser in the state s receives a message m for the browser's IP address a, the browser adds m to the subterm

[11] A `Strict-Transport-Security` header also contains a lifetime in seconds. After the lifetime has expired, the browser will remove the domain from the STS list if no new `Strict-Transport-Security` header was received in the meantime. This lifetime is not reflected in our model.

PROCESSING INPUT MESSAGE ***m***
m = **FULLCORRUPT:** Set *isCorrupted* := FULLCORRUPT.
m = **LIMITEDCORRUPT:** Clean secrets, windows, cookies, storage, set *isCorrupted* := LIMITEDCORRUPT.
m = **TRIGGER:** nondeterministically choose *action*:
  action **is** `script`**:** Call script of some active document. Outputs new state and command *cmd*.
    cmd = **HREF:** Initiate HTTP(S) request to URL in link.
    cmd = **IFRAME:** Create subwindow, initiate request to load URL into iframe.
    cmd = **FORM:** Initiate HTTP(S) GET/POST request to given URL with form data.
    cmd = **SETSCRIPT:** Change script in given document.
    cmd = **SETSCRIPTSTATE:** Change state of script in given document.
    cmd = **XMLHTTPREQUEST:** Initiate XMLHttpRequest.
    cmd = **BACK or FORWARD:** Navigate given window.
    cmd = **CLOSE:** Close given window.
    cmd = **POSTMESSAGE:** Send postMessage to specified document.
    cmd = **WS_OPEN:** Initiate HTTP(S) request to create new WebSocket connection.
    cmd = **WS_SEND:** Send WebSocket message over established WebSocket connection.
    cmd = **RTC_CREATE_PEERCONNECTION:** Create a new WebRTC connection object.
    cmd = **RTC_GET_OFFER:** Create and return WebRTC offer document.
    cmd = **RTC_SET_REMOTE:** Consume remote WebRTC offer.
    cmd = **RTC_GET_IA_INFO:** Return information needed for identity assertion.
    cmd = **RTC_SET_IA:** Store the local identity assertion in WebRTC connection.
    cmd = **RTC_GET_CHECK_IA_INFO:** Return information to check remote identity assertion against.
    cmd = **RTC_CHECKED_IA:** Set flag that remote identity assertion was checked.
    cmd = **RTC_SEND:** Send WebRTC message over established WebRTC connection.
  action **is** `urlbar`**:** Initiate request to some URL in new window.
  action **is** `reload`**:** Reload some document.
  action **is** `forward`**:** Navigate some window forward.
  action **is** `back`**:** Navigate some window back.
m **is a DNS response:** Send HTTP(S) request that was waiting for DNS resolution.
m **is a HTTP(S) response:** (Decrypt m,) handle headers (`Set-Cookie`, `Location`, etc.), find reference:
  reference to window: Create document in window.
  reference to document: Add response body to document's script input.
  reference to websocket: Finish WebSocket connection setup.
m **is a WebSocket message:** (Decrypt m,) find WebSocket connection, deliver data to respective document.
m **is a WebRTC message:** Find WebRTC connection, decrypt data, deliver data to respective document.

Figure 2.4. The basic structure of the web browser relation R^p with an extract of the most important processing steps, in the case that the browser is not already corrupted. Includes the WebRTC extensions presented in Section 2.12.

pendingRequests of its state.[12] The browser is now in a new state, say s'. It then creates and sends an event $\langle a', a, m' \rangle$ with the receiver address a' being a nondeterministically chosen IP address, the sender address being a, and the message m chosen nondeterministically from $d(s')$.

If the browser is not corrupted, the input message m is expected to be one of the special messages TRIGGER, FULLCORRUPT, LIMITEDCORRUPT, an HTTP(S) response, a DNS response, a WebSocket message, or a WebRTC message. Other types of messages are discarded without any change in the browser's state. The browser will then act as follows:

[12] The subterm is chosen arbitrarily, any other subterm would work as well.

$m = \texttt{TRIGGER}$. Upon receipt of this message, the browser nondeterministically chooses one of five *actions*:

Action is `script`: Some active (sub)window is chosen nondeterministically. Then the script of the active document of that window is triggered, as described in the next subsection.

Action is `urlbar`: A new HTTP(S) `GET` request (i.e., an HTTP(S) request with method `GET`) is created where the URL is some message derivable from the current state of the browser. However, nonces may not be used. This models the user typing in a URL herself, but we do not allow her to use secrets in the URL, e.g., passwords or session tokens. Otherwise, the attacker would trivially learn all of the user's secrets. A new window is created to show the response.

Action is `reload`: Some active (sub)window is chosen nondeterministically and the document in that window is reloaded.

Action is `forward` or `back`: Some active window or subwindow is chosen nondeterministically and navigated forward or back, respectively (cf. Section 2.10.3).

$m = \texttt{FULLCORRUPT}$. If the browser receives this message, it sets *isCorrupted* to `FULLCORRUPT`. From then on the browser is corrupted as described above. Unlike for limited corruption (see next paragraph), the state of the browser is not cleared when this command is received. This means that the attacker gains full access to the browser's internal state, including all secrets.

$m = \texttt{LIMITEDCORRUPT}$. If the browser receives this message, it first removes the user secrets, open windows and documents, all *session* cookies, all sessionStorage data, and all pending DNS/HTTP(S) requests from its current state. LocalStorage data and persistent cookies are not deleted. The browser then sets *isCorrupted* to `LIMITEDCORRUPT` (and hence, from then on is corrupted). As mentioned in Section 2.10.2, this models that the browser is closed by a user and that then the browser is used by another, potentially malicious user (an attacker), such as in an Internet café.

m is a DNS response. When a DNS response is received (and its nonce is contained in *pendingDNS*), this means that there is an HTTP(S) request waiting for this response (recall Section 2.10.5 above). Therefore, the corresponding HTTP(S) request will be dispatched.

m is a HTTP(S) response. The browser performs the following steps:

(I) The browser identifies the corresponding HTTP(S) request q (if any) and the window or document from which q originated using the data recorded in *pendingRequests*. If q was an encrypted HTTPS request, m is decrypted using the symmetric key recorded together with q in *pendingRequests*.

(II) If there is a `Set-Cookie` header in the response, its contents are evaluated: The cookie's name, value, and attributes (`httpOnly`, `secure`, `session`) are saved in the browser's list

of cookies. If a cookie with the same name already exists, the old values and attributes are overwritten, as specified in [RFC6265].

(III) If there is a `Strict-Transport-Security` header in the response, the domain of q is added to the term sts. As defined in [RFC6797], all future requests to this domain, if not already HTTPS requests, are automatically modified to use HTTPS. This includes requests made by the user (`urlbar` action above.)

(IV) If there is a `Location` header (with some URL u) in the response and the HTTP status code is 303 or 307, the browser re-sends the original request to the URL u (unless the original request was an XMLHttpRequest and u does not have the same origin as the initial request's URL, in which case the browser aborts). In line with [RFC7231], if the status code is 307, the browser retains the original request method and body in the redirected request. For 303, if the original request's method is not GET or HEAD, the browser will change the method to GET and discard the request body. The `Origin` header value is replaced by a null value ($\perp$) as defined in the W3C Cross-Origin Resource Sharing specification [Ann14].

(V) Otherwise, if no redirection is requested, the browser does the following:

a) If the request originated from a **window**, a new document is created from the response body. For this, the response body is expected to be a term of the form $\langle script, state\rangle$ where $script$ is the name of a script and $state$ is a term used as the script's initial state. The document is then added to the window the reference points to, it becomes the active document, and the successor of the currently active document. All previously existing successors are removed.

b) If the request originated from a **document** (and hence, was the result of an XML-HttpRequest), the body of the response is appended to the script input term of the document. When later the script of this document is activated, it can read and process the response.

c) If the message is a response to a **WebSocket connection establishment message**, the browser expects a status code of `101` and an `Upgrade` header just as in the request. If this is the case, the browser stores the nonce and symmetric encryption key (if any) used in the HTTP messages for future use in the WebSocket connection in its local state subterm $wsConnections$.

***m* is a WebSocket message.** To handle an incoming WebSocket message, the browser checks its list of open WebSocket connections for an entry matching the nonce (and possibly the encryption key) in the WebSocket message and adds the data contained in the WebSocket message to the script inputs of the script that initiated the connection.

***m* is a WebRTC message.** See Section 2.12.

2.10.10. Executing a Script

As described above, a browser, upon receiving a trigger message, can nondeterministically execute a script in any active document. The script is provided with a term of the form

$$\langle \mathit{tree}, \mathit{docnonce}, \mathit{scriptstate}, \mathit{scriptinputs}, \mathit{cookies}, \mathit{localStorage}, \mathit{sessionStorage}, \mathit{ids}, \mathit{secrets}\rangle \,.$$

The components of the term contain (in the order shown)

- document and window references of all active documents and subwindows,[13], and, only for same-origin documents, information about the documents' origins, scripts, script states and script inputs,
- the nonce of the document into which this script was loaded,
- the last state of the script,
- the input history (i.e., previous inputs from postMessages, XMLHttpRequests, WebSocket and WebRTC messages) of the script (as recorded in the document),
- cookies (names and values only) indexed with the document's domain, except for `httpOnly` cookies,
- localStorage data for the document's origin,
- sessionStorage data that is indexed with the document's origin and the reference of the document's top-level window,
- identities of the browser, and
- secrets indexed with the document's origin.

Now, according to the definition of scripts, the script outputs a term. The browser expects terms of the form

$$\langle \mathit{state}, \mathit{cookies}, \mathit{localStorage}, \mathit{sessionStorage}, \mathit{cmd}\rangle$$

(and otherwise ignores the output) where *state* is an arbitrary term describing the new state of the script, *cookies* is a sequence of name/value pairs, *localStorage* and *sessionStorage* are arbitrary terms, and *cmd* is a term which is interpreted as a command which is to be processed by the browser. The old state of the script recorded in the document is replaced by the new one (*state*), the local/session storage data recorded in the browser for the document's origin (and top-level window reference) is replaced by *localStorage*/*sessionStorage*, and the old cookie store of the document's origin is updated using *cookies* similar to the case of HTTP(S) responses with

[13] We over-approximate here: In real-world browsers, only a limited set of window handles are available to a script. Our approach is motivated by the fact that in some cases windows can be navigated by names (without a handle). However, as we will see, specific restrictions for navigating windows and accessing/changing their data apply.

`Set-Cookie` headers, except that now no httpOnly cookies can be set or replaced, as defined in [RFC6265]. For details, see Line 12 of Algorithm A.8 and Definition 46 in Appendix A.

Subsequently, *cmd* (if not empty) is interpreted by the browser as described next. For most commands, the browser expects additional parameters.

***cmd* = `HREF`:** (Parameters: URL, window reference, and a flag indicating whether the `Referer` header should be suppressed.) A new `GET` request to the given URL is initiated. If the window reference is `_BLANK`, the response to the request will be shown in a new *auxiliary* window. This new window will carry the reference to its opener, namely the reference to the window in which the script was running. Otherwise, if the window reference is not `_BLANK`, the corresponding window is navigated (upon receipt of the response and only if it is active) to the given URL.

Navigation of windows is subject to several restrictions. We closely follow the rules defined in [Ber+17], Subsection 5.1.4: A window A can navigate a window B if the active documents of both are same origin, or B is an ancestor window of A and B is a top-level window, or if there is an ancestor window of B whose active document has the same origin as the active document of A (including A itself). Additionally, A may navigate B if B is an auxiliary window and A is allowed to navigate the opener of B.

***cmd* = `IFRAME`:** (Parameters: URL, window reference.) Provided that the active document in the referenced window is same origin, create a new subwindow in that document and initiate an HTTP `GET` request to the given URL for that subwindow.

***cmd* = `FORM`:** (Parameters: URL, method, form data, window reference.) Initiate a new request using the specified method for the given URL. If the method `GET`, the form data is transferred as URL parameters, otherwise it is put in the request's body. The window reference determines, just like in the case of `HREF`, in what window the response is shown. Again the same restrictions for navigating other windows as in the case of `HREF` apply. For this request an `Origin` header is set if the method is `POST`. Its value is the origin of the document.

***cmd* = `SETSCRIPT`:** (Parameters: window reference, script name.) Replace the script of the active document in the referenced window by the script with the given name, provided that the active document in that window is same origin.

***cmd* = `SETSCRIPTSTATE`:** (Parameters: window reference, term.) Change the state of the script of the active document in the referenced window to a new term , provided that the active document in the window is same origin.

***cmd* = `XMLHTTPREQUEST`:** (Parameters: URL, method, data, XMLHttpRequest reference.) Initiate a request with the given method and data to the given URL, provided that the URL is same origin and the method is not is not `CONNECT`, `TRACE`, or `TRACK`.[14] The `Origin`

[14]These methods are forbidden to prevent certain attacks, in accordance with [Fetch].

header is set as in the case of FORM.

***cmd* = BACK or FORWARD:** (Parameter: window reference.) Replace the active document in the given window by its predecessor/successor in the window's history.[15] Again, the same restrictions for navigating windows as in the case of HREF apply.

***cmd* = CLOSE:** (Parameter: window reference.) Close the given window, i.e., remove it from the list of windows in which it is contained. The same restrictions for navigating windows as in the case of HREF apply.

***cmd* = POSTMESSAGE:** (Parameters: message, window reference, origin.) The message, the origin of the sending document, and a reference to its window are appended to the input history of the active document in the given window, unless that document's origin does not match the given origin (and the given origin is not $\perp$).

***cmd* = WS_OPEN:** (Parameters: URL, WebSocket reference.) Create a new WebSocket connection to the given URL and identified by the given WebSocket reference.

***cmd* = WS_SEND:** (Parameters: WebSocket reference, data.) Send data in a WebSocket message using an already established WebSocket connection (identified by the WebSocket reference).

***cmd* = RTC_*:** We present the commands for WebRTC connection establishment separately in Section 2.12.

The script execution ends after the interpretation of the command.

2.11. Generic HTTPS Servers

Our generic framework for HTTPS servers defines a basic set of algorithms that can handle incoming HTTPS requests, but also send its own HTTPS requests (and, to this end, handle DNS as well).

Except for the handling of incoming HTTPS requests, the model follows the relevant parts from the browser model: the framework prescribes that an HTTPS server must have a state containing *pendingDNS* and *pendingRequests* subterms, and these are used just as in browsers. Our framework currently does not handle cookies or strict transport security when dealing with outgoing HTTPS requests, but these and other features could be added easily.

Corruption is also included in the generic HTTPS server model. It follows the same idea as corruption in the browser model: A corrupted server collects all incoming messages and sends all messages derivable from its knowledge. Unlike browsers, the generic web server model only implements full corruption.

[15]Note that navigating a window using the back/forward buttons does not trigger a reload of the affected documents. While real world browser may chose to refresh a document in this case, we assume that the complete state of a previously viewed document is restored. A reload can be triggered nondeterministically at any point by the browser.

Concrete instantiations of servers using the framework must (at least) define algorithms for handling HTTPS responses, and HTTPS requests. They can also provide algorithms to handle trigger messages and other kinds of messages.

2.12. Extension: WebRTC

Although being relatively comprehensive, the WIM cannot capture all current and future web technologies. We therefore, in this section, demonstrate that the WIM can be extended easily, even for a complex technology such as WebRTC.

WebRTC can be used to establish direct connections between two browsers to transmit video, audio, or data streams.[16] In order to establish a WebRTC connection, web servers are needed to (1) deliver, to the browser(s) participating in a WebRTC session, JavaScript which then uses the WebRTC JavaScript API to establish a WebRTC connection, and (2) broker connections between the participating browsers as long as no direct WebRTC connection has been established.

Taking the scenario of a video connection between two users as an example, it is not necessary that the two users visit the same web site to establish the connection, i.e., WebRTC can connect browsers across origins (as long as the two origins are cooperating actively).

To authenticate users of a WebRTC session, the JavaScript delivered to the browser can either chose to only establish connections between authenticated and authorized users using a traditional cookie-based session management, or to use specific features of WebRTC that allow users to use a federated identity management to prove their identities (peer authentication), as we will see in more detail below.

2.12.1. WebRTC and the WebRTC Model

To explain WebRTC and our model of it, we first describe a WebRTC connection establishment procedure between two web browsers (Browser A and Browser B) using a common server (`example.com`). We here assume that `example.com` has authenticated the users of the respective browsers. In the next example, we will then introduce the authentication features of WebRTC.

The flow (Figure 2.5) shows the following steps:

- First, both browsers load some web page from the server of `example.com`.
- Second, the documents in both browsers call a WebRTC API function that creates and initializes a new WebRTC connection object [1].
- So far, both browsers performed the same steps. Now, one of the browsers (Browser A) creates a so-called *offer*, a description of the WebRTC connection properties [2]. It contains, for example, networking and media stream details, but also identity information (see next

[16] In practice and in our model, one or both of the endpoints might also be servers instead of browsers. This is used, for example, to stream a live video feed from a server to a browser or to connect the browser's user to a traditional telephone system.

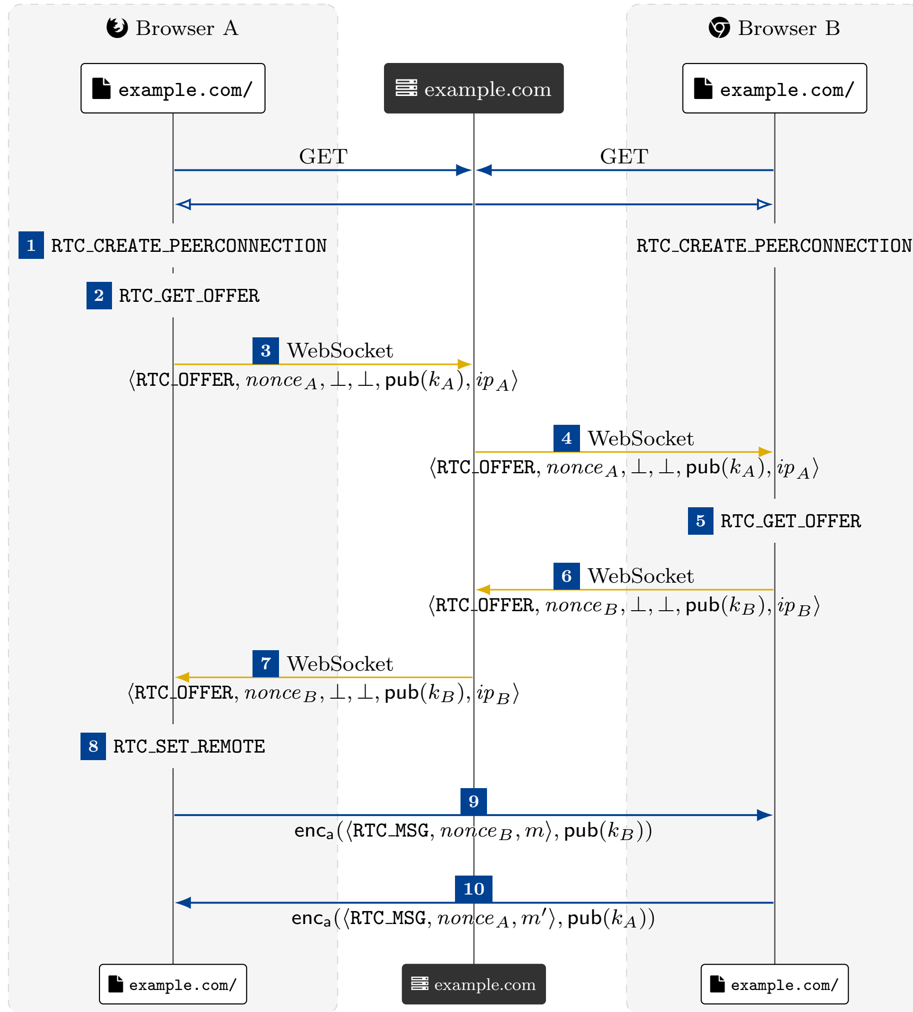


Figure 2.5. Simple example flow of WebRTC without peer authentication.

example), and cryptographic keys. In our example (and model), the offer contains a freshly chosen nonce that identifies the connection, the public key of the browser's (freshly chosen) key pair, and the browser's IP address.

– The offer is now sent, for example via WebSockets, to `example.com` and from there to Browser B [3], [4].

– Browser B now feeds the initial offer into the WebRTC API and creates an *answer* [5]. At the abstraction level of our model, answers and offers are of the same structure and contain the same data. We therefore, in our formal representation of WebRTC, call the answers "offers" as well.

– The answer is now transmitted back to Browser A [6], [7].

– The script in Browser A receives the answer and feeds it into the WebRTC API [8].

– The WebRTC connection is now configured in both browsers and the browsers can send messages back and forth [9], [10]. The messages contain the nonces identifying the respective connections in each browser and the payload (e.g., frames of a video stream) encrypted with the public keys exchanged earlier.

It is easy to see that in this example, browsers rely on example.com for authentication. WebRTC peer authentication can be used by browsers to authenticate themselves to each other.

The basic principle of peer authentication is that an identity provider (IdP) checks a user's identity (for example, using a combination of username and password) and then testifies for the identity of the user, for example by signing the user's public key, creating a so-called *identity assertion* (IA).[17]

To check the identity of a peer, a browser can ask the same IdP to verify the IA provided during the connection establishment.

Figure 2.6 shows a WebRTC flow where Browser A authenticates to Browser B. In more complex scenarios, authentication can be mutual, i.e., Browser B could also authenticate to Browser A. Since the steps are the same (with Browsers A and B interchanged), we omitted them from the figure and the following explanation.

To authenticate to Browser B, Browser A performs the following additional steps:

– When creating the peer connection object, the JavaScript in Browser A provides an IdP domain to the browser, say `idp.example`. The browser then triggers the WebRTC peer authentication: It opens a new window[18] [1] and loads a document from a well-known URI at `idp.example` [2]. This document is called a *proxy script*, since it acts as a proxy between the JavaScript from `example.com` and the IdP.

– Now, the JavaScript in the IdP window retrieves the information that is needed to create the IA from the calling window (e.g., the public key of the WebRTC connection) [3].

– The IdP now creates the IA. Typically, this involves one or more requests to the IdP's web server, which are out of the scope of the specification.

– The proxy script now calls an API function to transmit the IA back to the calling window for it to be used during the peer connection [4].

– The IA and the IdP domain become part of the offer that is created by the WebRTC API. Just as before, this offer is now, as part of the connection establishment, transmitted to Browser B.

In a similar way, Browser B now checks the IA of Browser A: It first opens a window with an IdP proxy script [5], which then retrieves the IA and the public key of Browser A [6]. If the IA

[17] Since WebRTC only defines an interface for peer authentication, details of the authentication are not specified, in particular the contents of the IA.

[18] In real-world browsers, a new "browsing context" is opened—roughly equivalent to a window without a user interface.

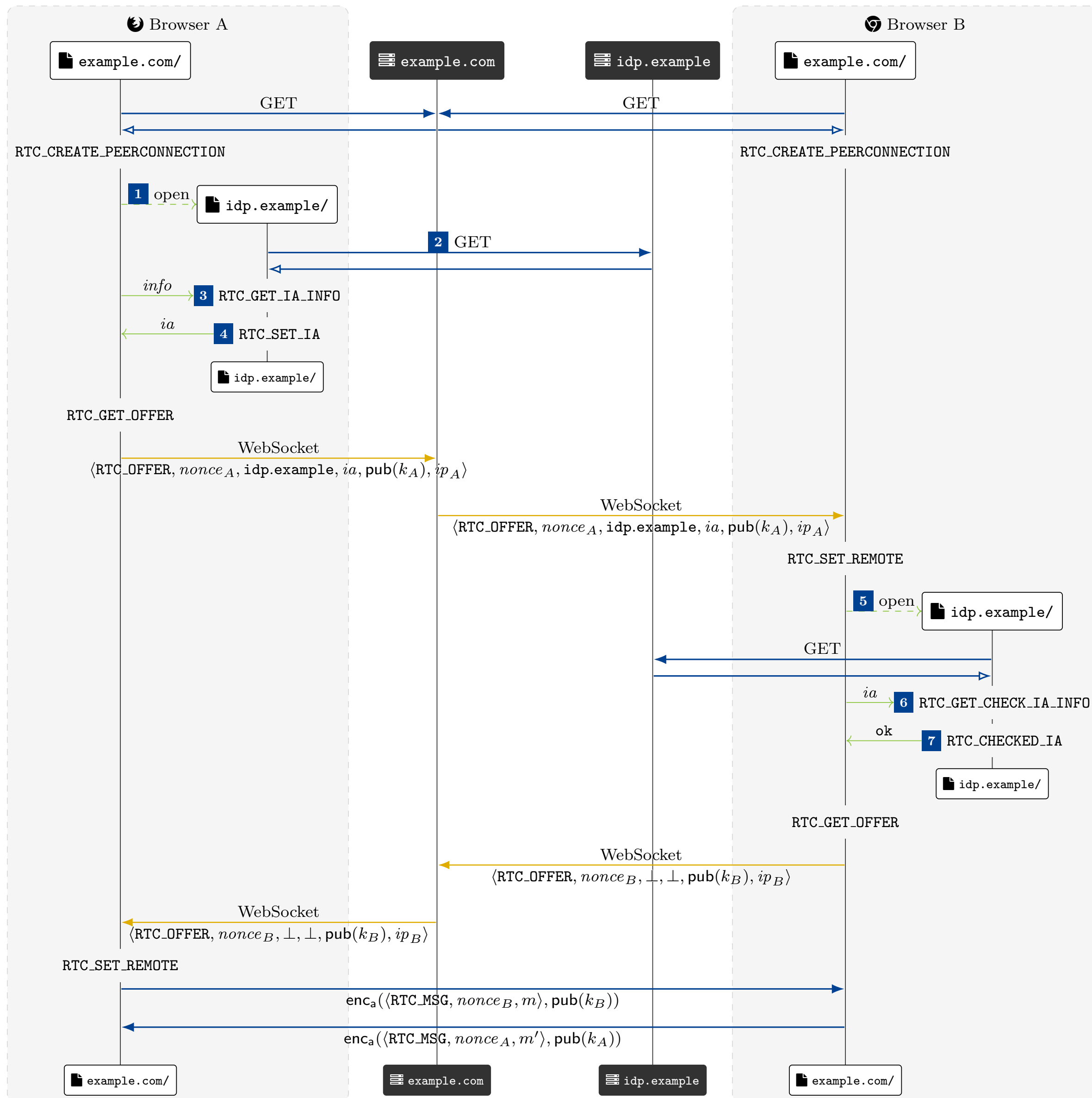


Figure 2.6. WebRTC example flow where Browser A authenticates itself to Browser B. For clarity of presentation, Browser B does not authenticate itself to browser A in this example.

has been checked successfully, the calling window is notified 7 and the WebRTC flow continues as in the previous example.

2.12.2. New Script Commands for WebRTC

To support WebRTC in browsers, we introduce new commands available to scripts additionally to those listed in Section 2.10.10. Some of the commands are only available to scripts in the IdP proxy window. If such a command is issued by a script not loaded in an IdP proxy window, the browser ignores the command. We introduce the following commands:

***cmd* = RTC_CREATE_PEERCONNECTION:** (Parameters: Domain of IdP, private key.) Create a new WebRTC connection object—more precisely, an entry in the browser state's subterm *rtcConnections* identified by a *WebRTC nonce* (chosen by the browser and returned to the script via the script inputs). In this entry, the browser tracks the properties of the WebRTC connection. If a domain of an IdP is given, the browser creates a new top-level window and, in this window, loads the IdP proxy script.

***cmd* = RTC_GET_OFFER:** (Parameters: WebRTC reference.) Create a new WebRTC offer/answer document containing the WebRTC reference, the identity assertion (if any), the public key chosen by the browser for this WebRTC connection, and the browser's IP address.

***cmd* = RTC_SET_REMOTE:** (Parameters: WebRTC reference, offer.) Put the information sent by the remote peer in an offer/answer document into the WebRTC connection object identified by the given WebRTC reference. Afterwards, run the code to check the remote's identity assertion, i.e., open a new window containing the IdP's proxy page, which can then use the commands `RTC_GET_CHECK_IA_INFO` and `RTC_GET_CHECK_IA_INFO` (see below).

***cmd* = RTC_GET_IA_INFO:** (Parameters: none.) This function can only be used from an IdP proxy script. It returns (by appending to the script's inputs), the WebRTC nonce identifying the connection for which the window was opened and the public key used by the browser in that connection. The proxy script is supposed to check the user's identity (for instance, using cookies and an XMLHttpRequest to the IdP server), and then use the next function to return a signed identity assertion to the browser.

***cmd* = RTC_SET_IA:** (Parameters: identity assertion.) This function can only be used from an IdP proxy script. Signals the browser to store the identity assertion created by the IdP in the WebRTC connection information, so that the browser can use this information in an offer/answer document.

***cmd* = RTC_GET_CHECK_IA_INFO:** (Parameters: none.) This function can only be used from a window opened for an IdP proxy script. It returns the information the IdP needs to check the validity of the identity assertion presented by the remote browser—in particular, the identity assertion itself, the nonce identifying the connection, and the remote browser's public key.

***cmd* = RTC_CHECKED_IA:** (Parameters: identity assertion.) This function can only be used from an IdP proxy script. Using this function, the proxy script signals to the browser that it has successfully checked the remote's identity assertion.

***cmd* = RTC_SEND:** (Parameters: WebRTC nonce, data.) Using the WebRTC connection identified by the WebRTC reference, send a WebRTC message over the network (with the data encrypted using the peer's public key).

3. Analysis of OAuth 2.0

In this chapter, we present our analysis of OAuth 2.0. We start with an introduction of basic concepts before we give an in-depth description of the four *modes*, also called *grants* or *grant types*, defined in [RFC6749]. Subsequently, we elaborate on the attacks uncovered during our analysis and fixes against these attacks. We then summarize the most important previously known attacks on OAuth 2.0 that have to be mitigated in order to prove the security of OAuth. In the last section, we outline our formal analysis and the proof. All technical details of the analysis and proof can be found in Appendix B.

3.1. OAuth 2.0 Basic Concepts

OAuth 2.0 was first intended only for *authorization*, i.e., users authorize client web sites to access their *protected resources* at resource servers by using a token issued by an authorization server. Accessing the protected resources can mean reading information from these resources (e.g., the user's private profile information) or modifying these resources on the user's behalf (e.g., uploading content to the user's profile at the resource server). Usually, authorization server and resource server are controlled by the same entity and we then use the term *OAuth Provider* (OAP) for this entity. For example, a user can use OAuth to authorize a photo printing service to download her (private) pictures from Facebook. In this case, the printing service is the client and Facebook is the OAuth Provider (since it operates the authorization and resource servers).

Roughly speaking, in the most common modes/grants, OAuth works as follows: If a user wants to authorize a client to access some of her data at an RS, the client redirects the user's browser to the AS, where the user authenticates and agrees to grant the client access to some of her user data at the RS. Then, the user is redirected back to the client with an *access token* or *authorization code* issued by the AS. If an authorization code was issued, the client can exchange it for an access token at the AS. The client can then use the access token as a credential at the RS to access the user's data.

OAuth is also commonly used for *authentication*, although it was not designed with authentication in mind. In this case, the client acts as an RP and the OAP (AS and RS) act as the IdP. A user can, for example, use her Facebook account, with Facebook being the IdP, to log in at some online community (the RP). Typically, in order to log in, the user authorizes the RP to access a unique user identifier at the IdP. The RP then retrieves this identifier and considers this user to be logged in.

In the following, we introduce the most important concepts used in OAuth 2.0.

3.1.1. Token Types

OAuth 2.0 defines three important types of tokens: *access tokens*, *authorization codes*, and *refresh tokens*.

The *access token* is issued by an AS and is what ultimately gives access to a resource at an RS. Access tokens are involved in every flow of OAuth. An access token is a so-called *bearer token*, which [RFC6750] defines as a "security token with the property that any party in possession of the token (a 'bearer') can use the token in any way that any other party in possession of it can. Using a bearer token does not require a bearer to prove possession of cryptographic key material (proof-of-possession)."

[RFC6749] does not mandate any specific structure or contents of the access token (it is "opaque to the client"), but there must be a way for the RS to check the validity of the access token and to learn the token's properties, for example, to which data the token gives access to. There are several ways to accomplish this: the token can be a cryptographically signed document which the RS can verify without contacting the AS (see, e.g., [RFC7519; RFC7523]), there can be a (proprietary) back-channel or shared state between the RS and the AS, or the RS can use *token introspection*, i.e., query information about the access token from the AS.[1] In our model, access tokens are nonces and the whole OAP is modeled as a single server which can immediately check the validity of a token from its state.

An *authorization code* is a temporary (single-use) token that can be issued to the client in the authorization code grant (see below). The client can exchange the authorization code for an access token at the token endpoint of the AS (see below). According to [RFC6749], authorization codes improve the security of OAuth 2.0: First, an AS can authenticate the client presenting the authorization code at the token endpoint (for example, using a client secret, see below). Second, with authorization codes, the access token can be transferred from the AS to the client directly without passing through the user's browser. This reduces chances for an inadvertent leak of the access token, and the access token is not exposed to the user.

Refresh tokens are credentials issued by an AS to a client and can be used to obtain further access tokens when the current one expires or becomes invalid, or to obtain other access tokens with different properties (see also Section 3.1.6 below). Unlike access tokens, refresh tokens are never sent to a resource server. The use of refresh tokens is optional. Since many real-world deployments do not use refresh tokens, and access tokens neither expire nor have varying properties in our model, we do not cover refresh tokens.

3.1.2. Endpoints

In OAuth, clients, RS, and AS need to know certain URIs of each other, called *endpoints*. An AS provides an *authorization endpoint* at which the user can authenticate to the AS and authorize a client to access her user data. The AS also provides a *token endpoint* at which the client

[1] A standard way for token introspection is defined in [RFC7662].

can request access tokens. A client provides one or more *redirection endpoints* to which the user's browser gets redirected by the by the AS after the user authenticated herself to the AS. The URIs of the endpoints are not fixed by the standard, but are communicated when a client registers itself at the OAP, as described below.

For all endpoints, [RFC6749] and the security recommendations in [RFC6819] recommend the use of HTTPS. We follow this recommendation in our analysis of OAuth.

3.1.3. Client Registration at the OAP

Before a client can interact with an OAP, the client needs to be registered at the OAP. The details of the registration process are out of the scope of the core OAuth protocol.[2] In practice, this process is usually a manual task. During the registration process, the OAP assigns credentials to the client, consisting of a client id and a client secret. The client may later use the client secret to authenticate to the AS. If the client cannot keep the OAuth client secret confidential, e.g., if the client is an in-browser app or a native application, the secret can be omitted. In [RFC6749], clients with client secrets are called *confidential clients*, while those without are called *public clients*. The OAuth client id always is public information. It is, for example, revealed to users in redirects issued by the client.

As mentioned above, a client registers one or more redirection endpoints at the OAP. If more than one redirection URI is registered, the client must specify which redirection URI is to be used in each run of the OAuth protocol. For simplicity of presentation, we assume that a client always specifies its choice, although this can be omitted if there exists only one (fixed) redirect URI. Depending on the implementation of an AS, a client may also register a pattern as a redirect URI and then specify the exact redirect URI during the OAuth run. In this case, the AS checks if the specified redirect URI matches this pattern.

With the registration, the operator of the client also needs to configure the endpoints of the OAP into its systems. Typically, the same endpoints are used for all clients.[3]

Our analysis presented in Section 4.5 covers all the options mentioned here: public and confidential clients, explicitly specified redirection URIs, and URI patterns.

3.1.4. Login Sessions and State

During an OAuth run, a user's browser is redirected from the client to the AS and back to the client. To keep track of the user's actions and to prevent Cross-Site Request Forgery attacks, the

[2] As introduced in Section 1.1.3, OpenID Connect provides features for the discovery of IdPs and dynamic registration of RPs. With [RFC7591], these mechanisms have been backported into OAuth, but they are not part of the core specification of OAuth 2.0.

[3] An attack on OAuth/OpenID Connect can be to social engineer the operator of the client to believe that a URL controlled by an adversary is the token endpoint or a resource server endpoint. In this case, the client would send access tokens or authorization codes to the attacker, which can use these to access protected resources at the original OAP. This attack has been described (briefly) in Section 8.3.2 of the Read and Write API Security Profile of the OpenID Financial API specification [SSN18]. In our analysis, we assume that the endpoints are configured properly, i.e., do not point to attacker-controlled servers.

client typically establishes a session with the browser before the first redirect.[4] To prevent CSRF attacks, the OAuth standard recommends that a client selects the so-called *state* parameter and binds this value to the session, e.g., by choosing a fresh nonce as the state value and storing the nonce in the session state. When the user later gets redirected back to the client, the state value must match the one stored in the session. To be effective against CSRF attacks, the state value must never leak to an attacker. Omitting or incorrectly using this parameter can lead to attacks described in [Ban+14; LM14; RFC6749; RFC6819; SB12]. In our analysis, we follow the recommendation of using the state parameter.

3.1.5. Tracking User Intention

Often, clients support more than one OAP for user authentication/authorization. For example, many web sites allow users to log in using Google or Facebook as the OAP.

A client needs to remember which OAP a user wanted to use when the user comes back from the AS. There are two different approaches to accomplish this in practice: First, the client can distinguish different OAPs by using separate redirection URIs for each OAP. We call this method *naïve user intention tracking*. Second, the client can store the user's choice in the session and use this information later. We call this *explicit user intention tracking*. In Section 3.3 we discuss the security implications of the choice of the user intention tracking method.

3.1.6. Further Recommendations and Options

The standard and the recommendations do not specify all implementation details. For example, the precise user interaction with a client, formatting details of messages, and the authentication of the user to an AS (e.g., username and password or other mechanisms) are not covered. In our security analysis of OAuth we follow all OAuth security recommendations as well as common best practices for state-of-the-art web applications in order to avoid all known attacks.

OAuth allows RPs to specify which *scope* of the user's data they are requesting access to at an RS. The scopes themselves are not defined in the standard and are considered an implementation detail of OAPs. Therefore, in our description and analysis of OAuth, we omit the scope parameter and assume that the user always grants full access to her data at the RS.

3.2. OAuth 2.0 Grant Types

With the basic concepts introduced, we now describe the four basic grant types (or modes) of OAuth defined in [RFC6749]. A client, when starting an OAuth flow, selects which grant it wants to use: It can redirect the user's browser to the authorization endpoint and use a parameter called *response_type* to select either the *authorization code grant* or the *implicit grant*. If it

[4] The OAuth standard [RFC6749] as well as the OAuth security recommendations [RFC6819] do not specify the session mechanism for clients. In our analysis we assume the usual session mechanism with session cookies following common best practices. For more details, see Section 3.4.5.

instead contacts the token endpoint first, it can select the *resource owner password credentials grant* or the *client credentials grant* using the *grant_type* parameter.

An OAP does not need to support all grant types. In practice, most OAPs only support one or two grant types.

3.2.1. Authorization Code Grant

In the authorization code grant, when the user tries to authorize a client to access her data at an RS (or tries to log in at a client, i.e., an RP), the client first redirects the user's browser to the AS. The user then authenticates to the AS, e.g., by providing her username and password, and finally is redirected back to the client along with an authorization code generated by the AS. The client now contacts the token endpoint of the AS with this authorization code (along with the client id and, if used, the client secret) and receives an access token, which the client can use as a credential to access the user's protected resources at the AS or to retrieve information about the user from the AS.

Step-by-Step Protocol Flow

The protocol flow of OAuth 2.0 in the authorization code grant looks as follows (see Figure 3.1):

- First, the user starts the OAuth flow, e.g., by clicking on a button to select an OAP. The user's choice is sent to the client, e.g., in an HTTP POST request [1].
- The client selects one of its redirection endpoint URIs, *redirect_uri*, and a nonce *state*. The *redirect_uri* will be used later in Step [7]. The client then redirects the browser to the authorization endpoint URI at the AS with the URL parameters *client_id*, *redirect_uri*, *state*, and *response_type* (value `code`) [2], [3].
- The AS then prompts the user to provide her username and password [4]. The user's browser sends this information to the AS [5].
- If the credentials are correct, the AS creates an authorization code, *code*, and redirects the user's browser to client's redirection endpoint URI *redirect_uri* with *code* and *state* as URI parameters [6], [7].
- If *state* matches the value stored in the user's session, the client contacts the token endpoint of the AS to exchange *code* for an access token. The client provides *code*, *client_id*, *client_secret*, and *redirect_uri* in this request [8].
- The AS verifies this information, i.e., it checks that *code* was issued for the client identified by *client_id*, that *client_secret* (if used) is the secret for *client_id*, that *redirect_uri* is the same as in Step [2], and that *code* has not been redeemed before. If these checks are successful, the AS issues an access token *access_token* [9].

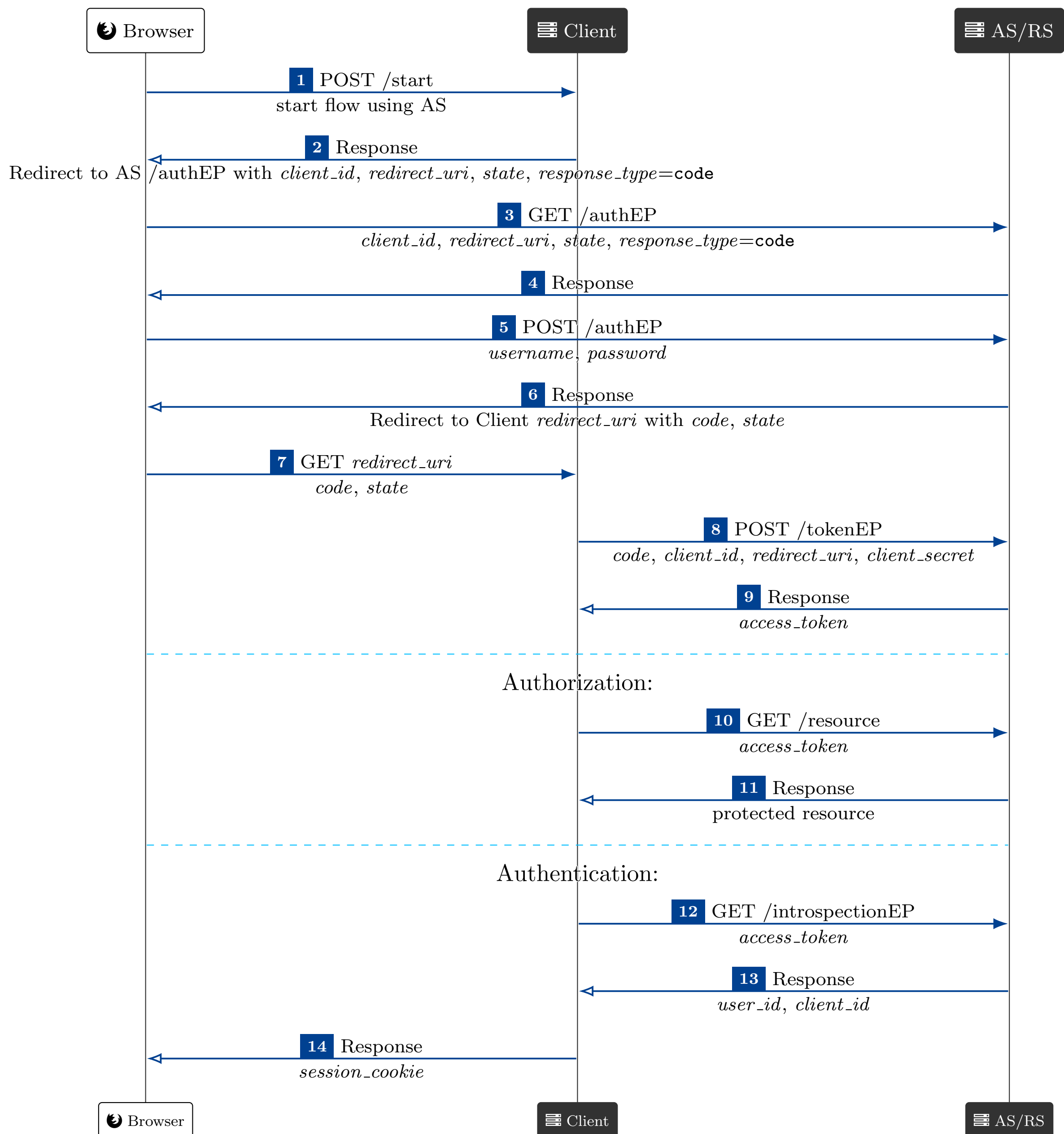


Figure 3.1. OAuth 2.0 authorization code grant. Data depicted below the arrows is either transferred in URI parameters, HTTP headers, or POST bodies.

- When OAuth is used for *authorization*, the client uses the access token to view or manipulate the protected resource at the RS (illustrated in Steps 10 and 11). For *authentication*, the client (in this case acting as an RP) fetches a user id (which uniquely identifies the user at the OAP) using the access token in Steps 12 and 13. The client then issues a session cookie to the user's browser as shown in Step 14.[5]

[5] A client might as well opt to re-use the session that was used for the login procedure. It is more secure, however, to issue a new session after login, see Section 3.4.5.

3.2.2. Implicit Grant

This grant is a simplified version of the authorization code grant: instead of providing an authorization code to a client, an AS directly delivers an access token to the client (via the user's browser).

Step-by-Step Protocol Flow

The implicit grant works as follows (see Figure 3.2):

- Steps 1, 2, 3, 4, and 5 are the same as in the authorization code grant, except that now the value of *response_type* is `token`.

- If the user's credentials are correct, the AS creates an access token, *access_token*, and redirects the user's browser to the client's redirection endpoint *redirect_uri*, where the AS appends *access_token* and *state* to the fragment of the redirection URI 6, 7.[6]

[6] The fragment is the last part of a URI, started by the '#' symbol. When the browser opens a URI, the information in the fragment is not transferred to the server.

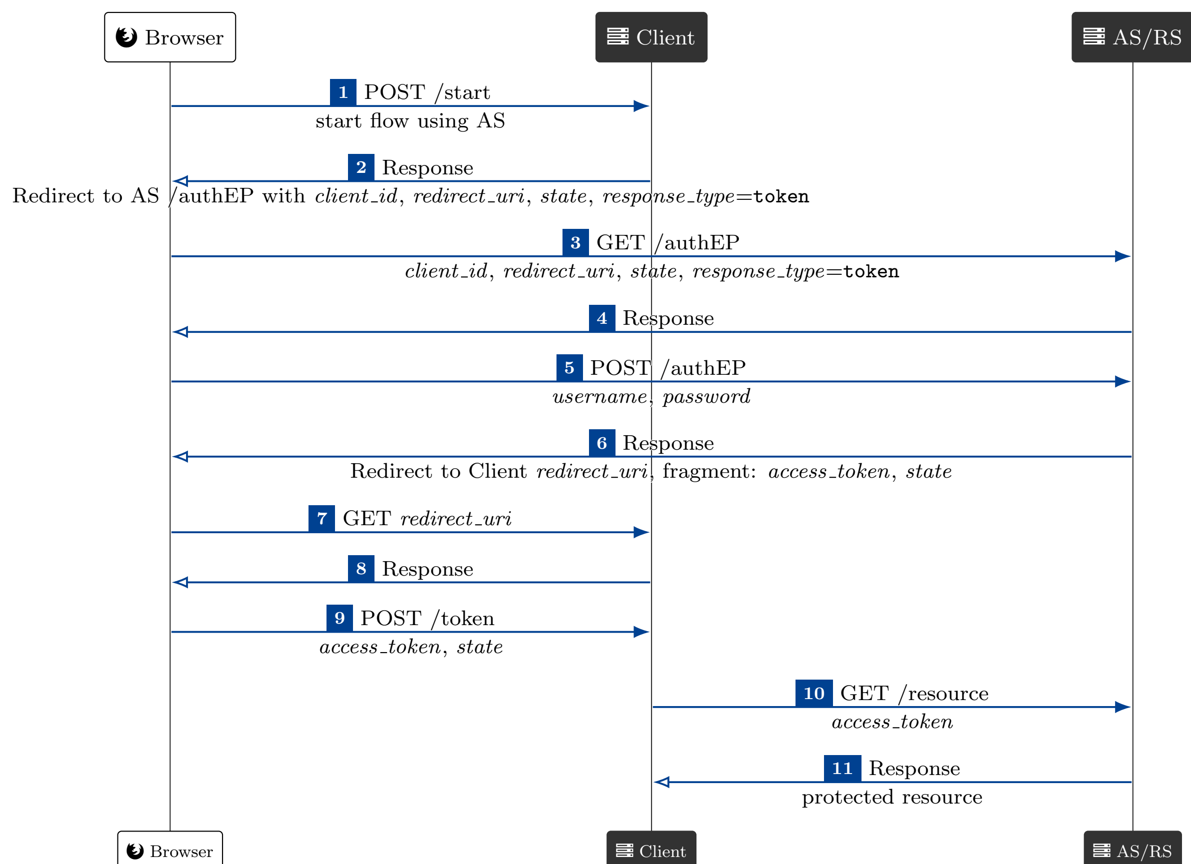


Figure 3.2. OAuth 2.0 implicit grant.

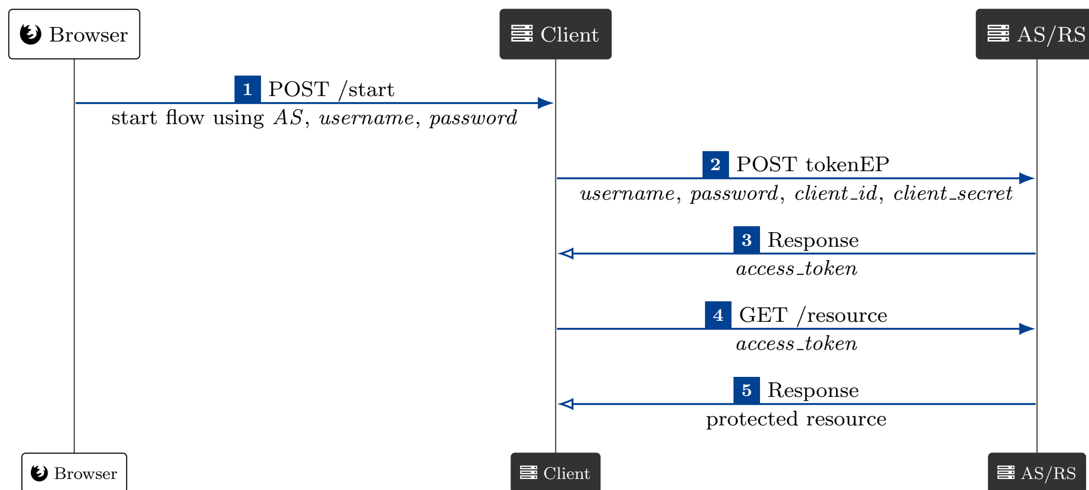


Figure 3.3. OAuth 2.0 resource owner password credentials grant.

– The browser follows the redirection [7]. Since the *access_token* and *state* are contained in the URI fragment, they are not transmitted to the client's server.

– To retrieve these values, the client delivers a document containing JavaScript code [8]. It retrieves *access_token* and *state* from the fragment and sends these to the client [9].

– The client then checks if *state* is the same as in the session. Just as in the authorization code grant, the client can now use *access_token* for authorization (illustrated in Steps [10] and [11]); authentication is analogous to Steps [12], [13], and [14] of Figure 3.1.[7]

Recall that in the implicit grant, an AS cannot verify the identity of the receiver of the access token, as a client does not authenticate itself to the AS (using *client_secret*). Hence, this grant type is more suitable for clients that do not have access to a secure, long-lived storage for client secrets (public clients) such as in-browser applications.

3.2.3. Resource Owner Password Credentials Grant

In this grant, the user gives her credentials for an AS directly to a client. The client can then authenticate to the AS on the user's behalf and retrieve an access token. The resource owner password credentials grant is intended for highly-trusted clients, such as the operating system of the user's device or highly-privileged applications, or if the previous two grants are not possible to perform (e.g., for applications without a web browser). In the following, we assume that the authorization/login process is started by the user using a web browser.

[7] For authentication, it is important to note that the response from the AS (Step [13]) includes the client's OAuth client id, which is checked by the client. This check prevents reuse of access tokens across clients in the OAuth implicit grant as explained in [Wan+13].

Step-by-Step Protocol Flow

The resource owner password credentials grant is depicted in Figure 3.3: In the first step, the user provides her username and password for the AS to the client [1]. Now, the client sends the username, the password, its *client_id* and *client_secret*[8] to the AS [2]. The AS then issues an access token, *access_token*, to the client [3]. Just as in the authorization code grant, the client can now use the access token for authorization (illustrated in Steps [4] and [5]) and authentication (as in Steps [12], [13], and [14] of Figure 3.1).

3.2.4. Client Credentials Grant

In contrast to the grant types shown above, this grant works without the user's interaction. Instead, it is started by a client in order to fetch an access token to access the client's own resources at an RS or to access resources at an RS the client is authorized to by other means. For example, Facebook allows clients to use the client credentials grant to obtain an access token to access reports of their advertisements' performance.

Step-by-Step Protocol Flow

The step-by-step description of the client credentials grant is as follows (see Figure 3.4): First, the client contacts the AS with its *client_id* and *client_secret* [1]. The AS now issues an *access_token* [2]. Just as in the authorization code grant, the client can now use *access_token* for authorization (illustrated in Steps [3] and [4]). In contrast to the other grants presented above, the access token is not bound to a specific user account, but only to the client. Therefore, the client cannot use this grant type for user authentication.

[8] In this grant type, if a client does not have an OAuth client secret for an AS, the *client_secret* and *client_id* parameters are *both* omitted in this request. This option is also covered by our analysis.

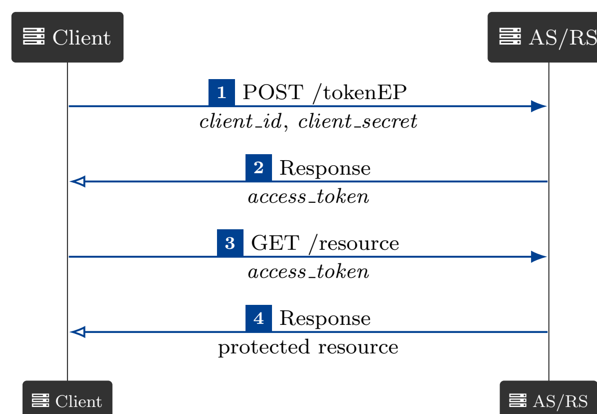


Figure 3.4. OAuth 2.0 client credentials grant.

3.3. New Attacks on OAuth

While trying to prove the security of OAuth based on the model presented in Chapter 2 and our model of OAuth (see Section 3.5), we found five previously unknown attacks, namely the *307 Redirect Attack*, the *AS Mix-Up Attack*, the *State Leak Attack*, the *Naïve Client Session Integrity Attack*, and the *Across-AS State Reuse Attack*. Table 3.1 shows the attacks and the security properties broken by them in different OAuth grants. As can be seen from the figure, all attacks target the authorization code grant and the implicit grant.

In this section, we provide, for each attack, the assumptions we have to make for the attack to work and their rationale, a detailed description of the attack, and easily implementable fixes. Our formal analysis of OAuth presented in the next section shows that these fixes are indeed sufficient to make OAuth secure against our strong attacker models. The attacks also apply to OpenID Connect (see Section 4.4). We have verified our attacks on actual implementations of OAuth and OpenID Connect and reported the attacks to the respective working groups who confirmed the attacks (see Section 5.1).

	attack on OAuth grant	
	authorization code	implicit
307 Redirect Attack	az + an	az + an
AS Mix-Up Attack	az + an	az + an
State Leak Attack	si	si
Naïve Client Session Integrity Attack	si	si
Across-AS State Reuse Attack	si	si

az: breaks authorization. **an:** breaks authentication. **si:** breaks session integrity.

Table 3.1. Overview of attacks on OAuth 2.0.

3.3.1. 307 Redirect Attack

In this attack, which breaks our authorization and authentication properties (see Section 3.5.3), the attacker (running a client) learns the user's credentials when the user logs in at an AS that uses the wrong HTTP redirection status code. While the attack itself is based on a simple error, to the best of our knowledge, this is the first description of an attack of this kind.

Assumptions

The main assumptions that we need to make for this attack to work are that

(1) the AS that is used for the login chooses the 307 HTTP status code when redirecting the user's browser back to the client (Step 6 in Figure 3.1), and

(2) the AS redirects the user immediately after the user entered her credentials (i.e., in the response to the HTTP POST request that contains the form data sent by the user's browser).

Assumption (1): This assumption is reasonable because neither [RFC6749] nor [RFC6819] specify the exact method of how to redirect. The OAuth standard rather explicitly permits any HTTP redirect:

> "While the examples in this specification show the use of the HTTP 302 status code, any other method available via the user-agent to accomplish this redirection is allowed and is considered to be an implementation detail."

Assumption (2): This assumption is reasonable as many examples for redirects immediately after entering the user credentials can be found in practice, for example at `github.com` (where, however, Assumption (1) is not satisfied.)

Attack

When a malicious client starts the authorization code or implicit grant of OAuth for a user, the user's browser is redirected to the AS and the user is prompted to enter her credentials. The AS then receives these credentials from the browser in a POST request. It checks the credentials and redirects the browser to the client's redirection endpoint in response to the POST request. Since the 307 status code is used for this redirection, the browser will send a POST request to the client that contains all form data from the previous request, including the user credentials. Since the client is run by the attacker, the attacker can use these credentials to impersonate the user.

Fix

Contrary to the current wording in the OAuth standard, the exact method of the redirect is not an implementation detail but essential for the security of OAuth. In the HTTP standard [RFC7231], only the 303 redirect is defined unambiguously to drop the body of an HTTP POST request instead of repeating it in the following request. Therefore, the OAuth standard should require 303 redirects for the steps mentioned above in order to fix this problem.

3.3.2. AS Mix-Up Attack

In this attack, which breaks our authorization and authentication properties (see Section 3.5.3), the attacker confuses a client about which AS the user chose at the beginning of the login/authorization process in order to acquire an authentication code or access token which can be used to impersonate the user or access user data.

This attack applies to the authorization code grant and the implicit grant of OAuth when explicit user intention tracking is used by the client. To launch the attack, the attacker manipulates the first request of the user such that the client thinks that the user wants to use an identity managed by an AS of the attacker (A-AS) while the user instead wishes to use her identity managed by an honest AS (H-AS). As a result, the client sends the authorization code

or the access token issued by H-AS to the attacker. The attacker then can use this information to login at the client under the user's identity or access the user's protected resources at the H-RS associated with H-AS. There is also a variant of the attack that does not require the attacker to manipulate any HTTP messages (and thus works with a web attacker instead of a network attacker), see below.

Assumptions

For the AS Mix-Up Attack to work, we need three assumptions that we further discuss below:

(1) the presence of a network attacker who can manipulate the request in which the user sends her identity to the client as well as the corresponding response to this request (see Steps 1 and 2 in Figure 3.1),

(2) a client which allows users to log in with identities from an honest OAP (H-AS/H-RS) and an attacker-controlled OAP (A-AS/A-RS), and

(3) a client that uses explicit user intention tracking and issues the same redirection URI to all ASs. (Alternatively, the attack works as well if the client issues different redirection URIs to different ASs, but internally treats them as the same URI.)

We emphasize that we do not assume that the user sends any secret (such as a password) over an unencrypted channel. The variant of this attack described below has slightly different assumptions.

Assumption (1): It would be unrealistic to assume that a network attacker can never manipulate Steps 1 and 2 in Figure 3.1.

First, these messages are sent between the user and the client, i.e., the attacker does not need to intercept server-to-server communication. He could, e.g., use ARP spoofing in a wifi network to mount the attack.

Second, the need for HTTPS for these steps is not obvious to users or developers, and the use of HTTPS is not suggested by the OAuth security recommendations, since the user only selects an AS at this point; credentials are not transferred.

Third, even if a client intends to use HTTPS also for the first request (as in our model), it has to protect itself against TLS stripping by adding the client domain to the browser's preloaded Strict Transport Security list [STSPre]. Other mitigations, such as the `Strict-Transport-Security` header, can be circumvented (see [Sel14]), and do not work on the very first connection between the user's browser and client. For example, when a user enters the address of a client into her browser, browsers by default try unencrypted connections. It is therefore unrealistic to assume that all clients are always protected against TLS stripping.

Our formal analysis presented in Section 4.5 shows that OAuth can be operated securely even if no HTTPS is used for the initial request (given that our fix, presented below, is applied).

Assumption (2): Clients may use different AS, some of which might be malicious, and hence, OAuth should provide security in this case. Using a technique called dynamic client registration, OAuth clients can even allow the ad-hoc use of any AS, including malicious ones [RFC7591]. This is particularly relevant in OpenID Connect, where this technique was first implemented.

Assumption (3): Typically, clients that use explicit user intention tracking do not register different redirection URIs for different AS, since it is not needed. In particular, for clients that allow for dynamic registration, using the same redirection endpoint for all AS is an obvious implementation choice. This is for example the case in the OAuth/OpenID Connect implementations *mod_auth_openidc* and *pyoidc*.

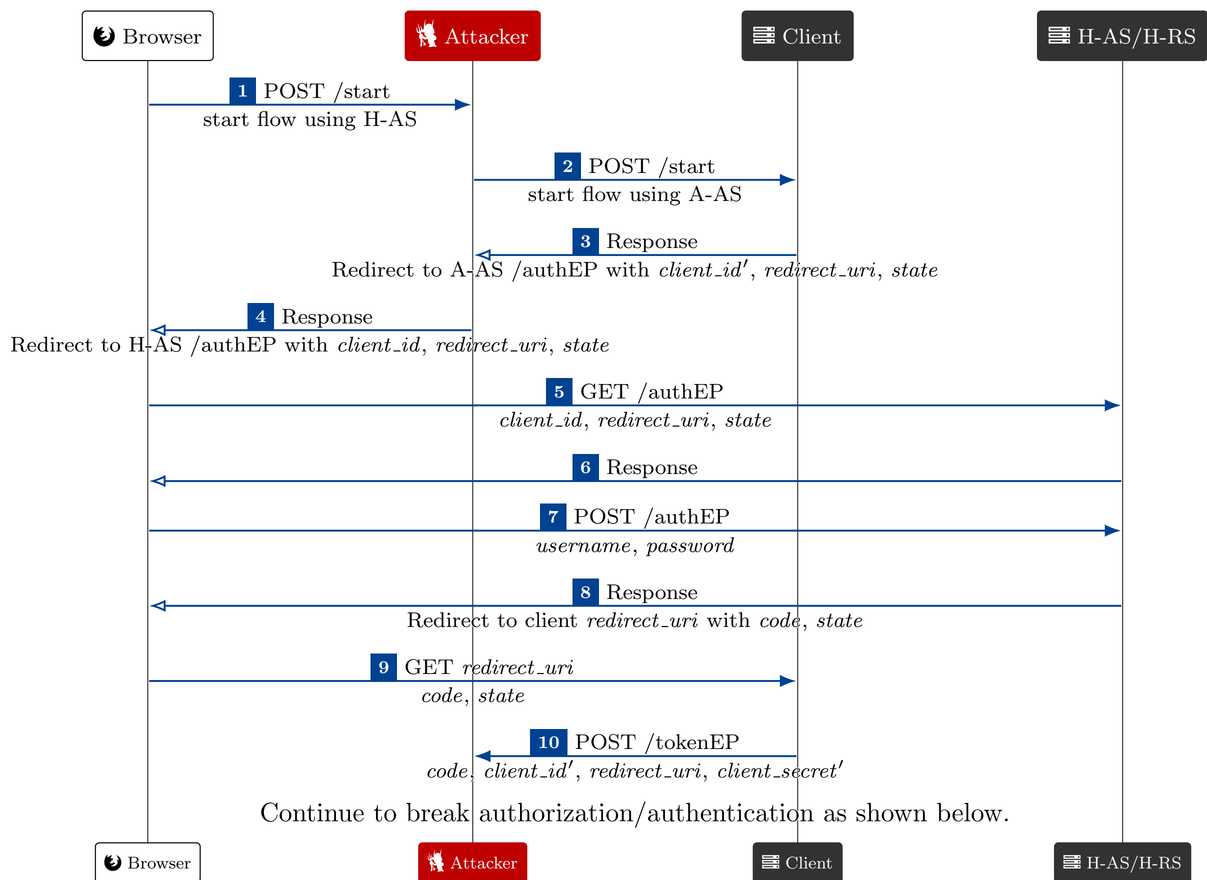


Figure 3.5. Start of the AS Mix-Up Attack on OAuth 2.0 authorization code grant.

Attack on Authorization Code Grant

We now describe the AS Mix-Up Attack on the OAuth authorization code grant. As mentioned, a very similar attack also applies to the implicit grant.

The AS Mix-Up Attack for the authorization code grant is depicted in Figure 3.5:

- Just as in a regular flow, the attack starts when the user selects that she wants to log in using H-AS 1.

– Now, the attacker intercepts the request intended for the client and modifies the content of this request by replacing H-AS by A-AS 2.

– The response of the client (containing a redirect to A-AS) is then again intercepted and modified by the attacker such that it redirects the user to H-AS 3, 4. The attacker also replaces the OAuth client id of the client at A-AS with the client id of the client at H-AS (which is public information). We assume that from this point on, in accordance with the OAuth security recommendations, the communication between the user's browser and H-AS and the client is encrypted by using HTTPS, and thus, cannot be inspected or altered by the attacker.

– The user's browser follows the redirection to H-AS, the user authenticates to H-AS and is redirected back to the client 5–8.

– The client believes, due to Step 2 of the attack, that the *code* contained in this redirect was issued by A-AS, rather than H-AS. The client therefore now tries to redeem this nonce for an access token at A-AS 10, rather than H-AS.

This leaks *code* to the attacker, which can now be used to break authentication or authorization properties.

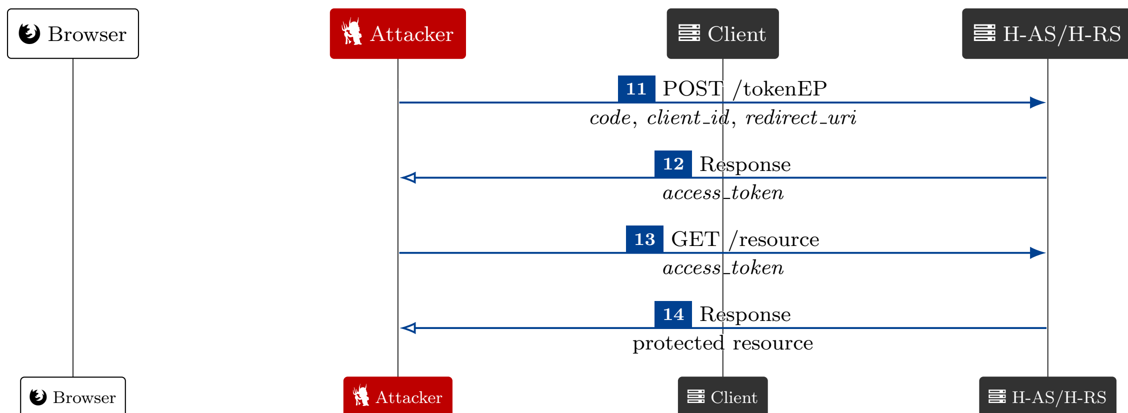


Figure 3.6. AS Mix-Up Attack on OAuth 2.0: Breaking authorization without code injection.

Breaking Authorization without Code Injection: If the client is a public client (i.e., does not have a client secret for H-AS), the attacker can now redeem *code* for an access token at the token endpoint of H-AS (Steps 11 and 12 in Figure 3.6). This access token allows the attacker to access protected resources of the user at H-AS (Steps 13 and 14). This breaks the authorization property (see Section 3.5.3).

Breaking Authorization with Code Injection: In the case that the client has to provide a client secret (confidential client), breaking authorization as presented before would not work (in the authorization code grant). However, with or without a client secret, the attacker could

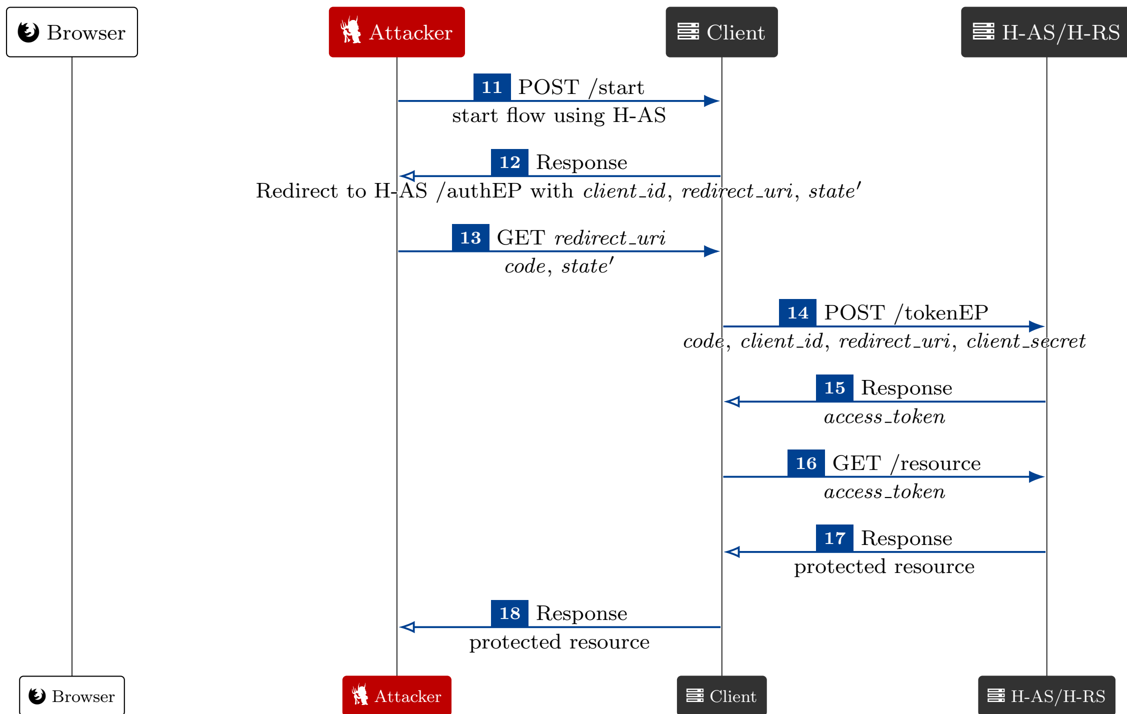


Figure 3.7. AS Mix-Up Attack on OAuth 2.0: Using code injection to break authorization.

instead launch a *code injection attack* (see Figure 3.7): The attacker starts a new OAuth flow at the client (using his own browser) 11. He selects H-AS as the AS for this flow and receives a redirect to H-AS, which he ignores 12. The redirect contains a cookie for a new login session and a fresh state parameter (*state*′). The attacker now crafts a request to the redirection endpoint of the client using *state*′ and *code* acquired earlier 13. Since the client will now provide its client secret to the token endpoint at H-AS 14, the client will receive an access token for the honest user's resources at H-RS 15, and the attacker will have access to these resources through the client 16–18. (The attacker does not learn an access token in this case.)

Breaking Authentication with Code Injection: Using code injection, the attacker can also break authentication and impersonate the honest user (see Figure 3.8). Just as before, the attacker, after obtaining *code*, starts a new login process at the client using H-AS 11, 12. Again as before, the attacker crafts a request to the redirection endpoint of the client using *state*′ and *code* acquired earlier 13. Now, the client will retrieve an access token from the token endpoint at H-AS (Steps 14 and 15) and subsequently use the token at the introspection endpoint to fetch the (honest) user's id (Steps 16 and 17). Being convinced that the attacker owns the honest user's account, the client issues a session cookie for this account to the attacker 18. As a result, the attacker is logged in at the client under the honest user's id. (Again, the attacker does not learn an access token.)

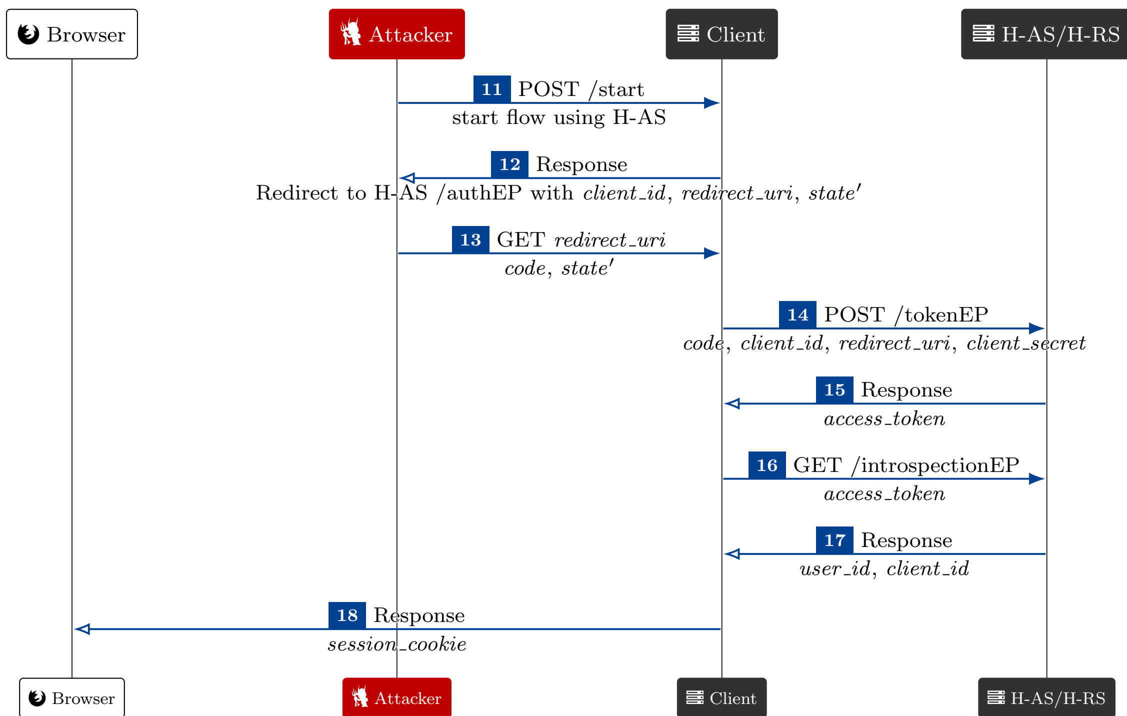


Figure 3.8. AS Mix-Up Attack on OAuth 2.0: Using code injection to break authentication.

Attack on the Implicit Grant

In the implicit grant, the attack works almost identical to the attack on the authorization code grant: As before, the attacker intercepts and modifies the first two messages. Unlike before, the user now returns from the authorization endpoint of H-AS with an access token instead of a code. The client believes that this access token was issued by A-AS and will therefore try to use the access token to access a resource at A-RS, leaking the token to the attacker. Since the access token is a bearer token (as explained above), the attacker can immediately use the token to break authorization, i.e., access the user's resources at H-RS. To break authentication, i.e., impersonate the honest user, the attacker can start a new login process (using his own browser) at the client with H-AS. Instead of following the redirect to the authorization endpoint at H-AS, the attacker can immediately forge a request to the redirection endpoint of the client using the captured access token. The client will now use the honest user's access token to retrieve user data at H-RS and the attacker will be logged in at the client under the honest user's identity.

Variant

There is also a variant of the AS Mix-Up Attack that only requires a web attacker (which does not intercept and manipulate network messages). In this variant, the user intends to log in with A-AS, but is redirected by A-AS to log in at H-AS. A user might "intend" to login with A-AS because A-AS disguises itself as H-AS (in the interface where the user selects the OAP), or because the client is misconfigured to use A-AS instead of H-AS.

In detail, the first four steps in Figure 3.5 are replaced by the following steps: First, the user starts a new OAuth flow with the client using A-AS. She is then redirected by the client to A-AS's authorization endpoint. Now, instead of prompting the user for her password, A-AS redirects the user to H-AS's authorization endpoint. As above, in this step, the attacker uses the state value he received from the browser plus the client id of client at H-AS. From here on, the attack proceeds exactly as in Figure 3.5 (Step 5 ff.).

Related Attacks

An attack in the same class, *cross social-network request forgery*, was outlined by Bansal, Bhargavan, Delignat-Lavaud, and Maffeis in [Ban+14]. It applies to clients with naïve user intention tracking (rather than explicit user intention tracking assumed in our AS Mix-Up Attack) in combination with AS, such as Facebook, that only loosely check the redirect URI. (Facebook, by default, only checks the origin of redirect URIs.) Our AS Mix-Up Attack works even if an AS strictly checks redirect URIs. The attack in [Ban+14] is described in the context of concrete social network implementations, but our findings show that this class of attacks is not merely an implementation error, but a more general problem in the OAuth standard, as confirmed by the IETF OAuth Working Group (see Section 5).

Another attack with a similar outcome, called *Malicious Endpoints Attack*, leveraging the OpenID Connect Discovery mechanism and therefore limited to OpenID Connect, was described in [Mla+16]. This attack assumes a CSRF vulnerability on the client's side.

Fix

A fundamental problem in the authorization code and implicit grants of the OAuth standard is a lack of reliable information in the redirect in Steps 6 and 7 in Figure 3.1 (even if HTTPS is used). The client does not receive information from where the redirect was initiated (when explicit user intention tracking is used) or receives information that can easily be spoofed (when naïve user intention tracking is used with OAPs such as Facebook). Hence, the client cannot check whether the information contained in the redirect stems from the AS that was indicated in Step 1.

Our fix therefore is to include the identity of the AS in the redirect URI in some form that cannot be influenced by the attacker, e.g., using a new URI parameter. Each AS should add such a parameter to the redirect URI. The client can then check that the parameter contains the identity of the AS it expects to receive the response from. (This could be used with either naïve or explicit user intention tracking, but to mitigate the Naïve Client Session Integrity Attack described below, we advise to use explicit user intention tracking only, see below.)

An early mitigation draft by the OAuth Working Group adopted this fix, with the parameter being called *iss* (for *issuer*) [JBS16]. The updated OAuth security recommendations, which we are working on together with the OAuth Working Group, recommend this parameter as well [Lod+18].

We show in Section 4.5 that this fix is indeed sufficient to mitigate the AS Mix-Up Attack (as well as the attacks pointed out in [Ban+14; Mla+16]).

3.3.3. State Leak Attack

Using the state leak attack, an attacker can force a browser to be logged in under the attacker's name at a client or force a client to use a resource of the attacker instead of a resource of the user. This attack, which breaks our session integrity property (see Section 3.5.3), enables what is often called session swapping or login CSRF [BJM08a].

State Leak from Client

After the user has authenticated to the AS in the authorization code grant, the user is redirected to the client (Step 7 in Figure 3.1). This request contains state and code as parameters. The response to this request (Step 14) can be a page containing a link to the attacker's website or some resource located at the attacker's website. (For example, it is not uncommon to include third-party advertisements, media files, or user-tracking scripts on such pages.) When the user clicks the link or the resource is loaded, the user's browser sends a request to the attacker. This request contains a `Referer` header with the full URI of the page the user was redirected to, which in this case contains state and code.

As the state value is supposed to protect the browser's session against CSRF attacks [RFC6819], the attacker can now use the leaked state value to perform a CSRF attack against the victim. For example, he can redirect the victim's browser to the client's redirection endpoint (again) and by this, overwrite the previously performed authorization. The user will then be logged in as the attacker.

Given the history of OAuth, leaks of sensitive data through the `Referer` header are not surprising. For example, the fact that the authorization code can leak through this header was described as an attack (in a similar setting) in [Hom14]. Since the authorization code is single-use only [RFC6749], it might already be redeemed by the time it is received by the attacker. State, however, is not limited to single use, making this attack easier to exploit in practice. Stealing the state value through the `Referer` header to break session integrity has not been reported as an attack before, as was confirmed by the IETF OAuth Working Group.

State Leak from AS

A variant of this attack exists if the login page at an AS contains links to external resources. If the user visits this page to authenticate at the AS and the browser follows links to external resources, the state is transferred in the `Referer` header. This variant is applicable to the authorization code grant and the implicit grant.

Fix

We suggest to limit state to a single use and to use the recently introduced Referrer Policies [ES17] to avoid leakage of the state (or code) to the attacker. Using Referrer Policies, a web server can instruct a web browser to (partially or completely) suppress the `Referer` header when the browser follows links in or loads resources for some web page. The `Referer` header can be blocked entirely, or it can be stripped down to the origin of the URI of the web page. Both options would prevent the state value from leaking. Referrer Policies are supported by all modern browsers.

In our OAuth model, clients and OAPs use Referrer Policies to specify that `Referer` headers sent when visiting links outgoing from any of their web pages may not contain more than the origin of the respective page. Our security proof shows that this measure is effective and protects against the attack in [Hom14] as well.

3.3.4. Naïve Client Session Integrity Attack

This attack again breaks the session integrity property for clients, where here we assume a client that uses naïve user intention tracking.[9]

Attack

First, an attacker starts a session with an honest AS (H-AS) to obtain an authorization code or access token for his own account. Next, when a user wants to log in at some client using A-AS (an AS controlled by the attacker), A-AS redirects the user back to the redirection URI assigned to H-AS at the client. To this redirection URI the A-AS attaches the state issued by the client, and the code or token obtained from H-AS. Now, since client performs naïve user intention tracking only, the client then believes that the user logged in at H-AS. Hence, the user is logged in at the client using the attacker's identity at H-AS or the client accesses the attacker's resources at H-AS believing that these resources are owned by the user.

Fix

The fix against the AS Mix-Up Attack (described above) does not work in this case: Since the client does not track where the user wanted to log in, it has to rely on parameters in the redirection URI which the attacker can easily spoof. Instead, we propose to always use explicit user intention tracking.

3.3.5. Across-AS State Reuse Attack

With this attack, an attacker can again break the session integrity property. The attack works if a client does not choose a fresh state value for each login/authorization attempt (and invalidates

[9] We may still assume that the OAuth state parameter is used, i.e., the client is not necessarily stateless.

any old values). A common approach is to choose a state value for a user's session with the client, but then leave this value unchanged (and valid) for multiple login attempts.

Attack

First, a user starts an OAuth flow at some client using a malicious AS (A-AS). A-AS learns the state value that is used in the current user session. Then, as soon as the user starts a new OAuth flow with the same client and an honest AS (H-AS), A-AS can use the known state value to mount a CSRF attack, breaking the session integrity property.

In this attack, the state value does not leak unintentionally (in contrast to the state leak attack).

Fix

Client implementations must ensure that a state that has been sent to one AS cannot be used for a login flow at another AS, or that state values that have been issued previously cannot be used for later login flows. For example, a client could choose a fresh nonce for state at each start of an OAuth flow and store that nonce in the user's session. If the client then uses explicit user intention tracking, an attacker would not be able to use the state in a different login flow.

Alternatively, [BLZ18] describes a mechanism to create signed state tokens that can only be used for one specific AS.

3.4. Other Attacks on OAuth

A number of attacks on OAuth 2.0 have been discovered in the past and respective security recommendations have been developed to implement and operate OAuth 2.0 securely. In this section, we summarize the most important attacks on OAuth and the security recommendations derived from these attacks. We have to follow these recommendations (in addition to those protecting against our new attacks) in our model in order to prove the security properties.

3.4.1. Code/Token/State Leakage

An attacker that has access to browsing histories (e.g., through malicious browser extensions) or logfiles of servers or proxies can steal authentication codes, access tokens, id tokens, or state values. The attacker can then, depending on the token, proceed as in the AS Mix-Up Attack or the State Leak Attack to break authentication, authorization, or session integrity. Such attacks have been dubbed *Cut-and-Paste Attacks* by the IETF OAuth working group [JBS16].

There are drafts for RFCs that tackle specific aspects of these leakage attacks, e.g., [BLZ18] which discusses binding the state parameter to the browser instance, and [Jon+] which proposes to bind the access token to a TLS session. Since these mitigations are still drafts, subject to change, and not implemented in the vast majority of OAuth deployments, we did not include

them in our model. They are, nonetheless, interesting candidates for future work in our model (see Chapter 6).

In the analysis, we assume that implementations keep logfiles and browsing histories (of honest browsers) secret, since otherwise, the attacks sketched above would easily break the security of OAuth. As mentioned above, we use Referrer Policies to protect against leakage through the `Referer` header.

3.4.2. CSRF Protection

Without proper CSRF protection, OAPs and clients can be vulnerable to CSRF attacks (as described in [Ban+14; SB12]). While [RFC6749] recommends the use of state to protect the core of the OAuth protocol, there are other endpoints that need to protected by effective mechanisms:

- **Clients** need to protect the endpoint where the OAuth flow is started (for the implicit and authorization code grants), the password login for the resource owner password credentials grant, and the URI to which the JavaScript posts the access token in the implicit mode.
- **OAPs** need to protect the endpoint to which the user credentials are posted.

In our model, we use `Origin` header checking for CSRF protection, which in practice might not be sufficient, since not all browsers support the `Origin` header yet. Therefore, we additionally recommend CSRF tokens [Ope18] or Same-Site Cookies [BW17] for these endpoints.

3.4.3. Third-Party Resources

Client and OAP websites that include active third-party content, in particular tracking or advertisement scripts, subject their users to token theft, phishing, and other attacks through the JavaScript delivered by these third parties. Malicious JavaScript running on an origin of an OAuth endpoint potentially has access to all tokens and cookies used in the OAuth flow. If possible, clients and OAPs should therefore avoid including third-party resources on the same origins as OAuth endpoints. For newer browsers, *subresource integrity* [Akh+16] can help to reduce the risks associated with embedding third-party resources. With subresource integrity, websites can instruct supporting web browsers to reject third-party content if this content does not match a specific hash.

In our model, we assume that clients and OAPs, as long as they are honest, do not include (untrusted) third-party JavaScript on their websites and do not have Cross-Site Scripting vulnerabilities. Otherwise, access tokens and authorization codes can be stolen in various ways, as described, among others, in [Ban+14; RFC6749; RFC6819; SB12].

3.4.4. Open Redirectors

In [RFC6819], an *open redirector* is described as "an endpoint using a parameter to automatically redirect a user agent to the location specified by the parameter value without any validation."

Open redirectors can allow an attacker to get access to code, state, or access token values. The preconditions are that the attacker can influence (parts of) the redirection URI such that the URI (a) points to an open redirector at the client, and (b) the URI is accepted by the AS, for example, because only a part of the redirection URI is actually checked by the AS. The attack then consists of the attacker influencing the redirection URI used in an OAuth flow such that the authorization endpoint redirects a user (with the secrets contained in the URI) back to an open redirector at the client's server, which then redirects the user to an attacker-controlled web site. The attacker can then perform the attacks we have seen above.

In order to prevent such attacks, clients must ensure that no registered redirection URI (and no URI matching a redirection URI pattern) points to an open redirector at the client. Additionally, open redirectors at the client web site should generally be avoided, since they can also be abused for phishing attacks (see [Ope17]). In our model, clients do not have open redirectors.

3.4.5. Session Handling

Sessions are typically identified by a nonce that is stored in the user's browser as a cookie. It is a well known best practice that cookies should make use of the *secure* attribute (i.e., the cookie is only ever used over HTTPS connections) and the *httpOnly* flag (i.e., the cookie is not accessible by JavaScript). Additionally, after the login, the client should replace the session ID of the user by a freshly chosen nonce in order to prevent session fixation attacks: Otherwise, a network attacker could set a login session cookie that is bound to a known state value into the user's browser, lure the user into logging in at the corresponding client, and then use the session cookie to access the user's data at the client (*session fixation*, see [Ope15; Zhe+15]). In our model, clients use two kinds of sessions: *Login sessions* (which are valid until just before a user is authenticated at the client) and *service sessions* (which signify that a user is already signed in to the client). For both sessions, cookies with the secure and httpOnly flags are used.

3.4.6. Access Token Introspection Client ID

When a client sends an access token to the introspection endpoint of an OAP for authentication (Step 12 in Figure 3.1), the OAP returns the user identifier and the client id for which the access token was issued (Step 13). The client must check that the returned client id is its own, otherwise a malicious client could impersonate an honest user at an honest client (see [RFC6749; Wan+13]). We therefore require this check also in our model.

3.5. Formal Analysis of OAuth 2.0

We now present our security analysis of OAuth. We start with an outline of our model before we introduce the security properties and state the main theorem, namely the security of OAuth w.r.t. these properties. We also provide a sketch of the proof.

The definitions of the security properties introduced in the following are similar to those for OpenID Connect, which we explain in more detail in Section 4.5.2. In order to avoid repetition, we here only provide an informal description.

See Appendices B.1–B.3 for the full details of the model, the security properties, and our proof.

3.5.1. Model: Design, Concepts, Limitations

Our model for OAuth is based on the Web Infrastructure Model presented in Chapter 2. We developed the OAuth model to adhere to [RFC6749], follow the security considerations described in [RFC6819], and include the mitigations against the attacks discussed above.

Design

Our comprehensive model of OAuth includes all configuration options of OAuth and makes as few assumptions as possible in order to strengthen our security results:

- Clients, OAPs, and browsers may run any of the **four OAuth grants** simultaneously. As described above, AS and RS run on the same server in our model, the OAP.
- Clients, OAPs and browsers can be **corrupted** by the attacker at any time.
- A client chooses **redirection URIs** explicitly or the OAP selects a redirection URI that was registered before. Redirection URIs can contain patterns. This covers all cases specified in the OAuth standard. We allow that OAPs do not strictly check the redirection URIs, and instead only check the origin.
- As in the OAuth standard, clients can be **public or confidential clients.** A single client may be a public client for one OAP and a confidential client for another OAP.
- Users may visit **HTTP and HTTPS URIs** of the servers in the model and parties are not required to use Strict-Transport-Security (STS), although we still recommend STS in practice. Web pages at clients can contain links to arbitrary external web sites.
- As usual in our web model, at any time the user can **navigate** backwards or forward in her browser history, navigate to any web page, open multiple windows, start simultaneous login flows using different or the same OAPs, etc.
- **User authentication at the authorization endpoint** of the OAP, which is out of the scope of OAuth, is performed using username and password. It is assumed that the user only ever sends her password over an encrypted channel and only to the AS this password was chosen for (or to trusted clients, as mentioned above). The user also does not reuse her password for different OAPs. Otherwise, a malicious OAP would be able to use the account of the user at an honest OAP.

Concepts Used in Our Model

We use the following concepts in our model and the security properties:

Protected Resources: Closely following [RFC6749], OAuth protected resources are an abstract concept for any resource a client could use at an RS after successful authorization. For example, if Facebook gives access to the friends list of a user to a client, this would be considered a protected resource. In our model, there is a mapping from (OAP, client, identity) to nonces (which model protected resources). In this mapping, the identity part can be $\perp$, modeling a resource that is acquired in the client credentials grant and thus not bound to a user.

Service Tokens: When OAuth is used for authentication, we assume that after successful login, the RP (client) sends a *service token* to the browser in a cookie (establishing a service session, cf. Section 3.4.5). The intuition is that with this service token a user can use the services of the RP. The service token consists of a nonce, the user's identifier, and the domain of the IdP (OAP) which was used in the login process. The service token is a generic model for any session mechanism the RP could use to track the user's login status (e.g., a cookie). We note that the actual session mechanism used by the RP *after* a successful login is out of the scope of OAuth, which is why we use the generic concept of a service token.

Trusted Clients: A browser can choose to launch the resource owner password credentials grant with any client, causing this client to know the password of the user. Clients, however, can become corrupted and thus leak the password to the attacker. Therefore, to define the security properties, we need the concept of *trusted clients*. Intuitively, this is a set of clients a user entrusts with her password. In particular, whether a client is trusted depends on the user. In our security properties, when we state that an adversary should not be able to impersonate a user u in a run, we would assume that all trusted clients of u have not become corrupted in this run.

Limitations

While our model of OAuth is very comprehensive, a few aspects of OAuth were not taken into consideration in our analysis: expiration and revocation of access tokens and session ids, user log out, error handling, and, as mentioned above, scopes and refresh tokens.

3.5.2. Model: Web Systems

Our model for OAuth defines two types of web systems: The *OAuth web system with a network attacker* for the analysis of authentication and authorization properties, and the *OAuth web system with web attackers* for the analysis of session integrity. (Recall that one network attacker subsumes multiple web attackers, as described in Section 2.2.)

The rationale behind using web attackers instead of a network attacker for the analysis of session integrity is that a network attacker can always forcefully log a user in under his own account by setting cookies from non-secure to secure origins [Zhe+15]. This is a common problem for cookie-based session management of web applications, independently of OAuth. In OAuth, a network attacker can use this attack vector to defeat the state parameter as a CSRF defense. We are interested in particular in this CSRF defense mechanism since it is contained in the OAuth specification, and therefore restrict our analysis of session integrity to web attackers.

We note, however, that more robust solutions for session integrity are conceivable, but are currently not standardized or not deployed. Such solutions could be based on TLS Token Binding [Pop+18] or JavaScript with Web Messaging or Web Storage.

OAuth Web System with a Network Attacker

We model OAuth as a class of web systems (in the sense of Section 2.7) that can contain an unbounded finite number of clients, OAPs, and browsers. We call a web system $\mathcal{OAuthWS}^n$ an OAuth web system with a network attacker if it is of the form described in what follows.

The **web system** consists of a network attacker, a finite set of web browsers, a finite set of web servers for the clients, and a finite set of web servers for the OAPs. The set of scripts consists of the three scripts *script_client_index*, *script_client_implicit*, and *script_oap_form*. We now briefly sketch clients, OAPs, and the scripts, with full details provided in Appendix B.

Each **client** is a web server modeled as an atomic Dolev-Yao process, including all OAuth modes, as well as the fixes and mitigations discussed before. The client can either (at any time) launch a client credentials grant or wait for users to start any of the other grants. As described in Section 3.4.5, the client manages two kinds of sessions: The login sessions, which are used only during the user login phase, and the service sessions modeled by a service token. When receiving a special message, a client can become corrupted and then behaves like an attacker process.

Each **OAP** is a web server modeled as an atomic Dolev-Yao process, again including all OAuth modes, as well as the fixes and mitigations discussed before. Users can authenticate to an OAP with their credentials. Just as clients, OAPs can become corrupted at any time.

The **scripts** which run in a user's browser are defined as follows:

- The script *script_client_index* is loaded from a client into a user's browser when the user visits the client's web site. It starts the authorization or login process.

- The script *script_client_implicit* is loaded into the user's browser from a client during an implicit grant flow to retrieve the data from the URI fragment. It extracts the access token and state from the fragment part of its own URI. The script then sends this information in the body of an HTTPS POST request to the client.

- The script *script_oap_form* is loaded from an OAP into the user's browser for user authentication at the OAP.

OAuth Web System with Web Attackers

In an *OAuth web system with web attackers*, the network attacker is replaced by an unbounded finite set of web attackers and a DNS server is introduced. We denote such systems by $\mathcal{OAuthWS}^w$ and use them for the analysis of session integrity properties.

3.5.3. Security Properties

Based on the formal OAuth model described above, we now formulate central security properties of OAuth, namely authorization, authentication, and session integrity (see Appendix B.2 for the full formal definitions).

Authorization

Intuitively, authorization for $\mathcal{OAuthWS}^n$ means that an attacker should not be able to obtain or use a protected resource available to some honest client at an OAP for some user unless, roughly speaking, the user's browser or the OAP is corrupted.

More formally, we say that $\mathcal{OAuthWS}^n$ is *secure w.r.t. authorization* if the following holds true: if at any point in a run of $\mathcal{OAuthWS}^n$ an attacker can obtain a protected resource available to some honest client r at an OAP i for some user u, then the OAP i is corrupt or, if $u \neq \bot$, we have that the browser of u or at least one of the trusted clients of u must be corrupted. Recall that if $u = \bot$, then the resource was acquired in the client credentials mode, and hence, is not bound to a user.

Authentication

Intuitively, authentication for $\mathcal{OAuthWS}^n$ means that an attacker should not be able to login at an (honest) client under the identity of a user unless, roughly speaking, the OAP involved or the user's browser is corrupted. As explained above, being logged in at a client under some user identity means to have obtained a service token for this identity from the client.

More formally, we say that $\mathcal{OAuthWS}^n$ is *secure w.r.t. authentication* if the following holds true: if at any point in a run of $\mathcal{OAuthWS}^n$ an attacker can obtain the service token that was issued by an honest client using some OAP i for a user u, then the OAP i, the browser of u, or at least one of the trusted clients of u must be corrupted.

Session Integrity

Intuitively, session integrity (for authorization) means that (a) a client should only be authorized to access some resources of a user when the user actually expressed the wish to start an OAuth flow before, and (b) if a user expressed the wish to start an OAuth flow using some honest OAP and a specific identity, then the OAuth flow is never completed with a different identity (in the same session); similarly for authentication.

More formally, we say that $\mathcal{OAuthWS}^w$ is *secure w.r.t. session integrity for authorization* if the following holds true: (a) If in a run $\mathcal{OAuthWS}^w$ an OAuth login flow is completed with a user's browser, then this user started an OAuth flow. (b) If in addition we assume that the OAP that is used in the completed flow is honest, then the flow was completed for the same identity for which the OAuth flow was started by the user. We say that the OAuth flow was completed (for some identity v) iff the client gets access to a protected resource (of v).

We say that $\mathcal{OAuthWS}^w$ is *secure w.r.t. session integrity for authentication* if the following holds true: (a) If in a run ρ of $\mathcal{OAuthWS}^w$ a user is logged in with some identity v, then the user started an OAuth flow. (b) If in addition the OAP that is used in that flow is honest, then the user is logged in under exactly the same identity for which the OAuth flow was started by the user.

3.5.4. The OAuth Security Theorem

We prove the following theorem (see Appendix B.3 for the proof):

Theorem 1 (Security of OAuth). Let $\mathcal{OAuthWS}^n$ be an OAuth web system with a network attacker, then $\mathcal{OAuthWS}^n$ is secure w.r.t. authorization and secure w.r.t. authentication. Let $\mathcal{OAuthWS}^w$ be an OAuth web system with web attackers, then $\mathcal{OAuthWS}^w$ is secure w.r.t. session integrity for authorization and authentication.

This trivially implies that authentication and authorization properties are satisfied also if web attackers are considered.

3.5.5. Proof of the OAuth Security Theorem: Outline

We first show three basic lemmas that apply to honest clients and capture specific technical details:

1. Messages transferred over HTTPS connections that were initiated by honest clients cannot be read or altered by other parties. In particular, honest clients do not leak the encryption keys to other parties.
2. HTTP(S) messages which await DNS resolution in a state of an honest client are later sent out over the network without being altered in between.
3. Honest clients never send messages to other clients or themselves, and they send only HTTPS messages that other clients cannot decrypt.

Authentication

We then prove the authentication property by contradiction. To this end, we show in three separate lemmas building on each other that (1) the attacker does not learn passwords of the user, (2) the attacker does not learn authorization codes that could be used to learn a relevant

access token, and (3) that the attacker in fact does not learn an access token that could be used to retrieve a service token as described in the authentication property. We finally show that there is no other way for an attacker to get hold of a service token (as described in the authentication property), and that therefore, the authentication property holds true.

Authorization

As above, we assume that the authorization property does not hold and lead this to a contradiction. The proof then builds upon the lemmas shown in the authentication proof. We show that the attacker would need to know an access token to acquire a protected resource. If the protected resource is bound to a user (i.e., it was not issued in the client credentials grant), then (3) from above applies and shows that the attacker cannot learn such an access token, and thus cannot learn this protected resource. If the protected resource was not assigned to a user (i.e., it was issued in the client credentials grant), then we can show that the attacker would need to know client secrets to get the protected resource. We show, however, that it is not possible for the attacker to learn the necessary client secrets (which are always required in the client credentials grant). Therefore, whether it is a user-bound protected resource or not, the attacker cannot learn it, leading our assumption to a contradiction.

Session Integrity

We first show session integrity for authorization. To this end, we show that an OAuth flow with an honest browser b and honest client r can only be completed when it was actively started by b, i.e., the correct script was run under an origin of r and this script started the login using some identity v. This is achieved by showing the existence of certain events, starting from the last event (where the flow is completed) and backtracing to a starting event. We then show that if i is also honest, the start and end events belong to the same flow, and that the identity v that was selected in this flow is exactly the same identity for which r accesses a resource in the last event. This is done by showing that all events (from the event where the identity was selected to the last event) are connected and that certain values (such as the chosen identity) are relayed correctly and not modified in between processing steps or messages. We then show that session integrity for authentication follows from session integrity for authorization.

3.5.6. Discussion of Results

Our results show that the OAuth standard is secure, i.e., provides strong authentication, authorization, and session integrity properties, when it is fixed according to our proposal and adheres to the OAuth security recommendations and best practices, as explained in Section 3.5.1. Depending on individual implementation choices, not all of these conditions can be met in all practical scenarios. For example, clients might run untrusted JavaScript on their websites. Nevertheless, our security results, for the first time, give precise implementation guidelines for

OAuth to be secure and also clearly show that if these guidelines are not followed, then the security of OAuth cannot be guaranteed.

4. Analysis of OpenID Connect

In this chapter, we present our analysis of OpenID Connect. First, we introduce the OpenID Connect protocol and the extensions analyzed in this thesis, building on concepts introduced in the previous chapter. Afterwards, we discuss attacks on OpenID Connect and finally present our formal analysis. We present the details of our analysis in Appendix C.

4.1. OpenID Connect Basic Concepts

As we have seen before, OAuth 2.0 is not only used for authorization, for which is was designed initially, but is also used for authentication. To this end, non-standardized ways for authentication in OAuth 2.0 were used, like the one presented in the previous chapter. The incorrect implementation of authentication in OAuth 2.0 led to security problems in the past [Bra12].

OpenID Connect was designed as a *standardized* authentication layer on top of OAuth (retaining the option for authorization) and provides other features not initially contained in OAuth: For example, an OIDC RP (client) can register itself at an IdP (also called *OpenID Provider, OP*) dynamically and automatically. This feature is called Dynamic Client Registration and replaces the manual registration process in OAuth outlined in Section 3.1.3. Often, Dynamic Client Registration is used together with another OIDC extension, Discovery, which enables a client to find the OP that is responsible for a certain user identity.

OIDC was defined by the OpenID Foundation in a *Core* document [Sak+14a] and in extension documents (e.g., [Sak+14b; SBJ14]). Supporting technologies were standardized at the IETF, e.g., [RFC7033; RFC7519].

Throughout this chapter, to align with the official OIDC terminology, we use the terms “RP” for the client and “IdP” or “OP” for the AS/RS.

4.1.1. Relationship to OAuth 2.0

At the core of each OIDC flow there is an OAuth flow. The Discovery and Dynamic Registrations extensions add new steps before the OAuth flow and OIDC introduces new parameters and a new token, the *id token*. The id token is created by the user’s identity provider and serves as a one-time proof of the user’s identity to the relying party. It is used as the primary means for user authentication. Some messages and tokens in OIDC can be cryptographically signed or encrypted while OAuth 2.0 does neither use signing nor encryption (besides HTTPS). The new hybrid flow combines features of the implicit grant and the authorization code grant. There are

no flows similar to the resource owner password credentials grant or the client credentials grant in OIDC.

Clearly, the addition of these features and their interplay introduce new potential security flaws. For example, an attacker, playing the role of an OP, can try to provide manipulated endpoint URIs to an RP during the Discovery phase of the protocol. This might lead to attacks, as shown in [Mla+16]. It is therefore not sufficient to analyze the security of OAuth 2.0 to derive any guarantees for OIDC, a new security analysis is required.

4.1.2. Authentication, ID Tokens, and Issuer Identifiers

The main goal of OpenID Connect is to *authenticate* a user to an RP, i.e., the RP gets assured of the identity of the user interacting with the RP. This assurance is based on id tokens. An id token is a JSON Web Token (JWT, see [RFC7519]) signed by the OP that carries *claims*, i.e., information about the user and meta data about the authentication process.

More precisely, an id token contains:

- A *user identifier* (unique at the respective OP) and the *issuer identifier* of the OP. The issuer identifier of an OP is an HTTPS URL without any query or fragment components. The path component may be used to host several different IdPs under a single domain. Both identifiers in combination serve as a global user identifier for authentication.
- The *audience*, i.e., the client id of the RP at the OP, which is assigned during registration.
- A *nonce* chosen by the RP during the authentication flow (optional).
- An expiration timestamp and a timestamp of the user's authentication at the OP to prevent replay attacks.
- Optionally, information about the particular method of authentication and other claims, such as further meta data about the user and a hash of some data sent outside of the id token.

When an RP validates an id token, it checks in particular whether the signature of the token is correct (we explain below how RP obtains the public key of the OP), the issuer identifier points to the OP currently used, the id token is issued for this RP (audience matches the RP's client id), the nonce is the one RP has chosen during this login flow (if any), and the token has not expired yet. If the id token is valid, the RP trusts the claims contained in the id token and is confident in the user's identity.

4.2. Discovery and Dynamic Registration Extensions

Just as in OAuth, an RP and an OP that want to run the OIDC protocol need to know the endpoint URLs of each other. Additionally, the RP needs to know its client id and client secret

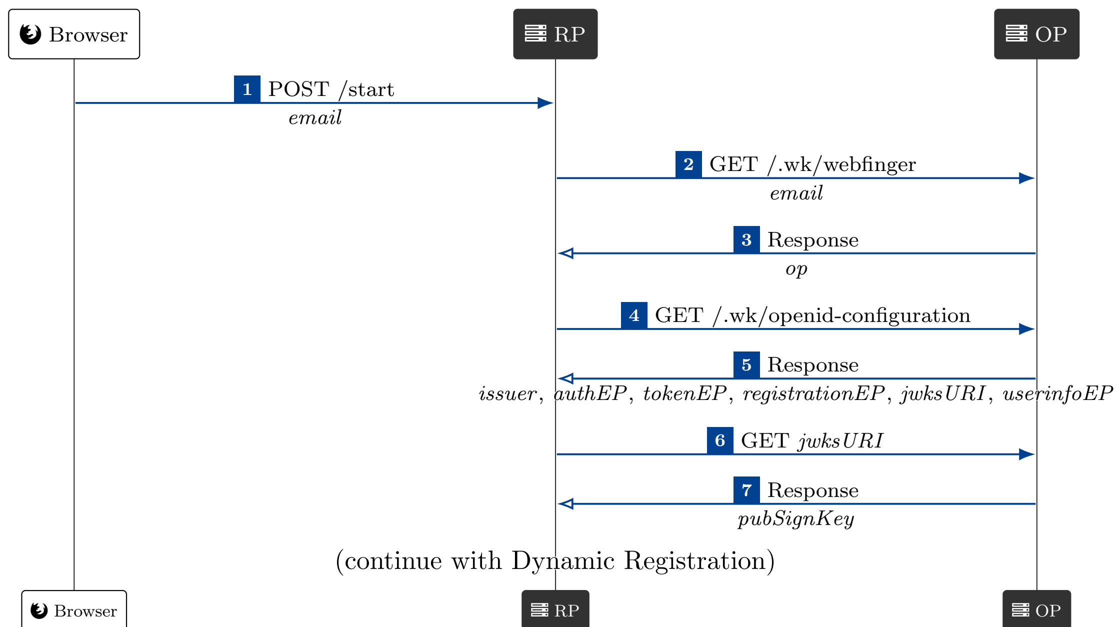


Figure 4.1. OpenID Connect Discovery extension protocol flow. As above, data shown below the arrows is either transferred in URI parameters, HTTP headers, or POST bodies. The server of the user's email domain is depicted as the same party as the OP.

at the OP and a public key of the OP to verify the signature of id tokens. This information can be exchanged either manually (as in OAuth) or automatically using the Discovery and Dynamic Registration extensions described in the following.

4.2.1. OpenID Connect Discovery

During the automated discovery as defined in [Sak+14b], the RP first uses the WebFinger protocol [RFC7033] to determine which OP is responsible for the user who wants to log in. The RP learns the issuer identifier of the OP and can retrieve the URLs of the authorization endpoint and the token endpoint from the OP. Furthermore, the RP receives a URL where it can retrieve the public key to verify the signature of the id token (*JWKS URI*, cf. [RFC7515]), and a URL where the RP can register itself at the OP (*client registration endpoint*).

Step-by-Step Protocol Flow

The flow defined by the Discovery extension is depicted in Figure 4.1. First, the user starts the login process by entering her email address in her browser (at some web page of an RP), which sends the email address to the RP in 1.[1] Now, the RP uses the OpenID Connect Discovery extension to gather information about the OP:

[1] In our examples and many real-world implementations, the user identifies herself by her email address, but other types of identifiers, such as personal URIs, are conceivable.

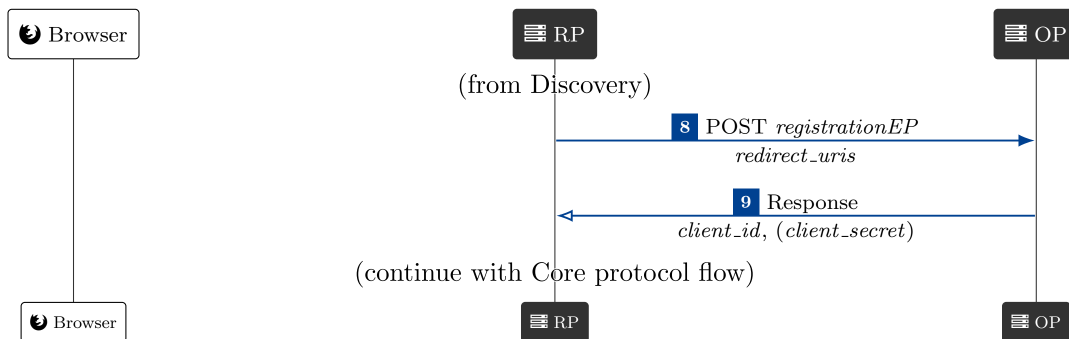


Figure 4.2. OpenID Connect Dynamic Registration extension protocol flow. This step is skipped if the RP is already registered at the OP.

– As the first step, the RP uses the WebFinger mechanism [RFC7033] to discover information about which OP is responsible for this user. For this discovery, the RP contacts the server of the user's email domain 2.

– The result of the WebFinger request contains the issuer identifier of the OP 3.

– With this information, the RP can continue the discovery by requesting the OIDC configuration from the OP 4, 5. This configuration contains meta data about the OP, including all endpoints of the OP and a URL where the RP can retrieve the public key of the OP (used to later verify the id token's signature).

– If the RP does not know this public key yet, the RP retrieves the key 6, 7.

This concludes the OIDC Discovery protocol.

4.2.2. OpenID Connect Dynamic Client Registration

If the RP has not registered itself at this OP before, it registers itself at the client registration endpoint using the Dynamic Client Registration protocol [SBJ14]: The RP sends a list of its redirection endpoint URLs to the OP and receives a new client id and (optionally) a client secret in return.

Step-by-Step Protocol Flow

The flow defined by the Dynamic Registration extension is depicted in Figure 4.2:

– The RP contacts the OP and provides its redirect URIs 8.

– In return, the OP issues a client id and (optionally) a client secret to the RP 9.

This concludes the OpenID Connect Dynamic Client Registration protocol.

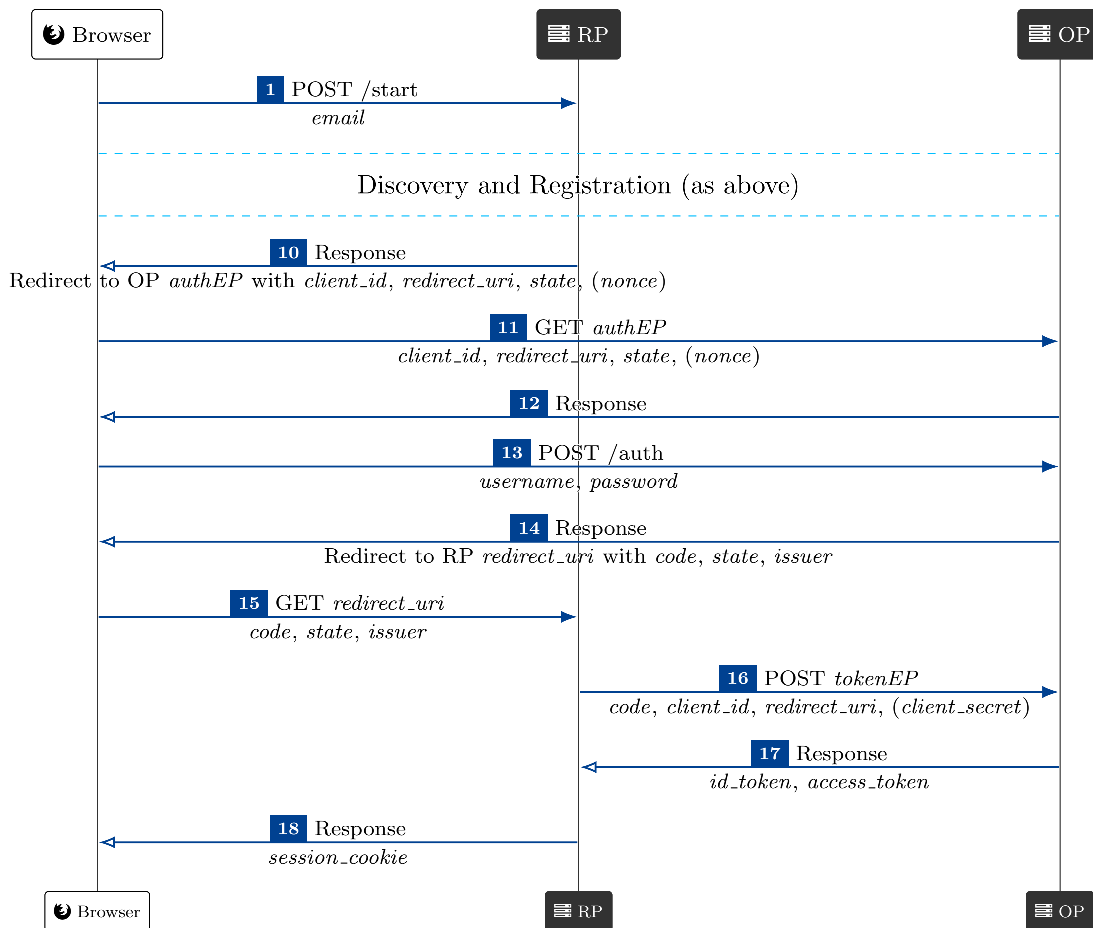


Figure 4.3. OpenID Connect authorization code flow.

4.3. OpenID Connect Flows

OIDC defines three *flows*: the *authorization code flow*, the *implicit flow*, and the *hybrid flow*. The first two are directly based on the respective OAuth grants. In the authorization code flow, the id token is retrieved by an RP from an OP in direct server-to-server communication (sometimes called *back channel*), and in the implicit flow, the id token is relayed from an OP to an RP via the user's browser (also called *front channel*). The hybrid flow is a combination of both flows and allows id tokens to be exchanged via the front and the back channel at the same time. We now provide a detailed description of all three flows.

4.3.1. Authorization Code Flow

In this flow, an RP redirects the user's browser to an OP. At the OP, the user authenticates herself and then the OP issues an authorization code to the RP. The RP now uses this authorization code to obtain an id token (and, optionally, an access token) from the OP.

Step-by-Step Protocol Flow

First, the Discovery and Dynamic Registration protocols, as presented above, are executed (if needed). Then, the core part of the OpenID Connect protocol starts (depicted in Figure 4.3):

- The RP redirects the user's browser to the OP [10]. This redirect contains the information that the authorization code flow is used, the client id of the RP, a redirect URI, and a state value. The redirect may also optionally include a nonce, which will be included in the id token issued later in this flow.

- This data is sent to the OP by the browser [11].

- The user authenticates to the OP [12], [13], and the OP redirects the user's browser back to the RP [14], [15]. The OP uses the redirect URI from the request in Step [11]. The redirect contains an authorization code, the state value received in Step [10], and the issuer identifier.[2]

- If the state value and the issuer identifier are correct, the RP contacts the OP at the token endpoint with the received authorization code, its client id, its client secret (if any), and the redirect URI used to obtain the authorization code [16].

- The OP sends a response with a fresh access token and an id token to the RP [17].

- If the id token is valid, the RP considers the user to be logged in under the identifier composed from the user id in the id token and the issuer identifier. Hence, the RP may set a session cookie at the user's browser [18]. Optionally, the RP can use the access token to access the user's resources at some RS.

4.3.2. Implicit Flow

This flow is similar to the authorization code flow, but instead of providing an authorization code, the OP issues an id token right away to the RP (via the user's browser) when the user authenticates to the OP.

Step-by-Step Protocol Flow

The protocol flow is depicted in Figure 4.4. The implicit flow differs only in its last part from the authorization code flow, i.e., the Steps [10]–[13] of the authorization code flow (Figure 4.3) are the same, with the exception that the nonce is mandatory in the implicit flow.

As already mentioned above, the OP does not issue an authorization code in Step [14] (Figure 4.4). Instead, the OP redirects the user's browser to the redirection endpoint at the RP, providing an id token, optionally an access token, the state value (as received in Step [11]), and the issuer identifier. These values are not provided as a URL parameter but in the URL

[2]The issuer identifier is included here as a fix against the AS Mix-Up attack, cf. Section 3.3.2.

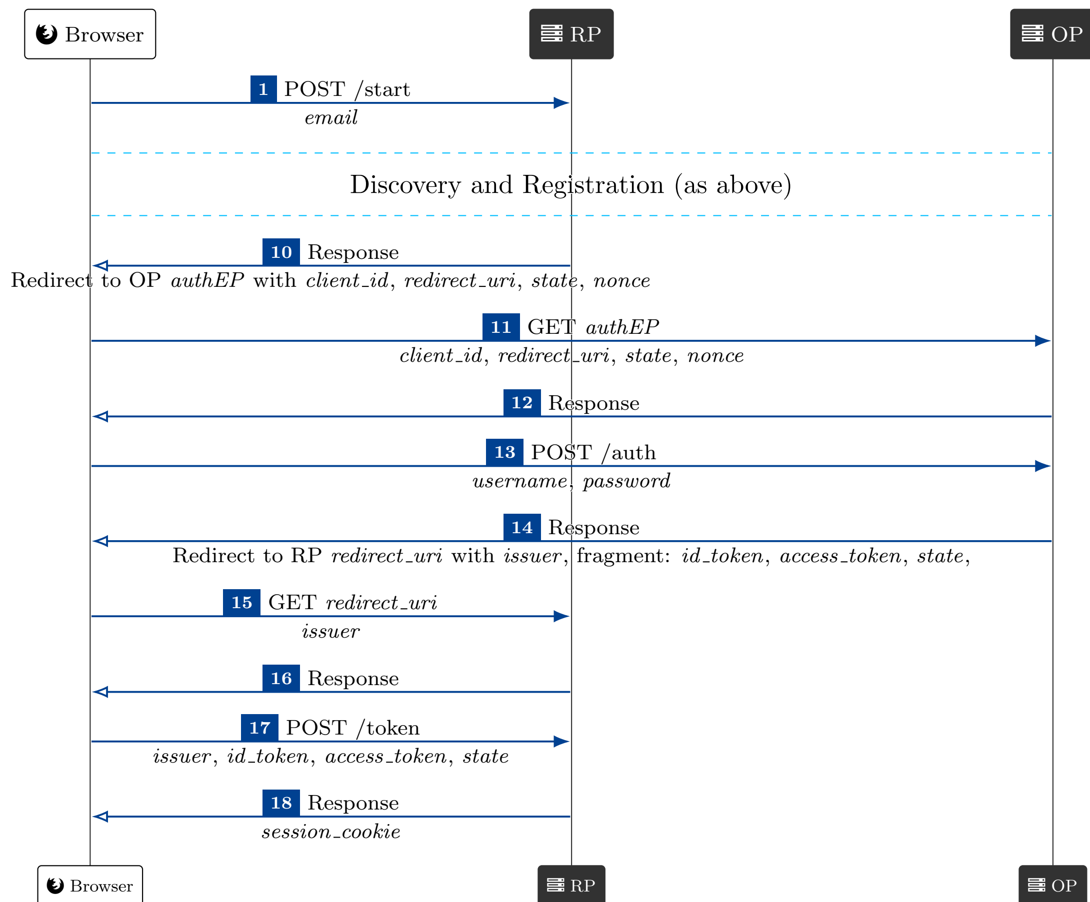


Figure 4.4. OpenID Connect implicit flow.

fragment instead, as in the OAuth implicit grant. Hence, the browser does not send them to the RP at first. Instead, the RP has to provide a JavaScript that retrieves these values from the fragment and sends them to the RP. If the id token is valid, the issuer is correct, and the state matches the one chosen by the RP for Step 10, the RP considers the user to be logged in and issues a session cookie.

4.3.3. Hybrid Flow

The hybrid flow (depicted in Figure 4.5) is a combination of the authorization code flow and the implicit flow: First, it works like the implicit flow, but when OP redirects the browser back to RP (Step 14), the OP issues an authorization code, and either an id token or an access token or both.[3] The RP then retrieves these values as in the implicit flow (as they are sent in the fragment like in the implicit flow) and uses the authorization code to obtain a (potentially second) id token and a (potentially second) access token from OP (Steps 18 f.).

[3]The choice of the OP to issue either an id token or an access token or both depends on the OP's configuration and the request in Step 11 in Figure 4.5.

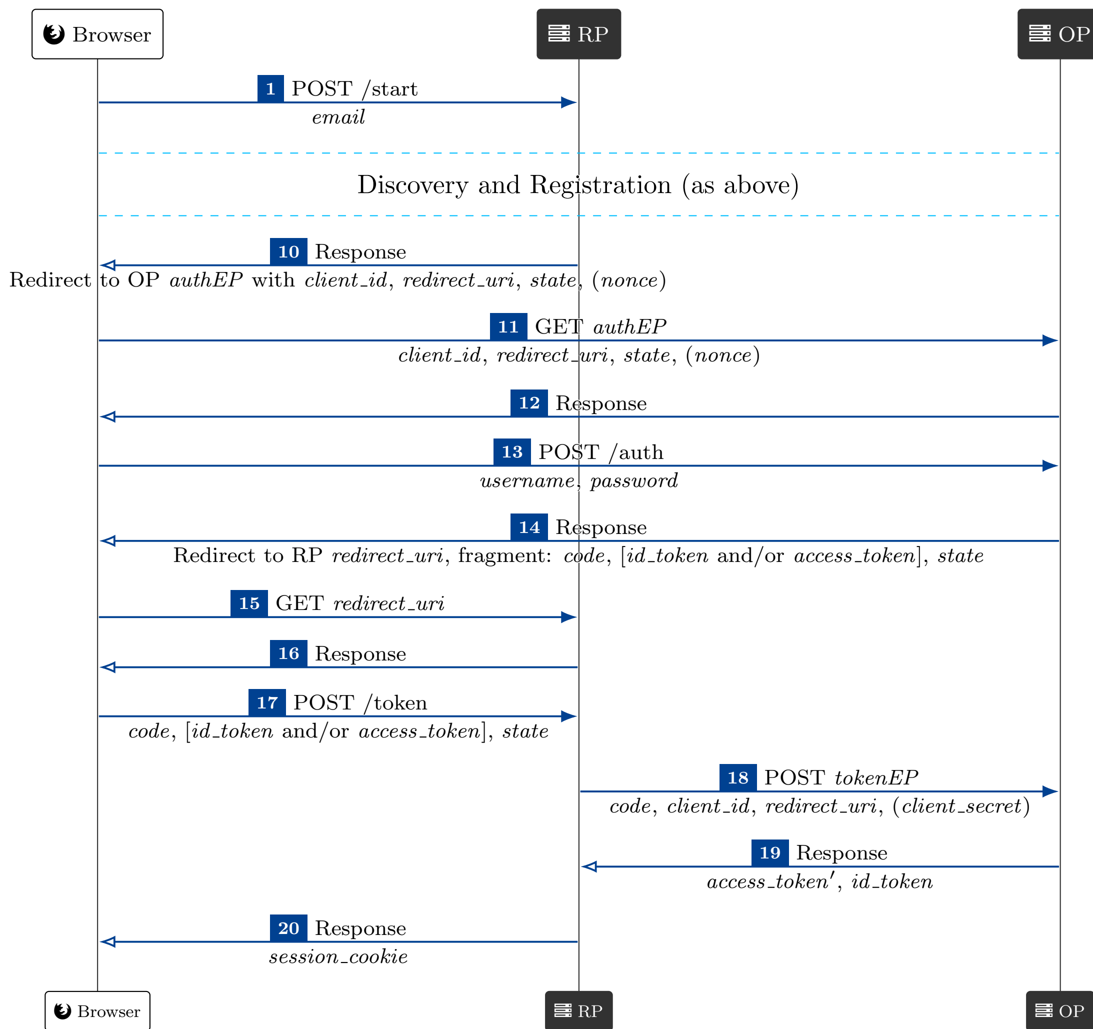


Figure 4.5. OpenID Connect hybrid flow.

4.4. Attacks on OpenID Connect

In this section, we revisit the attacks on OAuth presented in Section 3.3 and discuss if and how they apply to OpenID Connect as well (cf. Table 4.1). We also discuss other attacks on OpenID Connect and present some new extensions and variants that have not been documented so far. We further show mitigations and implementation guidelines that help to avoid all of these attacks. Attacks that are already covered in the OIDC Core standard [Sak+14a] itself are not listed here.

Unless noted otherwise, we incorporate the mitigations and adhere to the guidelines established in the following also in our model. We thereby show that the mitigations and guidelines are effective and sufficient to protect not only against the attacks described here but also against other, potentially unknown types of attacks captured by our security properties. This section also serves to show the state of the art regarding OpenID Connect security.

	attack on OpenID Connect flow		
	authorization code	implicit	hybrid
307 Redirect Attack	az + an	az + an	az + an
AS Mix-Up Attack	az + an	–	az + an*
State Leak	si	si	si
Naïve Client Session Integrity Attack	si	si	si
Across-AS State Reuse Attack	si	si	si

az: breaks authorization. **an:** breaks authentication. **si:** breaks session integrity. **–:** not applicable. ***** restriction: if client secrets are used, either authorization or authentication is broken, depending on implementation details.

Table 4.1. Overview of attacks on OpenID Connect.

4.4.1. AS Mix-Up Attacks

Recall that in the AS Mix-Up Attack presented in Section 3.3.2 and its variants, the aim was to confuse the RP about the identity of the OP. In all cases, the user was tricked into using an *honest* OP to authenticate to an *honest* RP, while the RP is made to believe that the user authenticated to the attacker. The RP therefore, after successful user authentication, tries to use the authentication token (authorization code or access token) at the attacker's server, by which the attacker learns this token and can impersonate the user or access the user's data at the OP.

AS Mix-Up Attack in OpenID Connect

Just as in the case of OAuth, the AS Mix-Up Attack on OpenID connect can be performed in several variants. Here, we first describe two variants to *start* the attack in the hybrid flow of OIDC. Below, we elaborate on three distinct methods to *continue* the attack that can be combined with both variants of starting the attack.

The start of the attack is depicted in Figure 4.6.

- To start the login flow, the user selects an OP at RP by entering her email address [1].

  This step is the main difference between the two variants to start the attack: In Variant 1, the user selects a malicious OP, say A-OP. In Variant 2, the user selects an honest OP, say H-OP, but the request is intercepted by the attacker and altered such that the attacker replaces the honest OP by A-OP (*email* is replaced by *email′* in Steps [1] and [2] in Figure 4.6).[4]

- Now, RP starts with the discovery phase of the protocol. Since RP thinks that the user wants to login with A-OP, it retrieves the OIDC configuration from A-OP [5], [6]. In this configuration, the attacker does not let all endpoint URLs point to himself, as would be usual for OIDC, but instead sets the authorization endpoint to be the one of H-OP.

- Next, the RP registers itself at A-OP [9], [10]. In this step, A-OP issues the same client id to RP which RP is registered with at H-OP (client ids are public). This is important as

[4]This initial request is often unencrypted in practice, see [FKS16].

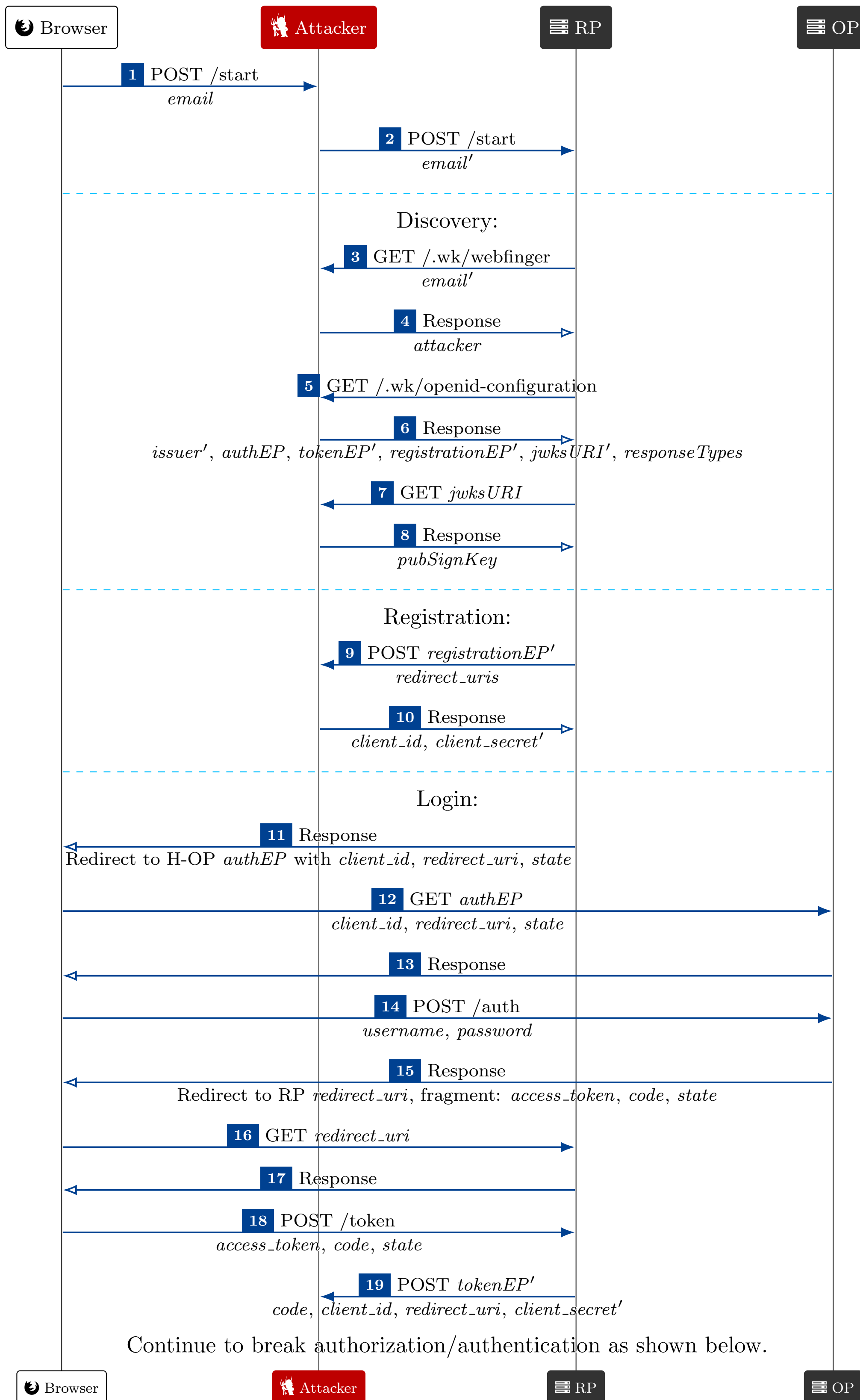


Figure 4.6. Start of the AS Mix-Up Attack on the OpenID Connect hybrid flow.

H-OP will later redirect the user's browser back to RP and checks the redirect URI based on the client id.

- Next, RP redirects the user's browser to H-OP (Variant 1) or A-OP (Variant 2) 11. In Variant 1 of the attack, a vigilant user might now be able to detect that she tried to log in using A-OP but instead is redirected to H-OP. This does not happen in Variant 2, but here the attacker needs to replace the redirection to A-OP by a redirection to H-OP.
- The user then authenticates at H-OP and is redirected back to RP along with an authorization code and an access token[5] 12–15.
- Now, RP retrieves the authorization code and the access token from the user's browser and continues the login flow 16, 18. As RP still assumes that A-OP is used in this case, it tries to redeem the authorization code for an id token (and a second access token) at A-OP 19.

The attacker can now continue the attack in different ways. We here present three interesting variants:

Breaking Authentication with Code Injection

As the authorization code has not been redeemed at H-OP yet, the code is still valid and the attacker may start a second login flow (pretending to be the user) at RP (see Figure 4.7). The attacker skips the authentication at H-OP and returns to RP with the authorization code he has learned before 22. RP now redeems this code at H-OP and receives an id token issued for the honest user and consequently assumes that the attacker has the identity of the user and logs him in 23–25.

This shows that, using the AS Mix-Up attack, an attacker can successfully impersonate users at RPs and access their data at honest OPs.

Breaking Authentication without Code Injection

In another variation of the attack, if H-OP does not issue client secrets to RPs, the attacker can also redeem the authorization code by himself (see Figure 4.8). In this case, the attacker receives an access token valid for the user's account. With this access token, he can retrieve data of the user or act on the user's behalf at H-OP. (As he redeems the authorization code, he cannot use it to log himself into the RP in this case.)

Breaking Authorization with Mock Tokens

In any case, the attacker can also respond to the authorization code sent to his token endpoint with a mock access token and a mock id token (which will not be used in the following). This is

[5] Depending on the sub-mode of the hybrid flow, OPs do not send id tokens in this step.

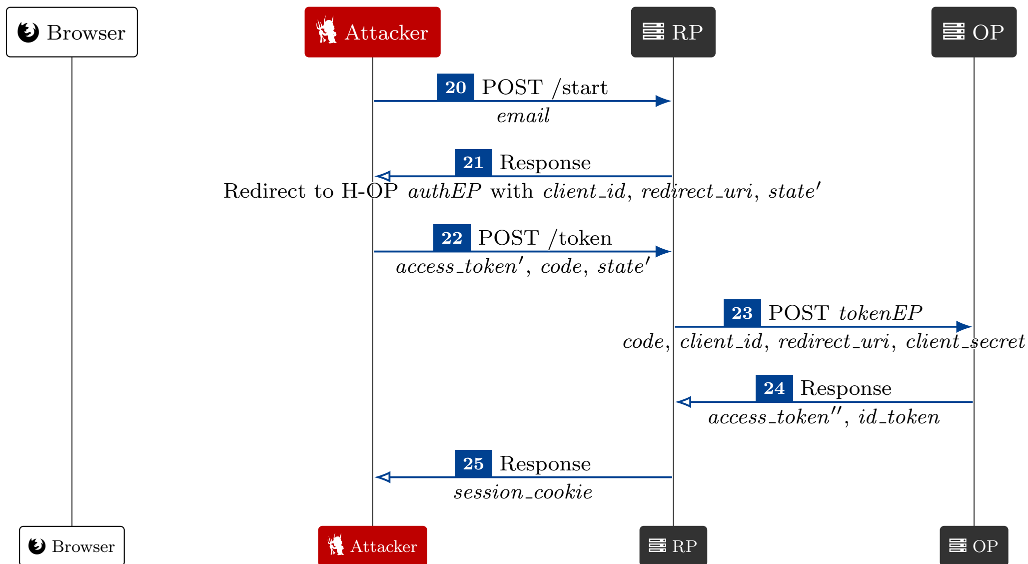


Figure 4.7. AS Mix-Up Attack on OIDC: Breaking authentication with code injection.

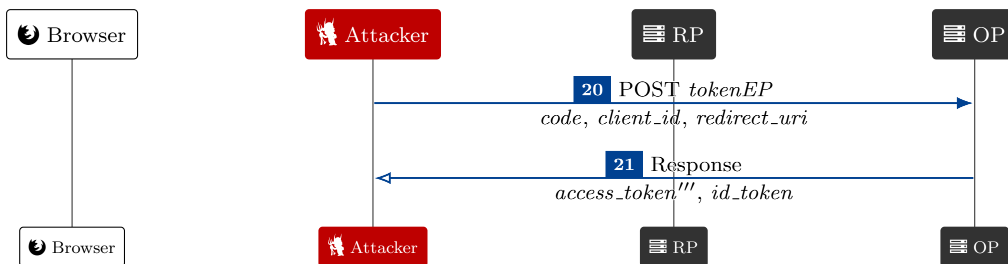


Figure 4.8. AS Mix-Up Attack on OIDC: Breaking authentication without code injection.

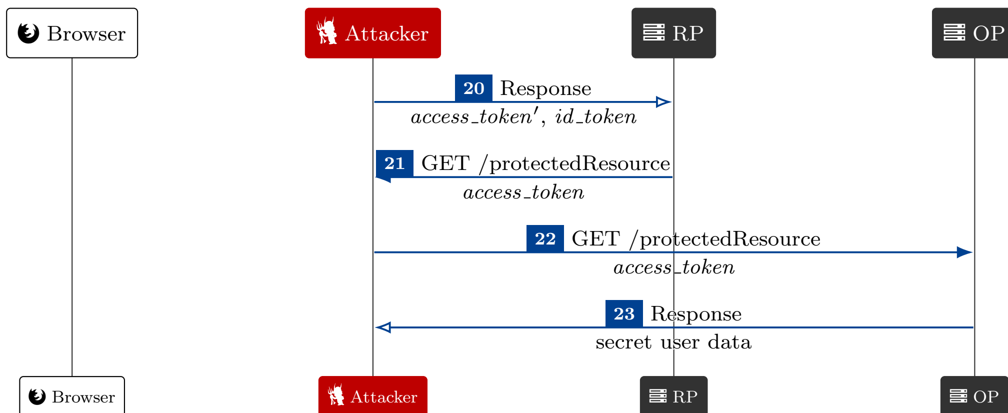


Figure 4.9. AS Mix-Up Attack on OIDC: Using a mock access token.

depicted in Figure 4.9, Step 20. In the next step, the RP *might* then use the access token learned from the honest OP to retrieve data of the user from A-OP 21.[6] Then the attacker learns also this access token, which (as described in the paragraph above) grants him unauthorized access to the user's account at H-OP 22, 23.

Fixing OpenID Connect

Our analysis of OpenID Connect shows that the mitigation presented in Section 3.3.2 is effective for OpenID Connect as well and protects against all attack variants shown above.

4.4.2. Attacks on the State Parameter

The state parameter is used in OpenID Connect, just as in OAuth, to protect against attacks on session integrity, i.e., to prevent an attacker from forcing a user to be logged in at some RP under the attacker's account. The use of the state parameter is recommended by the OIDC standard [Sak+14a].

All of the attacks on the state parameter presented in Section 3.3 apply to OpenID Connect as well: The State Leak attack, the Naïve Client Session Integrity attack, and the Across-AS State Reuse attack. Therefore, the fixes proposed in Section 3.3 should be used in OpenID Connect setups as well.

4.4.3. 307 Redirect Attack

Since OIDC is based on OAuth, the 307 Redirect Attack (Section 3.3.1) applies to OIDC as well. Assumption (1) in Section 3.3.1 is still reasonable: The OIDC standard does not define the redirection code or method. In our model, we therefore exclusively use the 303 status code, which does not instruct the browser to re-send form data.

4.4.4. Server-Side Request Forgery

Server-Side Request Forgery (SSRF) attacks can arise when an attacker can instruct a server to send HTTP(S) requests to other hosts, causing unwanted side-effects or revealing information [Pel+16]. For example, if an attacker can instruct a server behind a corporate firewall to send requests to other hosts behind this firewall, the attacker might be able to call services or to scan the internal network (using timing attacks). He might also instruct the server to retrieve very large documents from other sources, thereby creating Denial of Service attacks.

The first SSRF attack on OIDC was described in [Mla+16], in the context of the OIDC Discovery extension: An attacker can set up a malicious discovery service that, when queried by an RP, answers with links to arbitrary, network-internal or external servers (in Step 5 of Figure 4.1).

[6] Depending on the RP implementation, the RP might choose to use the mock access token or the one learned from the honest OP in this step. In the real-world implementation mod_auth_openidc, the access token from the honest OP was used.

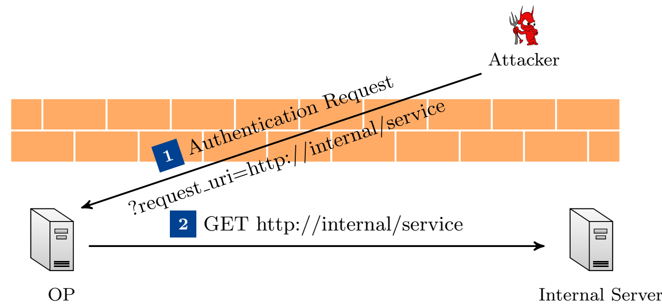


Figure 4.10. Server-Side Request Forgery in OIDC with an OP and an internal server behind a firewall.

We point out that not only RPs using the Discovery extension can be affected by SSRF vulnerabilities, but also OPs, even if they do not use the Discovery extension. The OIDC Core standard [Sak+14a] defines in Section 6.2.2 a way to indirectly pass the parameters for the authorization request (cf. Step 11 in Figure 4.5). To this end, RPs can create a JSON document containing the parameters that are to be passed indirectly (e.g., *redirect_uri*) and make this document available at some URI. Then, this URI is passed in the authorization request in a new parameter, *request_uri*. The OP then fetches this document and uses the parameters contained therein as if they were contained in the authorization request URI. The attacker can therefore easily mount an SSRF attack against the OP: He only needs to access the authorization endpoint of the targeted OP and provide a crafted URI (for example, pointing to an internal server) in the *request_uri* parameter. The OP will then try to access the *request_uri* and the attacker can, for example, determine by the timing whether the internal server exists, receive an error message containing further information about the internal server, or even cause some unwanted action on the internal server. Such an attack is depicted in Figure 4.10.

This new attack vector shows that not only RPs but also OPs have to protect themselves against SSRF by using appropriate filtering and limiting mechanisms to restrict unwanted requests that originate from a web server (see [Pel+16]).

Since SSRF attacks depend heavily on the structure of and (vulnerable) services on an internal network, and often also on timing and performance properties, they are not part of our model.

4.4.5. CSRF Attacks and Third-Party Login Initiation

Some endpoints in OIDC need protection against Cross-Site Request Forgery in addition to the protection that the state parameter provides, e.g., by checking the `Origin` header. Our analysis (see Section 4.5) shows that it is sufficient for the RP to protect the URI on which the login flow is started (otherwise, an attacker could start such a login flow using his own identity in a user's browser) and for the OP to protect the URI where the user submits her credentials (otherwise, an attacker could submit his credentials instead). The redirection endpoint at the RP is sufficiently protected if the state parameter does not leak to the attacker.

In the OIDC Core standard [Sak+14a], a so-called *login initiation endpoint* is described

which allows a third party to start a login flow by redirecting a user to this endpoint, passing the identity of an OP in the request. The RP will then start a login flow at the given OP. Members of the OIDC foundation confirmed to us that this endpoint is essentially an intentional CSRF protection bypass. We therefore recommend that login initiation endpoints should not be implemented (they are not a mandatory feature), or the endpoints should require explicit confirmation by the user.

4.5. Formal Analysis of OpenID Connect

In the following, we present our security analysis of OIDC, including a formal model of OIDC, the specifications of central security properties, our theorem which establishes the security of OIDC in our model, and a sketch of the proof (see the appendix for details).

The formal model of OIDC is based on the Web Infrastructure Model and is derived by closely following the OIDC standards Core, Discovery, and Dynamic Client Registration [Sak+14a; Sak+14b; SBJ14]. We formalize the main security properties for OIDC (authentication, authorization, session integrity for authentication and authorization) and secondary security properties. They capture important aspects of the security of OIDC, for example, regarding the outcome of the dynamic client registration. Finally, we state and prove our main theorem.

4.5.1. Model

Our model of OIDC comprises, as mentioned above, the OIDC Core standard [Sak+14a] plus the Discovery [Sak+14b] and Dynamic Client Registration [SBJ14] extensions.

More specifically, our model includes all features of OIDC that are commonly found in real-world implementations, for example, all three flows (implicit, authorization code, and hybrid flow), a detailed model of the Discovery (including the WebFinger protocol) and Dynamic Registration phases, including dynamic exchange of signing keys, and all relevant endpoints. RPs, IdPs (OPs), and browsers can be corrupted by the adversary dynamically.

We do not model detailed user claims (information about the user that can be retrieved from OPs). In our model, users have identities, but no other properties. We also do not model logout, self-issued OIDC providers (defined by [Sak+14a] as "personal, self-hosted OPs that issue self-signed ID Tokens"), and Authentication Class Reference (ACR) and Authentication Methods Reference (AMR) values that can be used to indicate the level of trust in a user authentication.[7]

Since the Web Infrastructure Model has no notion of time, we do not model expiry dates, for example, for the ID token and instead overapproximate by assuming that these tokens do not expire.

As in the OAuth model, we have two versions of our OIDC model, one with a network attacker and one with an unbounded number of web attackers (see Section 3.5.2 for the rationale).

[7] For example, these values would indicate a higher level of trust for a two-factor user authentication than for a password-based authentication.

OIDC Web System with a Network Attacker

An *OIDC web system with a network attacker* ($\mathcal{OIDCWS}^n$) consists of a network attacker, a finite set of web browsers, a finite set of web servers for the RPs, and a finite set of web servers for the OPs. All non-attacker parties are initially honest, but can become corrupted dynamically upon receiving a special message and then behave just like a web attacker process.

Web Servers: Our models for OPs and RPs follow the OIDC standard closely and include the mitigations discussed in Section 4.4.

An RP waits for users to start a login flow and then nondeterministically decides which flow to use. If needed, it then starts the discovery and dynamic registration phase of the protocol, and finally redirects the user to the OP for user authentication. Afterwards, it processes the received tokens. It then uses these tokens according to their type (e.g., using an access token, the RP would retrieve an id token from the OP). If an id token is received that passes all checks, the user will be logged in. Just as in our model for OAuth, RPs manage two kinds of sessions: The login sessions, which are used only during the user login phase, and service sessions.

The OP provides several endpoints according to its role in the login process. This includes the endpoints needed for the discovery and registration phases, which, in real-world deployments, may reside on different servers. For example, the OP provides its own OIDC configuration at the path `/.wk/openid-configuration`, and receives authentication and token requests.

Scripts: Three scripts can be sent from honest OPs and RPs to web browsers.

- The script *script_rp_index* is sent by an RP when the user visits web site of RP. It starts the login process.
- The script *script_rp_get_fragment* is sent by an RP during an implicit or hybrid flow to retrieve the data from the URI fragment. It extracts the access token, the authorization code, and the state from the fragment part of its own URI and sends this information in the body of a POST request back to the RP.
- Finally, OP uses the script *script_op_form* for user authentication.

OIDC Web System with Web Attackers

In addition to $\mathcal{OIDCWS}^n$, we also consider a class of web systems where the network attacker is replaced by an unbounded finite set of web attackers and a DNS server is introduced. We denote such a system by $\mathcal{OIDCWS}^w$ and call it an *OIDC web system with web attackers.*

4.5.2. Main Security Properties

Our primary security properties capture authentication, authorization and session integrity for authentication and authorization. We present these security properties in detail in the following. Supporting definitions can be found in the appendix.

Authentication Property

The most important property for OIDC is the authentication property. In short, it captures that a network attacker (and therefore also web attackers) should be unable to log in as an honest user at an honest RP using an honest OP.

Before we define the authentication property in more detail, recall that in our modeling, an RP uses two kinds of sessions: login sessions, which are only used for the login flow, and service sessions, which are used after a user/browser was logged in (see Section 3.4.5 for details). When a login session has finished successfully (i.e., the RP received a valid id token), the RP uses a fresh nonce as the service session id, stores this id in the session data of the login session, and sends the service session id as a cookie to the browser. In the same step, the RP also stores the issuer (say, d) that was used in the login flow and the identity (email address) of the user (say, id) as a pair $\langle d, id\rangle$, to be used as the global user identifier.

Now, our authentication property defines that a network attacker should be unable to get hold of a service session id by which the attacker would be considered to be logged in at an honest RP under an identity governed by an honest OP for an honest user/browser.

In order to define the authentication property formally, we first need to define the precise notion of a service session. In the following, as introduced in Section 2.7, (S, E, N) denotes a configuration in the run ρ with its components S, a mapping from processes to states of these processes, E, a set of events in the network that are waiting to be delivered to some party, and N, a set of nonces that have not been used yet. By $\mathsf{governor}(id)$ we denote the OP that is responsible for a given user identity (email address) id, and by $\mathsf{dom}(\mathsf{governor}(id))$, we denote the set of domains that are owned by this OP. By $S(r).\texttt{sessions}[lsid]$ we denote a data structure in the state of r that contains information about the login session identified by $lsid$. This data structure contains, for example, the identity for which the login session with the id $lsid$ was started and the service session id that was issued after the login session.

We can now define that there is a service session identified by a nonce n for an identity id at some RP r iff there exists a login session (identified by some nonce $lsid$) such that n is the service session associated with this login session, and r has stored that the service session is logged in for the id id using an issuer d (which is some domain of the governor of id).

Definition 1 (Service Sessions). We say that there is a *service session identified by a nonce n for an identity id at some RP r* in a configuration (S, E, N) of a run ρ of an OIDC web system iff there exists some login session id $lsid$ and a domain $d \in \mathsf{dom}(\mathsf{governor}(id))$ such that $S(r).\texttt{sessions}[lsid][\texttt{loggedInAs}] \equiv \langle d, id\rangle$ and $S(r).\texttt{sessions}[lsid][\texttt{serviceSessionId}] \equiv n$.

By $d_\emptyset(S(\mathsf{attacker}))$ we denote all terms that can be computed (more formally, derived in the usual Dolev-Yao style [DY83]) from the attacker's knowledge in the state S. We can now define that an OIDC web system with a network attacker is secure w.r.t. authentication iff the attacker can never get hold of a service session id (n) that was issued by an honest RP r for an identity id of an honest user (browser) at some honest OP (governor of id).

Definition 2 (Authentication Property). Let $\mathcal{OIDCWS}^n$ be an OIDC web system with a network attacker. We say that $\mathcal{OIDCWS}^n$ *is secure w.r.t. authentication* iff for every run ρ of $\mathcal{OIDCWS}^n$, every configuration (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}$ owned by b with $\mathsf{governor}(id)$ being an honest OP, every service session identified by some nonce n for id at r, we have that n is not derivable from the attackers knowledge in S (i.e., $n \notin d_\emptyset(S(\mathsf{attacker}))$).

Authorization Property

Intuitively, authorization for OIDC means that a network attacker should not be able to obtain or use a protected resource available to some honest RP at an OP for some user unless certain parties involved in the authorization process are corrupted. As the access control for such protected resources relies only on access tokens, we require that an attacker does not learn access tokens that would allow him to gain unauthorized access to these resources.

To define the authorization property formally, we need to reason about the state of an honest OP, say i. In this state, i creates *records* which contain information about successful authentications of users at i. Such records are stored in $S(i).\texttt{records}$ (with S as above). One such record, say x, contains the authenticated user's identity in $x[\texttt{subject}]$, two[8] access tokens in $x[\texttt{access_tokens}]$, and the client id of the RP in $x[\texttt{client_id}]$.

We can now define the authorization property. It defines that an OIDC web system with a network attacker is secure w.r.t. authorization iff the attacker cannot get hold of an access token that is stored in one of i's records for an identity of an honest user/browser b and an honest RP r.

Definition 3 (Authorization Property). Let $\mathcal{OIDCWS}^n$ be an OIDC web system with a network attacker. We say that $\mathcal{OIDCWS}^n$ *is secure w.r.t. authorization* iff for every run ρ of $\mathcal{OIDCWS}^n$, every configuration (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every $i \in \mathsf{OP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}^i$ owned by b, every nonce n, every term $x \in^{\langle\rangle} S(i).\texttt{records}$ with $x[\texttt{subject}] \equiv id$, $n \in^{\langle\rangle} x[\texttt{access_tokens}]$, and the client id $x[\texttt{client_id}]$ having been issued by i to r, we have that n is not derivable from the attackers knowledge in S (i.e., $n \notin d_\emptyset(S(\mathsf{attacker}))$).

Session Integrity for Authentication

The two session integrity properties capture that an attacker should be unable to forcefully log a user/browser in to some RP. This includes attacks such as CSRF and session swapping. As mentioned above, we define these properties over $\mathcal{OIDCWS}^w$.

For session integrity for authentication we say that a user/browser that is logged in at some RP must have expressed her wish to be logged in to that RP in the beginning of the login flow. It is important to note that not even a malicious OP should be able to forcefully log in its users

[8]In the hybrid mode, OPs can issue two access tokens, cf. Section 4.3.3.

(more precisely, its user's browsers) at an honest RP. If the OP is honest, then the user must additionally have authenticated herself at the OP with the same user account that RP uses for her identification. This excludes, for example, cases where (1) the user is forcefully logged in to an RP by an attacker that plays the role of an OP, and (2) where an attacker can force an honest user to be logged in at some RP under a false identity issued by an honest OP.

In our formal definition of session integrity for authentication (below), $\mathsf{loggedIn}^{Q}_{\rho}(b, r, u, i, \mathit{lsid})$ denotes that in the processing step Q (see below), the browser b was authenticated (logged in) to an RP r using the OP i and the identity u in an RP login session with the session id lsid. (Here, the processing step Q corresponds to Step 18 in Figure 4.3.) The user authentication in the processing step Q is characterized by the browser b receiving the service session id cookie that results from the login session lsid.

By $\mathsf{started}^{Q'}_{\rho}(b, r, \mathit{lsid})$ we denote that the browser b, in the processing step Q' triggered the script *script_rp_index* to start a login session which has the session id lsid at the RP r. (Compare Section 2.10.10 on how browsers handle scripts.) Here, Q' corresponds to Step 1 in Figure 4.3.

By $\mathsf{authenticated}^{Q''}_{\rho}(b, r, u, i, \mathit{lsid})$ we denote that in the processing step Q'', the user/browser b authenticated to the OP i. In this case, authentication means that the user filled out the login form (in *script_op_form*) at the OP i and, by this, consented to be logged in at r (as in Step 13 in Figure 4.3).

Using these notations, we can now define security w.r.t. session integrity for authentication of an OIDC web system with web attackers in a straightforward way from our informal definition above:

Definition 4 (Session Integrity for Authentication). Let $\mathcal{OIDCWS}^w$ be an OIDC web system with web attackers. We say that $\mathcal{OIDCWS}^w$ *is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{OIDCWS}^w$, every processing step Q in ρ with $Q = (S, E, N) \to (S', E', N')$ (for some S, S', E, E', N, N'), every browser b that is honest in S, every $i \in \mathsf{OP}$, every identity u, every $r \in \mathsf{RP}$ that is honest in S, every nonce lsid, with $\mathsf{loggedIn}^{Q}_{\rho}(b, r, u, i, \mathit{lsid})$, we have that (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b, r, \mathit{lsid})$, and (2) if i is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}^{Q''}_{\rho}(b, r, u, i, \mathit{lsid})$.

Session Integrity for Authorization

For session integrity for authorization we say that if an RP uses some access token at some OP in a session with a user, then that user expressed her wish to authorize the RP to interact with *some* OP. One cannot guarantee that the OP with which RP interacts is the one the user authorized the RP to interact with. This is because the OP might be malicious. In this case, for example in the discovery phase, the malicious OP might just claim (in Step 3 in Figure 4.1) that some other OP is responsible for the authentication of the user. If, however, the OP the user is logged in with is honest, then it should be guaranteed that the user authenticated to

that OP and that the OP the RP interacts with on behalf of the user is the one intended by the user.

For the formal definition, we use two additional predicates: $\mathsf{usedAuthorization}^Q_\rho(b, r, i, lsid)$ means that the RP r, in a login session (session id $lsid$) with the browser b used some access token to access services at the OP i. By $\mathsf{actsOnUsersBehalf}^Q_\rho(b, r, u, i, lsid)$ we denote that the RP r not only used *some* access token, but used one that is bound to the user's identity at the OP i.

Again, starting from our informal definition above, we define security w.r.t. session integrity for authorization of an OIDC web system with web attackers in a straightforward way (and similarly to session integrity for authentication):

Definition 5 (Session Integrity for Authorization). Let $\mathcal{OIDCWS}^w$ be an OIDC web system with web attackers. We say that $\mathcal{OIDCWS}^w$ *is secure w.r.t. session integrity for authorization* iff for every run ρ of $\mathcal{OIDCWS}^w$, every processing step Q in ρ with $Q = (S, E, N) \to (S', E', N')$ (for some S, S', E, E', N, N'), every browser b that is honest in S, every $i \in \mathsf{OP}$, every identity u, every $r \in \mathsf{RP}$ that is honest in S, every nonce $lsid$, we have that (1) if $\mathsf{usedAuthorization}^Q_\rho(b, r, i, lsid)$, then there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_\rho(b, r, lsid)$, and (2) if i is honest in S and $\mathsf{actsOnUsersBehalf}^Q_\rho(b, r, u, i, lsid)$, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}^{Q''}_\rho(b, r, u, i, lsid)$.

4.5.3. Secondary Security Properties

We define the following secondary security properties that capture specific aspects of OIDC. We use these secondary security properties during our proof of the above main security properties. Nonetheless, these secondary security properties are important and interesting in their own right, since the security of OIDC depends on them.

In the first lemma, we capture that if a relying party requests the issuer identifier from an identity provider (cf. Steps 2–3 in Figure 4.1), then the RP will only receive an origin that belongs to this OP in the response. In other words, honest OPs do not use attacker-controlled domains as issuer identifiers, and the attacker is unable to alter this information on the way to the RP. The RP stores the mapping from email addresses to issuer identifiers in the so-called *issuer cache*.

Lemma 1 (Integrity of Issuer Cache). For any run ρ of an OIDC web system $\mathcal{OIDCWS}^n$ with a network attacker or an OIDC web system $\mathcal{OIDCWS}^w$ with web attackers, every configuration (S, E, N) in ρ, every OP i that is honest in S, every identity $id \in \mathsf{ID}$ with $\mathsf{governor}(id) = i$, every relying party r that is honest in S, we have that $S(r).\mathtt{issuerCache}[id] \equiv \langle\rangle$ (not set) or $S(r).\mathtt{issuerCache}[id] \in \mathsf{dom}(i)$.

In a similar way, the next lemma captures that (1) honest OPs only use endpoints under their control in their OIDC configuration document (cf. Steps 4–5 in Figure 4.1) and that (2) this

information cannot be altered by an attacker. RPs store OIDC configurations in their *OIDC configuration cache* as a mapping from issuer identifiers to OIDC configuration documents.

Lemma 2 (Integrity of OIDC Configuration Cache). For any run ρ of an OIDC web system with a network attacker $\mathcal{OIDCWS}^n$ or an OIDC web system with web attackers $\mathcal{OIDCWS}^w$, every configuration (S, E, N) in ρ, every OP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every relying party r that is honest in S, we have that $S(r).\mathtt{oidcConfigCache}[d] \equiv \langle\rangle$ (not set) or $S(r).\mathtt{oidcConfigCache}[d] \equiv [\mathtt{issuer}: d, \mathtt{auth_ep}: u_1, \mathtt{token_ep}: u_2, \mathtt{jwks_ep}: u_3, \mathtt{reg_ep}: u_4]$ with u_l, $l \in \{1, 2, 3, 4\}$, being URLs, $u_l.\mathtt{host} \in \mathsf{dom}(i)$, and $u_l.\mathtt{protocol} \equiv \mathsf{S}$.

Again similarly to the above, the third lemma captures that RPs receive only "correct" signing keys from honest OPs, i.e., keys that belong to the respective OP (cf. Steps 6–7 in Figure 4.1). RPs store public signing keys of OPs in their *JWKS cache* (again as a mapping from issuer identifiers to keys).

Lemma 3 (Integrity of JWKS Cache). For any run ρ of an OIDC web system with a network attacker $\mathcal{OIDCWS}^n$ or an OIDC web system with web attackers $\mathcal{OIDCWS}^w$, every configuration (S, E, N) in ρ, every OP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every relying party r that is honest in S, we have that $S(r).\mathtt{jwksCache}[d] \equiv \langle\rangle$ (not set) or $S(r).\mathtt{jwksCache}[d] \equiv \mathsf{pub}(S(i).\mathtt{jwks})$.

The following lemma captures that honest RPs register only redirection URIs that point to themselves and that these URIs always use HTTPS. Recall that when an RP registers at an OP, the OP issues a freshly chosen client id to the RP and then stores RP's redirection URIs. In a state $S(i)$ of an OP, for a given client id c the list of redirection URIs is stored in $S(i).\mathtt{clients}[c][\mathtt{redirect_uris}]$.

Lemma 4 (Integrity of Client Registration). For any run ρ of an OIDC web system with a network attacker $\mathcal{OIDCWS}^n$ or an OIDC web system with web attackers $\mathcal{OIDCWS}^w$, every configuration (S, E, N) in ρ, every OP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every relying party r that is honest in S, every client id c that has been issued to r by i, every URL $u \in^{\langle\rangle} S(i).\mathtt{clients}[c][\mathtt{redirect_uris}]$, we have that $u.\mathtt{host} \in \mathsf{dom}(r)$ and $u.\mathtt{protocol} \equiv \mathsf{S}$.

The following lemma formalizes an important, yet expected property of $\mathcal{OIDCWS}^n$: Attackers cannot learn user's passwords. More precisely, we define that $\mathsf{secretOfID}(id)$, which denotes the password for a given identity id, is not known to any party except for the browser b owning the id and the identity provider i governing the id (as long as b and i are honest).

Lemma 5 (Third parties do not learn passwords). For any run ρ of an OIDC web system with a network attacker $\mathcal{OIDCWS}^n$ or an OIDC web system with web attackers $\mathcal{OIDCWS}^w$, every configuration (S, E, N) in ρ, every OP i that is honest in S, every identity $id \in \mathsf{ID}$ with $\mathsf{governor}(id) = i$, every browser b with $b = \mathsf{ownerOfID}(id)$ that is honest in S, every $p \in \mathcal{W} \setminus \{b, i\}$, we have that $\mathsf{secretOfID}(id) \notin d_\emptyset(S^l(p))$.

The following lemma states that attackers also cannot learn ID tokens that were issued by honest OPs for honest RPs and identities of honest browsers.

Lemma 6 (Attacker does not Learn ID Tokens). For any run ρ of an OIDC web system with a network attacker $\mathcal{OIDCWS}^n$ or an OIDC web system with web attackers $\mathcal{OIDCWS}^w$, every configuration (S, E, N) in ρ, every OP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every identity $id \in \mathsf{ID}$ with $\mathsf{governor}(id) = i$ and with $b = \mathsf{ownerOfId}(id)$ being an honest browser (in S), every relying party r that is honest in S, every client id c that has been issued to r by i, every term y, every id token $t = \mathsf{sig}([\mathtt{iss}: d, \mathtt{sub}: id, \mathtt{aud}: c, \mathtt{nonce}: y], k)$ with $k = S(i).\mathtt{jwks}$, every attacker process a, we have that $t \notin d_\emptyset(S(a))$.

Finally, the following lemma shows that if an honest browser logs in at an honest RP using an honest OP, then the attacker cannot learn the state value used in this login flow.

Lemma 7 (Third parties do not learn state). There exists no run ρ of an OIDC web system with web attackers $\mathcal{OIDCWS}^w$, no configuration (S, E, N) of ρ, no $r \in \mathsf{RP}$ that is honest in S, no $i \in \mathsf{OP}$ that is honest in S, no browser b that is honest in S, no nonce $lsid \in \mathcal{N}$, no domain $h \in \mathsf{dom}(r)$ of r, no terms g, x, y, $z \in \mathcal{T}_{\mathcal{N}}$, no cookie $c := \langle \mathtt{sessionId}, \langle lsid, x, y, z\rangle\rangle$, no atomic Dolev-Yao process $p \in \mathcal{W} \setminus \{b, i, r\}$ such that (1) $S(r).\mathtt{sessions}[lsid] \equiv g$, (2) $g[\mathtt{state}] \in d_\emptyset(S(p))$, (3) $S(r).\mathtt{issuerCache}[g[\mathtt{identity}]] \in \mathsf{dom}(i)$, and (4) $c \in^{\langle\rangle} S(b).\mathtt{cookies}[h]$.

4.5.4. The OpenID Connect Security Theorem

The following theorem states that OIDC is secure w.r.t. authentication and authorization in presence of the network attacker, and that OIDC is secure w.r.t. session integrity for authentication and authorization in presence of web attackers.

Theorem 2 (Security of OpenID Connect). Let $\mathcal{OIDCWS}^n$ be an OIDC web system with a network attacker. Then, $\mathcal{OIDCWS}^n$ is secure w.r.t. authentication and authorization. Let $\mathcal{OIDCWS}^w$ be an OIDC web system with web attackers. Then, $\mathcal{OIDCWS}^w$ is secure w.r.t. session integrity for authentication and authorization.

4.5.5. Proof of the OpenID Connect Security Theorem: Outline

Here, we only give a very rough overview of our proof. For full details we refer to the appendix.

For authentication and authorization, we first show that Lemmas 1–6 hold true. (The proofs for Lemmas 1–4 build upon each other.) We then assume that the authentication/authorization properties do not hold, i.e., that there is a run ρ of $\mathcal{OIDCWS}^n$ that does not satisfy authentication or authorization, respectively. Using Lemmas 1–6, it then only requires a few steps to lead the respective assumption to a contradication and thereby show that $\mathcal{OIDCWS}^n$ enjoys authentication/authorization.

For the session integrity properties, we follow a similar scheme. Since the state value is essential for session integrity, we first show Lemma 7, which essentially says that a web attacker is unable

to get hold of the state value that is used in a session between an honest browser b, an honest RP r, and an honest OP i. We then show session integrity for authentication/authorization by starting from the latest “known” processing steps in the respective flows (e.g., for authentication, $\mathsf{loggedIn}_\rho^Q(b, r, u, i, lsid)$) and tracking through the OIDC flows to show the existence of the earlier processing steps (e.g., $\mathsf{started}_\rho^{Q'}(b, r, lsid)$) and their respective properties.

4.5.6. Discussion of Results

We were able to prove, for the first time, the security of OpenID Connect. Our analysis shows that the fixes we devised for OAuth are also effective and sufficient to protect OpenID Connect. The secondary security properties show in particular that the Discovery and Dynamic Registration extensions work “as expected” and do not introduce new security vulnerabilities. This is an important result for the security of OpenID Connect.

5. Impact

Numerous services rely on OAuth and OpenID Connect to protect their users' data, for example, when authorizing critical financial transactions (PayPal [Pay], OpenID Financial API [SSN18]). Many of these services were potentially affected by the attacks we found, and it was therefore important to disclose the attacks responsibly.

After finding the attacks using our formal model, we first verified all attacks on existing, real-world implementations of OAuth and OpenID Connect. We then disclosed the attacks, most importantly the AS Mix-Up attack, to the OAuth and OpenID Connect standardization bodies. We are currently working in the OAuth Working Group on creating a new internet standard containing recommendations on mitigations against the attacks.

5.1. Verification

We verified the AS Mix-Up and 307 Redirect attacks on the Apache web server module *mod_auth_openidc*, an implementation of an OpenID Connect (and therefore also OAuth) client which is maintained by a member of the IETF OAuth Working Group. We also verified the AS Mix-Up Attack on the Python implementation *pyoidc*. We verified the State Leak Attack on a version of the Facebook PHP SDK and the Naïve Client Session Integrity and Across-AS State Reuse attacks on `nytimes.com`.[1]

5.2. Disclosure

In November 2015, we reported the attacks to the IETF OAuth Working Group and the OpenID Connect Working Group at the OpenID Foundation that confirmed the attacks. Simultaneously, we also notified the affected implementors of OAuth and OpenID Connect listed above. The OAuth Working Group invited us to discuss our findings and potential countermeasures at an emergency meeting in the following month. In January 2016, the OAuth Working Group published a security advisory on its mailing list [Tsc16] warning users of the AS Mix-Up Attack and the variant in [Mla+16].

[1] We found that *mod_auth_openidc* and `nytimes.com` are not susceptible to the State Leak attack since the user is immediately redirected to another web page at the same client after the login/authorization.

5.3. Follow-Up

During the emergency meeting, we proposed a workshop series to provide a forum for the exchange and cooperation between researchers, industry experts and standardization groups in the area of OAuth security. The idea was well received and so we hosted the first OAuth Security Workshop (OSW) at our university in July 2016. Since then, the workshop has been held annually and continues to foster the exchange in the areas of web authentication, authorization, and privacy. We continue to present and discuss our work at this workshop series.

We also joined the IETF OAuth Working Group to develop a new RFC codifying updated security recommendations for OAuth and OpenID Connect based on our findings [Lod+18]. This RFC contains advice on all attacks presented in this work and discusses potential mitigations. It is designed as a Best Current Practice (BCP) document that, in contrast to regular RFCs like [RFC6819], can be updated in the future.

6. Conclusion and Future Work

In this thesis, we have presented two important contributions to the area of web security: First, the most detailed and comprehensive formal model for the web infrastructure to date, and second, a detailed and precise security analysis of the widely used SSO standards OAuth and OpenID Connect, revealing previously unknown attacks.

Our Web Infrastructure Model captures many important features of the web with an unrivaled level of precision. The model is suitable for the analysis of a very wide range of web applications and standards, and with the WebRTC and WebSockets extensions, it supports even more web APIs than in the original publications. Nonetheless, some web features are not currently represented in the model and, since the web is developing at a high pace, there will always be room for elaborating on the model. For example, it could be extended to support complex Cross-Origin Resource Sharing requests, or upcoming technologies such as HTTP Token Binding [Pop+18] or the Web Authentication API [Bal+18].

As we have seen in our case studies, our analyses capitalize on the high level of detail of our pencil-and-paper model. As discussed earlier, achieving a similar level of detail in tools for automated proofs can be challenging. A mechanization of the web model, however, is certainly desirable: encoding the web model in a programming language and using tools to assist in creating and verifying proofs could enable an easier reuse of existing proofs and models, and it could help make formal methods more accessible to developers of applications and protocols. Such an approach could also facilitate the (automated) translation of real-world-applications into accurate models and vice-versa.

Using the Web Infrastructure Model as the foundation, we carried out the first extensive formal security analyses of the OAuth 2.0 and OpenID Connect standards. The detailed models created to this end comprise all grant types (flows) of OAuth and OIDC and take into account a large range of available options, the OpenID Connect Discovery and Dynamic Registration extensions, as well as corrupted browsers, clients, and OAPs/OPs.

Demonstrating the usefulness of our manual approach, our in-depth analyses revealed five previously unknown attacks on OAuth and OpenID Connect. The attacks break authorization, authentication, and session integrity in realistic settings. We verified all attacks on real-world implementations, proposed fixes, and joined the OAuth Working Group to codify the mitigations into official standards.

We showed that both, OAuth and OpenID Connect, can in fact be operated securely: With our mitigations in place and under assumptions that can be met in real-world deployments, we proved authentication, authorization and session integrity properties for both standards. Our

assumptions are documented in detail and can serve as guidelines for the secure implementation and usage of OAuth and OIDC. The fact that OAuth and OIDC are two of the most widely deployed authorization and authentication systems in the web and the basis for other protocols makes our findings particularly relevant.

Our definitions and proofs for OAuth and OIDC are excellent starting points for the thorough security analysis of technologies related to OAuth such as *PKCE*, *OAuth Token Binding*, and *OAuth MTLS*. PKCE (Proof Key for Code Exchange, [RFC7636]) was developed to protect OAuth even if an attacker can observe the authorization code, as possible on mobile operating systems. OAuth Token Binding [Jon+] limits access tokens, authorization codes, and refresh tokens to specific TLS connections. MTLS (mutual TLS, [Cam+18]) can be used for client authentication and to bind access tokens to client certificates. All of these techniques can and should be evaluated in our model.

In this work we have demonstrated that challenging security problems in the web can be tackled using formal models and rigorous proofs. With our accurate model and the case studies, we have laid a solid foundation for a more formal treatment of web security. Our hope is that our work will lead to formal methods being used more frequently and at an earlier stage in the development of web protocols, standards, and applications, making the web more secure.

A. The Web Infrastructure Model

In this appendix, we present the Web Infrastructure Model as proposed in [FKS14] and extended in [FKS15a; FKS15b; FKS16; FKS17]. The model also includes the WebRTC and WebSocket extensions presented in Section 2.

A.1. Communication Model

We start with details and definitions on the basic communication model.

A.1.1. Terms, Messages and Events

The signature Σ for the terms and messages considered in this work is the union of the following pairwise disjoint sets of function symbols:

- constants $C = \mathsf{IPs} \cup \mathbb{S} \cup \{\top, \bot, \diamond\}$ where the three sets are pairwise disjoint, $\mathbb{S}$ is interpreted to be the set of ASCII strings (including the empty string ε), and IPs is interpreted to be a set of (IP) addresses,
- function symbols for public keys, (a)symmetric encryption/decryption, and signatures: $\mathsf{pub}(\cdot)$, $\mathsf{enc_a}(\cdot,\cdot)$, $\mathsf{dec_a}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$, $\mathsf{dec_s}(\cdot,\cdot)$, $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{checksig}(\cdot,\cdot,\cdot)$, and $\mathsf{extractmsg}(\cdot)$,
- n-ary sequences $\langle\rangle, \langle\cdot\rangle, \langle\cdot,\cdot\rangle, \langle\cdot,\cdot,\cdot\rangle$, etc., and
- projection symbols $\pi_i(\cdot)$ for all $i \in \mathbb{N}$.

For strings (elements in $\mathbb{S}$), we use a specific font. For example, $\mathtt{HTTPReq}$ and $\mathtt{HTTPResp}$ are strings. We denote by $\mathsf{Doms} \subseteq \mathbb{S}$ the set of domains, e.g., $\mathtt{example.com} \in \mathsf{Doms}$. We denote by $\mathsf{Methods} \subseteq \mathbb{S}$ the set of methods used in HTTP requests, e.g., $\mathtt{GET}$, $\mathtt{POST} \in \mathsf{Methods}$.

Definition 6 (Nonces and Terms). By $X = \{x_1, x_2, \dots\}$ we denote a set of variables and by $\mathcal{N}$ we denote an infinite set of constants (*nonces*) such that Σ, X, and $\mathcal{N}$ are pairwise disjoint. For $N \subseteq \mathcal{N}$, we define the set $\mathcal{T}_N(X)$ of *terms* over $\Sigma \cup N \cup X$ inductively: (1) If $t \in N \cup X$, then t is a term. (2) If $f \in \Sigma$ is an n-ary function symbol in Σ for some $n \geq 0$ and $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term.

The equational theory associated with the signature Σ is given in Figure A.1. By $\equiv$ we denote the congruence relation on $\mathcal{T}_{\mathcal{N}}(X)$ induced by this theory. For example, we have that $\pi_1(\mathsf{dec_a}(\mathsf{enc_a}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{pub}(k)), k)) \equiv \mathsf{a}$.

$$
\begin{aligned}
\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) &= x \\
\mathsf{dec_s}(\mathsf{enc_s}(x, y), y) &= x \\
\mathsf{checksig}(\mathsf{sig}(x, y), \mathsf{pub}(y)) &= \top \\
\mathsf{extractmsg}(\mathsf{sig}(x, y)) &= x \\
\pi_i(\langle x_1, \dots, x_n \rangle) &= x_i \ \text{ if } 1 \le i \le n \\
\pi_j(\langle x_1, \dots, x_n \rangle) &= \diamond \ \text{ if } j \notin \{1, \dots, n\}
\end{aligned}
$$

Figure A.1. Equational theory for Σ.

Definition 7 (Ground Terms, Messages, Placeholders, Protomessages). By $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$, we denote the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*. The set $\mathcal{M}$ of messages (over $\mathcal{N}$) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$.

We define the set $V_{\text{process}} = \{\nu_1, \nu_2, \dots\}$ of variables (called placeholders) used by processes (see below). The set $\mathcal{M}^\nu := \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$ is called the set of *protomessages*, i.e., messages that can contain placeholders.

Example 1. For example, $k \in \mathcal{N}$ and $\mathsf{pub}(k)$ are messages, where k typically models a private key and $\mathsf{pub}(k)$ the corresponding public key. For constants a, b, c and the nonce $k \in \mathcal{N}$, the message $\mathsf{enc_a}(\langle a, b, c \rangle, \mathsf{pub}(k))$ is interpreted to be the sequence of constants a, b, c encrypted by the public key $\mathsf{pub}(k)$.

Definition 8 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \dots, x_n\})$, and $t_1, \dots, t_n \in \mathcal{T}_N$. By $\tau[t_1/x_1, \dots, t_n/x_n]$ we denote the (ground) term obtained from τ by replacing all occurrences of x_i in τ by t_i, for all $i \in \{1, \dots, n\}$.

Definition 9 (Events and Protoevents). An *event over* IPs *and* $\mathcal{M}$ is a term of the form $\langle a, f, m \rangle$, for a, $f \in \mathsf{IPs}$ and $m \in \mathcal{M}$, where a is interpreted to be the receiver address and f is the sender address. We denote by $\mathcal{E}$ the set of all events. Events over IPs and $\mathcal{M}^\nu$ are called *protoevents* and are denoted $\mathcal{E}^\nu$. By $2^{\mathcal{E}\langle\rangle}$ (or $2^{\mathcal{E}^\nu\langle\rangle}$, respectively) we denote the set of all sequences of (proto)events, including the empty sequence (e.g., $\langle\rangle$, $\langle\langle a, f, m\rangle, \langle a', f', m'\rangle, \dots\rangle$, etc.).

A.1.2. Notations

Definition 10 (Normal Form). Let t be a term. The *normal form* of t is acquired by reducing the function symbols from left to right as far as possible using the equational theory shown in Figure A.1. For a term t, we denote its normal form as $t{\downarrow}$.

Definition 11 (Pattern Matching). Let *pattern* $\in \mathcal{T}_{\mathcal{N}}(\{*\})$ be a term containing the wildcard (variable $*$). We say that a term t *matches pattern* iff t can be acquired from *pattern* by replacing each occurrence of the wildcard with an arbitrary term (which may be different for each instance

$$[k_1{:}v_1, \ldots, k_n{:}v_n]\,[k_i] = v_i$$
$$[k_1{:}v_1, \ldots, k_n{:}v_n] - k_i = [k_1{:}v_1, \ldots, k_{i-1}{:}v_{i-1}, k_{i+1}{:}v_{i+1} \ldots, k_n{:}v_n]$$

Figure A.2. Dictionary operators with $1 \leq i \leq n$.

of the wildcard). We write $t \sim \mathit{pattern}$. For a sequence of patterns *patterns* we write $t \mathrel{\dot\sim} \mathit{patterns}$ to denote that t matches at least one pattern in *patterns*.

For a term t' we write $t'|\,\mathit{pattern}$ to denote the term that is acquired from t' by removing all immediate subterms of t' that do not match *pattern*.

Example 2. For example, for a pattern $p = \langle \top, * \rangle$ we have that $\langle \top, 42 \rangle \sim p$, $\langle \bot, 42 \rangle \not\sim p$, and

$$\langle \langle \bot, \top \rangle, \langle \top, 23 \rangle, \langle \mathtt{a}, \mathtt{b} \rangle, \langle \top, \bot \rangle \rangle |\, p = \langle \langle \top, 23 \rangle, \langle \top, \bot \rangle \rangle \ .$$

Definition 12 (Sequence Notations). For a sequence $t = \langle t_1, \ldots, t_n \rangle$ and a set s we use $t \subset^{\langle\rangle} s$ to say that $t_1, \ldots, t_n \in s$. We define $x \in^{\langle\rangle} t \iff \exists i : t_i = x$. For a term y we write $t +^{\langle\rangle} y$ to denote the sequence $\langle t_1, \ldots, t_n, y \rangle$. For a sequence $r = \langle r_1, \ldots, r_m \rangle$ we write $t \cup r$ to denote the sequence $\langle t_1, \ldots, t_n, r_1, \ldots, r_m \rangle$. For a finite set M with $M = \{m_1, \ldots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \ldots, m_n \rangle$. (The order of the elements does not matter; one is chosen arbitrarily.)

Definition 13. A *dictionary over X and Y* is a term of the form

$$\langle \langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle \rangle$$

where $k_1, \ldots, k_n \in X$, $v_1, \ldots, v_n \in Y$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \ldots, n\}$, an *element* of the dictionary with key k_i and value v_i. We often write $[k_1{:}v_1, \ldots, k_i{:}v_i, \ldots, k_n{:}v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle \rangle$. We denote the set of all dictionaries over X and Y by $[X \times Y]$.

We note that the empty dictionary is equivalent to the empty sequence, i.e., $[] = \langle\rangle$. Figure A.2 shows the short notation for dictionary operations. For a dictionary $z = [k_1{:}v_1, k_2{:}v_2, \ldots, k_n{:}v_n]$ we write $k \in z$ to say that there exists i such that $k = k_i$. We write $z[k_j]$ to refer to the value v_j. If a dictionary contains two elements $\langle k, v \rangle$ and $\langle k, v' \rangle$, then the notations and operations for dictionaries apply nondeterministically to one of both elements. If $k \notin z$, we set $z[k] := \langle\rangle$.

Given a term $t = \langle t_1, \ldots, t_n \rangle$, we can refer to any subterm using a sequence of integers. The subterm is determined by repeated application of the projection π_i for the integers i in the sequence. We call such a sequence a *pointer*:

Definition 14. A *pointer* is a sequence of non-negative integers. We write $\tau.\overline{p}$ for the application of the pointer $\overline{p}$ to the term τ. This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.

Example 3. For the term $\tau = \langle a, b, \langle c, d, \langle e, f\rangle\rangle\rangle$ and the pointer $\overline{p} = \langle 3, 1\rangle$, the subterm of τ at the position $\overline{p}$ is $c = \pi_1(\pi_3(\tau))$. Also, $\tau.3.\langle 3, 1\rangle = \tau.3.\overline{p} = \tau.3.3.1 = e$.

To improve readability, we try to avoid writing, e.g., $o.2$ or $\pi_2(o)$ in this document. Instead, we will use the names of the components of a sequence that is of a defined form as pointers that point to the corresponding subterms. E.g., if an *Origin* term is defined as $\langle \mathit{host}, \mathit{protocol}\rangle$ and o is an Origin term, then we can write $o.\mathtt{protocol}$ instead of $\pi_2(o)$ or $o.2$. See also Example 4.

A.1.3. Atomic Processes, Systems and Runs

An atomic process takes its current state and an event as input, and then (nondeterministically) outputs a new state and a set of events.

Definition 15 (Generic Atomic Processes and Systems). A *(generic) atomic process* is a tuple

$$p = (I^p, Z^p, R^p, s_0^p)$$

where $I^p \subseteq \mathsf{IPs}$, $Z^p \subseteq \mathcal{T}_{\mathcal{N}}$ is a set of states, $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{E}^{\nu}\langle\rangle} \times \mathcal{T}_{\mathcal{N}}(V_{\text{process}}))$ (input event and old state map to sequence of output events and new state), and $s_0^p \in Z^p$ is the initial state of p. For any new state s and any sequence of nonces $(\eta_1, \eta_2, \dots)$ we demand that $s[\eta_1/\nu_1, \eta_2/\nu_2, \dots] \in Z^p$. A *system* $\mathcal{P}$ is a (possibly infinite) set of atomic processes.

We use configurations to capture the state of a system:

Definition 16 (Configurations). A *configuration of a system* $\mathcal{P}$ is a tuple (S, E, N) where the state of the system S maps every atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$, the sequence of waiting events E is an infinite sequence[1] $(e_1, e_2, \dots)$ of events waiting to be delivered, and N is an infinite sequence of nonces $(n_1, n_2, \dots)$.

Definition 17 (Concatenating terms and sequences). For a term $a = \langle a_1, \dots, a_i\rangle$ and a sequence $b = (b_1, b_2, \dots)$, we define the *concatenation* as $a \cdot b := (a_1, \dots, a_i, b_1, b_2, \dots)$.

Definition 18 (Subtracting from Sequences). For a sequence X and a set or sequence Y we define $X \setminus Y$ to be the sequence X where for each element in Y, a nondeterministically chosen occurence of that element in X is removed.

A system can transition from one configuration into another in a so-called processing step. Intuitively, for a processing step, we select one of the processes in $\mathcal{P}$, and call it with one of the events in the list of waiting events E. In its output (new state and output events), we replace any occurrences of placeholders ν_x by "fresh" nonces from N (which we then remove from N). The output events are then prepended to the list of waiting events, and the state of the process is reflected in the new configuration.

[1] Here: Not in the sense of terms as defined earlier.

Definition 19 (Processing Steps). A *processing step of the system $\mathcal{P}$* is of the form

$$(S, E, N) \xrightarrow[p \to E_{\text{out}}]{e_{\text{in}} \to p} (S', E', N')$$

where

- (S, E, N) and (S', E', N') are configurations of $\mathcal{P}$,
- $e_{\text{in}} = \langle a, f, m \rangle \in E$ is an event,
- $p \in \mathcal{P}$ is a process, and
- E_{out} is a sequence (term) of events

such that there exists

- a sequence (term) $E^{\nu}_{\text{out}} \subseteq 2^{\mathcal{E}^{\nu}\langle\rangle}$ of protoevents,
- a term $s^{\nu} \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
- a sequence $(v_1, v_2, \ldots, v_i)$ of all placeholders appearing in E^{ν}_{out} or s^{ν}, and
- a sequence $N^{\nu} = (\eta_1, \eta_2, \ldots, \eta_i)$ of the first i elements in N

with

- $((e_{\text{in}}, S(p)), (E^{\nu}_{\text{out}}, s^{\nu})) \in R^p$ and $a \in I^p$,
- $E_{\text{out}} = E^{\nu}_{\text{out}}[\eta_1/v_1, \ldots, \eta_i/v_i]$,
- $S'(p) = s^{\nu}[\eta_1/v_1, \ldots, \eta_i/v_i]$ and $S'(p') = S(p')$ for all $p' \neq p$,
- $E' = E_{\text{out}} \cdot (E \setminus \{e_{\text{in}}\})$, and
- $N' = N \setminus N^{\nu}$.

We may omit the superscript and/or subscript of the arrow.

A sequence of configurations linked by processing steps is called a run:

Definition 20 (Runs). Let $\mathcal{P}$ be a system, E^0 be sequence of events, and N^0 be a sequence of nonces. A *run ρ of a system $\mathcal{P}$ initiated by E^0 with nonces N^0* is a finite sequence of configurations $((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \ldots)$ such that $S^0(p) = s^p_0$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ for all $0 \leq i < n$ (finite run) or for all $i \geq 0$ (infinite run).

We denote the state $S^n(p)$ of a process p at the end of a run ρ by $\rho(p)$.

Usually, we will initiate runs with a set E^0 containing infinite trigger events of the form $\langle a, a, \texttt{TRIGGER} \rangle$ for each $a \in \mathsf{IPs}$, interleaved by address.

A.1.4. Atomic Dolev-Yao Processes

We next define atomic Dolev-Yao processes, for which we require that the messages and states that they output can be computed (more formally, derived) from the current input event and state. For this purpose, we first define what it means to derive a message from given messages.

Definition 21 (Deriving Terms). Let M be a set of ground terms. We say that *a term m can be derived from M with placeholders V* if there exist $n \geq 0$, $m_1, \dots, m_n \in M$, and $\tau \in \mathcal{T}_\emptyset(\{x_1, \dots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \dots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V.

For example, we have that $a \in d_\emptyset(\{\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k)), k\})$. We can now define atomic Dolev-Yao processes based on generic atomic processes. We also define an atomic attacker process based on the atomic Dolev-Yao process:

Definition 22 (Atomic Dolev-Yao Process). An *atomic Dolev-Yao process (or simply, a DY process)* is a tuple $p = (I^p, Z^p, R^p, s_0^p)$ such that (I^p, Z^p, R^p, s_0^p) is a generic atomic process and for all events $e \in \mathcal{E}$, sequences of protoevents E, $s \in \mathcal{T}_\mathcal{N}$, $s' \in \mathcal{T}_\mathcal{N}(V_\text{process})$, with $((e, s), (E, s')) \in R^p$ it holds true that E, $s' \in d_{V_\text{process}}(\{e, s\})$.

Definition 23 (Atomic Attacker Process). An *(atomic) attacker process for a set of sender addresses $A \subseteq \mathsf{IPs}$* is an atomic DY process $p = (I, Z, R, s_0)$ such that for all events e, and $s \in \mathcal{T}_\mathcal{N}$ we have that $((e, s), (E, s')) \in R$ iff $s' = \langle e, E, s\rangle$ and $E = \langle\langle a_1, f_1, m_1\rangle, \dots, \langle a_n, f_n, m_n\rangle\rangle$ with $n \in \mathbb{N}$, $a_1, \dots, a_n \in \mathsf{IPs}$, $f_1, \dots, f_n \in A$, $m_1, \dots, m_n \in d_{V_\text{process}}(\{e, s\})$.

Algorithms

We often define a relation R^p of an atomic Dolev-Yao process using a nondeterministic algorithm A. Besides the usual pseudocode syntax and the notations introduced before, we use the following notations in algorithms:

- The notation **let** $n \leftarrow N$ is used to describe that n is chosen nondeterministically from the set N.
- We write **for each** $s \in M$ **do** to denote that the following commands are repeated for every element in M, where the variable s is the current element. The order in which the elements are processed is chosen nondeterministically.
- We write, for example,

  let x, y **such that** $\langle \texttt{Constant}, x, y\rangle \equiv t$ **if possible; otherwise** doSomethingElse

  for some variables x, y, a string `Constant`, and some term t to express that $x := \pi_2(t)$, and $y := \pi_3(t)$ if $\texttt{Constant} \equiv \pi_1(t)$ and if $|\langle \texttt{Constant}, x, y\rangle| = |t|$, and that otherwise x and y are not set and doSomethingElse is executed.

– We write **stop** x to denote that the algorithm stops with the output x. We omit x to denote that there is no output.

We say that an algorithm A defines a relation R^p of an atomic Dolev-Yao process in the following sense: The pair

$$\big((\langle\langle a, f, m\rangle\rangle, s)\,, (M, s')\big)$$

belongs to R^p iff A (or any of the functions called therein), when given $(\langle a, f, m\rangle, s)$ as input, terminates with the command **stop** M, s', i.e., with output M and s'.

A.2. Scripts

We define scripts, which model client-side scripting technologies, such as JavaScript. Scripts are defined analog to DY processes.

Definition 24 (Placeholders for Scripts). By $V_{\text{script}} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts.

Definition 25 (Scripts). A *script* is a relation $R \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ such that for all $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ with $(s, s') \in R$ it follows that $s' \in d_{V_{\text{script}}}(s)$.

A script is called by the browser which provides it with state information s, such as the script's last state and limited information about the browser's state. The script then outputs a term s', which represents the new internal state and some command which is interpreted by the browser. The term s' may contain variables $\lambda_1, \dots$ which the browser will replace by otherwise unused placeholders $\nu_1, \dots$ which will be replaced by nonces once the browser DY process finishes. This provides the script with a way to get "fresh" nonces.

We also define the *attacker script* R^{att}:

Definition 26 (Attacker Script). The attacker script R^{att} outputs everything that is derivable from the input, i.e., $R^{\text{att}} = \{(s, s') \mid s \in \mathcal{T}_{\mathcal{N}}, s' \in d_{V_{\text{script}}}(s)\}$.

Like atomic Dolev-Yao processes, scripts are also usually defined using algorithms. We then say that an algorithm A defines a script in the following sense: The pair (in, out) belongs to the script iff A (or any of the functions called therein), when given in as input, terminates with the command **stop** out.

A.3. Web System

The web infrastructure and web applications are formalized by what is called a web system. A web system contains, among others, a (possibly infinite) set of DY processes, modeling web browsers, web servers, DNS servers, and attackers (which may corrupt other entities, such as browsers).

Definition 27. A *web system* $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ is a tuple with its components defined as follows:

- The first component, $\mathcal{W}$, denotes a system (a set of DY processes) and is partitioned into the sets Hon, Web, and Net of honest, web attacker, and network attacker processes, respectively.

  Every $p \in \mathsf{Web} \cup \mathsf{Net}$ is an attacker process for some set of sender addresses $A \subseteq \mathsf{IPs}$. For a web attacker $p \in \mathsf{Web}$, we require its set of addresses I^p to be disjoint from the set of addresses of all other web attackers and honest processes, i.e., $I^p \cap I^{p'} = \emptyset$ for all $p' \in \mathsf{Hon} \cup \mathsf{Web}$. Hence, a web attacker cannot listen to traffic intended for other processes. Also, we require that $A = I^p$, i.e., a web attacker can only use sender addresses it owns. Conversely, a network attacker may listen to all addresses (i.e., no restrictions on I^p) and may spoof all addresses (i.e., the set A may be IPs).

  Every $p \in \mathsf{Hon}$ is a DY process which models either a *web server*, a *web browser*, or a *DNS server*, as further described in the following subsections. Just as for web attackers, we require that p does not spoof sender addresses and that its set of addresses I^p is disjoint from those of other honest processes and the web attackers.

- The second component, $\mathcal{S}$, is a finite set of scripts such that $R^{\text{att}} \in \mathcal{S}$.

- The third component, script, is an injective mapping from $\mathcal{S}$ to $\mathbb{S}$, i.e., by script every $s \in \mathcal{S}$ is assigned its string representation $\mathsf{script}(s)$.

- Finally, E^0 is an (infinite) sequence of events, containing an infinite number of events of the form $\langle a, a, \mathtt{TRIGGER} \rangle$ for every $a \in \bigcup_{p \in \mathcal{W}} I^p$.

A *run of* $\mathcal{WS}$ is a run of $\mathcal{W}$ initiated by E^0.

A.4. Message and Data Formats

We now provide some more details about data and message formats that are needed for the formal treatment of the web model.

A.4.1. URLs

Definition 28. A *URL* is a term of the form

$$\langle \mathtt{URL}, \mathit{protocol}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{fragment} \rangle$$

with the subterms $\mathit{protocol} \in \{\mathtt{P}, \mathtt{S}\}$ (for **p**lain HTTP and **s**ecure HTTPS), $\mathit{host} \in \mathsf{Doms}$, $\mathit{path} \in \mathbb{S}$, $\mathit{parameters} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, and $\mathit{fragment} \in \mathcal{T}_{\mathcal{N}}$. The set of all valid URLs is URLs.

The *fragment* part of a URL can be omitted when writing the URL. Its value is then defined to be $\bot$.

Example 4. For the URL $u = \langle \texttt{URL}, a, b, c, d\rangle$, we have that $u.\texttt{protocol} = a$. If, in the algorithm described later, we write $u.\texttt{path} := e$ then $u = \langle \texttt{URL}, a, b, c, e\rangle$ afterwards.

A.4.2. Origins

Definition 29. An *origin* is a term of the form $\langle host, protocol\rangle$ with $host \in \mathsf{Doms}$ and $protocol \in \{\texttt{P}, \texttt{S}\}$. We write $\mathsf{Origins}$ for the set of all origins.

Example 5. For example, $\langle \texttt{FOO}, \texttt{S}\rangle$ is the HTTPS origin for the domain FOO, while $\langle \texttt{BAR}, \texttt{P}\rangle$ is the HTTP origin for the domain BAR.

A.4.3. Cookies

Definition 30. A *cookie* is a term of the form $\langle name, content\rangle$ where $name \in \mathcal{T}_{\mathcal{N}}$, and *content* is a term of the form $\langle value, secure, session, httpOnly\rangle$ where $value \in \mathcal{T}_{\mathcal{N}}$, $secure, session, httpOnly \in \{\top, \bot\}$. We write $\mathsf{Cookies}$ for the set of all cookies and $\mathsf{Cookies}^\nu$ for the set of all cookies where names and values are defined over $\mathcal{T}_{\mathcal{N}}(V)$.

If the *secure* attribute of a cookie is set, the browser will not transfer this cookie over unencrypted HTTP connections. If the *session* flag is set, this cookie will be deleted as soon as the browser is closed. The *httpOnly* attribute controls whether JavaScript has access to this cookie.

Cookies of the form described here are contained in HTTP(S) requests and in the browser state. In HTTP(S) responses, only the components *name* and *value* are transferred as a pairing of the form $\langle name, value\rangle$.

A.4.4. HTTP Messages

Definition 31. An *HTTP request* is a term of the form shown in (A.1). An *HTTP response* is a term of the form shown in (A.2).

$$\langle \texttt{HTTPReq}, nonce, method, host, path, parameters, headers, body\rangle \tag{A.1}$$

$$\langle \texttt{HTTPResp}, nonce, status, headers, body\rangle \tag{A.2}$$

The components are defined as follows:

- $nonce \in \mathcal{N}$ serves to map each response to the corresponding request,
- $method \in \mathsf{Methods}$ is one of the HTTP methods,
- $host \in \mathsf{Doms}$ is the host name in the Host header of HTTP/1.1,

- $path \in \mathbb{S}$ is a string indicating the requested resource at the server side,
- $status \in \mathbb{S}$ is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard),
- $parameters \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$ contains URL parameters,
- $headers \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$ contains request/response headers, and
- $body \in \mathcal{T}_{\mathcal{N}}$ represents the request/response body.

We write $\mathsf{HTTPRequests}/\mathsf{HTTPResponses}$ for the set of all HTTP requests or responses, respectively.

The following request/response headers are created or understood by our browser model presented later:

- $\langle \mathtt{Origin}, o\rangle$ where o is an origin or $\diamond$ (the "null" origin),
- $\langle \mathtt{Set\text{-}Cookie}, c\rangle$ where c is a sequence of cookies,
- $\langle \mathtt{Cookie}, c\rangle$ where $c \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$ (in this header, only names and values of cookies are transferred),
- $\langle \mathtt{Location}, l\rangle$ where $l \in \mathsf{URLs}$,
- $\langle \mathtt{Referer}, r\rangle$ where $r \in \mathsf{URLs}$,
- $\langle \mathtt{Strict\text{-}Transport\text{-}Security}, \top\rangle$,
- $\langle \mathtt{Authorization}, \langle username, password\rangle\rangle$ where $username, password \in \mathbb{S}$,
- $\langle \mathtt{ReferrerPolicy}, p\rangle$ where $p \in \{\mathtt{noreferrer}, \mathtt{origin}\}$, and
- $\langle \mathtt{Upgrade}, \mathtt{websocket}\rangle$.

Example 6 (HTTP Request and Response).

$$\begin{aligned} r :=& \langle \mathtt{HTTPReq}, n_1, \mathtt{POST}, \mathtt{example.com}, \mathtt{/show}, \langle\langle \mathtt{index}, \mathtt{1}\rangle\rangle, \\ & [\mathtt{Origin} : \langle \mathtt{example.com}, \mathtt{S}\rangle], \langle \mathtt{foo}, \mathtt{bar}\rangle\rangle \end{aligned} \tag{A.3}$$

$$s := \langle \mathtt{HTTPResp}, n_1, \mathtt{200}, \langle\langle \mathtt{Set\text{-}Cookie}, \langle\langle \mathtt{SID}, \langle n_2, \bot, \bot, \top\rangle\rangle\rangle\rangle\rangle, \langle \mathtt{somescript}, x\rangle\rangle \tag{A.4}$$

An HTTP `POST` request for the URL `http://example.com/show?index=1` is shown in (A.3), with an `Origin` header and a body that contains $\langle \mathtt{foo}, \mathtt{bar}\rangle$. A possible response is shown in (A.4), which contains an httpOnly cookie with name `SID` and value n_2 as well as the string representation `somescript` of the script $\mathsf{script}^{-1}(\mathtt{somescript})$ (which should be an element of $\mathcal{S}$) and its initial state x.

Encrypted HTTP Messages

For HTTPS, requests are encrypted using the public key of the server. Such a request contains an (ephemeral) symmetric key chosen by the client that issued the request. The server is supported to encrypt the response using the symmetric key.

Definition 32. An *encrypted HTTP request* is of the form $\mathsf{enc_a}(\langle m, k' \rangle, k)$, where $k \in \mathit{terms}$, $k' \in \mathcal{N}$, and $m \in \mathsf{HTTPRequests}$. The corresponding *encrypted HTTP response* would be of the form $\mathsf{enc_s}(m', k')$, where $m' \in \mathsf{HTTPResponses}$. We call the sets of all encrypted HTTP requests and responses $\mathsf{HTTPSRequests}$ or $\mathsf{HTTPSResponses}$, respectively.

We say that an HTTP(S) response matches or corresponds to an HTTP(S) request if both terms contain the same nonce.

Example 7.

$$\mathsf{enc_a}(\langle r, k' \rangle, \mathsf{pub}(k_{\text{example.com}})) \tag{A.5}$$

$$\mathsf{enc_s}(s, k') \tag{A.6}$$

The term (A.5) shows an encrypted request (with r as in (A.3)). It is encrypted using the public key $\mathsf{pub}(k_{\text{example.com}})$. The term (A.6) is a response (with s as in (A.4)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (A.5).

A.4.5. DNS Messages

Definition 33. A *DNS request* is a term of the form $\langle \mathtt{DNSResolve}, \mathit{domain}, \mathit{nonce} \rangle$ where $\mathit{domain} \in \mathsf{Doms}$, $\mathit{nonce} \in \mathcal{N}$. We call the set of all DNS requests $\mathsf{DNSRequests}$.

Definition 34. A *DNS response* is a term of the form $\langle \mathtt{DNSResolved}, \mathit{domain}, \mathit{result}, \mathit{nonce} \rangle$ with $\mathit{domain} \in \mathsf{Doms}$, $\mathit{result} \in \mathsf{IPs}$, $\mathit{nonce} \in \mathcal{N}$. We call the set of all DNS responses $\mathsf{DNSResponses}$.

DNS servers are supposed to include the nonce they received in a DNS request in the DNS response that they send back so that the party which issued the request can match it with the request.

A.4.6. WebSocket Messages

Definition 35. A *WebSocket Message* is a term of the form $\langle \mathtt{WS_MSG}, \mathit{nonce}, \mathit{data} \rangle$ where $\mathit{nonce} \in \mathcal{N}$ and $\mathit{data} \in \mathcal{T}_{\mathcal{N}}$.

Definition 36. An *Encrypted WebSocket Message* is a term of the form $\mathsf{enc_s}(m, k)$ where m is a WebSocket Message and k is a symmetric key (a nonce).

A.4.7. WebRTC Messages

Definition 37. A *WebRTC Offer* or *WebRTC Answer* is a term of the form

$$\langle \texttt{RTC_OFFER}, \mathit{nonce}, \mathit{idp}, \mathit{ia}, \mathit{pubkey}, \mathit{addr} \rangle$$

where nonce, $\mathit{pubkey} \in \mathcal{N}$, $\mathit{addr} \in \mathsf{IPs}$, and ia and $\mathit{data} \in \mathcal{T}_{\mathcal{N}}$. We use the identifier $\texttt{RTC_OFFER}$ for both offer and answer documents.

Definition 38. A *WebRTC Message* is a term of the form $\mathsf{enc_a}(\langle \texttt{RTC_MSG}, \mathit{nonce}, \mathit{data} \rangle, k,)$ where nonce, $k \in \mathcal{N}$ and $\mathit{data} \in \mathcal{T}_{\mathcal{N}}$.

A.5. DNS Server Model

We consider a flat DNS model in which DNS queries are answered directly by one DNS server and always with the same address for a domain.

Definition 39. A *DNS server* d (in a flat DNS model) is modeled as an atomic DY process $(I^d, \{s_0^d\}, R^d, s_0^d)$. It has a finite set of addresses I^d and its initial (and only) state s_0^d encodes a mapping from domain names to addresses of the form

$$s_0^d = \langle \langle \texttt{domain}_1, a_1 \rangle, \langle \texttt{domain}_2, a_2 \rangle, \ldots \rangle .$$

Algorithm 2.1 defines the relation R^d.

A.6. Web Browser Model

Following the description of the browser model in Section 2.10, we now present our formal definition of web browsers. This section is devided into four parts: First, we introduce notations to represent windows and documents in the browser state. Afterwards, we define the set of states of web browsers. We then present the relation defining the behavior of web browsers in our model. Finally, we put these parts together in the definition of web browser atomic processes.

A.6.1. Windows, Documents, and Related Notations

We represent windows and documents opened in a web browser by terms contained in the browser's state (see below). Each window/document in a web browser is represented by one specific term.

Definition 40 (Window). A *window* is a term of the form $w = \langle \mathit{nonce}, \mathit{documents}, \mathit{opener} \rangle$ with $\mathit{nonce} \in \mathcal{N}$, $\mathit{documents} \subset^{\langle\rangle} \mathsf{Documents}$ (defined below), $\mathit{opener} \in \mathcal{N} \cup \{\bot\}$ where $d.\texttt{active} = \top$ for exactly one $d \in^{\langle\rangle} \mathit{documents}$ if $\mathit{documents}$ is not empty (we then call d the *active document of w*).

We write $\mathsf{Windows}$ for the set of all windows. We write $w.\mathtt{activedocument}$ to denote the active document inside window w if it exists and $\langle\rangle$ else. We will refer to the window nonce as *(window) reference*.

A window a may have opened a top-level window b (i.e., a window term which is not a subterm of a document term). In this case, the *opener* part of the term b is the nonce of a, i.e., $b.\mathtt{opener} = a.\mathtt{nonce}$.

The documents contained in a window term to the left of the active document are the previously viewed documents (available to the user via the "back" button) and the documents in the window term to the right of the currently active document are documents available via the "forward" button.

Definition 41 (Document). A *document* d is a term of the form

$$\langle \mathit{nonce}, \mathit{location}, \mathit{headers}, \mathit{referrer}, \mathit{script}, \mathit{scriptstate}, \mathit{scriptinputs}, \mathit{subwindows}, \mathit{active}\rangle$$

where $\mathit{nonce} \in \mathcal{N}$, $\mathit{location} \in \mathsf{URLs}$, $\mathit{headers} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{referrer} \in \mathsf{URLs} \cup \{\bot\}$, $\mathit{script} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{scriptstate} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{scriptinputs} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{subwindows} \subset^{\langle\rangle} \mathsf{Windows}$, $\mathit{active} \in \{\top, \bot\}$. A *restricted document* is a term of the form $\langle \mathit{nonce}, \mathit{subwindows}\rangle$ with nonce, $\mathit{subwindows}$ as above. A window $w \in^{\langle\rangle} \mathit{subwindows}$ is called a *subwindow* (of d).

We write $\mathsf{Documents}$ for the set of all documents. For a document term d we write $d.\mathtt{origin}$ to denote the origin of the document, i.e., the term $\langle d.\mathtt{location}.\mathtt{host}, d.\mathtt{location}.\mathtt{protocol}\rangle \in \mathsf{Origins}$. We will refer to the document nonce as *(document) reference*.

Given two windows w and w' we write $w \xrightarrow{\text{childof}} w'$ if $w \in^{\langle\rangle} w'.\mathtt{activedocument}.\mathtt{subwindows}$. We write $\xrightarrow{\text{childof}^+}$ for the transitive closure.

Definition 42 (WebRTC Connection Record). A *WebRTC Connection Record* is a term of the form

$$\langle \mathit{docnonce}, \mathit{privkey}, \mathit{idp}, \mathit{ia}, \mathit{windownonce}, \mathit{remoteDescription}, \mathit{remoteAuthenticated}\rangle.$$

A.6.2. Web Browser States $Z_{\mathsf{webbrowser}}$

We now introduce the web browser state, which is the most complex term used in our model.

Recall from Section 2.10.5 that HTTP(S) connections are tracked using *references*. These are defined as a pairing of a an identifier for the type of the request and a unique part. The identifier can be *XHR* for XMLHttpRequests, *WS* for websocket requests, or *REQ* for normal HTTP(S) requests. The unique part is a term containing a nonce. In the browser state, we will introduce the subterms *pendingDNS*, *pendingRequests*, *wsConnections*. In these subterms, the references (or just their unique part) are used to identify individual entries.

Definition 43. The *set of states $Z_{webbrowser}$ of a web browser atomic Dolev-Yao process* consists of the terms of the form

$$\langle windows, ids, secrets, cookies, localStorage, sessionStorage, keyMapping, sts, DNSaddress, pendingDNS, pendingRequests, wsConnections, rtcConnections, isCorrupted\rangle$$

with the subterms as follows:

- $windows \subset^{\langle\rangle} \mathsf{Windows}$ contains a list of window terms (representing top-level windows, or browser tabs) which contain documents, which in turn can contain further window terms (iframes).
- $ids \subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ is a list of identities that are owned by this browser (i.e., belong to the user of the browser).
- $secrets \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ contains a list of secrets that are associated with certain origins (i.e., passwords of the user of the browser at certain websites).
- *cookies* is a dictionary over Doms and sequences of $\mathsf{Cookies}$ modelling cookies that are stored for specific domains.
- $localStorage \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ stores the data saved by scripts using the localStorage API (separated by origins).
- $sessionStorage \in \left[OR \times \mathcal{T}_{\mathcal{N}}\right]$ for $OR := \{\langle o, r\rangle \,|\, o \in \mathsf{Origins},\ r \in \mathcal{N}\}$ similar to localStorage, but the data in sessionStorage is additionally separated by top-level windows.
- $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ maps domains to TLS encryption keys.
- $sts \subset^{\langle\rangle} \mathsf{Doms}$ stores the list of domains that the browser only accesses via TLS (strict transport security).
- $DNSaddress \in \mathsf{IPs}$ defines the IP address of the DNS server.
- $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$ contains one pairing per unanswered DNS query of the form $\langle reference, request, url\rangle$. In these pairings, *reference* is an HTTP(S) request reference (as above), *request* contains the HTTP(S) message that awaits DNS resolution, and *url* contains the URL of said HTTP request. The pairings in *pendingDNS* are indexed by the DNS request/response nonce.
- $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ contains pairings of the form $\langle reference, request, url, key, f\rangle$ with *reference*, *request*, and *url* as in *pendingDNS*, *key* is the symmetric encryption key if HTTPS is used or $\perp$ otherwise, and f is the IP address of the server to which the request was sent.

- $wsConnections \in \mathcal{T}_{\mathcal{N}}$ is a list of pairings of the form $\langle reference, nonce, key, f \rangle$ where $nonce$ is a nonce that is used in WebSocket messages and the other terms are used analogously to above.

- $rtcConnections \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$ maps nonces to WebRTC Connection Records.

- $isCorrupted \in \{\bot, \texttt{FULLCORRUPT}, \texttt{LIMITEDCORRUPT}\}$ specifies the corruption level of the browser.

In corrupted browsers, certain subterms are used in different ways (e.g., *pendingRequests* is used to store all observed messages).

A.6.3. Web Browser Relation $R_{\text{webbrowser}}$

We now define the relation $R_{\text{webbrowser}}$ of a standard HTTP browser. To this end, we first introduce functions that are used for defining the browser main algorithm. We then define the browser relation.

Helper Functions

In the following description of the web browser relation $R_{\text{webbrowser}}$ we use the helper functions $\mathsf{Subwindows}$, Docs, Clean, $\mathsf{CookieMerge}$ and $\mathsf{AddCookie}$.

Given a browser state s, $\mathsf{Subwindows}(s)$ denotes the set of all pointers[2] to windows in the window list $s.\texttt{windows}$, their active documents, and (recursively) the subwindows of these documents. We exclude subwindows of inactive documents and their subwindows. With $\mathsf{Docs}(s)$ we denote the set of pointers to all active documents in the set of windows referenced by $\mathsf{Subwindows}(s)$.

Definition 44. For a browser state s we denote by $\mathsf{Subwindows}(s)$ the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in^{\langle\rangle} s.\texttt{windows}$ there is a $\overline{p} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p} = w$. (2) For all $\overline{p} \in \mathsf{Subwindows}(s)$, the active document d of the window $s.\overline{p}$ and every subwindow w of d there is a pointer $\overline{p'} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p'} = w$.

Given a browser state s, the set $\mathsf{Docs}(s)$ of pointers to active documents is the minimal set such that for every $\overline{p} \in \mathsf{Subwindows}(s)$, there is a pointer $\overline{p'} \in \mathsf{Docs}(s)$ with $s.\overline{p'} = s.\overline{p}.\texttt{activedocument}$.

By $\mathsf{Subwindows}^{+}(s)$ and $\mathsf{Docs}^{+}(s)$ we denote the respective sets that also include the inactive documents and their subwindows.

The function Clean will be used to determine which information about windows and documents the script running in the document d has access to.

[2] Recall the definition of a pointer in Definition 14.

Definition 45. Let s be a browser state and d a document. By $\mathsf{Clean}(s, d)$ we denote the term that equals $s.\mathtt{windows}$ but with (1) all inactive documents removed (including their subwindows etc.), (2) all subterms that represent non-same-origin documents w.r.t. d replaced by a restricted document d' with the same nonce and the same subwindow list, and (3) the values of the subterms $\mathtt{headers}$ for all documents set to $\langle\rangle$. (Non-same-origin documents on all levels are replaced by their corresponding restricted document.)

The function $\mathsf{CookieMerge}$ merges two sequences of cookies together: When used in the browser, *oldcookies* is the sequence of existing cookies for some origin, *newcookies* is a sequence of new cookies that was output by some script. The sequences are merged into a set of cookies using an algorithm that is based on the *Storage Mechanism* algorithm described in [RFC6265].

Definition 46. For a sequence of cookies (with pairwise different names) *oldcookies* and a sequence of cookies *newcookies*, the set $\mathsf{CookieMerge}(\mathit{oldcookies}, \mathit{newcookies})$ is defined by the following algorithm: From *newcookies* remove all cookies c that have $c.\mathtt{content}.\mathtt{httpOnly} \equiv \top$. For any $c, c' \in^{\langle\rangle} \mathit{newcookies}$, $c.\mathtt{name} \equiv c'.\mathtt{name}$, remove the cookie that appears left of the other in *newcookies*. Let m be the set of cookies that have a name that either appears in *oldcookies* or in *newcookies*, but not in both. For all pairs of cookies $(c_{\text{old}}, c_{\text{new}})$ with $c_{\text{old}} \in^{\langle\rangle} \mathit{oldcookies}$, $c_{\text{new}} \in^{\langle\rangle} \mathit{newcookies}$, $c_{\text{old}}.\mathtt{name} \equiv c_{\text{new}}.\mathtt{name}$, add c_{new} to m if $c_{\text{old}}.\mathtt{content}.\mathtt{httpOnly} \equiv \bot$ and add c_{old} to m otherwise. The result of $\mathsf{CookieMerge}(\mathit{oldcookies}, \mathit{newcookies})$ is m.

The function $\mathsf{AddCookie}$ adds a cookie c received in an HTTP response to the sequence of cookies contained in the sequence *oldcookies*. It is again based on the algorithm described in [RFC6265] but simplified for the use in the browser model.

Definition 47. For a sequence of cookies (with pairwise different names) *oldcookies* and a cookie c, the sequence $\mathsf{AddCookie}(\mathit{oldcookies}, c)$ is defined by the following algorithm: Let $m := \mathit{oldcookies}$. Remove any c' from m that has $c.\mathtt{name} \equiv c'.\mathtt{name}$. Append c to m and return m.

The function $\mathsf{NavigableWindows}$ returns a set of windows that a document is allowed to navigate. We closely follow [Ber+17], Section 5.1.4 for this definition.

Definition 48. The set $\mathsf{NavigableWindows}(\overline{w}, s')$ is the set $\overline{W} \subseteq \mathsf{Subwindows}(s')$ of pointers to windows that the active document in $\overline{w}$ is allowed to navigate. The set $\overline{W}$ is defined to be the minimal set such that for every $\overline{w'} \in \mathsf{Subwindows}(s')$ the following is true:

- If $s'.\overline{w'}.\mathtt{activedocument}.\mathtt{origin} \equiv s'.\overline{w}.\mathtt{activedocument}.\mathtt{origin}$ (i.e., the active documents in $\overline{w}$ and $\overline{w'}$ are same-origin), then $\overline{w'} \in \overline{W}$, and
- If $s'.\overline{w} \xrightarrow{\text{childof}^*} s'.\overline{w'} \wedge \nexists\, \overline{w''} \in \mathsf{Subwindows}(s')$ with $s'.\overline{w'} \xrightarrow{\text{childof}^*} s'.\overline{w''}$ ($\overline{w'}$ is a top-level window and $\overline{w}$ is an ancestor window of $\overline{w'}$), then $\overline{w'} \in \overline{W}$, and

– If $\exists \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w'} \xrightarrow{\text{childof}^+} s'.\overline{p}$ $\wedge\, s'.\overline{p}.\mathtt{activedocument.origin} = s'.\overline{w}.\mathtt{activedocument.origin}$ ($\overline{w'}$ is not a top-level window but there is an ancestor window $\overline{p}$ of $\overline{w'}$ with an active document that has the same origin as the active document in $\overline{w}$), then $\overline{w'} \in \overline{W}$, and

– If $\exists \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w'}.\mathtt{opener} = s'.\overline{p}.\mathtt{nonce} \wedge \overline{p} \in \overline{W}$ ($\overline{w'}$ is a top-level window—it has an opener—and $\overline{w}$ is allowed to navigate the opener window of $\overline{w'}$, $\overline{p}$), then $\overline{w'} \in \overline{W}$.

Functions

In the description of the following functions, we use a, f, m, and s as read-only global input variables. All other variables are local variables or arguments.

In several places throughout the algorithms we use placeholders to generate "fresh" nonces (Definition 6). Table A.1 shows a list of all placeholders used.

Placeholder	Usage
ν_1	Algorithm A.11, window nonces
ν_2	Algorithm A.11, HTTP request nonce
ν_3	Algorithm A.11, lookup key for pending HTTP requests entry
ν_4	Algorithm A.8, HTTP request nonce (multiple lines)
ν_5	Algorithm A.8, subwindow nonce
ν_6	Algorithm A.10, HTTP request nonce
ν_7	Algorithm A.10, document nonce
ν_8	Algorithm A.4, lookup key for pending DNS entry
ν_9	Algorithm A.1, window nonce
ν_{10}	Algorithm A.7, window nonce
ν_{11}	Algorithm A.7, HTTP request nonce
ν_{12}	Algorithm A.8, WebRTC connection nonce
$\nu_{13}, \ldots$	Algorithm A.8, replacement for placeholders in script output

Table A.1. List of placeholders used in browser algorithms.

The function GETNAVIGABLEWINDOW (Algorithm A.1) is called by the browser to determine the window that is *actually* navigated when a script in the window $s'.\overline{w}$ provides a window reference for navigation (e.g., for opening a link). When it is given a window reference (nonce) *window*, this function returns a pointer to a selected window term in s':

– If *window* is the string `_BLANK`, a new window is created and a pointer to that window is returned.

– If *window* is a nonce (reference) and there is a window term with a reference of that value in the windows in s', a pointer $\overline{w'}$ to that window term is returned, as long as the window is navigable by the current window's document (as defined by NavigableWindows above).

Algorithm A.1 Web Browser Model: Determine window for navigation.

```
 1: function GETNAVIGABLEWINDOW(w̄, window, noreferrer, s')
 2:     if window ≡ _BLANK then   → Open a new window when _BLANK is used
 3:         if noreferrer ≡ ⊤ then
 4:             let w' := ⟨ν9, ⟨⟩, ⊥⟩
 5:         else
 6:             let w' := ⟨ν9, ⟨⟩, s'.w̄.nonce⟩
 7:         let s'.windows := s'.windows +⟨⟩ w' and let w̄' be a pointer to this new element in s'
 8:         return w̄'
 9:     let w̄' ← NavigableWindows(w̄, s') such that s'.w̄'.nonce ≡ window if possible; otherwise
    return w̄
10:     return w̄'
```

In all other cases, $\overline{w}$ is returned instead (the script navigates its own window).

The function GETWINDOW (Algorithm A.2) takes a window reference as input and returns a pointer to a window as above, but it checks only that the active documents in both windows are same-origin. It creates no new windows.

Algorithm A.2 Web Browser Model: Determine same-origin window.

```
1: function GETWINDOW(w̄, window, s')
2:     let w̄' ← Subwindows(s') such that s'.w̄'.nonce ≡ window if possible; otherwise return w̄
3:     if s'.w̄'.activedocument.origin ≡ s'.w̄.activedocument.origin then
4:         return w̄'
5:     return w̄
```

The function CANCELNAV (Algorithm A.3) is used to stop any pending requests for a specific window. From the pending requests and pending DNS requests it removes any requests with the given window reference n.

Algorithm A.3 Web Browser Model: Cancel pending requests for given window.

```
1: function CANCELNAV(reference, s')
2:     remove all ⟨⟨REQ, reference⟩, req, key, f⟩ from s'.pendingRequests for any req, key, f
3:     remove all ⟨x, ⟨⟨REQ, reference⟩, message, url⟩⟩ from s'.pendingDNS
       ↪ for any x, message, url
4:     return s'
```

The function HTTP_SEND (Algorithm A.4) takes an HTTP request *message* as input, adds `Cookie` and `Origin` headers to the message, creates a DNS request for the hostname given in the request and stores the request in s'.`pendingDNS` until the DNS resolution finishes. For normal HTTP requests, *reference* is a window reference. For XMLHttpRequests, *reference* is a value of the form $\langle document, nonce\rangle$ where *document* is a document reference and *nonce* is some nonce that was chosen by the script that initiated the request. *url* contains the full URL of the request (this is mainly used to retrieve the protocol that should be used for this message, and to store the fragment identifier for use after the document was loaded). *origin* is the `Origin` header value that is to be added to the HTTP request.

Algorithm A.4 Web Browser Model: Prepare headers, do DNS resolution, save message.

```
1: function HTTP_SEND(reference, message, url, origin, referrer, referrerPolicy, s')
2:     if message.host ∈⟨⟩ s'.sts then
3:         let url.protocol := S
4:     let cookies := ⟨{⟨c.name, c.content.value⟩|c ∈⟨⟩ s'.cookies [message.host]
   ↪   ∧ (c.content.secure ⟹ (url.protocol = S))}⟩
5:     let message.headers[Cookie] := cookies
6:     if origin ≢ ⊥ then
7:         let message.headers[Origin] := origin
8:     if referrerPolicy ≡ noreferrer then
9:         let referrer := ⊥
10:    if referrer ≢ ⊥ then
11:        if referrerPolicy ≡ origin then   → Reduce Referer to origin.
12:            let referrer := ⟨URL, referrer.protocol, referrer.host, /, ⟨⟩, ⊥⟩
13:        let referrer.fragment := ⊥   → Browsers do not send fragment ids in the Referer header.
14:        let message.headers[Referer] := referrer
15:    let s'.pendingDNS[ν_8] := ⟨reference, message, url⟩
16:    stop ⟨⟨s'.DNSaddress, a, ⟨DNSResolve, message.host, ν_8⟩⟩⟩, s'
```

The functions NAVBACK (Algorithm A.5) and NAVFORWARD (Algorithm A.6), navigate a window forward or backward. More precisely, they deactivate one document and activate that document's succeeding document or preceding document, respectively. If no such successor/predecessor exists, the functions do not change the state.

Algorithm A.5 Web Browser Model: Navigate a window backward.

```
1: function NAVBACK(w̄', s')
2:     if ∃j̄ ∈ ℕ, j̄ > 1 such that s'.w̄'.documents.j̄.active ≡ ⊤ then
3:         let s'.w̄'.documents.j̄.active := ⊥
4:         let s'.w̄'.documents.(j̄ − 1).active := ⊤
5:         let s' := CANCELNAV(s'.w̄'.nonce, s')
```

Algorithm A.6 Web Browser Model: Navigate a window forward.

```
1: function NAVFORWARD(w̄', s')
2:     if ∃j̄ ∈ ℕ such that s'.w̄'.documents.j̄.active ≡ ⊤
           ↪ ∧ s'.w̄'.documents.(j̄ + 1) ∈ Documents then
3:         let s'.w̄'.documents.j̄.active := ⊥
4:         let s'.w̄'.documents.(j̄ + 1).active := ⊤
5:         let s' := CANCELNAV(s'.w̄'.nonce, s')
```

The function RTC_CHECK_REMOTE_IA (Algorithm A.7) starts the check of identity assertion presented by a remote server. To this end, the function opens an IdP proxy window. The script in the proxy window has access to the identity assertion via the command `RTC_GET_CHECK_IA_INFO` and can issue the command `RTC_CHECKED_IA` when it has successfully checked the IA.

Algorithm A.7 Web Browser Model: Check remote server's WebRTC identity assertion.

1: **function** RTC_CHECK_REMOTE_IA($nonce$, s')
2: **let** $idp := s'.\texttt{rtcConnections}[nonce].\texttt{remoteDescription.idp}$
3: **if** $idp \not\equiv \bot$ **then**
4: **let** $windownonce := \nu_{10}$
5: **let** $w' := \langle windownonce, \langle\rangle, \bot\rangle$
6: **let** $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
7: **let** $path := \texttt{.wk/idp-proxy}$
8: **let** $url := \langle \texttt{URL}, \texttt{S}, idp, path, \langle\rangle, \langle\rangle\rangle$
9: **let** $req := \langle \texttt{HTTPReq}, \nu_{11}, \texttt{GET}, idp, path, \langle\rangle, \langle\rangle, \langle\rangle\rangle$
10: **let** $s'.\texttt{rtcConnections}[nonce].\texttt{windownonce} := windownonce$
11: **call** HTTP_SEND($\langle \texttt{REQ}, windownonce\rangle$, req, url, $\bot$, $\bot$, $\bot$, s')

The function RUNSCRIPT (Algorithm A.8) performs a script execution step of the script in the document $s'.\overline{d}$ (which is part of the window $s'.\overline{w}$). A new script and document state is chosen according to the relation defined by the script and the new script and document state is saved. Afterwards, the *command* that the script issued is interpreted.

Algorithm A.8 Web Browser Model: Execute a script.

1: **function** RUNSCRIPT($\overline{w}$, $\overline{d}$, s')
2: **let** $tree := \mathsf{Clean}(s', s'.\overline{d})$
3: **let** $cookies := \langle\{\langle c.\texttt{name}, c.\texttt{content.value}\rangle | c \in^{\langle\rangle} s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$
↪ $\wedge\ c.\texttt{content.httpOnly} = \bot$
↪ $\wedge\ \left(c.\texttt{content.secure} \implies \left(s'.\overline{d}.\texttt{origin.protocol} \equiv \texttt{S}\right)\right)\}\rangle$
4: **let** $tlw \leftarrow s'.\texttt{windows}$ **such that** tlw is the top-level window containing $\overline{d}$
5: **let** $sessionStorage := s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right]$
6: **let** $localStorage := s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$
7: **let** $secrets := s'.\texttt{secrets}\left[s'.\overline{d}.\texttt{origin}\right]$
8: **let** $R \leftarrow \mathsf{script}^{-1}(s'.\overline{d}.\texttt{script})$
9: **let** $in := \langle tree, s'.\overline{d}.\texttt{nonce}, s'.\overline{d}.\texttt{scriptstate}, s'.\overline{d}.\texttt{scriptinputs}, cookies,$
↪ $localStorage, sessionStorage, s'.\texttt{ids}, secrets\rangle$
10: **let** $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$, $cookies' \leftarrow \mathsf{Cookies}^{\nu}$, $localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$, $sessionStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
↪ $command \leftarrow \mathcal{T}_{\mathcal{N}}(V)$, $out^{\lambda} := \langle state', cookies', localStorage', sessionStorage', command\rangle$
↪ **such that** $(in, out^{\lambda}) \in R$
11: **let** $out := out^{\lambda}[\nu_{13}/\lambda_1, \nu_{14}/\lambda_2, \dots]$
12: **let** $s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right] := \langle\mathsf{CookieMerge}(s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right], cookies')\rangle$
13: **let** $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right] := localStorage'$
14: **let** $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right] := sessionStorage'$
15: **let** $s'.\overline{d}.\texttt{scriptstate} := state'$
16: **let** $referrer := s'.\overline{d}.\texttt{location}$
17: **let** $referrerPolicy := s'.\overline{d}.\texttt{headers}[\texttt{ReferrerPolicy}]$
18: **let** $docorigin := s'.\overline{d}.\texttt{origin}$

19: **switch** *command* **do**
20:   **case** $\langle \mathtt{HREF}, url, hrefwindow, noreferrer\rangle$
21:     **let** $\overline{w}' := \mathsf{GETNAVIGABLEWINDOW}(\overline{w}, hrefwindow, noreferrer, s')$
22:     **let** $req := \langle \mathtt{HTTPReq}, \nu_4, \mathtt{GET}, url.\mathtt{host}, url.\mathtt{path}, url.\mathtt{parameters}, \langle\rangle, \langle\rangle\rangle$
23:     **if** $noreferrer \equiv \top$ **then**
24:       **let** $referrerPolicy := \mathtt{noreferrer}$
25:     **let** $s' := \mathsf{CANCELNAV}(s'.\overline{w}'.\mathtt{nonce}, s')$
26:     **call** $\mathsf{HTTP_SEND}(\langle \mathtt{REQ}, s'.\overline{w}'.\mathtt{nonce}\rangle, req, url, \bot, referrer, referrerPolicy, s')$
27:   **case** $\langle \mathtt{IFRAME}, url, window\rangle$
28:     **let** $\overline{w}' := \mathsf{GETWINDOW}(\overline{w}, window, s')$
29:     **let** $req := \langle \mathtt{HTTPReq}, \nu_4, \mathtt{GET}, url.\mathtt{host}, url.\mathtt{path}, url.\mathtt{parameters}, \langle\rangle, \langle\rangle\rangle$
30:     **let** $w' := \langle \nu_5, \langle\rangle, \bot\rangle$
31:     **let** $s'.\overline{w}'.\mathtt{activedocument.subwindows} := s'.\overline{w}'.\mathtt{activedocument.subwindows} +^{\langle\rangle} w'$
32:     **call** $\mathsf{HTTP_SEND}(\langle \mathtt{REQ}, \nu_5\rangle, req, url, \bot, referrer, referrerPolicy, s')$
33:   **case** $\langle \mathtt{FORM}, url, method, data, hrefwindow\rangle$
34:     **if** $method \notin \{\mathtt{GET}, \mathtt{POST}\}$ **then**
35:       **stop**
36:     **let** $\overline{w}' := \mathsf{GETNAVIGABLEWINDOW}(\overline{w}, hrefwindow, \bot, s')$
37:     **if** $method = \mathtt{GET}$ **then**
38:       **let** $body := \langle\rangle$
39:       **let** $parameters := data$
40:       **let** $origin := \bot$
41:     **else**
42:       **let** $body := data$
43:       **let** $parameters := url.\mathtt{parameters}$
44:       **let** $origin := docorigin$
45:     **let** $req := \langle \mathtt{HTTPReq}, \nu_4, method, url.\mathtt{host}, url.\mathtt{path}, parameters, \langle\rangle, body\rangle$
46:     **let** $s' := \mathsf{CANCELNAV}(s'.\overline{w}'.\mathtt{nonce}, s')$
47:     **call** $\mathsf{HTTP_SEND}(\langle \mathtt{REQ}, s'.\overline{w}'.\mathtt{nonce}\rangle, req, url, origin, referrer, referrerPolicy, s')$
48:   **case** $\langle \mathtt{SETSCRIPT}, window, script\rangle$
49:     **let** $\overline{w}' := \mathsf{GETWINDOW}(\overline{w}, window, s')$
50:     **let** $s'.\overline{w}'.\mathtt{activedocument.script} := script$
51:     **stop** $\langle\rangle, s'$
52:   **case** $\langle \mathtt{SETSCRIPTSTATE}, window, scriptstate\rangle$
53:     **let** $\overline{w}' := \mathsf{GETWINDOW}(\overline{w}, window, s')$
54:     **let** $s'.\overline{w}'.\mathtt{activedocument.scriptstate} := scriptstate$
55:     **stop** $\langle\rangle, s'$
56:   **case** $\langle \mathtt{XMLHTTPREQUEST}, url, method, data, xhrreference\rangle$
57:     **if** $method \in \{\mathtt{CONNECT}, \mathtt{TRACE}, \mathtt{TRACK}\} \wedge xhrreference \notin \{\mathcal{N}, \bot\}$ **then**
58:       **stop**
59:     **if** $url.\mathtt{host} \not\equiv docorigin.\mathtt{host} \vee url \not\equiv docorigin.\mathtt{protocol}$ **then**
60:       **stop**
61:     **if** $method \in \{\mathtt{GET}, \mathtt{HEAD}\}$ **then**
62:       **let** $data := \langle\rangle$
63:       **let** $origin := \bot$
64:     **else**
65:       **let** $origin := docorigin$
66:     **let** $req := \langle \mathtt{HTTPReq}, \nu_4, method, url.\mathtt{host}, url.\mathtt{path}, url.\mathtt{parameters}, \langle\rangle, data\rangle$
67:     **let** $reference := \langle \mathtt{XHR}, s'.\overline{d}.\mathtt{nonce}, xhrreference\rangle$
68:     **call** $\mathsf{HTTP_SEND}(reference, req, url, origin, referrer, referrerPolicy, s')$

69: **case** $\langle \texttt{BACK}, \mathit{window}\rangle$
70: **let** $\overline{w}' :=$ GETNAVIGABLEWINDOW($\overline{w}$, window, $\bot$, s')
71: NAVBACK($\overline{w'}$, s')
72: **stop** $\langle\rangle$, s'
73: **case** $\langle \texttt{FORWARD}, \mathit{window}\rangle$
74: **let** $\overline{w}' :=$ GETNAVIGABLEWINDOW($\overline{w}$, window, $\bot$, s')
75: NAVFORWARD($\overline{w'}$, s')
76: **stop** $\langle\rangle$, s'
77: **case** $\langle \texttt{CLOSE}, \mathit{window}\rangle$
78: **let** $\overline{w}' :=$ GETNAVIGABLEWINDOW($\overline{w}$, window, $\bot$, s')
79: **remove** $s'.\overline{w'}$ from the sequence containing it
80: **stop** $\langle\rangle$, s'
81: **case** $\langle \texttt{POSTMESSAGE}, \mathit{window}, \mathit{message}, \mathit{origin}\rangle$
82: **let** $\overline{w}' \leftarrow$ Subwindows(s') **such that** $s'.\overline{w}'.\texttt{nonce} \equiv \mathit{window}$
83: **if** $\exists \overline{j} \in \mathbb{N}$ **such that** $s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{active} \equiv \top \wedge$
↪ $(\mathit{origin} \not\equiv \bot \implies s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{origin} \equiv \mathit{origin})$ **then**
84: **let** $s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{scriptinputs} := s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{scriptinputs}$
↪ $+^{\langle\rangle} \langle \texttt{POSTMESSAGE}, s'.\overline{w}.\texttt{nonce}, \mathit{docorigin}, \mathit{message}\rangle$
85: **stop** $\langle\rangle$, s'
86: **case** $\langle \texttt{WS_OPEN}, \mathit{url}, \mathit{wsreference}\rangle$
87: **let** $\mathit{headers} := \langle\langle \texttt{Upgrade}, \texttt{websocket}\rangle\rangle$
88: **let** $\mathit{req} := \langle \texttt{HTTPReq}, \nu_4, \texttt{GET}, \mathit{url}.\texttt{host}, \mathit{url}.\texttt{path}, \mathit{url}.\texttt{parameters}, \mathit{headers}, \langle\rangle\rangle$
89: **let** $\mathit{reference} := \langle \texttt{WS}, s'.\overline{d}.\texttt{nonce}, \mathit{wsreference}\rangle$
90: **call** HTTP_SEND($\mathit{reference}$, req, url, $\mathit{docorigin}$, $\mathit{referrer}$, $\mathit{referrerPolicy}$, s')
91: **case** $\langle \texttt{WS_SEND}, \mathit{wsreference}, \mathit{data}\rangle$
92: **let** $\mathit{reference} := \langle \texttt{WS}, s'.\overline{d}.\texttt{nonce}, \mathit{wsreference}\rangle$
93: **let** $\mathit{nonce}, \mathit{key}, f$ **such that** $\langle \mathit{reference}, \mathit{nonce}, \mathit{key}, f\rangle \in^{\langle\rangle} s'.\texttt{wsConnections}$
↪ **if possible; otherwise stop**
94: **let** $\mathit{msg} := \langle \texttt{WS_MSG}, \mathit{nonce}, \mathit{data}\rangle$
95: **if** $\mathit{key} \not\equiv \bot$ **then**
96: **let** $\mathit{msg} := \mathsf{enc}_\mathsf{s}(\mathit{msg}, \mathit{key})$
97: **stop** $\langle\langle f, a, \mathit{msg}\rangle\rangle$, s'
98: **case** $\langle \texttt{RTC_CREATE_PEERCONNECTION}, \mathit{idp}, \mathit{privkey}\rangle$ → *idp* contains only the domain of the IdP.
99: **let** $s'.\overline{d}.\texttt{scriptinputs} := s'.\overline{d}.\texttt{scriptinputs} +^{\langle\rangle} \langle \texttt{RTC_PEERCONNECTION}, \nu_{12}\rangle$
100: **if** $\mathit{idp} \not\equiv \bot$ **then** → If IdP is used, create new window for IdP proxy script.
101: **let** $\mathit{windownonce} := \nu_5$
102: **let** $w' := \langle \mathit{windownonce}, \langle\rangle, \bot\rangle$
103: **let** $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
104: **let** $\mathit{path} :=$ .wk/idp-proxy
105: **let** $\mathit{url} := \langle \texttt{URL}, \texttt{S}, \mathit{idp}, \mathit{path}, \langle\rangle, \langle\rangle\rangle$ → Not implemented: subprotocol.
106: **let** $\mathit{req} := \langle \texttt{HTTPReq}, \nu_4, \texttt{GET}, \mathit{idp}, \mathit{path}, \langle\rangle, \langle\rangle, \langle\rangle\rangle$
107: **else**
108: **let** $\mathit{windownonce} := \nu_5$
109: **let** $\mathit{record} := \langle s'.\overline{d}.\texttt{nonce}, \mathit{privkey}, \mathit{idp}, \bot, \mathit{windownonce}, \bot, \bot, \bot, \bot, \bot\rangle$
110: **let** $s'.\texttt{rtcConnections}[\nu_{12}] := \mathit{record}$ → Create/store new WebRTC Connection Record.
111: **if** $\mathit{idp} \not\equiv \bot$ **then** → If using IdP proxy, finish with new HTTP request.
112: **call** HTTP_SEND($\langle \texttt{REQ}, \mathit{windownonce}\rangle$, req, url, $\bot$, $\bot$, $\bot$, s')
113: **case** $\langle \texttt{RTC_GET_OFFER}, \mathit{nonce}\rangle$
114: **if** $s'.\texttt{rtcConnections}[\mathit{nonce}].\texttt{docnonce} \not\equiv s'.\overline{d}.\texttt{nonce}$ **then**
115: **stop**

116: let $conn$:= s'.rtcConnections[$nonce$]
117: let $offeranswer$:= ⟨RTC_OFFER, $nonce$, .idp, $conn$.ia, pub($conn$.privkey), a⟩
118: let $s'.\overline{d}$.scriptinputs := $s'.\overline{d}$.scriptinputs $+^{\langle\rangle}$ $offeranswer$
119: **case** ⟨RTC_SET_REMOTE, $nonce$, $offeranswer$⟩
120: **if** s'.rtcConnections[$nonce$].docnonce $\not\equiv$ $s'.\overline{d}$.nonce **then**
121: **stop**
122: **let** s'.rtcConnections[$nonce$].remoteDescription := $offeranswer$
123: **call** RTC_CHECK_REMOTE_IA($nonce$, s') → Check the remote's identity assertion.
124: **case** ⟨RTC_GET_IA_INFO⟩ → Called by IdP proxy to receive data to be signed.
125: **let** $nonce \leftarrow \mathcal{N}$ **such that** s'.rtcConnections[$nonce$].windownonce $\equiv$ $s'.\overline{w}$.nonce
↪ **if possible; otherwise stop**
126: **let** $iarequest$:= ⟨RTC_IA_REQUEST, $nonce$, pub(s'.rtcConnections[$nonce$].privkey)⟩
127: **let** $s'.\overline{d}$.scriptinputs := $s'.\overline{d}$.scriptinputs $+^{\langle\rangle}$ $iarequest$
128: **case** ⟨RTC_SET_IA, ia⟩ → Called by IdP proxy to commit the IA.
129: **let** $nonce \leftarrow \mathcal{N}$ **such that** s'.rtcConnections[$nonce$].windownonce $\equiv$ $s'.\overline{w}$.nonce
↪ **if possible; otherwise stop**
130: **let** s'.rtcConnections[$nonce$].ia := ia
131: **case** ⟨RTC_GET_CHECK_IA_INFO⟩ → Called by IdP proxy to receive the IA to check.
132: **let** $nonce \leftarrow \mathcal{N}$ **such that** s'.rtcConnections[$nonce$].windownonce $\equiv$ $s'.\overline{w}$.nonce
↪ **if possible; otherwise stop**
133: **let** $remote$:= s'.rtcConnections[$nonce$].remoteDescription
134: **let** $data$:= ⟨RTC_REMOTE_IA, $remote$.nonce, $remote$.pubkey, $remote$.ia⟩
135: **let** $s'.\overline{d}$.scriptinputs := $s'.\overline{d}$.scriptinputs $+^{\langle\rangle}$ $data$
136: **case** ⟨RTC_CHECKED_IA, ia⟩ → Called by IdP proxy when the IA was checked successfully.
137: **let** $nonce \leftarrow \mathcal{N}$ **such that** s'.rtcConnections[$nonce$].windownonce $\equiv$ $s'.\overline{w}$.nonce
↪ **if possible; otherwise stop**
138: **if** s'.rtcConnections[$nonce$].remoteDescription.ia $\not\equiv$ ia **then**
139: **stop**
140: **let** s'.rtcConnections[$nonce$].remoteAuthenticated := $\top$
141: **case** ⟨RTC_SEND, $nonce$, $data$⟩
142: **if** s'.rtcConnections[$nonce$].docnonce $\not\equiv$ $s'.\overline{d}$.nonce **then**
143: **stop**
144: **let** $remote$:= s'.rtcConnections[$nonce$].remoteDescription
145: **stop** ⟨⟨$remote$.addr, a, ⟨RTC_MSG, $remote$.nonce, $\mathsf{enc_a}$($data$, $remote$.pubkey)⟩⟩⟩, s'
146: **case** else
147: **stop**

The function DELIVER_TO_DOC (Algorithm A.9) is a helper function that adds data to the script inputs of a document identified by the document reference. This function is used in the following algorithms to deliver WebSocket, WebRTC, and XMLHttpRequest messages to scripts.

Algorithm A.9 Web Browser Model: Deliver a message to the script in a document.

1: **function** DELIVER_TO_DOC($docnonce$, $data$, s')
2: **let** $\overline{w} \leftarrow$ Subwindows(s'), $\overline{d}$ **such that** $s'.\overline{d}$.nonce $\equiv$ $docnonce$ $\wedge$ $s'.\overline{d} = s'.\overline{w}$.activedocument
↪ **if possible; otherwise stop**
3: **let** $s'.\overline{d}$.scriptinputs := $s'.\overline{d}$.scriptinputs $+^{\langle\rangle}$ $data$

The function PROCESSRESPONSE (Algorithm A.10) is responsible for processing an HTTP response (*response*) that was received as the response to a request (*request*) that was sent earlier. The *reference* identifies this request, which can either be a normal HTTP connection, an XMLHttpRequest, or a WebSocket connection establishment request.

The function first saves any cookies that were contained in the response to the browser state (using the AddCookie helper function shown above), then checks whether a redirection is requested (`Location` header). If that is the case, the function prepares and sends a new HTTP request, according to the rules in the HTTP/1.1 [RFC7231] and Fetch [Fetch] standards. If no redirection is requested, the function creates a new document (for normal HTTP requests), delivers the contents of the response to the respective receiver (for XMLHttpRequest responses), or finishes the WebRTC connection establishment.

Finally, the browser main function (Algorithm A.11) brings all parts together and defines $R_{\text{webbrowser}}$. First, the algorithm checks whether the browser is already corrupted. If that is the case, it collects all incoming messages and derives a new messages from its state and sends it out over the network. If the browser is not corrupted, it handles trigger messages (to trigger a nondeterministic action by the browser), corruption messages, or DNS/HTTP/WebSocket/WebRTC messages (see Section 2.10 for details).

A.6.4. Definition of Web Browsers

Finally, we define web browser atomic Dolev-Yao processes as follows:

Definition 49 (Web Browser atomic Dolev-Yao Process). A web browser atomic Dolev-Yao process b is an atomic Dolev-Yao process $b = (I^b, Z_{\text{webbrowser}}, R_{\text{webbrowser}}, {s_0}^b)$ for a set I^b of addresses, $Z_{\text{webbrowser}}$ and $R_{\text{webbrowser}}$ as defined above, and an initial state ${s_0}^b \in Z_{\text{webbrowser}}$.

Algorithm A.10 Web Browser Model: Process an HTTP response.

1: **function** PROCESSRESPONSE($response$, $reference$, $request$, $requestUrl$, key, f, s')

— Process headers in response —

2: **if** `Set-Cookie` $\in$ $response$.headers **then**
3: **for each** $c \in^{\langle\rangle}$ $response$.headers [`Set-Cookie`], $c \in$ Cookies **do**
4: **let** s'.cookies [$request$.host] := AddCookie(s'.cookies [$request$.host] , c)
5: **if** `Strict-Transport-Security` $\in$ $response$.headers $\wedge$ $requestUrl$.protocol $\equiv$ S **then**
6: **let** s'.sts := s'.sts $+^{\langle\rangle}$ $request$.host
7: **if** `Referer` $\in$ $request$.headers **then**
8: **let** $referrer$:= $request$.headers[`Referer`]
9: **else**
10: **let** $referrer$:= $\bot$

11: **if** $\mathtt{Location} \in \mathit{response}.\mathtt{headers} \wedge \mathit{response}.\mathtt{status} \in \{303, 307\}$ **then**
12: **let** $\mathit{url} := \mathit{response}.\mathtt{headers}[\mathtt{Location}]$
13: **if** $\mathit{url}.\mathtt{fragment} \equiv \bot$ **then**
14: **let** $\mathit{url}.\mathtt{fragment} := \mathit{requestUrl}.\mathtt{fragment}$
15: **let** $\mathit{method}' := \mathit{request}.\mathtt{method}$
16: **let** $\mathit{body}' := \mathit{request}.\mathtt{body}$
17: **if** $\mathit{response}.\mathtt{status} \equiv 303 \wedge \mathit{request}.\mathtt{method} \not\in \{\mathtt{GET}, \mathtt{HEAD}\}$ **then**
18: **let** $\mathit{method}' := \mathtt{GET}$
19: **let** $\mathit{body}' := \langle\rangle$
20: **if** $\mathtt{Origin} \in \mathit{request}.\mathtt{headers} \wedge \mathit{method}' \equiv \mathtt{POST}$ **then**
21: **let** $\mathit{origin} := \diamond$
22: **else**
23: **let** $\mathit{origin} := \bot$
24: **if** $\pi_1(\mathit{reference}) \not\equiv \mathtt{XHR}$ **then** → Do not redirect XHRs.
25: **let** $\mathit{req} := \langle \mathtt{HTTPReq}, \nu_6, \mathit{method}', \mathit{url}.\mathtt{host}, \mathit{url}.\mathtt{path}, \mathit{url}.\mathtt{parameters}, \langle\rangle, \mathit{body}' \rangle$
26: **let** $\mathit{referrerPolicy} := \mathit{response}.\mathtt{headers}[\mathtt{ReferrerPolicy}]$
27: **call** $\mathsf{HTTP_SEND}(\mathit{reference}, \mathit{req}, \mathit{url}, \mathit{origin}, \mathit{referrer}, \mathit{referrerPolicy}, s')$

——— Deliver/process data in response ———

28: **switch** $\pi_1(\mathit{reference})$ **do**
29: **case** $\mathtt{REQ}$ → normal response
30: **let** $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\mathtt{nonce} \equiv \pi_2(\mathit{reference})$ ↪ **if possible; otherwise stop**
31: **if** $\mathit{response}.\mathtt{body} \not\sim \langle *, * \rangle$ **then**
32: **stop** $\{\}, s'$
33: **let** $\mathit{script} := \pi_1(\mathit{response}.\mathtt{body})$
34: **let** $\mathit{scriptstate} := \pi_2(\mathit{response}.\mathtt{body})$
35: **let** $d := \langle \nu_7, \mathit{requestUrl}, \mathit{response}.\mathtt{headers}, \mathit{referrer}, \mathit{script}, \mathit{scriptstate}, \langle\rangle, \langle\rangle, \top \rangle$
36: **if** $s'.\overline{w}.\mathtt{documents} \equiv \langle\rangle$ **then**
37: **let** $s'.\overline{w}.\mathtt{documents} := \langle d \rangle$
38: **else**
39: **let** $\overline{i} \leftarrow \mathbb{N}$ **such that** $s'.\overline{w}.\mathtt{documents}.\overline{i}.\mathtt{active} \equiv \top$
40: **let** $s'.\overline{w}.\mathtt{documents}.\overline{i}.\mathtt{active} := \bot$
41: **remove** $s'.\overline{w}.\mathtt{documents}.(\overline{i}+1)$ and all following documents from $s'.\overline{w}.\mathtt{documents}$
42: **let** $s'.\overline{w}.\mathtt{documents} := s'.\overline{w}.\mathtt{documents} +^{\langle\rangle} d$
43: **stop** $\{\}, s'$
44: **case** $\mathtt{XHR}$ → process XHR response
45: **let** $\mathit{headers} := \mathit{response}.\mathtt{headers} - \mathsf{Set\text{-}Cookie}$
46: **let** $m := \langle \mathtt{XMLHTTPREQUEST}, \mathit{headers}, \mathit{response}.\mathtt{body}, \pi_3(\mathit{reference}) \rangle$
47: **call** $\mathsf{DELIVER_TO_DOC}(\pi_2(\mathit{reference}), m, s')$
48: **stop** $\{\}, s'$
49: **case** $\mathtt{WS}$ → process WebSocket response
50: **if** $\mathit{response}.\mathtt{status} \not\equiv 101 \vee \mathit{response}.\mathtt{headers}[\mathtt{Upgrade}] \not\equiv \mathtt{websocket}$ **then**
51: **stop**
52: **let** $\mathit{wsconn} := \langle \mathit{reference}, \mathit{request}.\mathtt{nonce}, \mathit{key}, f \rangle$
53: **let** $s'.\mathtt{wsConnections} := s'.\mathtt{wsConnections} +^{\langle\rangle} \mathit{wsconn}$

Algorithm A.11 Web Browser Model: Main Algorithm.

Input: $\langle a, f, m\rangle, s$

1: **let** $s' := s$

——— Check if browser is corrupted ———

2: **if** s.isCorrupted $\not\equiv \bot$ **then**
3: **let** s'.pendingRequests $:= \langle m, s$.pendingRequests$\rangle$ → Collect incoming messages
4: **let** $n \leftarrow \mathbb{N}$
5: **let** $m'_1, \dots, m'_n \leftarrow d_V(s')$ → Create n new messages nondeterministically.
6: **let** $a'_1, \dots, a'_n \leftarrow$ IPs
7: **stop** $\langle\langle a'_1, a, m'_1\rangle, \dots, \langle a'_n, a, m'_n\rangle\rangle$, s'

——— Receive trigger message ———

8: **if** $m \equiv$ TRIGGER **then**
9: **let** $switch \leftarrow \{$script, urlbar, reload, forward, back$\}$
10: **let** $\overline{w} \leftarrow$ Subwindows(s') **such that** $s'.\overline{w}$.documents $\neq \langle\rangle$
↪ **if possible; otherwise stop** → Pointer to some window.
11: **let** $\overline{tlw} \leftarrow \mathbb{N}$ **such that** $s'.\overline{tlw}$.documents $\neq \langle\rangle$
↪ **if possible; otherwise stop** → Pointer to some top-level window.
12: **if** $switch \equiv$ script **then** → Run some script.
13: **let** $\overline{d} := \overline{w} +^{\langle\rangle}$ activedocument
14: **call** RUNSCRIPT$(\overline{w}, \overline{d}, s')$
15: **else if** $switch \equiv$ urlbar **then** → Create some new request.
16: **let** $newwindow \leftarrow \{\top, \bot\}$
17: **if** $newwindow \equiv \top$ **then** → Create a new window.
18: **let** $windownonce := \nu_1$
19: **let** $w' := \langle windownonce, \langle\rangle, \bot\rangle$
20: **let** s'.windows $:= s'$.windows $+^{\langle\rangle} w'$
21: **else** → Use existing top-level window.
22: **let** $windownonce := s'.\overline{tlw}.nonce$
23: **let** $protocol \leftarrow \{$P, S$\}$
24: **let** $host \leftarrow$ Doms
25: **let** $path \leftarrow \mathbb{S}$
26: **let** $fragment \leftarrow \mathbb{S}$
27: **let** $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
28: **let** $url := \langle$URL, $protocol, host, path, parameters, fragment\rangle$
29: **let** $req := \langle$HTTPReq, ν_2, GET, $host, path, parameters, \langle\rangle, \langle\rangle\rangle$
30: **call** HTTP_SEND$(\langle$REQ, $windownonce\rangle, req, url, \bot, \bot, \bot, s')$
31: **else if** $switch \equiv$ reload **then** → Reload some document.
32: **let** $url := s'.\overline{w}$.activedocument.location
33: **let** $req := \langle$HTTPReq, ν_2, GET, url.host, url.path, url.parameters, $\langle\rangle, \langle\rangle\rangle$
34: **let** $referrer := s'.\overline{w}$.activedocument.referrer
35: **let** $s' :=$ CANCELNAV$(s'.\overline{w}$.nonce$, s')$
36: **call** HTTP_SEND$(\langle$REQ, $s'.\overline{w}$.nonce$\rangle, req, url, \bot, referrer, \bot, s')$
37: **else if** $switch \equiv$ forward **then**
38: NAVFORWARD$(\overline{w}, s')$
39: **else if** $switch \equiv$ back **then**
40: NAVBACK$(\overline{w}, s')$

——— Change corruption status ———

41: **else if** $m \equiv$ FULLCORRUPT **then** → Request to corrupt browser
42: **let** s'.isCorrupted $:=$ FULLCORRUPT
43: **stop** $\langle\rangle$, s'

44: **else if** $m \equiv \texttt{LIMITEDCORRUPT}$ **then** → Close the browser
45: **let** $s'.\texttt{secrets} := \langle\rangle$
46: **let** $s'.\texttt{windows} := \langle\rangle$
47: **let** $s'.\texttt{pendingDNS} := \langle\rangle$
48: **let** $s'.\texttt{pendingRequests} := \langle\rangle$
49: **let** $s'.\texttt{sessionStorage} := \langle\rangle$
50: **let** $s'.\texttt{cookies} \subset^{\langle\rangle} \mathsf{Cookies}$ **such that**
↪ $(c \in^{\langle\rangle} s'.\texttt{cookies}) \iff (c \in^{\langle\rangle} s.\texttt{cookies} \wedge c.\texttt{content}.\texttt{session} \equiv \bot)$
51: **let** $s'.\texttt{isCorrupted} := \texttt{LIMITEDCORRUPT}$
52: **stop** $\langle\rangle$, s'

—— Plain-text messages ——

53: **else if** $m \in \mathsf{DNSResponses}$ **then** → DNS response
54: **if** $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin \mathsf{IPs}$
↪ $\vee\ m.\texttt{domain} \not\equiv \pi_2(s.\texttt{pendingDNS}[m.\texttt{nonce}]).\texttt{host}$ **then**
55: **stop**
56: **let** $\langle \mathit{reference}, \mathit{message}, \mathit{url}\rangle := s.\texttt{pendingDNS}[m.\texttt{nonce}]$
57: **if** $\mathit{url}.\texttt{protocol} \equiv \mathsf{S}$ **then**
58: **let** $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle \mathit{reference}, \mathit{message}, \mathit{url}, \nu_3, m.\texttt{result}\rangle$
59: **let** $\mathit{message} := \mathsf{enc_a}(\langle \mathit{message}, \nu_3\rangle, s'.\texttt{keyMapping}[\mathit{message}.\texttt{host}])$
60: **else**
61: **let** $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle \mathit{reference}, \mathit{message}, \mathit{url}, \bot, m.\texttt{result}\rangle$
62: **let** $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
63: **stop** $\langle\langle m.\texttt{result}, a, \mathit{message}\rangle\rangle$, s'
64: **else if** $\pi_1(m) \equiv \texttt{HTTPResp} \wedge \exists\, \langle \mathit{reference}, \mathit{request}, \mathit{url}, \bot, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
↪ **such that** $m.\texttt{nonce} \equiv \mathit{request}.\texttt{nonce}$ **then** → Plain HTTP Response
65: **remove** $\langle \mathit{reference}, \mathit{request}, \mathit{url}, \bot, f\rangle$ **from** $s'.\texttt{pendingRequests}$
66: **call** PROCESSRESPONSE$(m, \mathit{reference}, \mathit{request}, \mathit{url}, \mathit{key}, f, s')$
67: **else if** $m.1 \equiv \texttt{WS_MSG} \wedge \exists\, \langle \mathit{reference}, \mathit{nonce}, \bot, f\rangle \in^{\langle\rangle} s'.\texttt{wsConnections}$
↪ **such that** $m.\texttt{nonce} \equiv \mathit{nonce}$ **then** → Plain Websocket Message
68: **call** DELIVER_TO_DOC$(\pi_2(\mathit{reference}), m, s')$

—— Encrypted messages ——

69: **else if** $\exists\, \langle \mathit{reference}, \mathit{request}, \mathit{url}, \mathit{key}, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
↪ **such that** $\pi_1(\mathsf{dec_s}(m, \mathit{key})) \equiv \texttt{HTTPResp}$ **then** → Encrypted HTTP response
70: **let** $m' := \mathsf{dec_s}(m, \mathit{key})$
71: **if** $m'.\texttt{nonce} \not\equiv \mathit{request}.\texttt{nonce}$ **then**
72: **stop**
73: **remove** $\langle \mathit{reference}, \mathit{request}, \mathit{url}, \mathit{key}, f\rangle$ **from** $s'.\texttt{pendingRequests}$
74: **call** PROCESSRESPONSE$(m', \mathit{reference}, \mathit{request}, \mathit{url}, \mathit{key}, f, s')$
75: **else if** $\exists\, \langle \mathit{reference}, \mathit{nonce}, \mathit{key}, f\rangle \in^{\langle\rangle} s'.\texttt{wsConnections}$
↪ **such that** $\pi_1(\mathsf{dec_s}(m, \mathit{key})) \equiv \texttt{WS_MSG}$ **then** → Encrypted Websocket Message
76: **let** $m' := \mathsf{dec_s}(m, \mathit{key})$
77: **if** $m'.\texttt{nonce} \not\equiv \mathit{nonce}$ **then**
78: **stop**
79: **call** DELIVER_TO_DOC$(\pi_2(\mathit{reference}), m', s')$
80: **else if** $\exists \langle \mathit{nonce}, \mathit{info}\rangle \in^{\langle\rangle} s'.\texttt{rtcConnections}$
↪ **such that** $\pi_1(\mathsf{dec_a}(m, \mathit{info}.\texttt{privkey})) \equiv \texttt{RTC_MSG}$ **then** → WebRTC message
81: **let** $m' := \mathsf{dec_a}(m, \mathit{info}.\texttt{privkey})$
82: **if** $m'.\texttt{nonce} \not\equiv \mathit{nonce}$ **then**
83: **stop**
84: **let** $\mathit{docnonce} := \mathit{info}.\texttt{docnonce}$
85: **call** DELIVER_TO_DOC$(\mathit{docnonce}, m', s')$
86: **stop**

A.7. Generic HTTPS Server Model

This base model can be used to ease modeling of HTTPS server atomic processes. It defines placeholder algorithms that can be replaced by more detailed algorithms to describe a concrete instance of an HTTPS server.

For these algorithms to work as expected, the state of the HTTPS server must contain (at least) a minimal set of subterms:

Definition 50 (Base state for an HTTPS server.). The states of HTTPS servers that are instantiations of this generic server must contain at least the following subterms: $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ (both containing arbitrary terms), $DNSaddress \in \mathsf{IPs}$ (containing the IP address of a DNS server), $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ (containing a mapping from domains to public keys), $tlskeys \in [\mathsf{Doms} \times \mathcal{N}]$ (containing a mapping from domains to private keys), and $corrupt \in \mathcal{T}_{\mathcal{N}}$ (either $\perp$ if the server is not corrupted, or an arbitrary term otherwise).

We note that in concrete instantiations of the generic HTTPS server model, there is no need to extract information from these subterms or alter these subterms.

Let ν_{n0} and ν_{n1} denote placeholders for nonces that are not used in the concrete instantiation of the server. Algorithm A.17 defines the relation of the generic HTTPS server. It works similar to the browser main algorithm (Algorithm A.11) and makes use of Algorithms A.13–A.16 to handle various types of incoming messages. In particular, the function PROCESS_HTTPS_REQUEST (Algorithm A.15) is called when an HTTPS request was received and successfully decrypted. An HTTPS response is handled by the function PROCESS_HTTPS_RESPONSE (Algorithm A.13), trigger messages are handled by PROCESS_TRIGGER (Algorithm A.14), and all other messages are handled by PROCESS_OTHER (Algorithm A.16). The function HTTPS_SIMPLE_SEND (Algorithm A.12) can be used to send HTTPS requests. In this regard, the generic HTTPS server works just like a (simplified) browser. For example, it first sends a DNS request to resolve the name into an IP address before sending out an HTTPS request.

Algorithm A.12 Generic HTTPS Server Model: Sending a DNS message.

1: **function** HTTPS_SIMPLE_SEND($reference$, $message$, s', a)
2: **let** s'.pendingDNS[ν_{n0}] := $\langle reference, message \rangle$
3: **stop** $\langle \langle s'.\mathtt{DNSaddress}, a, \langle \mathtt{DNSResolve}, message.\mathtt{host}, \nu_{n0} \rangle \rangle \rangle$, s'

Algorithm A.13 Generic HTTPS Server Model: Default HTTPS response handler.

1: **function** PROCESS_HTTPS_RESPONSE(m, $reference$, $request$, key, a, f, s')
2: **stop**

Algorithm A.14 Generic HTTPS Server Model: Default trigger event handler.

1: **function** PROCESS_TRIGGER(s')
2: **stop**

Algorithm A.15 Generic HTTPS Server Model: Default HTTPS request handler.

1: **function** PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2: **stop**

Algorithm A.16 Generic HTTPS Server Model: Default handler for other messages.

1: **function** PROCESS_OTHER(m, a, f, s')
2: **stop**

Algorithm A.17 Generic HTTPS Server Model: Main relation of a generic HTTPS server.

Input: $\langle a, f, m\rangle, s$

1: **let** $s' := s$

——— Check if browser is corrupted or becomes corrupted ———

2: **if** $s'.\mathtt{corrupt} \not\equiv \bot \vee m \equiv \mathtt{CORRUPT}$ **then**
3: **let** $s'.\mathtt{corrupt} := \langle\langle a, f, m\rangle, s'.\mathtt{corrupt}\rangle$
4: **let** $m' \leftarrow d_V(s')$
5: **let** $a' \leftarrow \mathsf{IPs}$
6: **stop** $\langle\langle a', a, m'\rangle\rangle$, s'

——— Decrypt HTTPS request ———

7: **if** $\exists\, m_{\text{dec}}$, k, k', $\mathit{inDomain}$ **such that** $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec}_{\mathsf{a}}(m, k') \wedge \langle \mathit{inDomain}, k'\rangle \in s.\mathtt{tlskeys}$ **then**
8: **let** n, method, path, $\mathit{parameters}$, $\mathit{headers}$, body **such that**
↪ $\langle \mathtt{HTTPReq}, n, \mathit{method}, \mathit{inDomain}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}\rangle \equiv m_{\text{dec}}$
↪ **if possible; otherwise stop**
9: **call** $\mathsf{PROCESS_HTTPS_REQUEST}(m_{\text{dec}}, k, a, f, s')$

——— DNS response ———

10: **else if** $m \in \mathsf{DNSResponses}$ **then**
11: **if** $m.\mathtt{nonce} \not\in s.\mathtt{pendingDNS} \vee m.\mathtt{result} \not\in \mathsf{IPs}$
↪ $\vee\ m.\mathtt{domain} \not\equiv s.\mathtt{pendingDNS}[m.\mathtt{nonce}].2.\mathtt{domain}$ **then**
12: **stop**
13: **let** $\langle \mathit{reference}, \mathit{request}\rangle := s.\mathtt{pendingDNS}[m.\mathtt{nonce}]$
14: **let** $s'.\mathtt{pendingRequests} := s'.\mathtt{pendingRequests}$
↪ $+^{\langle\rangle}\ \langle \mathit{reference}, \mathit{request}, \nu_{n1}, m.\mathtt{result}\rangle$
15: **let** $\mathit{message} := \mathsf{enc}_{\mathsf{a}}(\langle \mathit{request}, \nu_{n1}\rangle, s'.\mathtt{keyMapping}[\mathit{request}.\mathtt{host}])$
16: **let** $s'.\mathtt{pendingDNS} := s'.\mathtt{pendingDNS} - m.\mathtt{nonce}$
17: **stop** $\langle\langle m.\mathtt{result}, a, \mathit{message}\rangle\rangle$, s'

——— Encrypted HTTPS response ———

18: **else if** $\exists\,\langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle \in^{\langle\rangle} s'.\mathtt{pendingRequests}$
↪ **such that** $\pi_1(\mathsf{dec}_{\mathsf{s}}(m, \mathit{key})) \equiv \mathtt{HTTPResp}$ **then**
19: **let** $m' := \mathsf{dec}_{\mathsf{s}}(m, \mathit{key})$
20: **if** $m'.\mathtt{nonce} \not\equiv \mathit{request}.\mathtt{nonce}$ **then**
21: **stop**
22: **remove** $\langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle$ **from** $s'.\mathtt{pendingRequests}$
23: **call** $\mathsf{PROCESS_HTTPS_RESPONSE}(m', \mathit{reference}, \mathit{request}, \mathit{key}, a, f, s')$
24: **stop**

——— Process was triggered ———

25: **else if** $m \equiv \mathtt{TRIGGER}$ **then**
26: **call** $\mathsf{PROCESS_TRIGGER}(s')$

——— Other type of message ———

27: **else**
28: **call** $\mathsf{PROCESS_OTHER}(m, a, f, s')$
29: **stop**

B. Analysis of OAuth 2.0

In this appendix, we present our model of OAuth, the formalization of the security properties, and the proof of the OAuth Security Theorem (Theorem 1).

B.1. Formal Model of OAuth with a Network Attacker

We model OAuth as a web system in the sense of Appendix A.3. We call a web system $\mathcal{OAuthWS}^n = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ an *OAuth web system with a network attacker* if it is of the form described in what follows.

B.1.1. Outline

The system $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Net}$ consists of a network attacker process (in Net), a finite set B of web browsers, a finite set $\mathsf{Clients}$ of web servers for the clients, a finite set OAP of web servers for the OAuth providers (each OAP playing the role of an AS and an RS), with $\mathsf{Hon} := \mathsf{B} \cup \mathsf{Clients} \cup \mathsf{OAP}$. More details on the processes in $\mathcal{W}$ are provided below. We do not model DNS servers, as they are subsumed by the network attacker. Table B.1 shows the set of scripts $\mathcal{S}$, their string representations that are defined by the mapping script, and the algorithms that define the respective scripts.

$s \in \mathcal{S}$	$\mathsf{script}(s)$	defined in
R^{att}	`att_script`	Definition 26
script_client_index	`script_client_index`	Algorithm B.1
script_client_implicit	`script_client_implicit`	Algorithm B.2
script_oap_form	`script_oap_form`	Algorithm B.3

Table B.1. List of scripts in $\mathcal{S}$, their respective string representations, and their definitions.

In the algorithms defining the scripts, we use the function $\mathsf{GETURL}(\mathit{tree}, \mathit{docnonce})$. We define this function as follows: It searches for the document with the identifier *docnonce* in the (cleaned) tree *tree* of the browser's windows and documents. It then returns the URL u of that document. If no document with nonce *docnonce* is found in the tree *tree*, $\Diamond$ is returned. We also use the helper function $\mathsf{GETDOCWINDOW}(\mathit{tree}, \mathit{docnonce})$. It returns the nonce of the window in *tree* that contains the document identified by *docnonce*.

The set E^0 contains only the trigger events as specified in Appendix A.3. This outlines $\mathcal{OAuthWS}^n$. We now define the DY processes in $\mathcal{OAuthWS}^n$ and their addresses, domain names, and secrets in more detail.

Algorithm B.1 Relation of *script_client_index*.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$

1: **let** $switch \leftarrow \{\texttt{auth}, \texttt{link}\}$
2: **if** $switch \equiv \texttt{auth}$ **then**
3:     **let** $url := \mathsf{GETURL}(tree, docnonce)$
4:     **let** $id \leftarrow ids$
5:     **let** $username := \pi_1(id)$
6:     **let** $domain := \pi_2(id)$
7:     **let** $interactive \leftarrow \{\bot, \top\}$
8:     **if** $interactive \equiv \top$ **then**
9:         **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/startInteractiveLogin}, \langle\rangle, \langle\rangle\rangle$
10:         **let** $command := \langle \texttt{FORM}, url', \texttt{POST}, domain, \bot\rangle$
11:     **else**
12:         **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/passwordLogin}, \langle\rangle, \langle\rangle\rangle$
13:         **let** $secret$ **such that** $secret = \mathsf{secretOfID}(id) \wedge secret \in secrets$ **if possible; otherwise** $\hookrightarrow$ **stop** $\langle s, cookies, localStorage, sessionStorage, \langle\rangle\rangle$
14:         **let** $command := \langle \texttt{FORM}, url', \texttt{POST}, \langle id, secret\rangle, \bot\rangle$
15:     **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$
16: **else**
17:     **let** $protocol \leftarrow \{\texttt{P}, \texttt{S}\}$
18:     **let** $host \leftarrow \mathsf{Doms}$
19:     **let** $path \leftarrow \mathbb{S}$
20:     **let** $fragment \leftarrow \mathbb{S}$
21:     **let** $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
22:     **let** $url := \langle \texttt{URL}, protocol, host, path, parameters, fragment\rangle$
23:     **let** $command := \langle \texttt{HREF}, url, \mathsf{GETDOCWINDOW}(tree, docnonce)), \bot\rangle$
24:     **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$

Algorithm B.2 Relation of *script_client_implicit*.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$

1: **let** $url := \mathsf{GETURL}(tree, docnonce)$
2: **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/receiveTokenFromImplicitGrant}, \langle\rangle, \langle\rangle\rangle$
3: **let** $body := \langle url.\texttt{fragment}[\texttt{access_token}], url.\texttt{fragment}[\texttt{state}], scriptstate\rangle$
4: **let** $command := \langle \texttt{FORM}, url', \texttt{POST}, body, \bot\rangle$
5: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$

Algorithm B.3 Relation of *script_oap_form*.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$

1: **let** $url := \mathsf{GETURL}(tree, docnonce)$
2: **let** $url.\texttt{path} \leftarrow \mathbb{S}$
3: **let** $formdata := scriptstate$
4: **let** $id \leftarrow ids$
5: **let** $secret \leftarrow secrets$
6: **let** $formdata := formdata +^{\langle\rangle} \langle \texttt{username}, id\rangle$
7: **let** $formdata := formdata +^{\langle\rangle} \langle \texttt{password}, secret\rangle$
8: **let** $command := \langle \texttt{FORM}, url, \texttt{POST}, formdata, \bot\rangle$
9: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$

B.1.2. Addresses and Domain Names

The set IPs contains for the network attacker in Net, every client in $\mathsf{Clients}$, every OAuth provider in OAP, and every browser in B a finite set of addresses each. By addr we denote the corresponding assignment from a process to its address. The set Doms contains a finite set of domains for every client in $\mathsf{Clients}$, every OAuth provider in OAP, and the network attacker in Net. Browsers (in B) do not have a domain.

By addr and dom we denote the assignments from atomic processes to sets of IPs and Doms, respectively.

B.1.3. Keys and Secrets

The set $\mathcal{N}$ of nonces is partitioned into five sets: an infinite sequence N, an infinite set K_{TLS}, an infinite set K_{sign}, and finite sets $\mathsf{Passwords}$, $\mathsf{ClientSecrets}'$ and $\mathsf{ProtectedResources}$. We thus have

$$\mathcal{N} = \underbrace{N}_{\text{infinite sequence}} \dot\cup \underbrace{K_{\text{TLS}}}_{\text{finite}} \dot\cup \underbrace{\mathsf{Passwords}}_{\text{finite}} \dot\cup \underbrace{\mathsf{ClientSecrets}'}_{\text{finite}} \dot\cup \underbrace{\mathsf{ProtectedResources}}_{\text{finite}} .$$

We then define $\mathsf{ClientSecrets} := \mathsf{ClientSecrets}' \cup \{\bot\}$. These sets are used as follows:

- The set N contains the nonces that are available to each DY process in $\mathcal{W}$, i.e., the set can be used to create a run of $\mathcal{W}$.

- The set K_{TLS} contains the keys that will be used for TLS encryption. Let $\mathsf{tlskey}\colon \mathsf{Doms} \to K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p we define $\mathit{tlskeys}^p$ to be a mapping from domains of p to corresponding TLS keys, or more formally, $\mathit{tlskeys}^p = \langle\{\langle d, \mathsf{tlskey}(d)\rangle \mid d \in \mathsf{dom}(p)\}\rangle$.

- The set $\mathsf{Passwords}$ is the set of passwords (secrets) the browsers share with the OAuth providers. These are the passwords the users use to log in at the OAPs.

- The set $\mathsf{ClientSecrets}$ is the set of passwords (secrets) the clients share with the OAuth providers. These are the passwords the clients use to log in at the OAPs. The passwords can also be blank ($\bot$).

- The set $\mathsf{ProtectedResources}$ contains a secret for each combination of OAP, client, and user. These are thought of as protected resources that only the resource owner (i.e., the user) should be able to read. (See also Definition 55.)

B.1.4. Identities, Passwords, and Protected Resources

Identities consist, like email addresses, of a username and a domain part. For our model, this is defined as follows:

Definition 51. An *identity* (email address) i is a term of the form $\langle name, domain\rangle$ with $name \in \mathbb{S}$ and $domain \in \mathsf{Doms}$.

Let ID be the finite set of identities. By ID^y we denote the set $\{\langle name, domain\rangle \in \mathsf{ID} \mid domain \in \mathsf{dom}(y)\}$.

We say that an ID is *governed* by the DY process to which the domain of the ID belongs. Formally, we define the mapping $\mathsf{governor} : \mathsf{ID} \to \mathcal{W}$, $\langle name, domain\rangle \mapsto \mathsf{dom}^{-1}(domain)$.

The governor of an ID will usually be an OAP, but could also be the attacker. Besides $\mathsf{governor}$, we define the following mappings:

- By $\mathsf{secretOfID} : \mathsf{ID} \to \mathsf{Passwords}$ we denote the bijective mapping that assigns secrets to all identities.

- Let $\mathsf{ownerOfSecret} : \mathsf{Passwords} \to \mathsf{B}$ denote the mapping that assigns to each secret a browser that *owns* this secret. Now, we define the mapping $\mathsf{ownerOfID} : \mathsf{ID} \to \mathsf{B}$, $i \mapsto \mathsf{ownerOfSecret}(\mathsf{secretOfID}(i))$, which assigns to each identity the browser that owns this identity (we say that the identity belongs to the browser).

- Let $\mathsf{trustedClients} : \mathsf{Passwords} \to 2^{\mathsf{Clients}}$ denote a mapping that assigns to each password a set of *trusted clients*. Intuitively a trusted client is a client the user entrusts with her password (in the resource owner password credentials grant of OAuth).

- Let $\mathsf{clientID} : (\mathsf{Client} \cup \{\bot\}) \times \mathsf{OAP} \to \mathbb{S} \cup \{\bot\}$ denote a mapping that assigns an OAuth client id for a client to each combination of a client and an OAuth provider. We require that $\mathsf{clientID}(\cdot, i)$ is bijective for all $i \in \mathsf{OAP}$ and that $\mathsf{clientID}(r, i) = \bot$ iff $r = \bot$ for all $i \in \mathsf{OAP}$.

- Let $\mathsf{secretOfClient} : \mathsf{Clients} \times \mathsf{OAP} \to \mathsf{ClientSecrets}$ denote a bijective mapping that assigns a client password (or the empty password $\bot$) to each combination of a client and an OAuth provider.

- As a shortcut, we define the mapping $\mathsf{secretOfClientID} : \mathbb{S} \times \mathsf{OAP} \to \mathsf{ClientSecrets}$ to return the client password to a client identified by an OAuth client id (at some specific OAuth provider), i.e., $\mathsf{secretOfClientID}(s, i)$ maps to $\mathsf{secretOfClient}(r, i)$ with r such that $s = \mathsf{clientID}(r, i)$.

- By $\mathsf{resourceOf} : \mathsf{OAP} \times (\mathsf{Clients} \cup \{\bot\}) \times (\mathsf{ID} \cup \{\bot\}) \to \mathsf{ProtectedResources}$ we denote the injective mapping that assigns a protected resource to each combination of user identity, OAP and client. We also include protected resources that are not assigned to a specific user (in this case, the user is $\bot$) and those that are not assigned to a specific client (the client then is $\bot$).

  A protected resource depends not only on the OAP and user ID but also the client. This is motivated by the fact that different clients may get access to different protected resources

at one OAP, even if they access the resources of the same user. In the resource owner password credentials mode, clients can also access resources that do not depend on the client, we then have that client is $\bot$.[1]

B.1.5. Corruption

Clients and OAPs can become corrupted: If they receive the message `CORRUPT`, they start collecting all incoming messages in their state and (upon triggering) send out all messages that are derivable from their state and collected input messages, just like the attacker process. We say that a client or an OAP is *honest* if the according part of their state ($s.\mathtt{corrupt}$) is $\bot$, and that they are corrupted otherwise.

We are now ready to define the processes in $\mathcal{W}$ in more detail.

B.1.6. Processes in $\mathcal{W}$ (Overview)

We first provide a brief overview of the processes in $\mathcal{W}$. All processes in $\mathcal{W}$ contain in their initial states all public keys and the private keys of their respective domains (if any). We define $I^p = \mathsf{addr}(p)$ for all $p \in \mathsf{Hon}$.

Network Attacker There is one atomic DY process $na \in \mathsf{Net}$ which is a network attacker (see Appendix A.3), who uses all addresses for sending and listening.

Browsers Each $b \in \mathsf{B}$ is a web browser as defined in Appendix A.6. The initial state contains all secrets owned by b, stored under the origins of the respective OAP and of all trusted clients for the respective secret. See Appendix B.1.8 for details.

Clients Each client is a web server modeled as an atomic DY process following the description in Chapter 3 including the fixes. The client can either (at any time) launch a client credentials grant or wait for users to start any of the other flows. The client manages two kinds of sessions: The *login sessions*, which are only used during the login phase of a user, and the *service sessions* (modeled by a *service token* as described above). When receiving a special message (`CORRUPT`) clients become corrupted, as described before.

OAuth Providers Each OAP is a web server modeled as an atomic DY process following the description in Chapter 3, again including the fixes. In particular, users can authenticate to the OAP with their credentials. Authenticated users can interact with the authorization endpoint of the OAP (e.g., to acquire an authorization code). Just as clients, OAPs can become corrupted.

[1] In the resource owner password credentials mode, the client gets the user's credentials and thus has full access to the user's account at OAP. This access is not bound to potential limitations that depend on the client's identity.

B.1.7. Network Attacker

As mentioned, the network attacker na is modeled to be a network attacker as specified in Appendix A.3. We allow it to listen to/spoof all available IP addresses, and hence, define $I^{na} = \mathsf{IPs}$. The initial state is $s_0^{na} = \langle \mathit{attdoms}, \mathit{tlskeys}, \mathit{signkeys} \rangle$, where $\mathit{attdoms}$ is a sequence of all domains along with the corresponding private keys owned by the attacker na, $\mathit{tlskeys}$ is a sequence of all domains and the corresponding public keys, and $\mathit{signkeys}$ is a sequence containing all public signing keys for all OAPs.

B.1.8. Browsers

Each $b \in \mathsf{B}$ is a web browser as defined in Appendix A.6, with $I^b := \mathsf{addr}(b)$ being its addresses.

To define the inital state, first let $\mathsf{ID}_b := \mathsf{ownerOfID}^{-1}(b)$ be the set of all IDs of b. We then define the set of passwords that a browser b gives to an origin o to consist of two parts: (1) If the origin belongs to an OAP, then the user's passwords of this OAP are contained in the set. (2) If the origin belongs to a client, then those passwords with which the user entrusts this client are contained in the set. To define this mapping in the initial state, we first define for some process p

$$\mathsf{Secrets}^{b,p} = \Big\{ s \;\Big|\; b = \mathsf{ownerOfSecret}(s) \wedge \big((\exists\, i : s = \mathsf{secretOfID}(i) \wedge i \in \mathsf{governor}^{-1}(p)) \vee (\exists\, R : p \in R \wedge s \in \mathsf{trustedClients}^{-1}(R))\big) \Big\} .$$

Then, the initial state s_0^b is defined as follows: the key mapping maps every domain to its public TLS key, according to the mapping tlskey; the DNS address is an address of the network attacker; the list of secrets contains an entry $\langle \langle d, \mathsf{S} \rangle, \langle \mathsf{Secrets}^{b,p} \rangle \rangle$ for each $p \in \mathsf{Clients} \cup \mathsf{OAP}$ and $d \in \mathsf{dom}(p)$; ids is $\langle \mathsf{ID}_b \rangle$; sts is empty.

B.1.9. Clients

A client $r \in \mathsf{Clients}$ is a web server modeled as an atomic DY process (I^r, Z^r, R^r, s_0^r) with the addresses $I^r := \mathsf{addr}(r)$.[2] Its initial state s_0^r contains its domains, the private keys associated with its domains, the DNS server address, and information about OAPs where the client is registered at. The full state additionally contains the sets of service tokens and login session identifiers the client has issued as well as information about pending DNS and pending HTTPS requests (similar to browsers or generic HTTPS servers). A client only accepts HTTPS requests.

A client manages two kinds of sessions: The *login sessions*, which are only used during the login phase of a user, and the *service sessions* (we call the session identifier of a service session

[2]We use r (for relying party) as a variable here and in the following instead of c to align with [FKS16] and the definitions in Appendix C. For the same reason, we use i (identity provider) instead of o for OAPs.

a *service token*). Service sessions allow a user to use the client's services. The ultimate goal of a login flow is to establish such a service session.

We have already described how r can become corrupted. In the following, we explain the behaviour of r during a login flow in more detail.

Initial Request In a typical flow, r will first receive an HTTP GET request from a browser for the path /. In this case, r returns the script `script_client_index`. Besides providing arbitrary links, this script allows users to start an OAuth flow in the browser. If an OAuth flow is started, the script nondeterministically chooses an identity of the user, i.e., a combination of a username and a domain of an OAP. Further this script nondeterministically decides whether an interactive login (i.e., authorization code grant or implicit grant) or a non-interactive login (i.e., resource owner password credentials grant) is used. If an interactive login is chosen, the script instructs the browser to send an HTTPS POST request to r for the path `/startInteractiveLogin`. This POST request contains in its body the domain of the OAP.[3] If the script chooses a non-interactive login, the domain of the OAP, the username, and the user's password are sent to r in an HTTPS POST request for the path `/passwordLogin`.

As the flow now forks into different branches, we explain each of these branches separately.

Interactive Login In this case, `script_client_index` has sent an HTTPS POST request for the path `/startInteractiveLogin` to r containing the domain of an OAP in its body. When r receives such a request, r nondeterministically decides whether the OAuth authorization grant or the OAuth implicit grant is used. Also, r nondeterministically either selects a redirect URI *redirect_uri* of its redirection endpoints and appends the domain of the OAP to this redirect URI, or selects no redirect URI. Further, r nondeterministically selects a (fresh) nonce *state* and a (fresh) nonce as the login session id. Now, r constructs and sends an HTTPS response containing an HTTP 303 location redirect or an HTTP 307 location redirect (chosen nondeterministically). The redirection points to the authorization endpoint at the OAP along with r's OAuth client id for this OAP, *state* and information which OAuth grant r has chosen. The response also contains a `Set-Cookie` header, which sets a cookie containing the login session id. r also creates a new record in the subterm *loginSessions* of its state. This record contains the login session id, the chosen OAP and grant type, and the redirection URI (if any).

Later in the flow, when OAP redirects the user's browser to r's redirection endpoint, r will receive an HTTPS GET request for the path `/redirectionEndpoint`. This request must contain a login session id cookie, which refers to the information stored in the subterm *loginSessions* in r's state. The request must also contain a parameter with the domain of the OAP and this domain must match the domain stored for this login session.

If r has stored that for this login session the OAuth authorization code grant is used, r checks if the *state* value in the parameter of the same name is correct, i.e., is congruent to the value

[3] While the script has selected an identity of the user, only the domain of the OAP is used here. During the authentication to the OAP, a different username may be chosen.

recorded in r's state. Then, r extracts the authorization code *code* from the parameters of the incoming request and prepares an HTTPS POST request to the OAP's token endpoint to obtain an access token. To this end, r first adds the authorization code to the request's body. If a redirect URI has been set by r before, the redirect URI is included in the request's body. If r knows an OAuth client secret for the OAP, r adds its OAuth client id and its OAuth client secret for the OAP to the header of the request, else r adds its OAuth client id for the OAP to the request's body. Now, r sends a DNS request for the domain of the OAP's token endpoint to the DNS server. Finally, r saves the prepared request and the HTTPS request received from the browser in its state. We continue our description in the OAuth authorization code grant in the paragraph *Token Response* below.

If the (incoming) HTTPS request's login session at r states that implicit grant is used, r instead sends an HTTPS response to the sender of the incoming message. This HTTPS response contains the script *script_client_implicit* and the initial state for this script in this response contains the domain of the OAP.

In a browser, this script extracts *access_token* and *state* from the fragment part of its URL and extracts the domain of the OAP from its initial state. The script then sends this information in the body of an HTTPS POST request for the path `/receiveTokenFromImplicitGrant` to r.

When r receives such an HTTPS POST request, r checks if this request contains a login session id cookie, which refers to the information stored in its state and if the values of *state* and *oap* (contained in the request) match the information there. Next, r prepares an HTTPS request to OAP's introspection endpoint containing the access token just received. r saves all information belonging to this new request and the (incoming) request it had just received in *pendingDNS* in its state and sends out a DNS request for the domain of the OAP's introspection endpoint to the DNS server.

We describe what happens when r later receives the response from OAP in the paragraph *Introspection Response* below.

Non-Interactive Login In this case, `script_client_index` has sent an HTTPS POST request for the path `/passwordLogin` to r containing a domain of an OAP, a username and a user's password in its body. Next, r constructs an HTTPS POST request to the token endpoint of the OAP. This request contains the username and the user's password in its body and if r knows an OAuth client secret for the OAP, the request contains an HTTP header with r's OAuth client id and OAuth client secret. r saves all information belonging to this new request and the (incoming) request r has just received in the subterm *pendingDNS* in r's state and sends out a DNS request for the domain of the OAP's token endpoint to the DNS server.

We describe what happens when r later receives the response from the OAP in the paragraph *Token Response* below.

Client Credentials Grant When r receives a `TRIGGER` message (which models that r nondeterministically starts an OAuth flow in the client credentials grant), r first nondeterministically

selects a domain of an OAP. Then, r constructs an HTTPS POST request to the token endpoint of the OAP. This request contains an HTTP header with r's OAuth client id and OAuth client secret.[4] r saves all information belonging to this (prepared) request in *pendingDNS* and sends out a DNS request for the domain of the OAP's token endpoint to the DNS server.

We describe what happens when r later receives the response from OAP in the paragraph *Token Response* below.

Token Response When r receives an encrypted HTTP response that matches a record in the subterm *pendingRequests* of its state and belongs to a request for an access token from an OAP (according to the information recorded in *pendingRequests*), then r extracts the access token and prepares an HTTPS request to the OAP's introspection endpoint containing the access token. r saves all information belonging to this new request in *pendingDNS*. Further, r also stores selected information, which is passed along in r's state in the corresponding record of the incoming request, such as the IP address of the sender and the HTTPS response key of the request which initiated r's request for the access token before. Then, r sends out a DNS request for the domain of the OAP's introspection endpoint to the DNS server.

Introspection Response When r receives an encrypted HTTP response that matches a record in the subterm *pendingRequests* in its state and this record belongs to a request to an OAP's introspection endpoint, r checks whether the response belongs to a flow in client credentials grant (according to the record). If that is the case, r stops. Otherwise, r nondeterministically proceeds with either an authorization flow or an authentication flow:

- If authorization is selected, r retrieves the protected resource from the OAP's response and sends out an HTTPS response to the IP address recorded in the record in *pendingRequests* (which contains the IP address of the browser, which initially sent either user credentials, an authorization code, or an access token).

- Else, authentication is selected. Now, if the response does not contain r's OAuth client id, r stops. Otherwise, r retrieves the user id from the response and nondeterministically chooses a fresh nonce as a service token. r records in its state that the service token belongs to the user identified by the user id at the OAP. Now, r sends out a response (as above) which contains the service token in a cookie.

In both cases, r replies with the script *script_client_index*, which provides arbitrary links and the possibility to start a new OAuth flow (see above).

This concludes the description of the behaviour of a client.

[4] In our model, r may even construct such a request if r does not have an OAuth client secret for the OAP. In this case, the symbol $\perp$ is placed in this header instead of an OAuth client secret. The OAP, however, will drop such a request, as it is not authenticated.

Formal description

We now provide the formal definition of r as an atomic DY process (I^r, Z^r, R^r, s_0^r). As mentioned, we define $I^r = \mathsf{addr}(r)$. Next, we define the set Z^r of states of r and the initial state s_0^r of r.

Definition 52. An *OAP registration record* is a term of the form

$$\langle \mathit{tokenEndpoint}, \mathit{authorizationEndpoint}, \mathit{introspectionEndpoint}, \mathit{clientId}, \mathit{clientPassword} \rangle$$

with $\mathit{tokenEndpoint}$, $\mathit{authorizationEndpoint}$, $\mathit{introspectionEndpoint} \in \mathsf{URLs}$, $\mathit{clientId} \in \mathbb{S}$, and $\mathit{clientPassword} \in \mathcal{N}$.

An *OAP registration record for an OAuth provider i at a client r* is an OAP registration record with $\mathit{tokenEndpoint}.\mathtt{host}$, $\mathit{authorizationEndpoint}.\mathtt{host}$, $\mathit{introspectionEndpoint}.\mathtt{host} \in \mathsf{dom}(i)$, $\mathit{clientId} = \mathsf{clientID}(r, i)$, and $\mathit{clientPassword} = \mathsf{secretOfClient}(r, i)$.

Definition 53. A *state $s \in Z^r$ of a client r* is a term of the form

$$\langle \mathit{DNSAddress}, \mathit{oaps}, \mathit{serviceTokens}, \mathit{loginSessions}, \\ \mathit{keyMapping}, \mathit{tlskeys}, \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{corrupt} \rangle$$

where $\mathit{DNSAddress} \in \mathsf{IPs}$, $\mathit{oaps} \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary of OAP registration records, $\mathit{serviceTokens} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{loginSessions} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary of login session records, $\mathit{keyMapping} \in [\mathbb{S} \times \mathcal{N}]$, $\mathit{tlskeys} = \mathit{tlskeys}^r$, $\mathit{pendingDNS} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{pendingRequests} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$.

An *initial state s_0^r of r* is a state of r with $s_0^r.\mathtt{oaps}$ being a dictionary that maps each domain of all OAuth providers i to an OAP registration record for i at r, $s_0^r.\mathtt{serviceTokens} = s_0^r.\mathtt{loginSessions} = \langle\rangle$, $s_0^r.\mathtt{corrupt} = \bot$, and $s_0^r.\mathtt{keyMapping}$ is the same as the keymapping for browsers above.

Algorithm B.4 specifies the relation R^r. In several places throughout this algorithm we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 6). Table B.2 shows a list of all placeholders used.

B.1.10. OAuth Providers

An OAuth provider $i \in \mathsf{OAPs}$ is a web server modeled as an atomic process (I^i, Z^i, R^i, s_0^i) with the addresses $I^i := \mathsf{addr}(i)$. Its initial state s_0^i contains a list of its domains and (private) TLS keys, the paths for the endpoints (authorization and token), a list of users, a list of clients, and information about the corruption status (initially, the OAP is not corrupted). Besides this, the full state of i further contains a list of issued authorization codes and access tokens.

Once the OAP becomes corrupted (when it receives the message $\mathtt{corrupt}$), it starts collecting all input messages and nondeterministically sending out whatever messages are derivable from its state.

Placeholder	Usage
ν_1	HTTP request nonce
ν_2	lookup key for pending DNS entry
ν_3	service token
ν_4	HTTPS response key
ν_5	HTTP request nonce
ν_6	lookup key for pending DNS entry
ν_7	CSRF token
ν_8	login session cookie
ν_9	HTTP request nonce
ν_{10}	lookup key for pending DNS entry
ν_{11}	HTTP request nonce
ν_{12}	lookup key for pending DNS entry
ν_{13}	HTTP request nonce
ν_{14}	lookup key for pending DNS entry

Table B.2. List of placeholders used in the client algorithm.

Otherwise, OAPs react to three types of requests:

Requests to the authorization endpoint path: In this case, the OAP expects a POST request containing valid user credentials. If the user credentials are not supplied, or the request is not a POST request, the answer contains a script which shows a form to the user to enter her user credentials. In our model, the script just extracts the user credentials from the browser and sends a request to the OAP containing the user credentials and any OAuth parameters contained in the original request (e.g., the intended redirect URI).

If the OAP received a POST request with valid user credentials, it checks the contained client identifier against its own list of clients. If the client identifier is unknown, the OAP aborts. Otherwise, it ensures that the redirect URI, if contained in the request, is valid. For this, it checks the list of redirect URIs stored along with the client identifier. If none of the redirect URIs match the redirect URI presented in the request (see “Matching Redirect URIs” below), the OAP aborts. If no redirect URI is provided in the request, the first URI in the list of redirect URIs is chosen as the redirect URI.

Now the OAP creates a new authorization code and saves this code together with the client identifier and the redirect URI (if provided in the request) to the list of authorization codes.

Now, if the response type parameter in the request is “code”, the OAP issues a `Location` redirect header to the redirect URI, appending (as parameters) the newly created authorization code and the state (if provided in the request).

If the reponse type is “token”, the OAP redirects the browser to the redirect URI, but appends the authorization code, the state (if provided) and a fixed string (containing the token type, which is “bearer”) to the hash of the redirect URI.

Requests to the token endpoint path: Requests to the token endpoint path are only accepted by the OAP if they are POST requests. The OAP then checks that the request either contains a valid client ID, provided as a parameter, or a pair of client ID and client password in a basic authentication header.

If the grant type parameter is *authorization code*, then the OAP checks that the authorization code delivered to it is contained in the list of codes. It checks that the client ID and redirect URI are the same as those stored in the list of codes. It then creates an access token and returns it in the HTTPS response (with token type "bearer").

If the grant type is *password*, the OAP checks the provided username and password and creates an access token as above.

If the grant type is *client credentials*, the OAP checks that the client was authorized with client ID and client password above. If so, it creates an access token as above.

Requests to the introspection endpoint path: In this case, the OAP expects an access token in the parameters of the request. If the access token is valid, the OAP returns the client and user id for which the access token was issued along with the protected resource for this client, user, and OAP.

Formal description

In the following, we first define the (initial) state of i formally and afterwards present the definition of the relation R^i.

To define the initial state, we need to add a list of all protected resources that this OAP manages. We therefore define

$$\mathit{srlist}^i := \langle \{\mathsf{resourceOf}(i, c, u) \mid c \in \mathsf{Clients} \cup \{\bot\}, u \in \mathsf{ID}\} \rangle$$

for some OAP i. (We do not use this term for term manipulations in the algorithm. Instead, this term ensures that the output of the atomic process is derivable from the input.)

Definition 54. A *state* $s \in Z^i$ *of an OAP* i is a term of the form $\langle \mathit{tlskeys}$, srlist, $\mathit{authEndpoint}$, $\mathit{tokenEndpoint}$, $\mathit{introspectEndpoint}$, $\mathit{clients}$, codes, $\mathit{corrupt}\rangle$ where $\mathit{tlskeys} = \mathit{tlskeys}^i$, $\mathit{srlist} = \mathit{srlist}^i$, $\mathit{authEndpoint}$, $\mathit{tokenEndpoint}$, $\mathit{introspectEndpoint} \in \mathbb{S}$, $\mathit{clients} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{codes} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{atokens} \in [\mathcal{N} \times \mathbb{S}]$.

An *initial state* s_0^i *of* i is a state of the form $\langle \mathit{tlskeys}^i, \mathit{srlist}^i, w, x, y, \mathit{clients}^i, \langle\rangle, \langle\rangle, \bot\rangle$ for some strings w, x and y and a dictionary $\mathit{clients}^i$ that for each client r contains an entry of the form $\langle \mathsf{clientID}(r, i), z\rangle$ where z is a sequence of URL terms that may contain the wildcard $*$ (see Definition 11) where for every $u \in^{\langle\rangle} z$ we have that $u.\mathtt{protocol} \equiv \mathtt{S}$, $u.\mathtt{host} \in \mathsf{dom}(r)$, $u.\mathtt{parameters}[\mathtt{iss}] \equiv d$ for some $d \in \mathsf{dom}(i)$, $u.\mathtt{parameters}[\mathtt{client_id}] \equiv \mathsf{clientID}(r, i)$, $u.\mathtt{fragment} \equiv \langle\rangle$, and $u.\mathtt{path} \equiv \mathtt{/redirectionEndpoint}$.

The relation R^i that defines the behavior of the OAP i is defined by Algorithm B.5.

Algorithm B.4 Relation of a Client R^r.

Input: $\langle a, f, m\rangle, s$

1: **if** s'.corrupt $\not\equiv \bot \vee m \equiv$ CORRUPT **then**
2: **let** s'.corrupt := $\langle\langle a, f, m\rangle, s'$.corrupt$\rangle$
3: **let** $m' \leftarrow d_V(s')$
4: **let** $a' \leftarrow$ IPs
5: **stop** $\langle\langle a', a, m'\rangle\rangle$, s'
6: **if** $\exists \langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle \in^{\langle\rangle} s'$.pendingRequests ↪ **such that** $\pi_1(\mathsf{dec_s}(m, \mathit{key})) \equiv$ HTTPResp **then** → Encrypted HTTP response
7: **let** $m' := \mathsf{dec_s}(m, \mathit{key})$
8: **if** m'.nonce $\not\equiv$ *request*.nonce **then**
9: **stop**
10: **remove** $\langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle$ **from** s'.pendingRequests
11: **let** *mode* := $\pi_1(\mathit{reference})$
12: **if** *mode* $\equiv$ code $\vee$ *mode* $\equiv$ password $\vee$ *mode* $\equiv$ client_credentials **then**
13: **let** *oap*, a', f', n', k' **such that** $\langle \mathit{mode}, \mathit{oap}, a', f', n', k'\rangle \equiv \mathit{reference}$ ↪ **if possible; otherwise stop**
14: **let** *token* := m'.body[access_token]
15: **let** *introspectionEndpoint* := s'.oaps[*oap*].introspectionEndpoint
16: **let** *parameters* := *introspectionEndpoint*.parameters
17: **let** *parameters* := *parameters* $+^{\langle\rangle}$ $\langle$token, *token*$\rangle$
18: **let** *host* := *introspectionEndpoint*.domain
19: **let** *path* := *introspectionEndpoint*.path
20: **let** *message* := $\langle$HTTPReq, ν_1, GET, *host*, *path*, *parameters*, $\langle\rangle, \langle\rangle\rangle$
21: **let** s'.pendingDNS[ν_2] := $\langle\langle$introspect, *mode*, *oap*, a', f', n', $k'\rangle$, *message*$\rangle$
22: **stop** $\langle\langle s'$.DNSaddress, a, $\langle$DNSResolve, *introspectionEndpoint*.domain, $\nu_2\rangle\rangle\rangle$, s'
23: **else if** *mode* $\equiv$ introspect **then**
24: **let** *resource*, *clientId*, *user* **such that** ↪ $\langle\langle$protected_resource, *resource*$\rangle$, $\langle$client_id, *clientId*$\rangle$, $\langle$user, *user*$\rangle\rangle \equiv m'$.body ↪ **if possible; otherwise stop**
25: **let** *mode*$'$, *oap*, a', f', n', k' **such that** $\langle$introspect, *mode*$'$, *oap*, a', f', n', $k'\rangle \equiv$ *reference* ↪ **if possible; otherwise stop**
26: **if** *mode*$' \equiv$ client_credentials **then**
27: **stop** → In client credential grant mode, no service token is issued.
28: **let** *goal* $\leftarrow$ {authz, authn} → Proceed with authorization or authentication.
29: **if** *goal* $\equiv$ authz **then**
30: **let** *headers* := $\langle\rangle$
31: **else**
32: **if** *clientId* $\equiv s'$.oaps[*oap*].clientId $\vee$ (*clientId* $\equiv \langle\rangle \wedge$ *mode* $\equiv$ password $\wedge$ ↪ s'.oaps[*oap*].clientPassword $\equiv \bot$) **then**
33: **if** *user* $\equiv \langle\rangle$ **then**
34: **stop**
35: **else**
36: **stop**
37: **let** *serviceToken* := ν_3
38: **let** s'.serviceTokens[*serviceToken*] := $\langle$*user*, *oap*$\rangle$
39: **let** *headers* := $\langle\langle$Set-Cookie, $\langle\langle$serviceToken, $\langle$*serviceToken*, $\bot, \bot, \top\rangle\rangle\rangle\rangle\rangle$
40: **let** *headers* := *headers* $+^{\langle\rangle}$ $\langle$ReferrerPolicy, origin$\rangle$
41: **let** $m' := \mathsf{enc_s}(\langle$HTTPResp, n', 200, *headers*, $\langle$script_client_index, $\langle\rangle\rangle\rangle, k')$
42: **stop** $\langle\langle f', a', m'\rangle\rangle, s'$
43: **stop**

```
44: else if m ∈ DNSResponses then   → Successful DNS response
45:     if m.nonce ∉ s.pendingDNS ∨ m.result ∉ IPs
            ↪ ∨ m.domain ≢ s.pendingDNS[m.domain].2.host then
46:         stop
47:     let ⟨reference, request⟩ := s.pendingDNS[m.nonce]
48:     let s′.pendingRequests := s′.pendingRequests +⟨⟩ ⟨reference, request, ν4, m.result⟩
49:     let message := enc_a(⟨request, ν4⟩, s′.keyMapping[request.host])
50:     let s′.pendingDNS := s′.pendingDNS − m.nonce
51:     stop ⟨⟨m.result, a, message⟩⟩, s′
52: else if m ≡ TRIGGER then   → Start Client Credentials Grant
53:     let oapEntry ← s′.oaps
54:     let oap := π1(oapEntry)
55:     let tokenEndpoint := s′.oaps[oap].tokenEndpoint   → tokenEndpoint is a URL
56:     let host := tokenEndpoint.domain
57:     let path := tokenEndpoint.path
58:     let parameters := tokenEndpoint.parameters
59:     let headers := ⟨⟨Authorization, ⟨s′.oaps[oap].clientId, s′.oaps[oap].clientPassword⟩⟩⟩
60:     let message :=
        ↪ ⟨HTTPReq, ν5, POST, host, path, parameters, headers, ⟨⟨grant_type, client_credentials⟩⟩⟩
61:     let s′.pendingDNS[ν6] := ⟨⟨client_credentials, oap, ⊥, ⊥, ⊥, ⊥⟩, message⟩
62:     stop ⟨⟨s′.DNSaddress, a, ⟨DNSResolve, oap.tokenEndpoint.domain, ν6⟩⟩⟩, s′
63: else   → Handle HTTP requests
64:     let m_dec, k, k′, inDomain such that
        ↪ ⟨m_dec, k⟩ ≡ dec_a(m, k′) ∧ ⟨inDomain, k′⟩ ∈ s.tlskeys
        ↪ if possible; otherwise stop
65:     let n, method, path, parameters, headers, body such that
        ↪ ⟨HTTPReq, n, method, inDomain, path, parameters, headers, body⟩ ≡ m_dec
        ↪ if possible; otherwise stop
66:     if path ≡ / then   → Serve index page.
67:         let headers := ⟨⟨ReferrerPolicy, origin⟩⟩
68:         let m′ := enc_s(⟨HTTPResp, n, 200, headers, ⟨script_client_index, ⟨⟩⟩⟩, k)
69:         stop ⟨⟨f, a, m′⟩⟩, s′
70:     else if path ≡ /startInteractiveLogin ∧ method ≡ POST then
71:         if headers[Origin] ≢ ⟨inDomain, S⟩ then   → CSRF protection.
72:             stop
73:         let oap := body
74:         if oap ∉ s′.oaps then
75:             stop
76:         let state := ν7
77:         let mode ← {code, token}
78:         let responseStatus ← {303, 307}
79:         let authEndpoint := s′.oaps[oap].authorizationEndpoint   → authEndpoint is a URL
80:         let authEndpoint.parameters := authEndpoint.parameters +⟨⟩ ⟨response_type, mode⟩
81:         let authEndpoint.parameters := authEndpoint.parameters +⟨⟩
            ↪ ⟨client_id, s′.oaps[oap].clientId⟩
82:         let authEndpoint.parameters := authEndpoint.parameters +⟨⟩ ⟨state, state⟩
```

83: **let** $redirectUri \leftarrow \{\bot, \top\}$
84: **if** $redirectUri \equiv \top$ **then**
85: **let** $tlskey' \leftarrow s'.\mathtt{tlskeys}$ → Choose one of client's domains nondeterministically
86: **let** $host' := \pi_1(tlskey')$
87: **let** $redirectUri := \langle \mathtt{URL}, \mathtt{S}, host', \mathtt{/redirectionEndpoint}, \langle\langle \mathtt{oap}, oap\rangle\rangle, \langle\rangle\rangle$
88: **let** $redirectUri.\mathtt{parameters} := redirectUri.\mathtt{parameters} +^{\langle\rangle} \langle \mathtt{redirect_uri}, mode\rangle$
89: **let** $loginSessionId := \nu_8$
90: **let** $session := \langle loginSessionId, \langle oap, state, mode, redirectUri\rangle\rangle$
91: **let** $s'.\mathtt{loginSessions} := s'.\mathtt{loginSessions} +^{\langle\rangle} session$
92: **let** $headers := \langle\langle \mathtt{Location}, authEndpoint\rangle\rangle$
93: **let** $headers := headers +^{\langle\rangle} \langle \mathtt{Set\text{-}Cookie}, \langle\langle \mathtt{loginSessionId}, \langle loginSessionId, \top, \top, \top\rangle\rangle\rangle\rangle$
94: **let** $headers := headers +^{\langle\rangle} \langle \mathtt{ReferrerPolicy}, \mathtt{origin}\rangle$
95: **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, responseStatus, headers, \bot\rangle, k)$
96: **stop** $\langle\langle f, a, m'\rangle\rangle, s'$
97: **else if** $path \equiv \mathtt{/redirectionEndpoint}$ **then**
98: **let** $loginSessionId := headers[\mathtt{Cookie}][\mathtt{loginSessionId}]$
99: **let** oap, $state$, $mode$, $redirectUri$ **such that** $\langle oap, state, mode, redirectUri\rangle \equiv$
↪ $s'.\mathtt{loginSessions}[loginSessionId]$ **if possible; otherwise stop**
100: **let** $clientId := s'.\mathtt{oaps}[oap].\mathtt{clientId}$
101: **if** $oap \not\equiv parameters[\mathtt{iss}] \vee clientId \not\equiv parameters[\mathtt{client_id}]$ **then**
102: **stop**
103: **if** $mode \equiv \mathtt{code}$ **then** → Continue Authorization Code Grant
104: **if** $parameters[\mathtt{state}] \not\equiv state$ **then**
105: **stop**
106: **let** $code := parameters[\mathtt{code}]$
107: **let** $trHeaders := \langle\rangle$
108: **let** $trBody := \langle\langle \mathtt{grant_type}, \mathtt{authorization_code}\rangle, \langle \mathtt{code}, code\rangle\rangle$
109: **if** $redirectUri \not\equiv \bot$ **then**
110: **let** $trBody := trBody +^{\langle\rangle} \langle \mathtt{redirect_uri}, redirectUri\rangle$
111: **let** $clientPassword := s'.\mathtt{oaps}[oap].\mathtt{clientPassword}$
112: **if** $clientPassword \equiv \bot$ **then**
113: **let** $trBody := trBody +^{\langle\rangle} \langle \mathtt{client_id}, clientId\rangle$
114: **else**
115: **let** $trHeaders := trHeaders +^{\langle\rangle}$
↪ $\langle \mathtt{Authorization}, \langle clientId, clientPassword\rangle\rangle$
116: **let** $te := s'.\mathtt{oaps}[oap].\mathtt{tokenEndpoint}$
117: **let** $message :=$
↪ $\langle \mathtt{HTTPReq}, \nu_9, \mathtt{POST}, te.\mathtt{domain}, te.\mathtt{path}, te.\mathtt{parameters}, trHeaders, trBody\rangle$
118: **let** $s'.\mathtt{pendingDNS}[\nu_{10}] := \langle\langle \mathtt{code}, oap, a, f, n, k\rangle, message\rangle$
119: **stop** $\langle\langle s'.\mathtt{DNSaddress}, a, \langle \mathtt{DNSResolve}, te.\mathtt{domain}, \nu_{10}\rangle\rangle\rangle, s'$
120: **else if** $mode \equiv \mathtt{token}$ **then** → Continue Implicit Grant
121: **let** $headers := \langle\langle \mathtt{ReferrerPolicy}, \mathtt{origin}\rangle\rangle$
122: **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, headers, \langle \mathtt{script_client_implicit}, oap\rangle\rangle, k)$
123: **stop** $\langle\langle f, a, m'\rangle\rangle, s'$
124: **stop**

125: **else if** $path \equiv$ /passwordLogin $\wedge$ $method \equiv$ POST **then**
126: **if** $headers$[Origin] $\not\equiv \langle inDomain, \mathtt{S}\rangle$ **then** → CSRF protection.
127: **stop**
128: **let** oap, $username$, $password$ **such that** $\langle\langle username, oap\rangle, password\rangle \equiv body$
↪ **if possible; otherwise stop**
129: **let** $trHeaders := \langle\rangle$
130: **let** $trBody := \langle\langle$grant_type, password$\rangle$, $\langle$username, $\langle username, oap\rangle\rangle$,
↪ $\langle$password, $password\rangle\rangle$
131: **let** $clientId := s'$.oaps[oap].clientId
132: **let** $clientPassword := s'$.oaps[oap].clientPassword
133: **if** $clientPassword \not\equiv \perp$ **then**
134: **let** $trHeaders := trHeaders +^{\langle\rangle}$ $\langle$Authorization, $\langle clientId, clientPassword\rangle\rangle$
135: **let** $te := s'$.oaps[oap].tokenEndpoint
136: **let** $message := \langle$HTTPReq, ν_{11}, POST, te.domain, te.path, te.parameters, $trHeaders, trBody\rangle$
137: **let** s'.pendingDNS[ν_{12}] $:= \langle\langle$password, $oap, a, f, n, k\rangle, message\rangle$
138: **stop** $\langle\langle s'$.DNSaddress, a, $\langle$DNSResolve, te.domain, $\nu_{12}\rangle\rangle\rangle$, s'
139: **else if** $path \equiv$ /receiveTokenFromImplicitGrant $\wedge$ $method \equiv$ POST **then**
140: **if** $headers$[Origin] $\not\equiv \langle inDomain, \mathtt{S}\rangle$ **then** → CSRF protection.
141: **stop**
142: **let** $loginSessionId := headers$[Cookie][loginSessionId]
143: **let** oap, $state$, $mode$, $redirectUri$ **such that** $\langle oap, state, mode, redirectUri\rangle \equiv$
↪ s'.loginSessions[$loginSessionId$] **if possible; otherwise stop**
144: **let** $token$ **such that** $\langle token, state, oap\rangle \equiv body$ **if possible; otherwise stop**
145: **let** $introspectionEndpoint := s'$.oaps[$oap$].introspectionEndpoint
146: **let** $parameters' := introspectionEndpoint$.parameters
147: **let** $parameters' := parameters' +^{\langle\rangle}$ $\langle$token, $token\rangle$
148: **let** $host := introspectionEndpoint$.domain
149: **let** $path' := introspectionEndpoint$.path
150: **let** $message := \langle$HTTPReq, ν_{13}, GET, $host, path', parameters', \langle\rangle, \langle\rangle\rangle$
151: **let** s'.pendingDNS[ν_{14}] $:= \langle\langle$introspect, implicit, $oap, a, f, n, k\rangle, message\rangle$
152: **stop** $\langle\langle s'$.DNSaddress, a, $\langle$DNSResolve, $introspectionEndpoint$.domain, $\nu_{14}\rangle\rangle\rangle$, s'
153: **stop**

Algorithm B.5 Relation of an OAP R^i.

Input: $\langle a, f, m\rangle, s$

1: **if** $s'.\mathtt{corrupt} \not\equiv \bot \vee m \equiv \mathtt{CORRUPT}$ **then**
2:     **let** $s'.\mathtt{corrupt} := \langle\langle a, f, m\rangle, s'.\mathtt{corrupt}\rangle$
3:     **let** $m' \leftarrow d_V(s')$
4:     **let** $a' \leftarrow \mathsf{IPs}$
5:     **stop** $\langle\langle a', a, m'\rangle\rangle$, s'
6: **let** $s' := s$
7: **let** m_{dec}, k, k', *inDomain* **such that**
    ↪ $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle inDomain, k'\rangle \in s.\mathtt{tlskeys}$
    ↪ **if possible; otherwise stop**
8: **let** n, *method*, *path*, *parameters*, *headers*, *body* **such that**
    ↪ $\langle \mathtt{HTTPReq}, n, method, inDomain, path, parameters, headers, body\rangle \equiv m_{\text{dec}}$
    ↪ **if possible; otherwise stop**
9: **if** $path \equiv s.\mathtt{authEndpoint}$ **then** → Authorization Endpoint.
10:     **if** $method \equiv \mathtt{GET} \vee (method \equiv \mathtt{POST} \wedge (body[\mathtt{username}] \equiv \langle\rangle \vee body[\mathtt{password}] \equiv \langle\rangle))$ **then**
11:         **let** $data := parameters$
12:         **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\langle \mathtt{ReferrerPolicy}, \mathtt{origin}\rangle\rangle, \langle \mathtt{script_oap_form}, data\rangle\rangle, k)$
13:         **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
14:     **else if** $method \equiv \mathtt{POST}$ **then**
15:         **if** $headers[\mathtt{Origin}] \not\equiv \langle inDomain, \mathtt{S}\rangle$ **then** → CSRF protection.
16:             **stop**
17:         **let** $username := body[\mathtt{username}]$
18:         **let** $password := body[\mathtt{password}]$
19:         **let** $clientid := body[\mathtt{client_id}]$
20:         **let** $allowedredirects := s.\mathtt{clients}[clientid]$
21:         **if** $password \not\equiv \mathsf{secretOfID}(username)$ **then**
22:             **stop**
23:         **if** $allowedredirects \equiv \langle\rangle$ **then**
24:             **stop**
25:         **let** $redirecturi := body[\mathtt{redirect_uri}]$
26:         **if** $redirecturi \not\equiv \langle\rangle$ **then**
27:             **if not** $redirecturi \mathrel{\dot\sim} allowedredirects$ **then**
28:                 **stop**
29:         **else**
30:             **let** $redirecturi \leftarrow allowedredirects$ → Take one from list of redir URIs.
31:         **if** $body[\mathtt{response_type}] \equiv \mathtt{code}$ **then** → Create authorization code.
32:             **let** $s'.\mathtt{codes} := s'.\mathtt{codes} +^{\langle\rangle} \langle \nu_1, \langle clientid, body[\mathtt{redirect_uri}], username\rangle\rangle$
33:             **let** $redirecturi.\mathtt{parameters} := redirecturi.\mathtt{parameters} +^{\langle\rangle} \langle \mathtt{code}, \nu_1\rangle$
34:             **let** $redirecturi.\mathtt{parameters} := redirecturi.\mathtt{parameters} +^{\langle\rangle} \langle \mathtt{state}, body[\mathtt{state}]\rangle$
35:             **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 303, \langle\langle \mathtt{Location}, redirecturi\rangle\rangle, \langle\rangle\rangle, k)$
36:             **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
37:         **else** → Assume response type token.
38:             **let** $s'.\mathtt{atokens} := s'.\mathtt{atokens} +^{\langle\rangle} \langle \nu_1, clientid, username\rangle$
39:             **let** $redirecturi.\mathtt{fragment} := redirecturi.\mathtt{fragment} +^{\langle\rangle} \langle \mathtt{access_token}, \nu_1\rangle$
40:             **let** $redirecturi.\mathtt{fragment} := redirecturi.\mathtt{fragment} +^{\langle\rangle} \langle \mathtt{token_type}, \mathtt{bearer}\rangle$
41:             **let** $redirecturi.\mathtt{fragment} := redirecturi.\mathtt{fragment} +^{\langle\rangle} \langle \mathtt{state}, body[\mathtt{state}]\rangle$
42:             **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 303, \langle\langle \mathtt{Location}, redirecturi\rangle\rangle, \langle\rangle\rangle, k)$
43:             **stop** $\langle\langle f, a, m'\rangle\rangle$, s'

44: **else if** $path \equiv s$.tokenEndpoint **then** → Token Endpoint.
45: **if** $method \not\equiv$ POST **then**
46: **stop**
47: **let** $auth := \bot$
48: **let** $clientid := \bot$
49: **if** $body$[client_id] $\not\equiv \langle\rangle$ **then** → Only client ID is provided, no password.
50: **let** $clientid := body$[client_id]
51: **let** $clientinfo := s$.clients[$clientid$]
52: **if** $clientinfo \equiv \langle\rangle \vee \mathsf{secretOfClientID}(clientid, i) \not\equiv \bot$ **then** → Empty client secret allowed?
53: **stop**
54: **else if** $headers$[Authorization].1 $\not\equiv \langle\rangle$ **then**
55: **let** $clientid := headers$[Authorization].1
56: **let** $clientpw := headers$[Authorization].2
57: **if** $\mathsf{secretOfClientID}(clientid, i) \not\equiv clientpw \vee clientpw \equiv \bot$ **then**
58: **stop**
59: **let** $auth := clientid$ → Authentication with client credentials.
60: **if** $body$[grant_type] $\equiv$ authorization_code **then**
61: **if** $clientid \equiv \bot$ **then**
62: **stop**
63: **let** $codeinfo := s$.codes[$body$[code]]
64: **if** $codeinfo \equiv \langle\rangle \vee codeinfo.1 \not\equiv clientid \vee codeinfo.2 \not\equiv body$[redirect_uri] **then**
65: **stop**
66: **let** s'.codes := s'.codes $-$ $body$[code]
67: **let** s'.atokens := s'.atokens $+^{\langle\rangle}$ $\langle \nu_1, clientid, codeinfo.3\rangle$
68: **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \langle\langle \mathtt{access_token}, \nu_1\rangle, \langle \mathtt{token_type}, \mathtt{bearer}\rangle\rangle\rangle, k)$
69: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
70: **else if** $body$[grant_type] $\equiv$ password **then**
71: **let** $username := body$[username]
72: **let** $password := body$[password]
73: **if** $password \not\equiv \mathsf{secretOfID}(username)$ **then**
74: **stop**
75: **let** s'.atokens := s'.atokens $+^{\langle\rangle}$ $\langle \nu_1, clientid, username\rangle$
76: **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \langle\langle \mathtt{access_token}, \nu_1\rangle, \langle \mathtt{token_type}, \mathtt{bearer}\rangle\rangle\rangle, k)$
77: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
78: **else if** $body$[grant_type] $\equiv$ client_credentials **then**
79: **if** $auth \equiv \bot$ **then**
80: **stop**
81: **let** s'.atokens := s'.atokens $+^{\langle\rangle}$ $\langle \nu_1, clientid, \bot\rangle$
82: **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \langle\langle \mathtt{access_token}, \nu_1\rangle, \langle \mathtt{token_type}, \mathtt{bearer}\rangle\rangle\rangle, k)$
83: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
84: **else if** $path \equiv s$.introspectEndpoint **then** → Introspection Endpoint.
85: **if** $method \not\equiv$ GET **then**
86: **stop**
87: **let** $atoken := parameters$[token]
88: **let** $clientid$, $userid$ **such that** $\langle atoken, clientid, userid\rangle \in^{\langle\rangle} s'$.atokens
↪ **if possible; otherwise stop**
89: **let** $secret := \mathsf{resourceOf}(i, clientid, userid)$
90: **let** $body' := \langle\langle \mathtt{protected_resource}, secret\rangle, \langle \mathtt{client_id}, clientid\rangle, \langle \mathtt{user}, userid\rangle\rangle$
91: **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, body'\rangle, k)$
92: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
93: **stop**

B.2. Formal Security Properties

The security properties for OAuth are formally defined as follows.

B.2.1. Authorization

Intuitively, authorization for $\mathcal{OAuthWS}^n$ means that an attacker should not be able to obtain or use a protected resource available to some honest client at an OAP for some user unless certain parties involved in the authorization process are corrupted.

Definition 55 (Authorization Property). Let $\mathcal{OAuthWS}^n$ be an OAuth web system with a network attacker. We say that $\mathcal{OAuthWS}^n$ *is secure w.r.t. authorization* iff for every run ρ of $\mathcal{OAuthWS}^n$, every state (S^j, E^j, N^j) in ρ, every OAP $i \in \mathsf{OAP}$, every $r \in \mathsf{Clients} \cup \{\bot\}$ with r being honest in S^j unless $r = \bot$, every $u \in \mathsf{ID} \cup \{\bot\}$, for $n = \mathsf{resourceOf}(i, r, u)$, n is derivable from the attackers knowledge in S^j (i.e., $n \in d_{\emptyset}(S^j(\mathsf{attacker}))$), it follows that

1. i is corrupted in S^j, or
2. $u \neq \bot$ and (i) the browser b owning u is fully corrupted in S^j or (ii) some $r' \in \mathsf{trustedClients}(\mathsf{secretOfID}(u))$ is corrupted in S^j.

The protected resource n being available to the attacker also models that the attacker can use a service of the OAP i under the name of the user u (e.g., the attacker can post to the Facebook wall of the victim).

B.2.2. Authentication

Intuitively, authentication for $\mathcal{OAuthWS}^n$ means that an attacker should not be able to login at an (honest) client under the identity of a user unless certain parties involved in the login process are corrupted. As explained above, being logged in at a client under some user identity means to have obtained a service token for this identity from the client.

Definition 56 (Authentication Property). Let $\mathcal{OAuthWS}^n$ be an OAuth web system with a network attacker. We say that $\mathcal{OAuthWS}^n$ *is secure w.r.t. authentication* iff for every run ρ of $\mathcal{OAuthWS}^n$, every state (S^j, E^j, N^j) in ρ, every $r \in \mathsf{Clients}$ that is honest in S^j, every $i \in \mathsf{OAP}$, every $g \in \mathsf{dom}(i)$, every $u \in \mathbb{S}$, every client service token of the form $\langle n, \langle u, g \rangle \rangle$ recorded in $S^j(r)$.`serviceTokens`, and n being derivable from the attackers knowledge in S^j (i.e., $n \in d_{\emptyset}(S^j(\mathsf{attacker}))$), then the browser b owning u is fully corrupted in S^j (i.e., the value of *isCorrupted* is `FULLCORRUPT`), some $r' \in \mathsf{trustedClients}(\mathsf{secretOfID}(\langle u, g \rangle))$ is corrupted in S^j, or i is corrupted in S^j.

B.2.3. Session Integrity for Authorization and Authentication

Before we can define the session integrity property for authorization and authentication, we need to define the notion of *Sessions* and, in particular, *OAuth Sessions.* These capture series

of processing steps related to a single OAuth flow. It is important to note that sessions here are not the same as sessions in the web which are usually identified by some session identifier in a cookie.

Notations

In the following, given a finite run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite run $\rho = ((S^0, E^0, N^0), \ldots)$, we denote by Q_i the processing step $(S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$ (with $i \geq 0$ and, for finite runs, $i < n$).

Definition 57 (Emitting Events). Given an atomic process p, an event e, and a finite run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite run $\rho = ((S^0, E^0, N^0), \ldots)$ we say that p *emits* e iff there is a processing step in ρ of the form

$$(S^i, E^i, N^i) \xrightarrow[p \rightarrow E]{} (S^{i+1}, E^{i+1}, N^{i+1})$$

for some $i \geq 0$ and a set of events E with $e \in E$. We also say that p *emits* m iff $e = \langle x, y, m \rangle$ for some addresses x, y.

Sessions and OAuth Sessions

We now define a relation between processing steps. Intuitively, we say that two processing steps are connected if one processing step causes the other. This can happen either directly (i.e., one DY process handles an event output by another process) or indirectly (e.g., a script that was loaded from an earlier message runs in a browser and outputs a new message).

Definition 58 (Connected Processing Steps). We say that two processing steps

$$\begin{aligned} Q_x &= (S^x, E^x, N^x) \xrightarrow[p_x \rightarrow E_{\text{out},x}]{e_{\text{in},x} \rightarrow p_x} (S^{x+1}, E^{x+1}, N^{x+1}) \text{ and} \\ Q_y &= (S^y, E^y, N^y) \xrightarrow[p_y \rightarrow E_{\text{out},y}]{e_{\text{in},y} \rightarrow p_y} (S^{y+1}, E^{y+1}, N^{y+1}) \end{aligned}$$

are *connected* iff (1) $e_{\text{in},y} \in E_{\text{out},x}$, or (2) p_y is a browser, $e_{\text{in},y}$ is a trigger event, the browser p_y selects to run a script (i.e., selects `script` in Line 9 of Algorithm A.11), and the document selected in Line 13 was created as the result of an HTTP(S) message in $E_{\text{out},x}$.

Based on the notion of connected processing steps, we now define sessions to be sequences of connected processing steps.

Definition 59 (Sessions). A *Session (in a run ρ of a web system)* is a sequence of processing steps $(Q_0, \ldots, Q_n)$ or $(Q_0, Q_1, \ldots)$ such that (1) for all Q_i with $i > 0$, Q_i is connected to some processing step in $(Q_0, \ldots, Q_{i-1})$, and (2) all processing steps appear in the same order as in ρ.

We can now define OAuth Sessions. Intuitively, an OAuth session starts when a user expresses her wish to use some identity at some client. Each session can only contain one such request. A session ends when a authorization or log in is complete (which does not necessarily happen in all OAuth Sessions).

Definition 60 (Start and End Processing Steps for OAuth). We write $\mathsf{startsOA}(Q, b, r, i)$ iff in the processing step Q the browser b triggers the script *script_client_index* which selects some domain of i (in Line 6 of Algorithm B.1) and instructs the browser b to send a message to r in Line 15.

We write $\mathsf{endsOA}(Q, b, r, i, t)$ iff the client r in the processing step Q receives an HTTPS response with a body of the form $\langle\langle \mathtt{protected_resource}, t\rangle, \langle \mathtt{client_id}, c\rangle, \langle \mathtt{user}, u\rangle\rangle$ for some terms c and u from i and emits an event in Line 42 of Algorithm B.4 that is addressed to b.

Definition 61 (OAuth Sessions). Let $\mathcal{OAuthWS}^w$ be an OAuth web system with web attackers and ρ be a run of $\mathcal{OAuthWS}^w$. An *OAuth Session in ρ by a browser b with a client r and an OAP i* is an infinite session $(Q_0, Q_1, \ldots)$ or a finite session $(Q_0, \ldots, Q_n)$ in ρ such that $\mathsf{startsOA}(Q_0, b, r, i)$, but there is no $j > 0$, i' such that $\mathsf{startsOA}(Q_j, b, r, i')$. If there are $j > 0$, t such that $\mathsf{endsOA}(Q_j, b, r, i, t)$, then the OAuth Session is finite and $n = j$.

We write $\mathsf{OASessions}(\rho, b, r, i)$ for the set of all OAuth Sessions in ρ by b with the client r and the OAP i.

We now introduce a notation to associate an OAuth Session with the identity that the browser selected during that session. This models the user intention to log in/authorize using a specific identity. This expression of intent can take place in one of two steps, either during the first request in an OAuth Session (in the resource owner password credentials grant) or at a later time when the user logs in at the OAP (in the implicit grant and the authorization code grant).

Definition 62 (Selected Identity in an OAuth Session). Given a run ρ of an an OAuth web system with a web attacker, a browser b, a client r, an OAP i, and an OAuth Session $o \in \mathsf{OASessions}(\rho, b, r, i)$ we write $\mathsf{selected}_{\mathrm{nia}}(o, b, r, \langle u, g\rangle)$ iff b in (the first processing step of) o selected $id \equiv \langle u, g\rangle$ in Line 4 of Algorithm B.1 and selected $interactive \equiv \bot$ in Line 7.

We write $\mathsf{selected}_{\mathrm{ia}}(o, b, r, \langle u, g\rangle)$ iff b in (the first processing step of) o selected $interactive \equiv \top$ in Line 7 and there is some Q' in o such that b triggers the script *script_oap_form* in Q' and selects $\langle u, g\rangle$ in Line 4 of Algorithm B.3 and sends a message out to i.

Session Integrity Property for Authorization

This security property captures that (a) a client should only be authorized to access some resources when the user actually expressed the wish to start an OAuth flow before, and (b) if a user expressed the wish to start an OAuth flow using some honest OAuth provider and a specific identity, then the OAuth flow is never completed with a different identity.

Definition 63 (Session Integrity for Authorization). Let $\mathcal{OAuthWS}^w$ be an OAuth web system with web attackers. We say that $\mathcal{OAuthWS}^w$ *is secure w.r.t. session integrity for authorization* iff for every run ρ of $\mathcal{OAuthWS}^w$, every processing step Q in ρ, every browser b that is honest in Q, every $r \in \mathsf{Clients}$ that is honest in Q, every $i \in \mathsf{OAP}$, every identity $\langle u, g \rangle$, some protected resource t, the following holds true: If $\mathsf{endsOA}(Q, b, r, i, t)$, then

(a) there is an OAuth Session $o \in \mathsf{OASessions}(\rho, b, r, i)$, and

(b) if i is honest in Q then Q is in o and we have that

$$\mathsf{selected}_{\text{ia}}(o, b, r, \langle u, g \rangle) \iff \big(t \equiv \mathsf{resourceOf}(i, r, \langle u, g \rangle)\big)$$

or

$$\mathsf{selected}_{\text{nia}}(o, b, r, \langle u, g \rangle) \iff \big(t \equiv \mathsf{resourceOf}(i, r', \langle u, g \rangle)\big)$$

for some $r' \in \{r, \perp\}$.

Session Integrity Property for Authentication

This security property captures that (a) a user should only be logged in when the user actually expressed the wish to start an OAuth flow before, and (b) if a user expressed the wish to start an OAuth flow using some honest OAuth provider and a specific identity, then user is not logged in under a different identity.

Definition 64 (Session Integrity for Authentication). Let $\mathcal{OAuthWS}^w$ be an OAuth web system with web attackers. We say that $\mathcal{OAuthWS}^w$ *is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{OAuthWS}^w$, every processing step Q_{login} in ρ, every browser b that is honest in Q_{login}, every $r \in \mathsf{Clients}$ that is honest in Q_{login}, every $i \in \mathsf{OAP}$, every identity $\langle u, g \rangle$, the following holds true: If in Q_{login} a service token of the form $\langle n, \langle \langle u', g' \rangle, m \rangle \rangle$ for a domain $m \in \mathsf{dom}(i)$ and some n, u', g' is created in r (in Line 38 of Algorithm B.4) and n is sent to the browser b, then

(a) there is an OAuth Session $o \in \mathsf{OASessions}(\rho, b, r, i)$, and

(b) if i is honest in Q_{login} then Q_{login} is in o and we have that

$$\big(\mathsf{selected}_{\text{ia}}(o, b, r, \langle u, g \rangle) \vee \mathsf{selected}_{\text{nia}}(o, b, r, \langle u, g \rangle)\big) \iff \big(\langle u, g \rangle \equiv \langle u', g' \rangle\big) \ .$$

B.3. Proof of the OAuth Security Theorem

Before we present the proof for Theorem 1, we first provide a high-level proof outline. We then show some general properties of OAuth web systems with a network attacker. Afterwards, we first prove the authentication property and then the authorization property.

B.3.1. Properties of $\mathcal{OAuthWS}^n$

Let $\mathcal{OAuthWS}^n = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ be an OAuth web system with a network attacker. Let ρ be a run of $\mathcal{OAuthWS}^n$. We write $s_x = (S^x, E^x, N^x)$ for the states in ρ.

Definition 65. We say that a term t *is derivably contained in (a term) t' for (a set of DY processes) P (in a processing step $s_i \to s_{i+1}$ of a run $\rho = (s_0, s_1, \ldots)$)* if t is derivable from t' with the knowledge available to P, i.e.,

$$t \in d_\emptyset(\{t'\} \cup \bigcup_{p \in P} S^{i+1}(p))\,.$$

Definition 66. We say that *a set of processes P leaks a term t (in a processing step $s_i \to s_{i+1}$) to a set of processes P'* if there exists a message m that is emitted (in $s_i \to s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \to s_{i+1}$. If we omit P', we define $P' := \mathcal{W} \setminus P$. If P is a set with a single element, we omit the set notation.

Definition 67. We say that an DY process p *created* a message m (at some point) in a run if m is derivably contained in a message emitted by p in some processing step and if there is no earlier processing step where m is derivably contained in a message emitted by some DY process p'.

Definition 68. We say that *a browser b accepted* a message (as a response to some request) if the browser decrypted the message (if it was an HTTPS message) and called the function PROCESSRESPONSE, passing the message and the request (see Algorithm A.10).

Definition 69. We say that an atomic DY process p *knows a term t* in some state $s = (S, E, N)$ of a run if it can derive the term from its knowledge, i.e., $t \in d_\emptyset(S(p))$.

Definition 70. We say that a *script initiated a request r* if a browser triggered the script (in Line 10 of Algorithm A.8) and the first component of the *command* output of the script relation is either `HREF`, `IFRAME`, `FORM`, `XMLHTTPREQUEST`, or `WS_OPEN` such that the browser issues the request r in the same step as a result.

The following lemma captures properties of a client when it uses HTTPS. For example, the lemma says that other parties cannot decrypt messages encrypted by the client.

Lemma 8 (Client messages are protected by HTTPS). If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{OAuthWS}^n$ an honest client r (I) emits an HTTPS request of the form

$$m = \mathsf{enc_a}(\langle req, k\rangle, \mathsf{pub}(k'))$$

(where req is an HTTP request, k is a nonce (symmetric key), and k' is the private key of some other DY process u), and (II) in the initial state s_0 the private key k' is only known to u, and (III) u never leaks k', then all of the following statements are true:

1. There is no state of $\mathcal{OAuthWS}^n$ where any party except for u knows k', thus no one except for u can decrypt *req*.

2. If there is a processing step $s_j \to s_{j+1}$ where the client r leaks k to $\mathcal{W} \setminus \{u, r\}$ there is a processing step $s_h \to s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, r\}$ or r is corrupted in s_j.

3. The value of the `Host` header in *req* is the domain that is assigned the public key $\mathsf{pub}(k')$ in client's keymapping s_0.`keyMapping` (in its initial state).

4. If r accepts a response (say, m') to m in a processing step $s_j \to s_{j+1}$ and r is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, r\}$ prior to s_j, then either u or r created the HTTPS response m' to the HTTPS request m, in particular, the nonce of the HTTP request *req* is not known to any atomic process p, except for the atomic DY processes r and u.

Proof. **(1)** follows immediately from the condition. If k' is initially only known to u and u never leaks k', i.e., even with the knowledge of all nonces (except for those of u), k' can never be derived from any network output of u, k' cannot be known to any other party. Thus, nobody except for u can derive *req* from m.

(2) We assume that r leaks k to $\mathcal{W} \setminus \{u, r\}$ in the processing step $s_j \to s_{j+1}$ without u prior leaking the key k to anyone except for u and r and that the client is not fully corrupted in s_j, and lead this to a contradiction.

The client is honest in s_i. From the definition of the client, we see that the key k is always a fresh nonce that is not used anywhere else. Further, the key is stored in *pendingRequests* (ν_4 in Lines 48f. of Algorithm B.4). The information from *pendingRequests* is not extracted or used anywhere else, except when handling the received messages, where the key is only checked against and used to decrypt the message (Lines 6ff. of Algorithm B.4). Hence, r does not leak k to any other party in s_j (except for u and r). This proves (2).

(3) Per the definition of clients (Algorithm B.4), a `Host` header is always contained in HTTP requests by clients. From Line 49 of Algorithm B.4 we can see that the encryption key for the request *req* was chosen using the `Host` header of the message. It is chosen from the *keyMapping* in client's state, which is never changed during ρ. This proves (3).

(4) An HTTPS response m' that is accepted by r as a response to m has to be encrypted with k. The nonce k is stored by the client in the *pendingRequests* state information (see Line 48 of Algorithm B.4). The client only stores freshly chosen nonces there (i.e., the nonces are not used twice, or for other purposes than sending one specific request). The information cannot be altered afterwards (only deleted) and cannot be read except when the client checks incoming messages. The nonce k is only known to u (which did not leak it to any other party prior to s_j) and r (which did not leak it either, as u did not leak it and r is honest, see (2)). This proves (4). □

On a high level, the following lemma shows that the contents in the list of pending HTTP requests are immutable.

Lemma 9 (Pending DNS messages become pending requests). Let r be some honest client in $\mathcal{OAuthWS}^n$, $\nu \in \mathcal{N}$, $l > 0$ such that (S^l, E^l, N^l) is a state in ρ, and let $ref \in \mathcal{T_N}$, $req \in$ $\mathsf{HTTPRequests}$ such that $S^l(r).\mathtt{pendingDNS} \equiv S^{l-1}(r).\mathtt{pendingDNS} +^{\langle\rangle} \langle \nu, \langle ref, req \rangle\rangle$. Then we have that $\forall l'$: if there exist ref', req', x, $y \in \mathcal{T_N}$ with $req.\mathtt{nonce} \equiv req'.\mathtt{nonce}$ and $\langle ref', req', x, y\rangle \in^{\langle\rangle} S^{l'}(r).\mathtt{pendingRequests}$ then $req \equiv req' \wedge ref \equiv ref'$.

PROOF. We first note that Algorithm B.4 modifies the subterm *pendingDNS* of the client's state only in such a way that entries are appended to or removed from this subterm, but never modified. Entries are appended in Lines 21, 61, 118, 137, and 151. At all these places in the algorithm, an HTTP message term, say *req*, having a fresh (HTTP) nonce, is appended (together with some term *ref*) to the subterm *pendingDNS*. (A processing step executing one of these parts of the algorithm results in the state (S^l, E^l, N^l) of ρ.) Entries are only removed in Line 50. In this part of the algorithm, a sequence $\langle ref'', req'', x, y\rangle$ with x, $y \in \mathcal{T_N}$ and $req'' \equiv req$ and $ref'' \equiv ref$ (which could not have been altered in any processing step) are appended to the subterm *pendingRequests* of client's state (in Line 48). Besides Line 10, where some entry is removed from this subterm, there is no other part of the algorithm that alters *pendingRequests* in any way. Hence, there we cannot have any state $(S^{l'}, E^{l'}, N^{l'})$ of ρ where we have an request in *pendingRequests* with the same (HTTP) nonce but a different req' or a different ref'. □

Lemma 10 (Clients never send requests to themselves). An honest client never sends an HTTP request to any client (including itself), and only sends HTTPS requests to clients that the receiving client cannot decrypt.

PROOF. Honest clients send HTTP requests only in Lines 20, 60, 117, 136, and 150. In all of these cases, they send the HTTPS request to an endpoint configured in the state (in `oaps`). With Definition 53, it follows that the domains to which these requests are sent, are never a domain of a client. All requests are sent over HTTPS, and the "correct" encryption keys (as stored in `keyMapping`) are used (i.e., even if the attacker changes the DNS response such that an HTTPS request is sent to a client, it cannot be decrypted by the client). □

B.3.2. Proof of Authentication

We here want to show that every OAuth web system is secure w.r.t. authentication, and therefore assume that there exists an OAuth web system that is not secure w.r.t. authentication. We then lead this to a contradiction, thereby showing that all OAuth web systems are secure w.r.t. authentication. In detail, we assume:

Assumption 1. There exists an OAuth web system with a network attacker $\mathcal{OAuthWS}^n$, a run ρ of $\mathcal{OAuthWS}^n$, a state (S^j, E^j, N^j) in ρ, some $r \in \mathsf{Clients}$ that is honest in S^j, some $i \in \mathsf{OAP}$

that is honest in S^j, some $g \in \mathsf{dom}(i)$, some $u \in \mathbb{S}$ with the browser b owning u being not fully corrupted in S^j and all $r' \in \mathsf{trustedClients}(\mathsf{secretOfID}(\langle u, g\rangle))$ being honest, some client service token of the form $\langle n, \langle u, g\rangle\rangle$ recorded in $S^j(r)$.`serviceTokens` such that n is derivable from the attackers knowledge in S^j (i.e., $n \in d_\emptyset(S^j(\mathsf{attacker}))$).

To show that this is a contradiction, we first show some lemmas:

Lemma 11 (Attacker does not learn passwords). There exists no $l \leq j$, (S^l, E^l, N^l) being a state in ρ such that $\mathsf{secretOfID}(u) \in d_\emptyset(S^l(\mathsf{attacker}))$.

PROOF. Let $s := \mathsf{secretOfID}(\langle u, g\rangle)$ and $TC := \mathsf{trustedClients}(s)$. Initially, in S^0, s is only contained in $S^0(b)$.`secrets`$[\langle d, \mathtt{S}\rangle]$ for any $d \in \bigcup_{r' \in TC} \mathsf{dom}(r') \cup \mathsf{dom}(i)$ and in no other states (or waiting events). By the definition of the browser, we can see that only scripts loaded from the origins $\langle d, \mathtt{S}\rangle$ can access s. We know that i and all $r' \in TC$ are honest (from the assumption). We therefore have that only the scripts *script_client_index*, *script_client_implicit*, and *script_oap_form* can access s (if loaded from their respective origins) and that the browser does not use or leak s in any other way. *script_client_implicit* does not use any browser secrets. We therefore focus on the remaining two scripts:

script_client_index. If this script was loaded and has access to s, it must have been loaded from origin $\langle d, \mathtt{S}\rangle$ for a domain d of some trusted client, say t $(\in TC)$. If *script_client_index* selects the secret s in Line 13 of Algorithm B.1, we know that it must have selected the id u in Line 4. We therefore know that in Line 14, the browser b is instructed to send (using HTTPS) $\langle u, s\rangle$ to the path `/passwordLogin` at d. If b sends such a request, t is the only party able to decrypt this request. This message is then processed by t according to Lines 125ff. There, username and password are forwarded to some OAP, say i', using an HTTPS POST request. More precisely, this request is sent to the domain of the token endpoint URL contained in the OAP registration record for the domain contained in u. From Definitions 52 and 53 and the fact that this part of the state (of clients) is never changed, we can see that the request is sent to a domain of i, and therefore $i' = i$. (The attacker can also not modify or read this request, see Lemma 8.) The body of the HTTPS POST request sent to i is of the following form:

$$\langle\langle \mathtt{grant_type}, \mathtt{password}\rangle, \langle \mathtt{username}, u\rangle, \langle \mathtt{password}, s\rangle\rangle\,.$$

Such a request can processed by the OAP only in Lines 70ff. of Algorithm B.5. There, the OAP checks s and discards it. Therefore, s does not leak from i, t, or b to the attacker (or any other party).

script_oap_form. If this script was loaded and has access to s, it must have been loaded from origin $\langle d, \mathtt{S}\rangle$ for a domain d of i. This script sends s to d in an HTTPS POST request. If b sends such a request, i is the only party able to decrypt this request. This message is then

processed by i according to Lines 14ff. of Algorithm B.5. There, the OAP i checks s and discards it. Therefore, s does not leak from i or b to the attacker (or any other party).

This proves Lemma 11. □

Lemma 12 (Attacker does not learn authorization codes). There exists no $l \leq j$, (S^l, E^l, N^l) being a state in ρ, $v \in \mathcal{N}$, $y \in \mathcal{T}_{\mathcal{N}}$ such that $v \in d_{\emptyset}(S^l(\mathsf{attacker}))$ and $\langle v, \langle \mathsf{clientID}(r, i), y, u \rangle\rangle \in^{\langle\rangle} S^l(i).\mathtt{codes}$.

Proof. $S^l(i).\mathtt{codes}$ is initially empty and appended to only in Line 32 of Algorithm B.5 (where an authorization code is created). From Line 14ff. it is easy to see that the request which triggers the creation of the authorization code must carry a valid password for the specific identity in the request body. With Lemma 11, we can see that such a request can not come from the attacker, as the attacker does not know the password needed in the request. It can also not originate from an OAP, as OAPs do not send requests. Further, the request can not originate from any corrupted party or an attacker-controlled origin in the honest browser (as otherwise there would be a flow where the attacker would learn the password by sending it to himself, which can be ruled out by Lemma 11). It is also impossible that the request originated from any non-attacker controlled origin in the honest browser: Such a request could be caused by either a Location redirect or a script. (We refer to the following as *.) A Location redirect must have been issued by an honest party (otherwise, the attacker would have learned the password by the time he issued the response, see Lemma 11). There are two occasions where honest parties issue Location redirect headers:

OAP in Lines 35/42 of Algorithm B.5 In this case, an HTTP status code of 303 is sent. While this causes the browser to do a new request, the new request has an empty body in any case.[5]

Client in Line 95 of Algorithm B.4 In this case, a 307 redirect could be issued, causing the browser to preserve the request body. We therefore have to check what could have caused the browser to issue a request that caused this Location redirect response, and what body could be contained in such a request. For clarity, we call the request causing the redirection m. It is clear that m cannot come from the attacker (as it contains the password). It must therefore come from an honest browser. If it was caused by a redirect in the honest browser, (*) applies recursively. Otherwise, there are three scripts that could send such a request to client: *script_client_index*, *script_client_implicit*, and *script_oap_form*. Of these, only *script_client_index* causes a request for the path `/startInteractiveLogin` (which triggers the redirection in Line 95 of Algorithm B.4), which, however, does not contain any secret.

[5] At this point it is important that a 303 redirect is performed, not a 307 redirect. See Line 19 of Algorithm A.10 for details.

A Location redirect can therefore be ruled out as the cause of the request. There are three scripts that could send such a request: *script_client_index*, *script_client_implicit*, and *script_oap_form*. The first two, *script_client_index*, *script_client_implicit*, do not send requests to any OAP (instead, they only send requests to the client that sent the scripts to the browser, OAP does not send these scripts to the browser). The latter script, *script_oap_form*, can send the request. In this (last remaining) case, the OAP responds with a `Location` redirect header in the response, which, among others, carries a URL containing the critical value v (in Line 35). In this case, the browser receives the response, and immediately triggers a new request to the redirection URL. This URL was composed by the OAP using the list of valid redirection URIs from $S^l(i)$.`clients`, a part of the state of i that is not changed during any run. Definition 54 defines how $S^l(i)$.`clients` is initialized: For the client id $c := \mathsf{clientID}(r, i)$, all redirection URLs carry hosts (domains) of r, have the protocol `S` (HTTPS), and contain a query parameter component identifying the OAP i. In the checks in Lines 20ff., it is ensured that in any case, this restriction on domain and protocol applies to the resulting redirection URI (called *redirecturi* in the algorithm) as well. Therefore, the browser's GET request which is triggered by the `Location` header and contains the value v is sent to r over HTTPS.

The client r can process such a GET request only in Lines 66 and 97 of Algorithm B.4. It is clear, that in Line 66, the value v does not leak to the attacker: An honest script is loaded into the browser, which does not use v in any form. If this script causes a request to the attacker (or causes a request which would be redirected to the attacker), the request does not contain v. In particular, v cannot be contained in the `Referer` header, because this is prevented by the Referrer Policy.

In Lines 97ff., v is forwarded to the OAP for checking its validity and retrieving the access token (there is also code for retrieving the access code from the implicit flow in this part of the code, which is not of interest here). When sending the authorization code, it is critical to ensure that v is forwarded to an honest OAP (in particular, i), and not to the attacker. This is ensured by checking the redirection URL parameters, which, as mentioned above, contain a hint for the OAP in use, in this case i. In Line 101 it is checked that the OAP, to which v is eventually sent, is i.

Therefore, we know that v is sent via POST to the honest OAP i. There, it can only be processed in Lines 44ff. Here, it is easy to see that the value v (called *body*[`code`] in the algorithm) is checked. However, the value is never sent out to any other party and therefore does not leak.

We have shown that the value v cannot be known to the attacker, which proves Lemma 12. □

Lemma 13 (Attacker does not learn access tokens). There exists no $l \leq j$, (S^l, E^l, N^l) being a state in ρ, $v \in \mathcal{N}$, such that $v \in d_\emptyset(S^l(\mathsf{attacker}))$ and $\langle v, \mathsf{clientID}(r, i), u\rangle \in^{\langle\rangle} S^l(i).\mathtt{atokens}$.

Proof. Initially, we have $S^0(i).\mathtt{atokens} \equiv \langle\rangle$. $S^l(i).\mathtt{atokens}$ is appended to only in Lines 38, 67, 75, and 81 (where in each an access token is issued) of Algorithm B.5 and not altered in any other way.

In Line 81, a term of the form $\langle *, *, \bot \rangle$ is appended, which is not of the form $\langle v, \mathsf{clientID}(r,i), u\rangle$. In what follows, we distinguish between the lines of Algorithm B.5 were $\langle v, \mathsf{clientID}(r,i), u\rangle$ is created:

Line 38. It is easy to see, that i must have received an HTTPS POST request containing an `Origin` header with one of its HTTPS origins and containing (in its body) a dictionary with the entries $\langle \mathtt{username}, u\rangle$, $\langle \mathtt{password}, \mathsf{secretOfID}(u)\rangle$, and $\langle \mathtt{client_id}, \mathsf{clientID}(r,i)\rangle$. (In this case, $\mathsf{clientID}(r,i) \neq \bot$, and therefore, $r \neq \bot$.) From Lemma 11 it follows that such a request cannot be assembled by the attacker. Also, neither an OAP nor a client sends such a request. Hence, this request must have be sent from a browser. In the browser, only the scripts `script_oap_form` and the attacker script R^{att} can instruct the browser to send such a request. From Lemma 11 we know that the attacker script cannot access $\mathsf{secretOfID}(u)$ (otherwise, there would be a run ρ' in which the attacker script would send $\mathsf{secretOfID}(u)$ to the attacker instead). Hence, this request must originate from a command returned by `script_oap_form` and it must be created by the browser b (which is $\mathsf{ownerOfID}(u)$). This script only sends such a request to its own origin, which must be an HTTPS origin (it would not have access to $\mathsf{secretOfID}(u)$ otherwise). The OAP responds with a `Location` redirect header in the response, which among others, carries a URL containing the critical value v (in Line 42) in the fragment of the URL. In this case, the browser receives the response, and immediately triggers a new request to the redirection URL. This URL was composed by the OAP using the list of valid redirection URIs from $S^l(i).\mathtt{clients}$, a part of the state of i that is not changed during any run. Definition 54 defines how $S^l(i).\mathtt{clients}$ is initialized: For the client id $c := \mathsf{clientID}(r,i)$, all redirection URLs carry hosts (domains) of r, have the protocol `S` (HTTPS), and contain a query parameter component identifying the OAP i. In the checks in Lines 20ff., it is ensured that in any case, this restriction on domain and protocol applies to the resulting redirection URI (called *redirecturi* in the algorithm) as well. Therefore, the browser's GET request which is triggered by the `Location` header and contains the value v in the fragment, is sent to r over HTTPS.

The client r can process such a GET request only in Lines 66 and 97 of Algorithm B.4. It is clear, that in Line 66, the value v does not leak to the attacker: The honest script *script_client_index* is loaded into the browser, which does not use v in any form.

In Lines 97ff., client's algorithm branches into two different flows: (1) The client takes some value from the URL parameters (which do not contain v) and sends it to some process. The client defers its response to the browser and will (later) only send out the response in Lines 37ff. This response, however, does not contain a script and hence, the browser will not be instructed to create any new messages from the resulting document. Hence, v does not leak in this case. (2) The client sends an HTTPS response containing the script *script_client_implicit* (and, in the script's initial state, a domain of i derived from the redirection URL), which takes v from the URL parameters and instructs the

browser to send an HTTPS POST request containing v and the domain of i to the script's (secure) origin at path `/receiveTokenFromImplicitGrant`. The client processes such a request in Lines 139ff. where it forwards v to the OAP for checking its validity. Here, it is critical to ensure that v is forwarded to an honest OAP (in particular, i), and not to the attacker. This is fulfilled since a domain of i is contained in the request's body, and, before forwarding, it is checked that v is only forwarded to this domain.

Therefore, we know that v is sent via GET to the honest OAP i. There, it can only be processed in Lines 84ff. Here, it is easy to see that the value v is never sent out to any other party and therefore does not leak.

Line 67. In this case, i must have received an HTTPS POST request carrying a dictionary in its body containing the entries $\langle \texttt{grant_type}, \texttt{authorization_code}\rangle$ and $\langle \texttt{code}, \mathit{code}\rangle$ with $\mathit{code} \in \mathcal{N}$ such that $\langle \mathit{code}, \langle \mathsf{clientID}(r,i), y, u\rangle\rangle \in^{\langle\rangle} S^{l'}(i).\texttt{codes}$ for some $y \in \mathcal{T}_{\mathcal{N}}$ and $l' \leq l$. (As above, $\mathsf{clientID}(r,i) \neq \bot$, and therefore, $r \neq \bot$.) From Lemma 12 it follows that such a request can neither be constructed by the attacker nor by a browser instructed by the attacker script R^{att}. In a browser, the remaining honest scripts do not instruct the browser to send such a request. (Honest) OAPs do not send such requests. Hence, such a request must have been constructed by an (honest) client. A client prepares such a request only in Lines 108ff. (of Algorithm B.4) and finally sends out this request in Line 51 (after a DNS response). With Lemma 9 and Lemma 8 we know that *reference* contains a term of the form $\langle \texttt{code}, \mathit{oap}, *, *, *, *\rangle$ with $\mathit{oap} \in \mathsf{dom}(i)$ (as the request was sent encrypted for and to i). When the client receives the response from i, it processes the response in Lines 6ff. where it distinguishes between two cases based on the first subterm in *reference*. As we know that this subterm is `code`, we have that the response is processed only in Lines 13ff. The client takes a subterm from the response's body which might contain[6] v in Line 14 and prepares an HTTPS POST request to an URL of i (which is taken from the subterm *oaps* of the client's state and this subterm is never altered and initially configured such that the URLs under the dictionary key *oap* are actually belonging to i). This HTTPS POST request contains v in the parameter `token`. This request is finally sent out this request in Line 51 (after a DNS response) encrypted for and to i.

It is now easy to see that i only processes the request in Lines 84ff. of Algorithm B.5. There, the OAP only checks the parameter `token` against its state and discards it afterwards. Hence, v does not leak.

Line 75. In this case, i must have received an HTTPS POST request carrying a dictionary in its body containing at least the three entries $\langle \texttt{grant_type}, \texttt{password}\rangle$, $\langle \texttt{username}, u\rangle$, and $\langle \texttt{password}, \mathsf{secretOfID}(u)\rangle$. From Lemma 11 it follows that such a request cannot be constructed by the attacker, dishonest scripts in browsers, or any other dishonest party. (Honest) OAPs do not construct such a request. All honest scripts do not instruct a

[6] The subterm actually is v.

browser to send such a request. Hence, the request must have been constructed by an honest client. A client prepares such a request only in Lines 130ff. (of Algorithm B.4) and finally sends out this request in Line 51 (after a DNS response). With Lemma 9 and Lemma 8 we know that *reference* contains a term of the form $\langle \texttt{password}, oap, *, *, *, * \rangle$ with $oap \in \mathsf{dom}(i)$ (as the request was sent encrypted for and to i). When the client receives the response from i, it processes this response in Lines 6ff., where the client distinguishes between two cases based on the first subterm in *reference*. As we know that this subterm is `code`, we have that the response is processed only in Lines 13ff. The client takes a subterm from the response's body which might contain[7] v in Line 14 and prepares an HTTPS POST request to an URL of i (which is taken from the subterm *oaps* of the client's state and this subterm is never altered and initially configured such that the URLs under the dictionary key *oap* are actually belonging to i). This HTTPS POST request contains v in the parameter `token`. This request is finally sent out this request in Line 51 (after a DNS response) encrypted for and to i. It is now easy to see that i only processes the request in Lines 84ff. (of Algorithm B.5). There, the OAP only checks the parameter `token` against its state and discards it afterwards. Hence, v does not leak.

We have shown that the value v cannot be known to the attacker, which proves Lemma 13. □

We can now show that Assumption 1 is a contradiction.

Lemma 14. Assumption 1 is a contradiction.

Proof. The service token $\langle n, \langle u, g \rangle \rangle$ can only be added to the state $S^j(r)$.`serviceTokens` in Line 37 of Algorithm B.4. To get to this point in the algorithm, in Line 25, it is checked that *reference* is a tupel of the form $\langle \texttt{introspect}, mode, g, a', f', n', k' \rangle$. This is taken from the pending requests, where the value is transferred to from the pending DNS subterm (see Lemma 9). Such a term (starting with `introspect`) is added to the `pendingDNS` subterm only in Lines 21 and 151. We can now do a case distinction between these two possibilities to identify the request m' to which the response containing the service token will be sent.

Subterm was added in Line 21. In Line 13, an entry of the form $\langle mode, g, a', f', n', k' \rangle$ must have existed as a reference in the pending HTTP requests, where *mode* is either `code` or `password`.[8] Such entries are created in the following lines:

Line 118. Here, a request m' must have been received which contained a valid authorization code for the identity u at the OAP i.[9] The attacker cannot know such an

[7] The subterm actually is v.

[8] If *mode* was `client_credentials`, no service token is created.

[9] Otherwise, the OAP would not have returned an access token for the identity u. As $g = oap$ is the value stored in the reference, it is also clear that the authorization code was, in fact, sent to i for retrieving the access token, and not to the attacker or another OAP. Also, the request to i was sent over HTTPS, and therefore, Lemma 8 applies.

authorization code (see Lemma 12). The client r does not send requests to itself or to other clients (see Lemma 10), and no OAPs send requests. Therefore, m' must have originated from an honest browser.

Line 137. In this case, a request m' was received which contained a valid username and password combination for u at i. (As above, we know that i was used to verify that information as g is a domain of i, and $oap = g$. Only the honest browser b and some clients know this password (see Lemma 11), but the clients would not send such a request. The request m' was therefore sent from the browser b.

Subterm was added in Line 151. If the subterm $\langle \texttt{introspect}, mode, g, a', f', n', k' \rangle$ was added in this line, the request causing this (m') must have carried a valid access token for the identity u at i. (As above, the access token was sent to i for validation.) The attacker does not know such an access token (see Lemma 13), and other clients or OAPs cannot send m'. Therefore, an honest browser must have sent m'.

We therefore have that in all cases, m' was sent by an honest browser. Further, m' must have been an HTTPS request (by the definition of clients). If the request was sent as the result of an XMLHttpRequest command from a script, that script must have been loaded from the origin $\langle g_r, \mathsf{S} \rangle$ with $g_r \in \mathsf{dom}(r)$. This is a contradiction (there are no honest scripts that use XMLHttpRequest). Otherwise, it was a "regular" request. In this case, the browser tries to load the service token as a document (which will fail). In particular, the service token $\langle n, \langle u, g \rangle \rangle$ never leaks to the attacker.

We therefore know that the attacker cannot know the service token, which is a contradiction to the assumption. □

B.3.3. Proof of Authorization

As above, we assume that there exists an OAuth web system that is not secure w.r.t. authorization and lead this to a contradiction. In the following, some of the lemmas shown in Appendix B.3.2 are used.

Assumption 2. There exists a run ρ of an OAuth web system with a network attacker $\mathcal{OAuthWS}^n$, a state (S^j, E^j, N^j) in ρ, an OAP $i \in \mathsf{OAP}$ that is honest in S^j, a client $r \in \mathsf{Clients} \cup \{\bot\}$ with r being honest in S^j unless $r = \bot$, some $u \in \mathsf{ID} \cup \{\bot\}$, some $n = \mathsf{resourceOf}(i, r, u)$, n being derivable from the attackers knowledge in S^j (i.e., $n \in d_\emptyset(S^j(\mathsf{attacker}))$), and $u = \bot$ or ((i) the browser b owning u is not fully corrupted in S^j and (ii) all $r' \in \mathsf{trustedClients}(\mathsf{secretOfID}(u))$ are honest S^j).

We first show the following lemma:

Lemma 15 (Attacker does not learn client secrets.). There exists no $l \leq j$, (S^l, E^l, N^l) being a state in ρ such that $\mathsf{secretOfClient}(r, i) \in d_\emptyset(S^l(\mathsf{attacker}))$ unless $\mathsf{secretOfClient}(r, i) \equiv \bot$.

PROOF. Following the definition of the initial states of all atomic processes (in particular Definition 53), initially, $\mathsf{secretOfClient}(r,i)$ is only known to r.

The secret is being used and sent out in an HTTPS message in Lines 53ff. of Algorithm B.4 The message is being sent to the token endpoint configured for i, which, according to Definition 52, bears a host name belonging to i. With the definition of *tlskeys* in Definition 53 and Lemma 8 it can be seen that this outgoing HTTP POST request can therefore only be read by the intended receiver, i.

In i, the message cannot be processed in the authentication endpoint, Lines 14 to 43 of Algorithm B.5, since it does not carry an `Origin` header. It can be processed in Lines 44 to 83. It is easy to see that the secret in the message is not used in any outgoing message, neither stored in the OAP's data structures. The message not be processed in Line 84ff., since it is a POST request.

The same applies when the client sends the password in Line 111ff. or Line 131ff. of Algorithm B.4.

Therefore, the secret $\mathsf{secretOfClient}(r,i)$ cannot be known to the attacker. □

Lemma 16. Assumption 2 is a contradiction.

PROOF. At the beginning of each run, the attacker cannot know n (as defined in the initial states). Only the OAP i can send out the protected resource n, in Line 91 of Algorithm B.5. In a state $(S^{l'}, E^{l'}, N^{l'})$ in ρ for some $l' < j$, for i to send out n, an HTTPS request must be received by i which contains, among others, an access token a such that $\langle a, \mathsf{clientID}(r,i), u\rangle \in^{\langle\rangle} S^{l'}(i).\mathtt{atokens}$. We therefore note that for the attacker to learn n, it has to know a. We also note that if r requests n at the OAP i, the attacker cannot read n or a from such messages (see Lemma 8).

We now have to distinguish two cases:

Anonymous Resource, i.e., $u \equiv \bot$. In this case, the access token a was chosen by i in Line 81 of Algorithm B.5. There, a is sent out in response to a request that must have contained the client credentials for r, where the client secret cannot be $\bot$ (see Line 54. With Lemma 15 we see that the attacker cannot send such a request, and therefore, cannot learn a. This implies that the attacker cannot send the request to learn n from i.

User Resource, i.e., $u \not\equiv \bot$. In this case, Lemma 13 shows that it is not possible for the attacker to send a request to learn n.

With this, we have shown that the attacker cannot learn n, and therefore, Assumption 2 is a contradiction. □

B.3.4. Proof of Session Integrity

Before we prove this property, we highlight that in the absence of a network attacker and with the DNS server as defined for $\mathcal{OAuthWS}^w$, HTTP(S) requests by (honest) parties can only be

answered by the owner of the domain the request was sent to, and neither the requests nor the responses can be read or altered by any attacker unless he is the intended receiver. This property is important for the following proof.

We further show the following lemma, which says that an attacker (under the assumption above) cannot learn a state value that is used in a login session between an honest browser, an honest OAP, and an honest client.

Lemma 17 (Third parties do not learn state). Let ρ be a run of an OAuth web system with web attackers $\mathcal{OAuthWS}^w$, (S^j, E^j, N^j) be a state of ρ, $r \in \mathsf{Clients}$ be a client that is honest in S_j, $i \in \mathsf{OAP}$ be an OAP that is honest in S_j, b be a browser that is honest in S_j.

Then there exists no $l \leq j$, with (S^l, E^l, N^l) being a state in ρ, a nonce $\mathit{loginSessionId} \in \mathcal{N}$, a nonce $\mathit{state} \in \mathcal{N}$, a domain $h \in \mathsf{dom}(r)$ of r, terms x, y, x', y', $z \in \mathcal{T}_{\mathcal{N}}$, cookie $c := \langle \mathtt{loginSessionId}, \langle \mathit{loginSessionId}, x', y', z \rangle \rangle$, an atomic DY process $p \in \mathcal{W} \setminus \{b, i, r\}$ such that $\mathit{state} \in d_{\emptyset}(S^l(p))$, $\langle \mathit{loginSessionId}, \langle g, \mathit{state}, x, y \rangle \rangle \in^{\langle\rangle} S^l(r).\mathtt{loginSessions}$ and $\langle h, c \rangle \in^{\langle\rangle} S^l(b).\mathtt{cookies}$.

Proof. To prove Lemma 17, we track where the login session identified by *loginSessionId* is created and used.

We have that $\langle h, c \rangle \in^{\langle\rangle} S^l(b).\mathtt{cookies}$. Login sessions are only created in Line 91 of Algorithm B.4 (and never altered afterwards). After the session identifier *loginSessionId* was chosen, its value is sent over the network to the party that requested the login. We have that for *loginSessionId*, this party must be b because only r can set the cookie c for the domain h in the state of b[10] and Line 91 of Algorithm B.4 is actually the only place where r does so.

Since b is honest, b follows the location redirect contained in the response sent by r. This location redirect contains the *state* (as a URL parameter). The redirect points to some domain of i.[11] The browser therefore sends (among others) *state* to i. Of all the endpoints at i where the request can be received, the authorization endpoint is the only endpoint where *state* could potentially leak to another party. (For all other endpoints, the value is dropped.) If the request is received at the authorization endpoint, *state* is only sent back to b in the initial scriptstate of *script_oap_form*. In this case, the script sends *state* back to i in a POST request to the authorization endpoint. In the steps outlined here, the value $\mathit{client_id} = \mathsf{clientIDOfClient}(r, i)$ is transferred alongside with *state* (and not altered in-between). Now, after receiving *state* and *client_id* in a POST request at the authorization endpoint, i looks up some redirection URI for *client_id*, which, by Definition 54, is some URI at a domain of r. The value *state* is appended to this URI (either as a parameter or in the fragment). The redirection to the redirection URI is then sent to the browser b. Therefore, b now sends a GET request to r.

If *state* is contained in the parameter, then *state* is immediately sent to r where it is compared to the stored login session records but neither stored nor sent out again. In each case, a script is sent back to b. The scripts that r can send out are *script_client_index* and *script_client_implicit*,

[10] We have only web attackers.
[11] This follows from Definition 52 and Definition 53.

$$d^{\text{req}}_{\text{auth}} \rightsquigarrow d^{\text{resp}}_{\text{auth}} \dashrightarrow e^{\text{req}}_{\text{auth}} \dashrightarrow d^{\text{req}}_{\text{cred}} \rightsquigarrow d^{\text{resp}}_{\text{cred}} \dashrightarrow e^{\text{req}}_{\text{cred}} \rightsquigarrow e^{\text{resp}}_{\text{cred}} \dashrightarrow d^{\text{req}}_{\text{intr}} \rightsquigarrow d^{\text{resp}}_{\text{intr}} \dashrightarrow e^{\text{req}}_{\text{intr}} \rightsquigarrow e^{\text{resp}}_{\text{intr}} \dashrightarrow e^{\text{resp}}_{\text{auth}} \tag{B.1}$$

$$e^{\text{req}}_{\text{auth}} \dashrightarrow d^{\text{req}}_{\text{tokn}} \rightsquigarrow d^{\text{resp}}_{\text{tokn}} \dashrightarrow e^{\text{req}}_{\text{tokn}} \rightsquigarrow e^{\text{resp}}_{\text{tokn}} \dashrightarrow d^{\text{req}}_{\text{intr}} \rightsquigarrow d^{\text{resp}}_{\text{intr}} \dashrightarrow e^{\text{req}}_{\text{intr}} \rightsquigarrow e^{\text{resp}}_{\text{intr}} \dashrightarrow e^{\text{resp}}_{\text{auth}} \tag{B.2}$$

$$e^{\text{req}}_{\text{auth}} \dashrightarrow d^{\text{req}}_{\text{intr}} \rightsquigarrow d^{\text{resp}}_{\text{intr}} \dashrightarrow e^{\text{req}}_{\text{intr}} \rightsquigarrow e^{\text{resp}}_{\text{intr}} \dashrightarrow e^{\text{resp}}_{\text{auth}} \tag{B.3}$$

Figure B.1. Events as described in Lemma 18. Here, $e^{\cdot}_{\cdot}$ denotes events containing HTTP(S) messages, $d^{\cdot}_{\cdot}$ denotes events containing DNS messages. (B.1) applies to the resource owner password credentials grant, (B.2) applies to the authorization code grant, and (B.3) applies to the implicit grant.

none of which cause requests that contain *state*. Also, since both scripts are always delivered with a restrictive `Referrer-Policy` header, any requests that are caused by these scripts (e.g., the start of a new login flow) do not contain *state* in the `Referer` header.[12]

If *state* is contained in the fragment, then *state* is not immediately sent to r, but instead, a request without *state* is sent to r. Since this is a GET request, r either answers with an empty response (Lines 39ff. of Algorithm B.4), a response containing *script_client_index* (Lines 66ff.), or a response containing *script_client_implicit* (Line 122). In case of the empty response, *state* is not used anymore by the browser. In case of *script_client_index*, the fragment is not used. (As above, there is no other way in which *state* can be sent out, also because the fragment part of an URL is stripped in the `Referer` header.) In the case of *script_client_implicit* being loaded into the browser, the script sends *state* in the body of an HTTPS request to r (using the path `/receiveTokenFromImplicitGrant`). When r receives this request, it does not send out *state* to any party (see Lines 139ff. of Algorithm B.4).

This shows that *state* cannot be known to any party except for b, i, and r. □

Definition 71. Let $e_1 = \langle a_1, f_1, m_1 \rangle$ and $e_2 = \langle a_2, f_2, m_2 \rangle$ be events with m_1 being a DNS request and m_2 being a DNS response or m_1 being an HTTP(S) request and m_2 being an HTTP(S) response. We say that the events *correspond* to each other if m_1 and m_2 use the same DNS/HTTP(S) message nonce, $a_1 = f_2$ and $a_2 = f_1$, and (for HTTP(S) messages) either both m_1 and m_2 are encrypted or both are not encrypted.

Given a run ρ, and two events e_1 and e_2 where e_1 is emitted in a processing step Q_1 in ρ before e_2 is emitted in a processing step Q_2 in ρ, we write $e_1 \rightsquigarrow e_2$ if e_1 corresponds to e_2 and we write $e_1 \dashrightarrow e_2$ if Q_1 is connected to Q_2.

Lemma 18. Given a run ρ, a client r, and a browser b, if r, in the run ρ, emits an event, say $e^{\text{resp}}_{\text{auth}}$, in Line 42 of Algorithm B.4 that is addressed to b, and b and r are not corrupted at this point in the run, then all of the following statements hold true:

(a) Events of one of the forms shown in Figure B.1 exist in ρ.

[12] Without the Referrer Policy, *state* could leak to a malicious OAP or other parties.

(b) The event $e^{\text{req}}_{\text{auth}}$ was emitted by b and is addressed to r.

(c) Let $e^{\text{resp}}_{\text{intr}} = \langle a^{\text{resp}}_{\text{intr}}, f^{\text{resp}}_{\text{intr}}, m^{\text{resp}}_{\text{intr}} \rangle$ with $f^{\text{resp}}_{\text{intr}}$ being an IP adress of some party, say, i. Then there is a Q_{starts} such that $\mathsf{startsOA}(Q_{\text{starts}}, b, r, i)$ and we have that (1) $d^{\text{req}}_{\text{auth}}$ was emitted in Q_{starts}, or (2) there are events

$$d^{\text{req}}_{\text{strt}} \rightsquigarrow d^{\text{resp}}_{\text{strt}} \dashrightarrow e^{\text{req}}_{\text{strt}} \rightsquigarrow e^{\text{resp}}_{\text{strt}}$$

such that $d^{\text{req}}_{\text{strt}}$ was emitted in Q_{starts} and $e^{\text{resp}}_{\text{strt}}$ was received by r before $e^{\text{req}}_{\text{auth}}$ was received by r.

Proof. **(a)** We have that $e^{\text{resp}}_{\text{auth}} = \langle a^{\text{resp}}_{\text{auth}}, f^{\text{resp}}_{\text{auth}}, m^{\text{resp}}_{\text{auth}} \rangle$ was emitted by r in Line 42 of Algorithm B.4. (We note that $a^{\text{resp}}_{\text{auth}}$ is an address of b.) This requires that r received (and further processed) an HTTPS response in $e^{\text{resp}}_{\text{intr}}$. Also, it is required that (before receiving this event) there is an entry in the state of r in the subterm `pendingRequests` of the form $\mathit{ref} = \langle \mathit{reference}, \mathit{request}, \mathit{key}, f \rangle$ for some terms *request*, *key*, and f. In this subterm, *request*.`nonce` must be the nonce used in the HTTPS response in $e^{\text{resp}}_{\text{intr}}$, and *reference* must be of the form $\langle \mathtt{introspect}, \mathit{mode}', \mathit{oap}, f^{\text{resp}}_{\text{auth}}, a^{\text{resp}}_{\text{auth}}, n', k' \rangle$ where n' is the nonce used in $m^{\text{resp}}_{\text{auth}}$, k' is the key used to encrypt $m^{\text{resp}}_{\text{auth}}$, and *oap* is some domain.

A subterm of the form of *ref* therefore had to be created in `pendingRequests` before. This term is only appended to in Line 48 of Algorithm B.4. There, the message in *request* was sent out because a DNS response with some message nonce n'' was received and in the state of r the following holds true: $\mathtt{pendingDNS}[n''] \equiv \langle \mathit{reference}, \mathit{request} \rangle$. Such entries in `pendingDNS` can only be created when a corresponding DNS request is sent out, which can happen in Lines 20, 60, 117, 136, and 150. We therefore have that the events $d^{\text{req}}_{\text{intr}}$, $d^{\text{resp}}_{\text{intr}}$, and $e^{\text{req}}_{\text{intr}}$ exist and have the mutual relations shown in (B.1), (B.2), and (B.3).

The string `introspect` is set as the first part of *reference* in Lines 151 and 21. We examine these cases separately.

In the case that *reference* was created in Line 151 (where also the second part of *reference* is set to `implicit`), an incoming HTTPS request from $a^{\text{resp}}_{\text{auth}}$, i.e., from b, must have been received. This shows the existence and mutual relations of all events depicted in (B.3) for the implicit grant.

Otherwise, *reference* was created in Line 21. This requires that r must have received an HTTPS response ($e^{\text{resp}}_{\text{cred}}$ or $e^{\text{resp}}_{\text{tokn}}$), that, as above, has a matching entry in `pendingRequests`, which, as above, was created by sending out an HTTPS request, which, again as above, was preceded by a DNS request and response. We therefore have that (in the resource owner password credentials grant) $d^{\text{req}}_{\text{cred}}$, $d^{\text{resp}}_{\text{cred}}$, $e^{\text{req}}_{\text{cred}}$, $e^{\text{resp}}_{\text{cred}}$ or (in the authorization code grant) $d^{\text{req}}_{\text{tokn}}$, $d^{\text{resp}}_{\text{tokn}}$, $e^{\text{req}}_{\text{tokn}}$, $e^{\text{resp}}_{\text{tokn}}$ exist and have the mutual relations shown in‘ (B.1) and (B.2), respectively.

It is further required that another reference term, *reference*$'$ was in `pendingRequests` when

$e^{\text{resp}}_{\text{cred}}$ or $e^{\text{resp}}_{\text{tokn}}$ was received. The term *reference'* must be of the following form:

$$\mathit{reference}' = \langle w, \mathit{oap}, f^{\text{resp}}_{\text{auth}}, a^{\text{resp}}_{\text{auth}}, n', k' \rangle$$

with $w \in \{\texttt{password}, \texttt{code}\}$.

Now, as above, we can check where *reference'* was created as an entry in `pendingDNS`. This can only happen in Line 137 ($w \equiv \texttt{password}$) and 118 ($w \equiv \texttt{code}$). In both cases, an incoming HTTPS request from $a^{\text{resp}}_{\text{auth}}$, i.e., from b, must have been received. This shows the existance and mutual relations of all events depicted in (B.2).

For (B.1), it is easy to see (as above) that $d^{\text{req}}_{\text{auth}}$ and $d^{\text{resp}}_{\text{auth}}$ exist and have the mutual relations as shown.

(b) As already shown above, in all cases, $e^{\text{req}}_{\text{auth}}$ was sent by b to r.

(c) We have that $e^{\text{resp}}_{\text{intr}}$ was received from i. Therefore, $e^{\text{req}}_{\text{intr}}$ must have been sent to i. Therefore, r requested the IP address of some domain of i in $d^{\text{req}}_{\text{intr}}$. This DNS request was created for the domain of a token endpoint which was looked up in an OAP registration record stored under the key *oap*. From Definitions 53 and 52 it follows that *oap* is a domain of i.

As above, we now have to distinguish where the value *reference* is created such that the first part is `introspect`. This can happen in Lines 21 and 151. We examine these cases separately.

– From **(a)** above we have that *reference'* (which contains *oap*) was created as an entry in `pendingDNS` in Line 137 or 118.

  In the case that *reference'* was created in Line 137 we have that the HTTPS request $e^{\text{req}}_{\text{auth}}$ (which was sent by b as shown above) must have been received by r and that this request was a POST request for the path `/passwordLogin`, with a message body *body* such that $\pi_2(\pi_1(\mathit{body})) \equiv \mathit{oap}$, and that contains an `Origin` header for some domain of r. Such a request can only be caused by *script_client_index* loaded into b from some domain of r. Hence, this script selected the domain *oap* in Line 6 of Algorithm B.1 and we have that $\mathsf{startsOA}(Q_{\text{auth}}, b, r, i)$ where Q_{auth} is the processing step that emitted $d^{\text{req}}_{\text{auth}}$.

  In the case that *reference'* was created in Line 118 we have that (*) the HTTPS request $e^{\text{req}}_{\text{auth}}$ must have been received by r and that in this request there is a cookie `loginSessionId` with a value, say, l such that in the state of r (when receiving the request) in the subterm `loginSessions` under the key l there is a sequence with the first element being *oap*.

  Since we have that $e^{\text{req}}_{\text{auth}}$ was sent by b (as shown above) we have that b must have received an HTTP(S) response from r which contains a `Set-Cookie` header for the cookie `loginSessionId` with the value l.[13] We denote the event of this message as $e^{\text{resp}}_{\text{strt}}$. This message must have been created in Line 95 and, in the same processing step, an entry in `loginSessions` under the key l as described above is created in Line 91. (There are no other places where login session entries are created.) We have that the corresponding

[13]This cookie cannot be set by any party except for r and there are no scripts sent out by r that set cookies.

request $e^{\text{req}}_{\text{strt}}$ is a POST request with an `Origin` header for some domain of r, the path `/startInteractiveLogin`, and that the body must be *oap*. As above, such a request can only be caused by *script_client_index* loaded into b from some domain of r. Hence, this script selected the domain *oap* in Line 6 of Algorithm B.1, which output an `HREF`-command to the browser to send $e^{\text{req}}_{\text{strt}}$ to r. This request is preceded by a pair of corresponding DNS messages $d^{\text{req}}_{\text{strt}}$ and $d^{\text{resp}}_{\text{strt}}$ as defined in the browser relation. We therefore have that $\mathsf{startsOA}(Q_{\text{strt}}, b, r, i)$ where Q_{strt} is the processing step that emitted $d^{\text{req}}_{\text{strt}}$.

– In the case that *reference* was created in Line 151 we have the same situation as in (*) and the proof continues exactly as in (*).

This concludes the proof of Lemma 18. □

Lemma 19. Let $\mathcal{OAuthWS}^w$ be an OAuth web system with web attackers, then $\mathcal{OAuthWS}^w$ is secure w.r.t. session integrity for authorization.

Proof. We have to show that for all OAuth web system with web attackers $\mathcal{OAuthWS}^w$, for every run ρ of $\mathcal{OAuthWS}^w$, every processing step Q_{ends} in ρ, every browser b that is honest in Q_{ends}, every $r \in \mathsf{Clients}$ that is honest in Q_{ends}, every $i \in \mathsf{OAP}$, every identity $\langle u, g\rangle$, some protected resource t, the following holds true: If $\mathsf{endsOA}(Q_{\text{ends}}, b, r, i, t)$, then

(a) there is an OAuth Session $o \in \mathsf{OASessions}(\rho, b, r, i)$, and

(b) if i is honest in Q_{ends} then Q_{ends} is in o and we have that

$$\mathsf{selected}_{\text{ia}}(o, b, r, \langle u, g\rangle) \iff \big(t \equiv \mathsf{resourceOf}(i, r, \langle u, g\rangle)\big)$$

or

$$\mathsf{selected}_{\text{nia}}(o, b, r, \langle u, g\rangle) \iff \big(t \equiv \mathsf{resourceOf}(i, r', \langle u, g\rangle)\big)$$

for some $r' \in \{r, \bot\}$.

We can see that Lemma 18 applies, since $\mathsf{endsOA}(Q_{\text{ends}}, b, r, i, t)$ where Q_{ends} is the processing step in which $e^{\text{resp}}_{\text{intr}}$ was received by r from i and $e^{\text{resp}}_{\text{auth}}$ was emitted to b. With Lemma 18 (c) and Definition 61 it immediately follows that there is an OAuth Session $o \in \mathsf{OASessions}(\rho, b, r, i)$.

For part (b), we now show the connection between Q_{ends} and o and show that one of the logical equivalences in (b) hold true. In the following, we therefore have that i is honest.

In Lemma 18 we have already shown the existence of and the relations between the events of one of the forms shown in Figure B.1. For any two events $e_1 \rightsquigarrow e_2$ in Figure B.1, the processing steps where these events where emitted are connected (as i and DNS servers are honest).

Authorization Code Mode. We now show that if the events are structured as shown in (B.2) in Figure B.1 then there also exist events as shown in (B.4) in Figure B.2. (The event $e^{\text{req}}_{\text{auth}}$ is the same in both figures.)

$$
\begin{aligned}
d^{\text{req}}_{\text{strt}} &\dashrightarrow d^{\text{resp}}_{\text{strt}} \dashrightarrow e^{\text{req}}_{\text{strt}} \dashrightarrow e^{\text{resp}}_{\text{strt}} \\
&\dashrightarrow d^{\text{req}}_{\text{aep1}} \dashrightarrow d^{\text{resp}}_{\text{aep1}} \dashrightarrow e^{\text{req}}_{\text{aep1}} \dashrightarrow e^{\text{resp}}_{\text{aep1}} \\
&\dashrightarrow d^{\text{req}}_{\text{aep2}} \dashrightarrow d^{\text{resp}}_{\text{aep2}} \dashrightarrow e^{\text{req}}_{\text{aep2}} \dashrightarrow e^{\text{resp}}_{\text{aep2}} \\
&\dashrightarrow d^{\text{req}}_{\text{auth}} \dashrightarrow d^{\text{resp}}_{\text{auth}} \dashrightarrow e^{\text{req}}_{\text{auth}}
\end{aligned}
\tag{B.4}
$$

$$
\begin{aligned}
d^{\text{req}}_{\text{strt}} &\dashrightarrow d^{\text{resp}}_{\text{strt}} \dashrightarrow e^{\text{req}}_{\text{strt}} \dashrightarrow e^{\text{resp}}_{\text{strt}} \\
&\dashrightarrow d^{\text{req}}_{\text{aep1}} \dashrightarrow d^{\text{resp}}_{\text{aep1}} \dashrightarrow e^{\text{req}}_{\text{aep1}} \dashrightarrow e^{\text{resp}}_{\text{aep1}} \\
&\dashrightarrow d^{\text{req}}_{\text{aep2}} \dashrightarrow d^{\text{resp}}_{\text{aep2}} \dashrightarrow e^{\text{req}}_{\text{aep2}} \dashrightarrow e^{\text{resp}}_{\text{aep2}} \\
&\dashrightarrow d^{\text{req}}_{\text{impl}} \dashrightarrow d^{\text{resp}}_{\text{impl}} \dashrightarrow e^{\text{req}}_{\text{impl}} \dashrightarrow e^{\text{resp}}_{\text{impl}} \\
&\dashrightarrow d^{\text{req}}_{\text{auth}} \dashrightarrow d^{\text{resp}}_{\text{auth}} \dashrightarrow e^{\text{req}}_{\text{auth}}
\end{aligned}
\tag{B.5}
$$

Figure B.2. Structure of run from start to redirection endpoint.

Since we have that $e^{\text{req}}_{\text{auth}}$ exists and was sent by b, the DNS messages $d^{\text{req}}_{\text{auth}}$ and $d^{\text{resp}}_{\text{auth}}$ (as shown) follow immediately. The request $e^{\text{req}}_{\text{auth}}$ contains a session cookie containing a session id, say, l. The request also contains a URI parameter `state` with some value, say, z.[14]

With Lemma 17, we can see that the attacker (or any other party except for i, b, and r) cannot instruct the browser to send $e^{\text{req}}_{\text{auth}}$. Also, r does not instruct the browser to send such a request, and neither does any honest script. The request must therefore have been caused by a redirection contained in an event $e^{\text{resp}}_{\text{aep2}}$ that was sent from i to b (see Line 35 of Algorithm B.5). (The redirection must have included the state parameter in the URI as above.) This requires that an event $e^{\text{resp}}_{\text{aep2}}$ was sent from b to i. (Which, as above, was preceded by DNS messages $d^{\text{req}}_{\text{aep2}}$ and $d^{\text{resp}}_{\text{aep2}}$.) This event must contain an HTTP(S) POST request, with an `Origin` header value of some domain of i, and in the body there must be a dictionary with an entry for the key `client_id` containing the client id $c = \mathsf{clientIDOfClient}(r, i)$, and an entry for the key `state` with the value z. (In this case, $c \neq \perp$.)

This request can only be caused by the script *script_oap_form* because of the `Origin` header value. This script extracted c and z from its initial scriptstate, which was a dictionary with the keys as above.[15] The initial scriptstate must have been sent by i in an event $e^{\text{resp}}_{\text{aep1}}$. Such an event can only be sent out in Line 12 of Algorithm B.5.

The event $e^{\text{resp}}_{\text{aep1}}$, as above, must have been preceded by connected events $d^{\text{req}}_{\text{aep1}}$, $d^{\text{resp}}_{\text{aep1}}$, and $e^{\text{req}}_{\text{aep1}}$. In $e^{\text{req}}_{\text{aep1}}$ the message must be an HTTP(S) request which must have two parameters, first, under the key `state`, the value z, and second, under the key `client_id`, the value l. (These parameters are used as the initial scriptstate for the script *script_oap_form* above.)

Similar to above, with Lemma 17, we have that the event $e^{\text{req}}_{\text{aep1}}$ (and, with that, $d^{\text{req}}_{\text{aep1}}$) must have been caused by a redirect that was sent from r to b. Such a response is only created by r

[14] From the proof of Lemma 18 we follow that $e^{\text{req}}_{\text{auth}}$ must be an HTTPS request for the path `/redirectionEndpoint` containing the parameters `code`, `state`, `iss`, and `client_id`.

[15] This initial scriptstate is never changed if the script runs under the origin of an honest OAP, which it does in this case.

in Line 95 of Algorithm B.4. Since the state value is always chosen freshly, and we have that in this case it is z, the event containing this redirect is $e^{\text{resp}}_{\text{strt}}$.

It is now easy to see that the sequence of processing steps emitting the events in (B.4) and (B.2) is a session (as in Definition 59), say, o. We already know that $\mathsf{startsOA}(Q_{\text{starts}}, b, r, i)$ where Q_{starts} is the processing step in which $d^{\text{req}}_{\text{strt}}$ was emitted. There is no other processing step in o in which the browser b triggers the script *script_client_index*. The processing step Q_{ends} (in which $e^{\text{resp}}_{\text{auth}}$ is emitted) is the only processing step in which r receives a protected resource from i and emits an event in Line 42 of Algorithm B.4. Therefore, o is an OAuth session, and Q_{ends} is in o.

We now show that

$$\mathsf{selected}_{\text{ia}}(o, b, r, \langle u, g\rangle) \iff \big(t \equiv \mathsf{resourceOf}(i, r, \langle u, g\rangle)\big)\ .$$

Iff $\mathsf{selected}_{\text{ia}}(o, b, r, \langle u, g\rangle)$ then we have that b in Q_{start} selected $\mathit{interactive} \equiv \top$ in Line 7 and there is some Q_{select} in o such that b triggers the script *script_oap_form* in Q_{select} and selects $\langle u, g\rangle$ in Line 4 of Algorithm B.3 and sends a message out to i.

We therefore have that Q_{select} is the processing step where $d^{\text{req}}_{\text{aep2}}$ was emitted. (This is the only processing step in which the browser triggers the script *script_oap_form*.) We have that in this step, the browser selected $\langle u, g\rangle$ in Line 4 of Algorithm B.3. Then, and only then, the HTTPS POST request in $e^{\text{req}}_{\text{aep2}}$ contained, in the body, the credentials (username and password) for the identity $\langle u, g\rangle$. From the proof of Lemma 18 we see that in $e^{\text{resp}}_{\text{strt}}$, in the redirection URI, and hence in the URI in $e^{\text{req}}_{\text{aep1}}$, the parameter `response_type` must be `code`. We therefore have that the initial scriptstate of *script_oap_form* in $e^{\text{resp}}_{\text{aep1}}$ contains the entry $\langle \texttt{response_type}, \texttt{code}\rangle$. Now, in $e^{\text{req}}_{\text{aep2}}$, the body also contains the same entry. Therefore, iff i receives $e^{\text{req}}_{\text{aep2}}$, then it creates an entry in the subterm `codes` of its state (in Line 32 of Algorithm B.5) of the form

$$\langle \mathit{code}, \langle c, \mathit{redirecturi}, \langle u, g\rangle\rangle\rangle$$

(where *redirecturi* is some URI and *code* is a freshly chosen nonce).

Then, and only then, $e^{\text{resp}}_{\text{aep2}}$ contains *code* in the parameter `code` of the location redirect URI (which is the URI for the HTTPS request in $e^{\text{req}}_{\text{auth}}$). Client sends (as shown in the proof of Lemma 18) *code* to OAP in $e^{\text{req}}_{\text{tokn}}$. This request contains the body

$$\langle\langle \texttt{grant_type}, \texttt{authorization_code}\rangle, \langle \texttt{code}, \mathit{code}\rangle\rangle\ .$$

Then, and only then, OAP processes $e^{\text{req}}_{\text{tokn}}$ (in Line 67 of Algorithm B.5) and creates an entry in the subterm `atokens` of its state of the form

$$\langle \mathit{atoken}, \langle c, \langle u, g\rangle\rangle\rangle$$

for a freshly chosen nonce *atoken* (as there exists an entry in the subterm `code` of the form

$\langle code, \langle c, redirecturi, \langle u, g\rangle\rangle\rangle$). Then and only then, *atoken* is contained in $e^{\text{resp}}_{\text{tokn}}$. Then and only then, r sends *atoken* to i in $e^{\text{req}}_{\text{intr}}$. (In this request, *atoken* is contained in the URI parameter `token`.)

Iff there is an entry of the form $\langle atoken, \langle c, \langle u, g\rangle\rangle\rangle$ in the subterm `atokens` in the state of i and i receives $e^{\text{req}}_{\text{intr}}$ (containing *atoken* as shown) then i processed $e^{\text{req}}_{\text{intr}}$ in Line 84ff. and emitted an event ($e^{\text{resp}}_{\text{intr}}$) containing $\mathsf{resourceOf}(i, r, \langle u, g\rangle)$.

Implicit Mode. This case is very similar to the authorization code grant above. We therefore only describe the differences between the two grants.

In this case, with the proof of Lemma 18, we have that $e^{\text{req}}_{\text{auth}}$ is an HTTPS POST request to the path `/receiveTokenFromImplicitGrant` with an `Origin` header being some domain of r. Further, as above, $e^{\text{req}}_{\text{auth}}$ contains the state z. This request must have been created in the browser by *script_client_implicit* running under an origin of r. This script retrieves the state value from the fragment of the URI from which the script was loaded. Therefore, there must have been a request, $e^{\text{req}}_{\text{impl}}$ containing such a fragment in the URI. This implies the presence of the events $d^{\text{req}}_{\text{impl}}$, $d^{\text{resp}}_{\text{impl}}$, and $e^{\text{resp}}_{\text{impl}}$.

We can now that Q_{ends} is in o and $\mathsf{selected}_{\text{ia}}(o, b, r, \langle u, g\rangle) \iff \big(t \equiv \mathsf{resourceOf}(i, r, \langle u, g\rangle)\big)$ by applying the same reasoning as above, with the following differences:

- The event $e^{\text{req}}_{\text{impl}}$ takes the role of $e^{\text{req}}_{\text{auth}}$ in the proof above.
- We can show that the sequence of processing steps emitting the events in (B.3) in Figure B.1 and (B.5) in Figure B.2 are the OAuth session o and (as above) that Q_{ends} is in o.
- Where the parameter `response_type` was `code` above, it now is `token`. The same applies to the initial scriptstate of *script_oap_form*.
- Instead of creating *code* in the processing step that emits $e^{\text{resp}}_{\text{aep2}}$, this step now creates an access token *token* (in the same way as the token was created in the authorization grant in the processing step that emits $e^{\text{resp}}_{\text{tokn}}$). The steps $d^{\text{req}}_{\text{tokn}}$, $d^{\text{resp}}_{\text{tokn}}$, $e^{\text{req}}_{\text{tokn}}$, and $e^{\text{resp}}_{\text{tokn}}$ are skipped.
- The redirection URI contained in $e^{\text{resp}}_{\text{aep2}}$ contains an access token instead of an authorization code, and the access token and the state value are contained in the fragment instead of in the parameters.
- As already discussed, $e^{\text{req}}_{\text{auth}}$ was created by the script *script_client_implicit* which relays the access token from the URI fragment to r.

Resource Owner Password Credentials Mode. It is easy to see that the sequence of processing steps emitting the events in (B.1) is a session (as in Definition 59), say, o. In this case, $\mathsf{startsOA}(Q_{\text{starts}}, b, r, i)$ holds true if Q_{starts} is the processing step in which $d^{\text{req}}_{\text{auth}}$ was emitted. As above, o is also an OAuth session, and Q_{ends} is in o.

We now show that

$$\mathsf{selected}_{\text{nia}}(o, b, r, \langle u, g\rangle) \iff \big(t \equiv \mathsf{resourceOf}(i, r', \langle u, g\rangle)\big)$$

for some $r' \in \{r, \bot\}$. Iff $\mathsf{selected}_{\text{nia}}(o, b, r, \langle u, g \rangle)$ then we have that b in Q_{start} selected $id \equiv \langle u, g \rangle$ in Line 4 of Algorithm B.1 and selected *interactive* $\equiv \bot$ in Line 7.

Then and only then, $e^{\text{req}}_{\text{auth}}$ is an HTTPS POST request for the path `/passwordLogin` with an `Origin` header containing some domain of r and with the identity $\langle u, g \rangle$ and the corresponding password, say p, in the body. Then and only then, the body in $e^{\text{req}}_{\text{cred}}$ is of the form

$$\langle \langle \texttt{grant_type}, \texttt{password} \rangle, \langle \texttt{username}, \langle u, g \rangle \rangle, \langle \texttt{password}, p \rangle \rangle \ .$$

Then, and only then, OAP processes $e^{\text{req}}_{\text{cred}}$ (in Line 70ff. of Algorithm B.5) and creates an entry in the subterm `atokens` of its state of the form

$$\langle \mathit{atoken}, \langle c', \langle u, g \rangle \rangle \rangle$$

for a freshly chosen nonce *atoken* (as there exists an entry in the subterm `code` of the form $\langle \mathit{code}, \langle c, \mathit{redirecturi}, \langle u, g \rangle \rangle \rangle$) and for $c' \in \{\mathsf{clientIDOfClient}(r, i), \bot\}$. Then and only then, *atoken* is contained in $e^{\text{resp}}_{\text{cred}}$. Then and only then, r sends *atoken* to i in $e^{\text{req}}_{\text{intr}}$. (In this request, *atoken* is contained in the URI parameter `token`.)

Iff there is an entry of the form $\langle \mathit{atoken}, \langle c', \langle u, g \rangle \rangle \rangle$ in the subterm `atokens` in the state of i and i receives $e^{\text{req}}_{\text{intr}}$ (containing *atoken* as shown) then i processed $e^{\text{req}}_{\text{intr}}$ in Line 84ff. and emitted an event ($e^{\text{resp}}_{\text{intr}}$) containing $\mathsf{resourceOf}(i, r, \langle u, g \rangle)$ if $c' \neq \bot$ and containing $\mathsf{resourceOf}(i, \bot, \langle u, g \rangle)$ otherwise. □

Lemma 20. Let $\mathcal{OAuthWS}^w$ be an OAuth web system with web attackers, then $\mathcal{OAuthWS}^w$ is secure w.r.t. session integrity for authentication.

Proof. We have that r sends a service token to b, and thus, $\mathsf{endsOA}(Q_{\text{login}}, b, r, it)$ for some term t. Since $\mathcal{OAuthWS}^w$ is secure w.r.t. session integrity for authorization, we have that (a) holds true. For (b), we see from Line 84ff. that honest OAPs, at their introspection endpoint, if they send out an HTTPS response, the body of that response is of the form

$$\langle \langle \texttt{protected_resource}, \mathsf{resourceOf}(i'', r'', \langle u'', g'' \rangle) \rangle, \langle \texttt{client_id}, c'' \rangle, \langle \texttt{user}, \langle u'', g'' \rangle \rangle \rangle$$

for any $\langle u'', g'' \rangle$ and some c'', i'', r''. We therefore have that

$$\big(t \equiv \mathsf{resourceOf}(i, r, \langle u, g \rangle)\big) \iff \big(\langle u, g \rangle \equiv \langle u', g' \rangle\big) \ .$$

Since $\mathcal{OAuthWS}^w$ is secure w.r.t. session integrity for authorization, we have that (b) holds true. □

With Lemma 14, Lemma 16, Lemma 19 and Lemma 20 we have proven Theorem 1. ■

C. Analysis of OpenID Connect

In this appendix, we present our model of OpenID Connect, the formalization of the security properties, and the proof of the OpenID Connect Security Theorem (Theorem 2).

C.1. Formal Model of OpenID Connect with a Network Attacker

We start with the full details of our formal model of OIDC which we use to analyze the authentication and authorization properties. This model contains a network attacker. We will later derive from this model a model where the network attacker is replaced by a web attacker. We use this modified model for the session integrity properties.

We model OIDC as a web system (in the sense of Appendix A.3). We call a web system $\mathcal{OIDCWS}^n = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ an *OIDC web system with a network attacker* if it is of the form described in what follows.

C.1.1. Outline

The system $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Net}$ consists of a network attacker process (in Net), a finite set B of web browsers, a finite set RP of web servers for the relying parties, a finite set OP of web servers for the identity providers, with $\mathsf{Hon} := \mathsf{B} \cup \mathsf{RP} \cup \mathsf{OP}$. More details on the processes in $\mathcal{W}$ are provided below. We do not model DNS servers, as they are subsumed by the network attacker. Table C.1 shows the set of scripts $\mathcal{S}$, their string representations that are defined by the mapping script, and the algorithms that define the respective scripts. The set E^0 contains only the trigger events as specified in Appendix A.3.

$s \in \mathcal{S}$	$\mathsf{script}(s)$	defined in
R^{att}	`att_script`	Definition 26
script_rp_index	`script_rp_index`	Algorithm C.1
script_rp_get_fragment	`script_rp_get_fragment`	Algorithm C.2
script_op_form	`script_op_form`	Algorithm C.3

Table C.1. List of scripts in $\mathcal{S}$, their respective string representations, and their definitions.

This outlines $\mathcal{OIDCWS}^n$. We now define the DY processes in $\mathcal{OIDCWS}^n$ and their addresses, domain names, and secrets in more detail.

Algorithm C.1 Relation of $script_rp_index$.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$
1: **let** $switch \leftarrow \{\texttt{auth}, \texttt{link}\}$
2: **if** $switch \equiv \texttt{auth}$ **then**
3: **let** $url := \mathsf{GETURL}(tree, docnonce)$
4: **let** $id \leftarrow ids$
5: **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/startLogin}, \langle\rangle, \langle\rangle\rangle$
6: **let** $command := \langle \texttt{FORM}, url', \texttt{POST}, id, \bot\rangle$
7: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$
8: **else**
9: **let** $protocol \leftarrow \{\texttt{P}, \texttt{S}\}$
10: **let** $host \leftarrow \mathsf{Doms}$
11: **let** $path \leftarrow \mathbb{S}$
12: **let** $fragment \leftarrow \mathbb{S}$
13: **let** $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
14: **let** $url := \langle \texttt{URL}, protocol, host, path, parameters, fragment\rangle$
15: **let** $command := \langle \texttt{HREF}, url, \bot, \bot\rangle$
16: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$

Algorithm C.2 Relation of $script_rp_get_fragment$.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$
1: **let** $url := \mathsf{GETURL}(tree, docnonce)$
2: **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/redirect_ep}, [\texttt{iss} : url.\texttt{parameters}[\texttt{iss}]], \langle\rangle\rangle$
3: **let** $command := \langle \texttt{FORM}, url', \texttt{POST}, url.\texttt{fragment}, \bot\rangle$
4: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$

Algorithm C.3 Relation of $script_op_form$.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$
1: **let** $url := \mathsf{GETURL}(tree, docnonce)$
2: **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/auth2}, \langle\rangle, \langle\rangle\rangle$
3: **let** $formData := scriptstate$
4: **let** $identity \leftarrow ids$
5: **let** $secret \leftarrow secrets$
6: **let** $formData[\texttt{identity}] := identity$
7: **let** $formData[\texttt{password}] := secret$
8: **let** $command := \langle \texttt{FORM}, url', \texttt{POST}, formData, \bot\rangle$
9: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$

C.1.2. Addresses and Domain Names

The set IPs contains for the network attacker in Net, every relying party in RP, every identity provider in OP, and every browser in B a finite set of addresses each. By addr we denote the corresponding assignment from a process to its address. The set Doms contains a finite set of domains for every relying party in RP, every identity provider in OP, and the network attacker in Net. Browsers (in B) do not have a domain.

By addr and dom we denote the assignments from atomic processes to sets of IPs and Doms, respectively.

C.1.3. Keys and Secrets

The set $\mathcal{N}$ of nonces is partitioned into five sets, an infinite sequence N, an finite set K_{TLS}, an finite set K_{sign}, and a finite set $\mathsf{Passwords}$. We thus have

$$\mathcal{N} = \underbrace{N}_{\text{infinite sequence}} \dot\cup \underbrace{K_{\text{TLS}}}_{\text{finite}} \dot\cup \underbrace{K_{\text{sign}}}_{\text{finite}} \dot\cup \underbrace{\mathsf{Passwords}}_{\text{finite}} .$$

These sets are used as follows:

- The set N contains the nonces that are available for each DY process in $\mathcal{W}$ (it can be used to create a run of $\mathcal{W}$).
- The set K_{TLS} contains the keys that will be used for TLS encryption. Let $\mathsf{tlskey}\colon \mathsf{Doms} \to K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p we define $\mathit{tlskeys}^p = \langle\{\langle d, \mathsf{tlskey}(d)\rangle \mid d \in \mathsf{dom}(p)\}\rangle$.
- The set K_{sign} contains the keys that will be used by OPs for signing id tokens. Let $\mathsf{signkey}\colon \mathsf{OP} \to K_{\text{sign}}$ be an injective mapping that assigns a (different) signing key to every OP.
- The set $\mathsf{Passwords}$ is the set of passwords (secrets) the browsers share with the identity providers. These are the passwords the users use to log in at the OPs.

C.1.4. Identities and Passwords

Identities are defined as in the model for OAuth (see Section B.1.4), except that we do not need the mappings $\mathsf{trustedClients}$, $\mathsf{clientID}$, $\mathsf{secretOfClient}$, and $\mathsf{secretOfClientID}$ in OpenID Connect.

C.1.5. Corruption

Just as in OAuth, RPs and OPs can become corrupted. See Section B.1.5 for details.

C.1.6. Network Attacker

The network attacker is defined as in the model for OAuth. See Section B.1.7 for details.

C.1.7. Browsers

Each $b \in \mathsf{B}$ is a web browser as defined in Appendix A.6, with $I^b := \mathsf{addr}(b)$ being its addresses.

To define the inital state, first let $\mathsf{ID}_b := \mathsf{ownerOfID}^{-1}(b)$ be the set of all IDs of b. We then define the set of passwords that a browser b gives to an origin o: If the origin belongs to an OP, then the user's passwords of this OP are contained in the set. To define this mapping in the initial state, we first define for some process p

$$\mathsf{Secrets}^{b,p} = \left\{s \mid b = \mathsf{ownerOfSecret}(s) \wedge (\exists\, i : s = \mathsf{secretOfID}(i) \wedge i \in \mathsf{ID}^p)\right\} .$$

Placeholder	Usage
ν_1	new login session id
ν_2	new HTTP request nonce
ν_3	new HTTP request nonce
ν_4	new service session id
ν_5	new HTTP request nonce
ν_6	new state value
ν_7	new *nonce* value (for the implicit flow)

Table C.2. List of placeholders used in the relying party algorithm.

Then, the initial state s_0^b is defined as follows: the key mapping maps every domain to its public (TLS) key, according to the mapping tlskey; the DNS address is an address of the network attacker; the list of secrets contains an entry $\langle\langle d, \mathtt{S}\rangle, \langle \mathsf{Secrets}^{b,p}\rangle\rangle$ for each $p \in \mathsf{RP} \cup \mathsf{OP}$ and $d \in \mathsf{dom}(p)$; *ids* is $\langle \mathsf{ID}_b\rangle$; *sts* is empty.

C.1.8. Relying Parties

A relying party $r \in \mathsf{RP}$ is a web server modeled as an atomic DY process (I^r, Z^r, R^r, s_0^r) with the addresses $I^r := \mathsf{addr}(r)$. Next, we define the set Z^r of states of r and the initial state s_0^r of r.

Definition 72. A *state* $s \in Z^r$ *of an RP* r is a term of the form

$$\langle \mathit{DNSaddress}, \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{corrupt}, \mathit{keyMapping}, \mathit{tlskeys}, \\ \mathit{sessions}, \mathit{issuerCache}, \mathit{oidcConfigCache}, \mathit{jwksCache}, \mathit{clientCredentialsCache}\rangle$$

with *DNSaddress*, *pendingDNS*, *pendingRequests*, *corrupt*, *keyMapping*, *tlskeys* as in Definition 50, $\mathit{sessions} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{issuerCache} \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{oidcConfigCache} \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{jwksCache} \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, and $\mathit{clientCredentialsCache} \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$.

An *initial state* s_0^r *of* r is a state of r with $s_0^r.\mathtt{corrupt} = \bot$, $s_0^r.\mathtt{keyMapping}$ being the same as the keymapping for browsers above, $s_0^r.\mathtt{tlskeys} = \mathit{tlskeys}^r$, $s_0^r.\mathtt{pendingDNS} = \langle\rangle$, $s_0^r.\mathtt{pendingRequests} = \langle\rangle$, $s_0^r.\mathtt{sessions} = \langle\rangle$, $s_0^r.\mathtt{issuerCache} = \langle\rangle$, $s_0^r.\mathtt{oidcConfigCache} = \langle\rangle$, $s_0^r.\mathtt{jwksCache} = \langle\rangle$, and $s_0^r.\mathtt{clientCredentialsCache} = \langle\rangle$.

The relation of a relying party r, R^r, is based on the generic HTTPS servers (see Appendix A.7). Algorithms C.4–C.9 further define the relation (we only specify those algorithms that differ from or do not exist in the generic server model). Table C.2 shows the list of placeholders used.

C.1.9. Identity Providers

An identity provider $i \in \mathsf{OP}$ is a web server modeled as an atomic process (I^i, Z^i, R^i, s_0^i) with the addresses $I^i := \mathsf{addr}(i)$. Next, we define the set Z^i of states of i and the initial state s_0^i of i.

Algorithm C.4 Relation of a Relying Party R^r: Processing HTTPS Responses.

```
 1: function PROCESS_HTTPS_RESPONSE(m, reference, request, key, a, f, s')
 2:     let session := s'.sessions[reference[session]]
 3:     let id := session[identity]
 4:     let issuer := s'.issuerCache[id]
 5:     if reference[responseTo] ≡ WEBFINGER then
 6:         let wf := m.body
 7:         if wf[subject] ≢ id then
 8:             stop
 9:         if wf[links][rel] ≢ OIDC_issuer then
10:             stop
11:         let s'.issuerCache[id] := wf[links][href]
12:         call START_LOGIN_FLOW(reference[session], s')
13:     else if reference[responseTo] ≡ CONFIG then
14:         let oidcc := m.body
15:         if oidcc[issuer] ≢ issuer then
16:             stop
17:         let s'.oidcConfigCache[issuer] := oidcc
18:         call START_LOGIN_FLOW(reference[session], s')
19:     else if reference[responseTo] ≡ JWKS then
20:         let s'.jwksCache[issuer] := m.body
21:         call START_LOGIN_FLOW(reference[session], s')
22:     else if reference[responseTo] ≡ REGISTRATION then
23:         let s'.clientCredentialsCache[issuer] := m.body
24:         call START_LOGIN_FLOW(reference[session], s')
25:     else if reference[responseTo] ≡ TOKEN then
26:         if token ∈⟨⟩ session[response_type] ∧ useAccessTokenNow ≡ ⊤ then
27:             call USE_ACCESS_TOKEN(reference[session], m.body[access_token], s')
28:         call CHECK_ID_TOKEN(reference[session], m.body[id_token], s')
29:     stop
```

Definition 73. A *state* $s \in Z^i$ *of an OP* i is a term of the form

$$\langle DNSaddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, \\ registrationRequests, clients, records, jwk\rangle$$

with $DNSaddress$, $pendingDNS$, $pendingRequests$, $corrupt$, $keyMapping$, $tlskeys$ as in Definition 50, $registrationRequests \in \mathcal{T}_{\mathcal{N}}$, $clients \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$, $records \in \mathcal{T}_{\mathcal{N}}$, and $jwk \in K_{\text{sign}}$.

An *initial state* s^i_0 *of* i is a state of i with $s^i_0.\mathtt{pendingDNS} = \langle\rangle$, $s^i_0.\mathtt{pendingRequests} = \langle\rangle$, $s^i_0.\mathtt{corrupt} = \bot$, $s^i_0.\mathtt{keyMapping}$ being the same as the keymapping for browsers above,

Placeholder	Usage
ν_1	new authorization code
ν_2, ν_3	new access tokens
ν_4	new client secret

Table C.3. List of placeholders used in the identity provider algorithm.

Algorithm C.5 Relation of a Relying Party R^r: Processing HTTPS Requests.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2:     if m.path ≡ / then   → Serve index page.
3:         let headers := [ReferrerPolicy:origin]
4:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, headers, ⟨script_rp_index, ⟨⟩⟩⟩, k)
5:         stop ⟨⟨f, a, m'⟩⟩, s'
6:     else if m.path ≡ /startLogin ∧ m.method ≡ POST then   → Serve start login request.
7:         if m.headers[Origin] ≢ ⟨m.host, S⟩ then   → CSRF protection.
8:             stop
9:         let id := m.body
10:        let sessionId := ν_1
11:        let session := [startRequest:[message:m, key:k, receiver:a, sender:f], identity:id]
12:        let s'.sessions[sessionId] := session
13:        call START_LOGIN_FLOW(sessionId, s')
14:    else if m.path ≡ /redirect_ep then
15:        let sessionId := m.headers[Cookie][sessionId]
16:        if sessionId ∉ s'.sessions then
17:            stop
18:        let session := s'.sessions[sessionId]
19:        let id := session[id]
20:        let issuer := s'.issuerCache[identity]
21:        if m.parameters[iss] ≢ issuer then
22:            stop
23:        let oidcConfig := s'.oidcConfigCache[issuer]
24:        let responseType := session[response_type]
25:        if responseType ≡ ⟨code⟩ then   → Auth. code mode.
26:            let data := m.parameters
27:        else → Hybrid or implicit mode.
28:            if m.method ≡ GET then
29:                let headers := ⟨⟨ReferrerPolicy, origin⟩⟩
30:                let m' := enc_s(⟨HTTPResp, m.nonce, 200, headers, ⟨script_rp_get_fragment, ⊥⟩⟩, k)
31:                stop ⟨⟨f, a, m'⟩⟩, s'
32:            else
33:                let data := m.body
34:        if data[state] ≢ session[state] then
35:            stop
36:        let s'.sessions[sessionId][redirectEpRequest] := [message:m, key:k, receiver:a, sender:f]
37:        if id_token ∈^⟨⟩ responseType then
38:            if code ∈^⟨⟩ responseType then   → In hybrid mode, only one of two id tokens is checked.
39:                let checkIdTokenNow ← {⊤, ⊥}
40:            else
41:                let checkIdTokenNow := ⊤
42:            if checkIdTokenNow ≡ ⊤ then   → Nondeterministically omit ID token check.
43:                call CHECK_ID_TOKEN(sessionId, data[id_token], s')
44:        let useAccessTokenNow ← {⊤, ⊥}
45:        if token ∈^⟨⟩ responseType ∧ useAccessTokenNow ≡ ⊤ then
46:            call USE_ACCESS_TOKEN(sessionId, m.body[access_token], s')
47:        if code ∈^⟨⟩ responseType then
48:            call SEND_TOKEN_REQUEST(sessionId, m.body[code], s')
49:    stop
```

Algorithm C.6 Relation of a Relying Party R^r: Request to token endpoint.

```
1:  function SEND_TOKEN_REQUEST(sessionId, code, s')
2:      let session := s'.sessions[sessionId]
3:      let identity := session[identity]
4:      let issuer := s'.issuerCache[identity]
5:      let credentials := s'.clientCredentialsCache[issuer]
6:      let headers := []
7:      let body := [grant_type:authorization_code, code:code, redirect_uri:session[redirect_uri]]
8:      let clientId := credentials[client_id]
9:      let clientSecret := credentials[client_secret]
10:     if clientSecret ≡ ⟨⟩ then
11:         let body[client_id] := clientId
12:     else
13:         let headers[Authorization] := ⟨clientId, clientSecret⟩
14:     let url := s'.oidcConfigCache[issuer][token_ep]
15:     let message := ⟨HTTPReq, ν_2, POST, url.domain, url.path, url.parameters, headers, body⟩
16:     call HTTPS_SIMPLE_SEND([responseTo:TOKEN, session:sessionId], message, s')
```

Algorithm C.7 Relation of a Relying Party R^r: Using the access token (no response expected).

```
1:  function USE_ACCESS_TOKEN(sessionId, token, s')
2:      let session := s'.sessions[sessionId]
3:      let identity := session[identity]
4:      let issuer := s'.issuerCache[identity]
5:      let headers := [Authorization : ⟨Bearer, token⟩]
6:      let url := s'.oidcConfigCache[issuer][token_ep]
7:      let url.path ← 𝕊
8:      let message := ⟨HTTPReq, ν_3, POST, url.domain, url.path, url.parameters, headers, ⟨⟩⟩
9:      call HTTPS_SIMPLE_SEND([responseTo:RESOURCE_USAGE, session:sessionId], message, s')
```

s_0^i.tlskeys $= tlskeys^i$, s_0^i.registrationRequests $= \langle\rangle$, s_0^i.clients $= an$, s_0^i.records $= \langle\rangle$, and s_0^i.jwk $=$ signkey(i).

Algorithms C.10 and C.11 define the relation R^i. Again, we only specify algorithms that differ from or do not exist in the generic server model. Table C.3 shows the list of placeholders used.

Algorithm C.8 Relation of a Relying Party R^r: Check ID token.

```
 1: function CHECK_ID_TOKEN(sessionId, id_token, s')
 2:     let session := s'.sessions[sessionId]
 3:     let identity := session[identity]
 4:     let issuer := s'.issuerCache[identity]
 5:     let oidcConfig := s'.oidcConfigCache[issuer]
 6:     let credentials := s'.clientCredentialsCache[issuer]
 7:     let jwks := s'.jwksCache[issuer]
 8:     let data := extractmsg(id_token)
 9:     if data[iss] ≢ issuer then
10:         stop
11:     if data[aud] ≢ credentials[client_id] then
12:         stop
13:     if checksig(id_token, jwks) ≢ ⊤ then
14:         stop
15:     if nonce ∈ session then
16:         if data[nonce] ≢ session[nonce] then
17:             stop
18:     let s'.sessions[sessionId][loggedInAs] := ⟨issuer, data[sub]⟩
19:     let s'.sessions[sessionId][serviceSessionId] := ν_4
20:     let request := session[redirectEpRequest]
21:     let headers := [ReferrerPolicy:origin]
22:     let headers[Set-Cookie] := [serviceSessionId:⟨ν_4, ⊤, ⊤, ⊤⟩]
23:     let m' := enc_s(⟨HTTPResp, request[message].nonce, 200, headers, ok⟩, request[key])
24:     stop ⟨⟨request[sender], request[receiver], m'⟩⟩, s'
```

Algorithm C.9 Relation of a Relying Party R^r: Continuing in the login flow.

```
1: function START_LOGIN_FLOW(sessionId, s')
2:     let redirectUris := {⟨URL, S, d, /redirect_ep, ⟨⟩, ⟨⟩⟩ | d ∈ dom(r)}   → Set of redirect URIs.
3:     let session := s'.sessions[sessionId]
4:     let identity := session[identity]
5:     if identity ∉ s'.issuerCache then
6:         let host := identity.domain
7:         let path := /.wk/webfinger
8:         let parameters := [resource : identity]
9:         let message := ⟨HTTPReq, ν5, GET, host, path, parameters, ⟨⟩, ⟨⟩⟩
10:        call HTTPS_SIMPLE_SEND([responseTo:WEBFINGER, session:sessionId], message, s')
11:    let issuer := s'.issuerCache[identity]
12:    if issuer ∉ s'.oidcConfigCache then
13:        let host := issuer
14:        let path := /.wk/openid-configuration
15:        let message := ⟨HTTPReq, ν5, GET, host, path, [], ⟨⟩, ⟨⟩⟩
16:        call HTTPS_SIMPLE_SEND([responseTo:CONFIG, session:sessionId], message, s')
17:    let oidcConfig := s'.oidcConfigCache[issuer]
18:    if issuer ∉ s'.jwksCache then
19:        let url := oidcConfig[jwks_uri]
20:        let message := ⟨HTTPReq, ν5, GET, url.host, url.path, [], ⟨⟩, ⟨⟩⟩
21:        call HTTPS_SIMPLE_SEND([responseTo:JWKS, session:sessionId], message, s')
22:    if issuer ∉ s'.clientCredentialsCache then
23:        let url := oidcConfig[reg_ep]
24:        let message := ⟨HTTPReq, ν5, POST, url.host, url.path, [], ⟨⟩, [redirect_uris : ⟨redirectUris⟩]⟩
25:        call HTTPS_SIMPLE_SEND([responseTo:REGISTRATION, session : sessionId], message, s')
26:    let credentials := s'.clientCredentialsCache[issuer]
27:    let responseType ← {⟨code⟩, ⟨id_token⟩, ⟨id_token, token⟩, ⟨code, id_token⟩,
                    ↪ ⟨code, token⟩, ⟨code, id_token, token⟩}
28:    let redirectUri ← redirectUris
29:    let data := [response_type:responseType, redirect_uri:redirectUri,
                    ↪ client_id:credentials[client_id], state:ν6]
30:    if code ∉⟨⟩ responseType then   → Implicit flow requires nonce.
31:        let data[nonce] := ν7
32:    let s'.sessions[sessionId] := s'.sessions[sessionId] ∪ data
33:    let authEndpoint := oidcConfig[auth_ep]
34:    let authEndpoint.parameters := authEndpoint.parameters ∪ data
35:    let headers := [Location:authEndpoint, ReferrerPolicy:origin]
36:    let headers[Set-Cookie] := [sessionId:⟨sessionId, ⊤, ⊤, ⊤⟩]
37:    let request := s'.sessions[sessionId][startRequest]
38:    let m' := enc_s(⟨HTTPResp, request[message].nonce, 303, headers, ⊥⟩, request[key])
39:    stop ⟨⟨request[sender], request[receiver], m'⟩⟩, s'
```

Algorithm C.10 Relation of an OP R^i: Processing HTTPS Requests.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2:   if m.path ≡ /.wk/webfinger then
3:     let user, domain such that ⟨user, domain⟩ ≡ m.parameters[resource]
       ↪ ∧⟨user, domain⟩ ∈ ID^i if possible; otherwise stop
4:     let descriptor := [subject:⟨user, domain⟩, links: [rel:OIDC_issuer, href:m.host]]
5:     let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, descriptor⟩, k)
6:     stop ⟨⟨f, a, m'⟩⟩, s'
7:   else if m.path ≡ /.wk/openid-configuration then
8:     let metaData := [issuer:m.host]
9:     let metaData[auth_ep] := ⟨URL, S, m.host, /auth, ⟨⟩, ⟨⟩⟩
10:    let metaData[token_ep] := ⟨URL, S, m.host, /token, ⟨⟩, ⟨⟩⟩
11:    let metaData[jwks_uri] := ⟨URL, S, m.host, /jwks, ⟨⟩, ⟨⟩⟩
12:    let metaData[reg_ep] := ⟨URL, S, m.host, /reg, ⟨⟩, ⟨⟩⟩
13:    let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, metaData⟩, k)
14:    stop ⟨⟨f, a, m'⟩⟩, s'
15:  else if m.path ≡ /jwks then
16:    let m' := enc_s(⟨HTTPResp, m.nonce, 201, ⟨⟩, pub(s'.jwk)⟩, k)
17:    stop ⟨⟨f, a, m'⟩⟩, s'
18:  else if m.path ≡ /reg ∧ m.method ≡ POST then
19:    let s'.registrationRequests := s'.registrationRequests +⟨⟩ ⟨m, k, a, f⟩
20:    stop    → Stop here to let attacker choose the client id.
21:  else if m.path ≡ /auth then
22:    if m.method ≡ GET then
23:      let data := m.parameters
24:    else if m.method ≡ POST then
25:      let data := m.body
26:    let headers := [ReferrerPolicy:origin]
27:    let m' := enc_s(⟨HTTPResp, m.nonce, 200, headers, ⟨script_op_form, data⟩⟩, k)
28:    stop ⟨⟨f, a, m'⟩⟩, s'
29:  else if m.path ≡ /auth2 ∧ m.method ≡ POST ∧ m.headers[Origin] ≡ ⟨m.host, S⟩ then
30:    let identity := m.body[identity]
31:    let password := m.body[password]
32:    if identity.domain ∉ dom(i) ∨ password ≢ secretOfID(identity) then
33:      stop
34:    let responseType := m.body[response_type]
35:    let clientId := m.body[client_id]
36:    let redirectUri := m.body[redirect_uri]
37:    let state := m.body[state]
38:    let nonce := m.body[nonce]
39:    if clientId ∉ s'.clients then
40:      stop
41:    let clientInfo := s'.clients[clientId]
42:    if redirectUri ∉⟨⟩ clientInfo[redirect_uris] then
43:      stop
44:    let record := [client_id:clientId]
45:    let record[redirect_uri] := redirectUri
46:    let record[subject] := identity
47:    let record[issuer] := m.host
48:    let record[nonce] := nonce
49:    let record[code] := ν_1
```

```
50:         let record[access_tokens] := ⟨ν₂, ν₃⟩
51:         let s'.records := s'.records +⟨⟩ record
52:         let responseData := []
53:         if code ∈⟨⟩ responseType then
54:             let responseData[code] := ν₁
55:         if token ∈⟨⟩ responseType then
56:             let responseData[access_token] := ν₂
57:             let responseData[token_type] := bearer
58:         if id_token ∈⟨⟩ responseType then
59:             let idTokenBody := [iss:record[issuer], sub:record[subject],
                                        ↪ aud:record[client_id], nonce:record[nonce]]
60:             let responseData[id_token] := sig(idTokenBody, s'.jwk)
61:         if state ≢ ⟨⟩ then
62:             let responseData[state] := state
63:         if responseType ≡ ⟨code⟩ then  → Authorization Code Flow
64:             let redirectUri.parameters := redirectUri.parameters ∪ responseData
65:         else  → Implicit/Hybrid Flow
66:             if code ∉⟨⟩ responseType ∧ id_token ∈⟨⟩ responseType ∧ nonce ≡ ⟨⟩ then
67:                 stop  → Nonce is required in implicit mode.
68:             let redirectUri.fragment := redirectUri.fragment ∪ responseData
69:         let redirectUri.parameters[iss] := record[issuer]
70:         let m' := enc_s(⟨HTTPResp, m.nonce, 303, ⟨⟨Location, redirectUri⟩⟩, ⟨⟩⟩, k)
71:         stop ⟨⟨f, a, m'⟩⟩, s'
72:     else if m.path ≡ /token ∧ m.method ≡ POST then
73:         if client_id ∈ m.body then  → Only client id is provided, no client secret.
74:             let clientId := m.body[client_id]
75:             let clientSecret := ⟨⟩
76:         else
77:             let clientId := m.headers[Authorization].username
78:             let clientSecret := m.headers[Authorization].password
79:         let clientInfo := s'.clients[clientId]
80:         if clientInfo ≡ ⟨⟩ ∨ clientInfo[client_secret] ≢ clientSecret then
81:             stop
82:         let code := m.body[code]
83:         let record, ptr such that record ≡ s'.records.ptr ∧ record[code] ≡ code ∧ code ≢ ⊥
          ↪ if possible; otherwise stop
84:         if record[client_id] ≢ clientId then
85:             stop
86:         if not (record[redirect_uri] ≡ m.body[redirect_uri]∨
          ↪ (|clientInfo[redirect_uris]| = 1 ∧ redirect_uri ∉ m.body)) then
87:             stop  → If only one redirect URI is registered, it can be omitted.
88:         let s'.records.ptr[code] := ⊥  → Invalidate code
89:         let accessTokenChoice ← {1, 2}
90:         let accessToken := record[access_tokens].accessTokenChoice
91:         let idTokenBody := [iss:record[issuer]]
92:         let idTokenBody[sub] := record[subject]
93:         let idTokenBody[aud] := record[client_id]
94:         let idTokenBody[nonce] := record[nonce]
95:         let id_token := sig(idTokenBody, s'.jwk)
96:         let body := [access_token:accessToken, token_type:bearer, id_token:id_token]
97:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, body⟩, k)
98:         stop ⟨⟨f, a, m'⟩⟩, s'
```

Algorithm C.11 Relation of an OP R^i: Processing other messages.

```
1: function PROCESS_OTHER(m, a, f, s')
2:     let clientId := m   → m is client id chosen by and sent by an attacker process.
3:     if clientId ∈ s'.clients then
4:         stop
5:     let m, k, a, f such that ⟨m, k, a, f⟩ ∈⟨⟩ s'.registrationRequests if possible; otherwise stop
6:     remove ⟨m, k, a, f⟩ from s'.registrationRequests
7:     let redirectUris := m.body[redirect_uris]
8:     let regResponse := [client_id:clientId]
9:     let issueSecret ← {⊤, ⊥}
10:    if issueSecret ≡ ⊤ then
11:        let clientSecret := ν_4
12:        let regResponse[client_secret] := clientSecret
13:    let clientInfo := regResponse
14:    let clientInfo[redirect_uris] := redirectUris
15:    let s'.clients[clientId] := clientInfo
16:    let m' := enc_s(⟨HTTPResp, m.nonce, 201, ⟨⟩, regResponse⟩, k)
17:    stop ⟨⟨f, a, m'⟩⟩, s'
```

C.2. Formal Security Properties

The security properties for OIDC are defined as follows.

C.2.1. Authentication

Intuitively, authentication for $\mathcal{OIDCWS}^n$ means that an attacker should not be able to login at an (honest) RP under the identity of a user unless certain parties involved in the login process are corrupted. As explained in Section 4.5.2, being logged in at an RP under some user identity means to have obtained a service token for this identity from the RP.

Definition 74 (Service Sessions). We say that there is a *service session identified by a nonce n for an identity id at some RP r* in a configuration (S, E, N) of a run ρ of an OIDC web system iff there exists some session id x and a domain $d \in \mathsf{dom}(\mathsf{governor}(id))$ such that $S(r).\mathtt{sessions}[x][\mathtt{loggedInAs}] \equiv \langle d, id \rangle$ and $S(r).\mathtt{sessions}[x][\mathtt{serviceSessionId}] \equiv n$.

Definition 75 (Authentication Property). Let $\mathcal{OIDCWS}^n$ be an OIDC web system with a network attacker. We say that *$\mathcal{OIDCWS}^n$ is secure w.r.t. authentication* iff for every run ρ of $\mathcal{OIDCWS}^n$, every configuration (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}$ owned by b with $\mathsf{governor}(id)$ being an honest OP, every service session identified by some nonce n for id at r, n is not derivable from the attackers knowledge in S (i.e., $n \notin d_\emptyset(S(\mathsf{attacker}))$).

C.2.2. Authorization

Intuitively, authorization for $\mathcal{OIDCWS}^n$ means that an attacker should not be able to obtain or use a protected resource available to some honest RP at an OP for some user unless certain parties involved in the authorization process are corrupted.

Definition 76. We say that a client id c *has been issued to r by i* iff i has sent a response to a registration request from r in Line 17 of Algorithm C.11 and this response contains c in its body under the dictionary key `client_id`.

Definition 77 (Authorization Property). Let $\mathcal{OIDCWS}^n$ be an OIDC web system with a network attacker. We say that *$\mathcal{OIDCWS}^n$ is secure w.r.t. authorization* iff for every run ρ of $\mathcal{OIDCWS}^n$, every configuration (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every $i \in \mathsf{OP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}^i$ owned by b, every nonce n, every term $x \in^{\langle\rangle} S(i).\mathtt{records}$ with $x[\mathtt{subject}] \equiv id$, $n \in^{\langle\rangle} x[\mathtt{access_tokens}]$, and the client id $x[\mathtt{client_id}]$ has been issued by i to r, we have that n is not derivable from the attackers knowledge in S (i.e., $n \notin d_\emptyset(S(\mathsf{attacker}))$).

C.2.3. Session Integrity for Authentication and Authorization

The two session integrity properties capture that an attacker should be unable to forcefully log a user in to some RP. This includes attacks such as CSRF and session swapping.

Session Integrity Property for Authentication

This security property captures that (a) a user should only be logged in when the user actually expressed the wish to start an OIDC flow before, and (b) if a user expressed the wish to start an OIDC flow using some honest identity provider and a specific identity, then user is not logged in under a different identity.

We first need to define notations for the processing steps that represent important events during a flow of an OIDC web system.

Definition 78 (User is logged in). For a run ρ of an OIDC web system with web attacker $\mathcal{OIDCWS}^w$ we say that a browser b was authenticated to an RP r using an OP i and an identity u in a login session identified by a nonce lsid in processing step Q in ρ with

$$Q = (S, E, N) \xrightarrow[r \to E_{\text{out}}]{} (S', E', N')$$

(for some S, S', E, E', N, N') and some event $\langle y, y', m \rangle \in E_{\text{out}}$ such that m is an HTTPS response matching an HTTPS request sent by b to r and we have that in the headers of m there is a header of the form $\langle \mathtt{Set\text{-}Cookie}, [\mathtt{serviceSessionId}{:}\langle \mathit{ssid}, \top, \top, \top \rangle] \rangle$ for some nonce ssid and we have that there is a term g such that $S(r).\mathtt{sessions}[\mathit{lsid}] \equiv g$, $g[\mathtt{serviceSessionId}] \equiv \mathit{ssid}$, and $g[\mathtt{loggedInAs}] \equiv \langle d, u \rangle$ with $d \in \mathsf{dom}(i)$. We then write $\mathsf{loggedIn}^Q_\rho(b, r, u, i, \mathit{lsid})$.

Definition 79 (User started a login flow). For a run ρ of an OIDC web system with web attacker $\mathcal{OIDCWS}^w$ we say that the user of the browser b started a login session identified by a nonce lsid at the RP r in a processing step Q in ρ if (1) in that processing step, the browser b was triggered, selected a document loaded from an origin of r, executed the script $\mathit{script_rp_index}$ in that document, and in that script, executed the Line 7 of Algorithm C.1, and (2) r sends an HTTPS response corresponding to the HTTPS request sent by b in Q and in that response, there is a header of the form $\langle \mathtt{Set\text{-}Cookie}, [\mathtt{sessionId}{:}\langle \mathit{lsid}, \top, \top, \top \rangle] \rangle$. We then write $\mathsf{started}^Q_\rho(b, r, \mathit{lsid})$.

Definition 80 (User authenticated at an OP). For a run ρ of an OIDC web system with web attacker $\mathcal{OIDCWS}^w$ we say that the user of the browser b authenticated to an OP i using an identity u for a login session identified by a nonce lsid at the RP r if there is a processing step Q in ρ with

$$Q = (S, E, N) \to (S', E', N')$$

(for some S, S', E, E', N, N') in which the browser b was triggered, selected a document loaded from an origin of i, executed the script $\mathit{script_op_form}$ in that document, and in that script, (1) in Line 4 of Algorithm C.3, selected the identity u, and (2) we have that the $\mathit{scriptstate}$ of that document, when triggered, contains a nonce s such that $\mathit{scriptstate}[\mathtt{state}] \equiv s$ and $S(r).\mathtt{sessions}[\mathit{lsid}][\mathtt{state}] \equiv s$. We then write $\mathsf{authenticated}^Q_\rho(b, r, u, i, \mathit{lsid})$.

Definition 81 (RP uses an access token). For a run ρ of an OIDC web system with web attacker $\mathcal{OIDCWS}^w$ we say that the RP r uses some access token t in a login session identified by the

nonce $lsid$ established with the browser b at an OP i if there is a processing step Q in ρ with

$$Q = (S, E, N) \rightarrow (S', E', N')$$

(for some S, S', E, E', N, N') in which

(1) r calls the function $\mathsf{USE_ACCESS_TOKEN}$ with the first two parameters being $lsid$ and t,

(2) $S(r).\mathtt{issuerCache}[S(r).\mathtt{sessions}[lsid][\mathtt{identity}]] \in \mathsf{dom}(i)$, and

(3) $\langle \mathtt{sessionid}, \langle lsid, y, z, z' \rangle\rangle \in^{\langle\rangle} S(b).\mathtt{cookies}[d]$ for $d \in \mathsf{dom}(r)$, $y, z, z' \in \mathcal{T}_{\mathcal{N}}$.

We then write $\mathsf{usedAuthorization}^Q_\rho(b, r, i, lsid)$.

Definition 82 (RP acts on the user's behalf). For a run ρ of an OIDC web system with web attacker $\mathcal{OIDCWS}^w$ we say that the RP r acts on behalf of the user with the identity u at an honest OP i in a login session identified by the nonce $lsid$ established with the browser b if there is a processing step Q in ρ with

$$Q = (S, E, N) \rightarrow (S', E', N')$$

(for some S, S', E, E', N, N') in which

(1) r calls the function $\mathsf{USE_ACCESS_TOKEN}$ with the first two parameters being $lsid$ and t,

(2) we have that there is a term g such that $g \in^{\langle\rangle} S(i).\mathtt{records}$ with $t \in^{\langle\rangle} g[\mathtt{access_tokens}]$ and $g[\mathtt{subject}] \equiv u$, and

(3) $\langle \mathtt{sessionid}, \langle lsid, y, z, z' \rangle\rangle \in^{\langle\rangle} S(b).\mathtt{cookies}[d]$ for $d \in \mathsf{dom}(r)$, $y, z, z' \in \mathcal{T}_{\mathcal{N}}$.

We then write $\mathsf{actsOnUsersBehalf}^Q_\rho(b, r, u, i, lsid)$.

For session integrity for authentication we say that a user that is logged in at some RP must have expressed her wish to be logged in to that RP in the beginning of the login flow. If the OP is honest, then the user must also have authenticated herself at the OP with the same user account that RP uses for her identification. This excludes, for example, cases where (1) the user is forcefully logged in to an RP by an attacker that plays the role of an OP, and (2) where an attacker can force a user to be logged in at some RP under a false identity issued by an honest OP.

Definition 83 (Session Integrity for Authentication). Let $\mathcal{OIDCWS}^w$ be an OIDC web system with web attackers. We say that $\mathcal{OIDCWS}^w$ *is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{OIDCWS}^w$, every processing step Q in ρ with

$$Q = (S, E, N) \rightarrow (S', E', N')$$

(for some S, S', E, E', N, N'), every browser b that is honest in S, every $i \in \mathsf{OP}$, every identity u, every $r \in \mathsf{RP}$ that is honest in S, every nonce lsid, and $\mathsf{loggedIn}_\rho^Q(b, r, u, i, \mathit{lsid})$ we have that (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, r, \mathit{lsid})$, and (2) if i is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, r, u, i, \mathit{lsid})$.

For session integrity for authorization we say that if an RP uses some access token at some OP in a session with a user, then that user expressed her wish to authorize the RP to interact with some OP. If the OP is honest, and the RP acts on the user's behalf at the OP (i.e., the access token is bound to the user's identity), then the user authenticated to the OP using that identity.

Definition 84 (Session Integrity for Authorization). Let $\mathcal{OIDCWS}^w$ be an OIDC web system with web attackers. We say that $\mathcal{OIDCWS}^w$ *is secure w.r.t. session integrity for authorization* iff for every run ρ of $\mathcal{OIDCWS}^w$, every processing step Q in ρ with

$$Q = (S, E, N) \rightarrow (S', E', N')$$

(for some S, S', E, E', N, N'), every browser b that is honest in S, every $i \in \mathsf{OP}$, every identity u, every $r \in \mathsf{RP}$ that is honest in S, every nonce lsid, we have that (1) if $\mathsf{usedAuthorization}_\rho^Q(b, r, i, \mathit{lsid})$ then there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, r, \mathit{lsid})$, and (2) if i is honest in S and $\mathsf{actsOnUsersBehalf}_\rho^Q(b, r, u, i, \mathit{lsid})$ then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, r, u, i, \mathit{lsid})$.

C.3. Proof of the OpenID Connect Security Theorem

Before we prove Theorem 2, we show some general properties of OIDC web systems with a network attacker. We then first prove the authentication, authorization, and session integrity properties separately. In the following, we use the terms introduced in Definitions 65–70.

C.3.1. Proof of Authentication

We here want to show that every OIDC web system is secure w.r.t. authentication, and therefore assume that there exists an OIDC web system that is not secure w.r.t. authentication. We then lead this to a contradiction, thereby showing that all OIDC web systems are secure w.r.t. authentication. In detail, we assume:

Lemma 21 (Integrity of Issuer Cache). For any run ρ of an OIDC web system $\mathcal{OIDCWS}^n$ with a network attacker or an OIDC web system $\mathcal{OIDCWS}^w$ with web attackers, every configuration (S, E, N) in ρ, every OP i that is honest in S, every identity $id \in \mathsf{ID}^i$, every relying party r that is honest in S, we have that $S(r).\mathtt{issuerCache}[id] \equiv \langle\rangle$ (not set) or $S(r).\mathtt{issuerCache}[id] \in \mathsf{dom}(i)$.

PROOF. Initially, the issuer cache of an honest relying party is empty. The issuer cache can only be modified in Line 11 of Algorithm C.4. There, the value of $S^l(r).\mathtt{issuerCache}[id']$ (for some $l < j$) is taken from an HTTPS response. The value of id' is taken from session data (Line 3) which is identified by a session id that is taken from the internal reference data of the incoming message. This internal reference data must have been created previously in Algorithm A.12 ($\mathsf{HTTPS_SIMPLE_SEND}$) which must have been called in Line 10 of Algorithm C.9 (since this is the only place where the reference data for a webfinger request is created). In this algorithm, it is easy to see that the request to which the request is sent (see Line 6) is the domain part of the identity. We therefore have that a webfinger request must have been sent (using HTTPS) to the OP i. (An attacker can neither decrypt any information from this request, nor spoof a response to this request. The request must therefore have been responded to by the honest OP.)

Since the path of this request is /.wk/webfinger, the OP can respond to this request only in Lines 3ff. of Algorithm C.10. Since the OP there chooses an issuer value that is one of its own domains (see Line 4), we finally have that $S(r).\mathtt{issuerCache}[id] \equiv \langle\rangle$ (if the response is blocked or the webfinger request was never sent) or we have that $S(r).\mathtt{issuerCache}[id] \in \mathsf{dom}(i)$, which proves the lemma. □

Lemma 22 (Integrity of oidcConfigCache). For any run ρ of an OIDC web system $\mathcal{OIDCWS}^n$ with a network attacker or an OIDC web system $\mathcal{OIDCWS}^w$ with web attackers, every configuration (S, E, N) in ρ, every OP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every relying party r that is honest in S, $l \in \{1, 2, 3, 4\}$ we have that $S(r).\mathtt{oidcConfigCache}[d] \equiv \langle\rangle$ (not set) or $S(r).\mathtt{oidcConfigCache}[d] \equiv [\mathtt{issuer}: d, \mathtt{auth_ep}: u_1, \mathtt{token_ep}: u_2, \mathtt{jwks_ep}: u_3, \mathtt{reg_ep}: u_4]$ with u_l being URLs, $u_l.\mathtt{host} \in \mathsf{dom}(i)$, and $u_l.\mathtt{protocol} \equiv \mathtt{S}$.

PROOF. This proof proceeds analog to the one for Lemma 21 with the following changes: First, the OIDC configuration cache is filled only in Line 17 of Algorithm C.4. It requires a request that was created in Line 16 of Algorithm C.9. This request was not sent to the domain contained in an ID (as above) but instead to the issuer (in this case, d). The issuer responds to this request in Lines 8ff. of Algorithm C.10. There, the issuer only choses the redirection endpoint URIs such that the host is the domain of the incoming request and the protocol is HTTPS (S). This proves the lemma. □

Lemma 23 (Integrity of JWKS Cache). For any run ρ of an OIDC web system $\mathcal{OIDCWS}^n$ with a network attacker or an OIDC web system $\mathcal{OIDCWS}^w$ with web attackers, every configuration (S, E, N) in ρ, every OP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every relying party r that is honest in S, we have that $S(r).\mathtt{jwksCache}[d] \equiv \langle\rangle$ (not set) or $S(r).\mathtt{jwksCache}[d] \equiv \mathsf{pub}(S(i).\mathtt{jwks})$.

PROOF. This proof proceeds analog to the one for Lemma 22. The relevant HTTPS request by r is created in Line 21 of Algorithm C.9, and responded to by the OP i in Lines 15ff. of Algorithm C.10. There, the OP chooses its own signature verification key to send in the response. This proves the lemma. □

Lemma 24 (Integrity of Client Registration). For any run ρ of an OIDC web system $\mathcal{OIDCWS}^n$ with a network attacker or an OIDC web system $\mathcal{OIDCWS}^w$ with web attackers, every configuration (S, E, N) in ρ, every OP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every relying party r that is honest in S, every client id c that has been issued to r by i, every URL $u \in^{\langle\rangle} S(i).\mathtt{clients}[c][\mathtt{redirect_uris}]$ we have that $u.\mathtt{host} \in \mathsf{dom}(r)$ and $u.\mathtt{protocol} \equiv \mathtt{S}$.

Proof. From Definition 76 it follows that an HTTPS request must have been sent from r to i in Lines 23ff. of Algorithm C.9. This request must have been processed by i in Lines 18ff. of Algorithm C.10, and, after receiving the client id from some other party (usually the attacker), in Algorithm C.11. From the latter algorithm it is easy to see that the redirection endpoint data must have been taken from r's initial registration request to create the dictionary stored in $S(i).\mathtt{clients}[c]$. This data, however, was chosen by r in Line 2 of Algorithm C.9 such that $u.\mathtt{host} \in \mathsf{dom}(r)$ and $u.\mathtt{protocol} \equiv \mathtt{S}$ for every $u \in^{\langle\rangle} S(i).\mathtt{clients}[c][\mathtt{redirect_uris}]$. □

Lemma 25 (Third Parties do not Learn Passwords). For any run ρ of an OIDC web system $\mathcal{OIDCWS}^n$ with a network attacker or an OIDC web system $\mathcal{OIDCWS}^w$ with web attackers, every configuration (S, E, N) in ρ, every OP i that is honest in S, every identity $id \in \mathsf{ID}^i$, every browser b with $b = \mathsf{ownerOfID}(id)$ that is honest in S, every $p \in \mathcal{W} \setminus \{b, i\}$ we have that $\mathsf{secretOfID}(id) \notin d_\emptyset(S^l(p))$.

Proof. Let $s := \mathsf{secretOfID}(id)$. Initially, in S^0, s is only contained in $S^0(b).\mathtt{secrets}[\langle d, \mathtt{S}\rangle]$ with $d \in \mathsf{dom}(i)$ and in no other states of any atomic processes (or in any waiting events). By the definition of the browser, we can see that only scripts loaded from the origins $\langle d, \mathtt{S}\rangle$ can access s. We know that i is an honest OP. Now, the only script that an honest OP sends to the browser is *script_op_form*. This scripts sends the form data only to its own origin, which means, that the form data is sent over HTTPS and to the honest OP. In this request, the script uses the path `/auth2`. There, identity and password are checked, but not used otherwise. Therefore, the form data cannot leak from the honest OP. It could, however, leak from the browser itself. The form data is sent via POST, and therefore, not used in any `Referer` headers. A redirection response from the server contains the status code 303, which implies that the browser does not send the form data again when following the redirection. Since there are also no other scripts from the same origin running in the browser which could access the form data, the password s cannot leak from the browser either. This proves Lemma 25. □

Lemma 26 (Attacker does not Learn ID Tokens). For any run ρ of an OIDC web system $\mathcal{OIDCWS}^n$ with a network attacker or an OIDC web system $\mathcal{OIDCWS}^w$ with web attackers, every configuration (S, E, N) in ρ, every OP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every identity $id \in \mathsf{ID}^i$ with $b = \mathsf{ownerOfID}(id)$ being an honest browser (in S), every relying party r that is honest in S, every client id c that has been issued to r by i, every term y, every id token $t = \mathsf{sig}([\mathtt{iss}: d, \mathtt{sub}: id, \mathtt{aud}: c, \mathtt{nonce}: y], k)$ with $k = S(i).\mathtt{jwks}$, every attacker process a we have that $t \notin d_\emptyset(S(a))$.

PROOF. The signing key k is only known to i initially and at least up to S (since i is honest). Therefore, only i can create t. There are two places where an honest OP can create such a token in Algorithm C.10: In Line 59 (immediately after receiving the user credentials) and in Lines 91ff. (after receiving an authorization code).

We now distinguish between these two cases to show that in either case, the attacker cannot get hold of an id token. We start with the first case.

ID token was created in Line 59. To create t, the OP i must have received a request to the path /`auth2` in Lines 29ff. of Algorithm C.10. It is clear that i sends the response to this request to the sender of the request, and, if that sender is honest, the response cannot be read by an attacker. The request must contain $\mathsf{secretOfId}(id)$. Only b and i know this secret (per Lemma 25). Since i does not send requests to itself, the request must have been sent from b. Since the `Origin` header in the request must be a domain of i, we know that the request was not initiated by a script other than i's own scripts, in particular, it must have been initiated by *script_op_form*.

Now it is easy to see that this script does not use the token t in any way after the token was returned from i, since the script uses a form post to transmit the credentials to i, and the window is subsequently navigated away. Instead, i provides an empty script in its response to b. This response contains a `Location` redirect header. It is now crucial to check that this location redirect does not cause the id token to be leaked to the attacker: With Lemma 24 we have that the redirection URIs that are registered at i for the client id c only point to domains of r (and use HTTPS).

We therefore know that b will send an HTTPS request (say m) containing t to r. We have to check whether r or a script delivered by r to b will leak t. Algorithm C.5 processes all HTTPS requests delivered to r. As i redirected b using the 303 status code, the request to r must be a GET request. Hence, r does not process this request in Lines 6ff. of Algorithm C.5. Lines 2ff. do only respond with a script and do not use t in any way. We are left with Lines 14ff. to be analyzed.

As in m the id token t is always contained in a dictionary under the key `id_token` and this dictionary is either in the parameters, the fragment, or the body of m, it is now easy to see that r does not store or send out t in any way.

We now have to check if a script delivered by r to b leads to t being leaked. First note that r always sets the header `ReferrerPolicy` to `origin` in every HTTP(S) response r sends out. Hence, t can never leak using the `Referer` header.

There are only two scripts that r may deliver: (1) The script *script_rp_index* either issues a `FORM` command to the browser, which does not contain t, or this script issues a `HREF` command to the browser for some URL, which also does not contain t. (2) The script *script_rp_get_fragment* takes the fragment of the current URL (which may be a dictionary that contains t under the key `id_token`) and the `iss` parameter and issues an HTTPS

request to r for the path /redirect_ep, which will be processed by r in Lines 14ff. of Algorithm C.5. Now, the same reasoning as above applies.

ID token was created in Lines 91ff. In this case, the id token is created by i only when an HTTPS request was received by i that matches the following criteria: (a) it must be for the path /token, (b) it must contain the client id c in the body (under the key client_id), and (c) it must contain a authorization code in the body (under the key code) that occurs in one of i's internal records with a matching subject, issuer, and nonce. To be more precise, the request must contain a code *code* such that there is a record *rec* with $rec \in^{\langle\rangle} S(i).\mathtt{records}$ and $rec[\mathtt{issuer}] \equiv d$, $rec[\mathtt{subject}] \equiv id$, $rec[\mathtt{client_id}] \equiv c$, and $rec[\mathtt{code}] \equiv c$. Such a record can only be created and the authorization code *code* issued under exactly the same circumstances that allow an id token (of the above form) to be created in Line 91. Similar to the reasoning for the id token above, we can follow that *code* does not leak to the attacker. (Here it is important that the iss parameter ensures that the code is not sent to third parties.)

We have therefore shown that no attacker process can get hold of the id token t. This proves the lemma. □

Assumption 3. There exists an OIDC web system $\mathcal{OIDCWS}^n$ with a network attacker such that there exists a run ρ of $\mathcal{OIDCWS}^n$, a configuration (S, E, N) in ρ, some $r \in \mathsf{RP}$ that is honest in S, some identity $id \in \mathsf{ID}$ with $\mathsf{governor}(id)$ being an honest OP (in S) and $\mathsf{ownerOfID}(id)$ being an honest browser (in S), some service session identified by some nonce n for id at r, and n is derivable from the attackers knowledge in S (i.e., $n \notin d_\emptyset(S(\mathsf{attacker}))$).

Lemma 27. Assumption 3 is a contradiction.

PROOF. We first recall how the service session identified by some nonce n for id at r is defined. It means that there is some session id x and a domain $d \in \mathsf{dom}(\mathsf{governor}(id))$ with $S(r).\mathtt{sessions}[x][\mathtt{loggedInAs}] \equiv \langle d, id\rangle$ and $S(r).\mathtt{sessions}[x][\mathtt{serviceSessionId}] \equiv n$. Now the assumption is that n is derivable from the attacker's knowledge. Since we have that $S(r).\mathtt{sessions}[x][\mathtt{serviceSessionId}] \equiv n$, we can check where and how, in general, service session ids can be created. It is easy to see that this can only happen in Algorithm C.8, where, in Line 19, the RP chooses a fresh nonce as the value for the service session id, in this case x. In the line before, it sets the value for $S(r).\mathtt{sessions}[x][\mathtt{loggedInAs}]$, in this case $\langle d, id\rangle$. In the Lines 9ff., r performs several checks to ensure the integrity and authenticity of the id token.

The function function CHECK_ID_TOKEN can be called in either (a) Line 43 of Algorithm C.5 or (b) in Line 28 of Algorithm C.4.

We can now distinguish between these two cases.

Case (a). In this case, we can easily see that the same party that finally receives the service session id x, must have provided, in an HTTPS request, an id token (say, t') with the

following properties (for some $l < j$):

$$\mathsf{extractmsg}(t')[\mathtt{iss}] \equiv d$$

$$\mathsf{extractmsg}(t')[\mathtt{sub}] \equiv id$$

$$\mathsf{extractmsg}(t')[\mathtt{aud}] \equiv S^l(r).\mathtt{clientCredentialsCache}[d][\mathtt{client_id}]$$

$$\mathsf{checksig}(t', \mathsf{pub}(S^l(i).\mathtt{jwks})) \equiv \top\ .$$

The attacker (and, by extension, any other party except for i, b, and r), however, cannot know such an id token (see Lemma 26). Since r and i do not send requests to r, the id token must have been sent by b to r. As the service session id x is only contained in a `Set-Cookie` header with the httpOnly and secure flags set, b will only ever send the service session id x to r (contained in a `Cookie` header). As b does not leak x in any other way and as r does not leak information sent in cookie headers, the service session id x does not leak.

Case (b). Otherwise, the party that finally receives the service session id x needs to provide a code c such that, when this code is sent to the token endpoint of i (Algorithm C.6), i responds with an id token matching the criteria listed in Case (a). This, however, would mean that an attacker, knowing this code, could do the same, violating Lemma 26. (Note that for every run where a client secret is associated with the client id there is also a run where the client secret is not used; the client secret does not prevent the attacker from requesting an id token at the token endpoint for a valid code.)

We therefore have shown that the attacker cannot know x, proving the lemma and showing that Assumption 3 is, in fact, a contradiction. □

C.3.2. Proof of Authorization

As above, we assume that there exists an OIDC web system that is not secure w.r.t. authorization and lead this to a contradiction.

Assumption 4. There exists an OIDC web system with a network attacker $\mathcal{OIDCWS}^n$, a run ρ of $\mathcal{OIDCWS}^n$, a state (S^j, E^j, N^j) in ρ, a relying party $r \in \mathsf{RP}$ that is honest in S^j, an identity provider $i \in \mathsf{OP}$ that is honest in S^j, a browser b that is honest in S^j, an identity $id \in \mathsf{ID}^i$ owned by b, a nonce n, a term $x \in^{\langle\rangle} S^j(i).\mathtt{records}$ with $x[\mathtt{subject}] \equiv id$, $n \in^{\langle\rangle} x[\mathtt{access_tokens}]$, and the client id $x[\mathtt{client_id}]$ has been issued by i to r, and n is derivable from the attackers knowledge in S^j (i.e., $n \in d_\emptyset(S^j(\mathsf{attacker}))$).

Lemma 28. Assumption 4 is a contradiction.

Proof. We have that $n \in d_\emptyset(S^j(\mathsf{attacker}))$ and therefore, there must have been a message from a third party to $\mathsf{attacker}$ (or any other corrupted party, which could have forwarded n to the

attacker) that contained n. We can now distinguish between the parties that could have sent n to the attacker (or to the corrupted party):

The access token n was sent by the browser b: We now track different cases in which the access token n can get into b's knowledge. We omit the cases in which b learns n from any dishonest party as in such a case there is a different run ρ' of $\mathcal{OIDCWS}^n$ in which this dishonest party immediately sends n to the attacker.

(I) First, we analyze the case in which b has learned n from an honest (in S^j) identity provider, say i'. In this case, b must have received an HTTPS response from i' (honest identity providers do not send out unencrypted HTTP responses). Honest identity providers send out HTTPS responses in Lines 6, 14, 17, 28, 71, and 98 of Algorithm C.10 and Line 17 of Algorithm C.11. It is easy to see that i' does not send out n in Lines 6, 14, 17, and 28 of Algorithm C.10 and Line 17 of Algorithm C.11 (given that the attacker does not know n), leaving Lines 71, and 98 of Algorithm C.10 to analyze.

(a) If i' sends out n in Line 71 of Algorithm C.10, b must have sent an HTTPS POST request bearing an `Origin` header for one of the domains of i' to i'. As i' only delivers the script *script_op_form*, only this script could have caused this request (using a `FORM` command). Hence, b will navigate the corresponding window to the location indicated in the `Location` header of the HTTPS response assembled in Lines 29ff. of Algorithm C.10. The body of this response can consist of an authorization code (a fresh nonce), an access token (a fresh nonce), and an id token consisting of one domain of i, a valid username for i', a client id, and a nonce (say n') from the request.

We now reason why i' must be i, and the access token in the response must be n. In the id token, only the client id and the nonce n' could be n. As the client id is always set by the attacker during registration, the client id cannot be n. The nonce n' originates from the request sent by b on the command of *script_op_form*. In this request, the nonce n' must be contained in the URL, which is the URL from which the script was loaded before. Hence, the browser must have been navigated to this URL. As the attacker does not know n at this point, only honest scripts or honest web servers could have navigated the browser to such an URL (containing n). Honest relying parties only populate the parameter `nonce` (bearing n') in such a redirect with a fresh nonce, honest identity providers do not populate such an URL parameter by themselves, but could have used this parameter in a redirect based on a registered redirect URL. As honest parties never register such a redirect URL, n' cannot be n. Hence, only the access token in the response above can be n. As the access token is a fresh nonce, we must have that i' is i and that i creates the term $x \in^{\langle\rangle} S^j(i).\texttt{records}$ with $x[\texttt{subject}] \equiv id$, $n \in^{\langle\rangle} x[\texttt{access_tokens}]$ (i will never create such a term at any other time), and the client id $x[\texttt{client_id}]$ has been issued by i to r. Hence, the location redirect issued by i must point to an URL of r with the path `/redirect_ep` (see Lemma 24) and this URL contains the parameter `iss` with a domain of i. The access token n is only contained in the fragment of this URL under the key `access_token`.

Now, b sends an HTTPS request to r. This request does not contain n (as it is placed in the

fragment part of the URL). The relying party r can (regardless of the path) send out only the scripts *script_rp_index* and *script_rp_get_fragment* as a response to such a request. The script *script_rp_index* ignores the fragment of its URL. The script *script_rp_get_fragement* takes the fragment of the URL and uses it as the body of a POST request to its own origin (which is r) with path `/redirect_uri`. When r processes this POST request, r only ever uses n in Line 46 of Algorithm C.5. There, the access token n and the value of the parameter `iss` (a domain of i) is processed by Algorithm C.7. From Lemma 22, we know that r will only send n to the token endpoint of i in an HTTPS request. This request is then processed by i in Lines 72ff. of Algorithm C.10. There, i only checks n, but does not send out n.

If b sends out a response, the same reasoning as above applies. Hence, we have that n does not leak to the attacker in this case.

(b) If i' sends out n in Line 98 of Algorithm C.10, we have that the response does not contain a script or a redirect. The browser would only interpret such a response if the request was caused by an `XMLHTTPREQUEST` command of a script. Honest scripts do not issue such a command, leaving only the attacker script as the only possible source for such a request. If i' is not i, it is easy to see that this response cannot contain n. The identity provider i only sends out n (taken from the subterm `records` from its state) if the request contains a valid authorization code for this access token. With the same reasoning as for the authentication property above, the attacker cannot know a valid id token for any user id owned by b. If the attacker would know a valid authorization code, he could retrieve a valid id token (for such a user id) from i. Hence, the attacker cannot know a valid authorization code. As this reasoning also applies for the attacker script, the attacker script could not have caused a request to i revealing n.

(II) Now, we analyze the case in which b received n from some honest (in S^j) relying party, say r'. In this case, b must have received an HTTPS response from r' (honest relying parties do not send out unencrypted HTTP responses). Honest relying parties only send out such HTTPS responses in Lines 5 and 30 of Algorithm C.5, Line 24 of Algorithm C.8, and Line 39 of Algorithm C.9. In the former three cases, r' only sends out fixed information and fresh nonces (either chosen by r' directly before sending out the message or the HTTPS nonce and key chosen by b when creating the request). In the latter case, r' (besides the pieces of information as before) also adds information from its OpenID Connect configuration cache (i.e., client id and authorization URL). From Lemma 22 and 24 we know that if r' gathered this information from an honest party, this information cannot contain n. As the attacker does not know n at this point, this registration information cannot contain n if r' gathered this information from a dishonest party. Hence, b cannot have learned n from any r'.

(III) b cannot have learned n from a different honest (in S^j) browser as honest browsers do not create messages that can be interpreted by honest browsers.

(IV) b cannot have learned n from the attacker, as the attacker does not know n at this point.

The access token n was sent by the OP i: We can see that access tokens are sent by the OP only after a request to the path (endpoints) `/auth2` or to the path and `/token`.

In case of a request to the path /auth2, a pair of access tokens is created and the first access token in the pair is returned from the endpoint. If the attacker would be able to learn n from this endpoint such that there exists a record $x \in^{\langle\rangle} S^j(i).\texttt{records}$ with $x[\texttt{subject}] \equiv \mathit{id}$, then the attacker would need to provide the user's credentials to the OP i. The attacker cannot know these credentials (Lemma 25), therefore the attacker cannot request n from this endpoint.

In case of a request to the path /token, the attacker would need to provide an authorization code that is contained in the same record (in this case x) as n. Now, recall that we have that $x[\texttt{subject}] \equiv \mathit{id}$ and $c := x[\texttt{client_id}]$ has been issued to r by i. We can now see that if the attacker would be able to send a request to the endpoint /token which would cause a response that contains n, the attacker would also be able to learn an id token of the form shown in Lemma 26 (the issuer is a domain of i, the subject is id, and the audience is c). This would be a contradiction to Lemma 26.

We can conclude that the access token n was not sent by the OP i.

The access token n was sent by the RP r: The only place where the (honest) RP uses an access token is in Algorithm C.7. There, the access token is sent to the domain of the token endpoint (compare Algorithm C.6, where the authorization code is sent to that endpoint). We can now see that the access token is always sent to i: If the access token would be sent to the attacker, so would the authorization code in Algorithm C.6, and Lemma 26 would not hold true. □

C.3.3. Proof of Session Integrity

Before we prove this property, we highlight that in the absence of a network attacker and with the DNS server as defined for $\mathcal{OIDCWS}^w$, HTTP(S) requests by (honest) parties can only be answered by the owner of the domain the request was sent to, and neither the requests nor the responses can be read or altered by any attacker unless he is the intended receiver.

We further show the following lemma, which says that an attacker (under the assumption above) cannot learn a *state* value that is used in a login session between an honest browser, an honest OP, and an honest RP.

Lemma 29 (Third parties do not learn state). There exists no run ρ of an OIDC web system with web attackers $\mathcal{OIDCWS}^w$, no configuration (S, E, N) of ρ, no $r \in \mathsf{RP}$ that is honest in S, no $i \in \mathsf{OP}$ that is honest in S, no browser b that is honest in S, no nonce $\mathit{lsid} \in \mathcal{N}$, no domain $h \in \mathsf{dom}(r)$ of r, no terms g, x, y, $z \in \mathcal{T}_{\mathcal{N}}$, no cookie $c := \langle \texttt{sessionId}, \langle \mathit{lsid}, x, y, z\rangle\rangle$, no atomic DY process $p \in \mathcal{W} \setminus \{b, i, r\}$ such that (1) $S(r).\texttt{sessions}[\mathit{lsid}] \equiv g$, (2) $g[\texttt{state}] \in d_\emptyset(S(p))$, (3) $S(r).\texttt{issuerCache}[g[\texttt{identity}]] \in \mathsf{dom}(i)$, and (4) $c \in^{\langle\rangle} S(b).\texttt{cookies}[h]$.

Proof. To prove Lemma 29, we track where the login session identified by lsid is created and used.

Login session ids are only chosen in Line 10 of Algorithm C.5. After the session id was chosen, its value is sent over the network to the party that requested the login (in Line 39 of

Algorithm C.9). We have that for *lsid*, this party must be b because only r can set the cookie c for the domain h in the state of b[1] and Line 39 of Algorithm C.9 is actually the only place where r does so.

Since b is honest, b follows the location redirect contained in the response sent by r. This location redirect contains *state* (as a URL parameter). The redirect points to some domain of i. (This follows from Lemma 22.) The browser therefore sends (among others) *state* in a GET request to i. Of all the endpoints at i where the request can be received, the authorization endpoint is the only endpoint where *state* could potentially leak to another party. (For all other endpoints, the value is dropped.) If the request is received at the authorization endpoint, *state* is only sent back to b in the initial scriptstate of *script_op_form*. In this case, the script sends *state* back to i in a POST request to the authorization endpoint. Now, i redirects the browser b back to the redirection URI that was passed alongside *state* from r via the browser to i. This redirection URI was chosen in Line 2 of Algorithm C.9 and therefore points to one of r's domains. The value *state* is appended to this URI (either as a parameter or in the fragment). The redirection to the redirection URI is then sent to the browser b. Therefore, b now sends a GET request to r.

If *state* is contained in the parameter, then *state* is immediately sent to r where it is compared to the stored login session records but neither stored nor sent out again. In each case, a script is sent back to b. The scripts that r can send out are *script_rp_index* and *script_rp_get_fragment*, none of which cause requests that contain *state* (recall that we are in the case where *state* is contained in the URI parameter, not in the fragment). Also, since both scripts are always delivered with a restrictive `Referrer-Policy` header, any requests that are caused by these scripts (e.g., the start of a new login flow) do not contain *state* in the `Referer` header.[2]

If *state* is contained in the fragment, then *state* is not immediately sent to r, but instead, a request without *state* is sent to r. Since this is a GET request, r either answers with a response that only contains the string `ok` but no script (Lines 24ff. of Algorithm C.8), a response containing *script_rp_index* (Lines 2ff. of Algorithm C.5), or a response containing *script_rp_get_fragment* (Line 30 of Algorithm C.5). In case of the `ok` response, *state* is not used anymore by the browser. In case of *script_rp_index*, the fragment is not used. (As above, there is no other way in which *state* can be sent out, also because the fragment part of an URL is stripped in the `Referer` header.) In the case of *script_rp_get_fragment* being loaded into the browser, the script sends *state* in the body of an HTTPS request to r (using the path `/redirect_ep`). When r receives this request, it does not send out *state* to any party (see Lines 14ff. of Algorithm C.5).

This shows that *state* cannot be known to any party except for b, i, and r. □

[1] We have only web attackers.

[2] Without the Referrer Policy, *state* could leak to a malicious OP or other parties.

Proof of Session Integrity for Authentication

To prove that every OIDC web system with web attackers is secure w.r.t. session integrity for authentication, we assume that there exists an OIDC web system with web attackers which is not secure w.r.t. session integrity for authentication and lead this to a contradiction.

Assumption 5. There exists an $\mathcal{OIDCWS}^w$ be an OIDC web system with web attackers, a run ρ of $\mathcal{OIDCWS}^w$, a processing step Q in ρ with

$$Q = (S, E, N) \rightarrow (S', E', N')$$

(for some S, S', E, E', N, N'), a browser b that is honest in S, an OP $i \in \mathsf{OP}$, an identity u that is owned by b, an RP $r \in \mathsf{RP}$ that is honest in S, a nonce $lsid$, with $\mathsf{loggedIn}_\rho^Q(b, r, u, i, lsid)$ and (1) there exists no processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, r, lsid)$, or (2) i is honest in S, and there exists no processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, r, u, i, lsid)$.

Lemma 30. Assumption 5 is a contradiction.

PROOF. **(1)** We have that $\mathsf{loggedIn}_\rho^Q(b, r, u, i, lsid)$. With Definition 78 we have that r sent out the service session id belonging to $lsid$ to b. (This can only happen when the function CHECK_ID_TOKEN (Algorithm C.8) was called with $lsid$ as the first parameter.) This means that r must have received a request from b containing a cookie with the name `sessionId` and the value $lsid$: The response by r (which we know was sent to b) was sent in Line 24 in Algorithm C.8. There, r looks up the address of b using the login session record under the key `redirectEpRequest`. This key is only ever created in Line 36 of Algorithm C.5. This line is only ever called when r receives an HTTPS request from b with the cookie as described.

We can now track how the cookie was stored in b: Since the cookie is stored under a domain of r and we have no network attacker, the cookie must have been set by r. This can only happen in Line 39 in Algorithm C.9. Similar to the `redirectEpRequest` session entry above, r sends this cookie as a response to a stored request, in this case, using the key `startRequest` in the session data. This key is only ever created in Lines 6ff. of Algorithm C.5. Hence, there must have been a request from b to r containing a POST request for the path `/startLogin` with an `Origin` header for an origin of r. There are only two scripts which could potentially send such a request, *script_rp_index* and *script_rp_get_fragment*. It is easy to see that only the former send requests of the kind described. We therefore have a processing step Q' that happens before Q in ρ with $\mathsf{started}_\rho^{Q'}(b, r, lsid)$.

(2) Again, we have that $\mathsf{loggedIn}_\rho^Q(b, r, u, i, lsid)$. Now, however, i is honest.

We first highlight that if r receives an HTTPS request, say m, which contains $state$ such that $S(r).\mathtt{sessions}[lsid][\mathtt{state}] \equiv state$ and contains a cookie with the name `sessionId` and the value $lsid$ then this request must have come from the browser b and be caused by a redirection from i or a script from r. From $\mathsf{loggedIn}_\rho^Q(b, r, u, i, lsid)$ it follows that there is a term g such that $S(r).\mathtt{sessions}[lsid] \equiv g$, and $g[\mathtt{loggedInAs}] \equiv \langle d, u\rangle$ with $d \in \mathsf{dom}(i)$. From the Algorithm C.8

we have that $S(r).\mathtt{issuerCache}[g[\mathtt{identity}]] \equiv d$. With Lemma 29 we have that only b, r, and i know state.

We can now show that m must have been caused by i by means of a Location redirect that was sent to b or by the script *script_rp_get_fragment*. First, neither r nor i send requests that contain cookies. The request must therefore have originated from b. Since no attacker knows *state*, the request cannot have been caused by any attacker scripts or by redirects from parties other than r or i (otherwise, there would be runs where the attacker learns *state*).

Redirects from r can be excluded, since r only sends a redirection in Line 39 in Algorithm C.9 but there, a freshly chosen state value is used, hence, there is only one processing step in which r uses *state* for this redirect. This is the processing step where r adds *state* to the session data stored under the key *lsid*. Since this is a session in which the honest OP i is used, and with Lemma 22, we have that r does not redirect to itself (but to i instead).

The scripts *script_rp_index* and *script_op_form* do not send requests with the *state* parameter. Therefore, the remaining causes for the request m are either the script *script_rp_get_fragment* or a location redirect from i.

If the request m was caused by *script_rp_get_fragment*, then it is easy to see from the definition of *script_rp_get_fragment* (Algorithm C.2) that this script only sends data from the fragment part of its own URI (except for the `iss` parameter) and it sends this data only to its own origin. This script therefore must have been sent to b by r, which only sends this script after receiving HTTPS request to the redirection endpoint (`/redirect_ep`). With the same reasoning as above this must have been caused by a location redirect from i.

For clarity, by m_{redir} we denote the response by i to the browser b containing this redirection. We now show that for m_{redir} to take place, there must have been a processing step Q'' (before Q) with $\mathsf{authenticated}_\rho^{Q''}(b, r, u', i, \mathit{lsid})$ for some identity u'.

In the honest OP i, there is only one place where a redirection happens, namely in Line 71 in Algorithm C.10. To reach this point, i must have received the login data for u' in the HTTPS request corresponding to m_{redir}. This must be a POST request with an `Origin` header containing an origin of i. As i only uses *script_op_form*, the request must have been caused by this script. Hence, we have $\mathsf{authenticated}_\rho^{Q''}(b, r, u', i, \mathit{lsid})$. We now only need to show that $u' = u$.

With $\mathsf{loggedIn}_\rho^{Q}(b, r, u, i, \mathit{lsid})$, we know that r must have called CHECK_ID_TOKEN (Algorithm C.8). We further have that $S(r).\mathtt{sessions}[\mathit{lsid}][\mathtt{loggedInAs}] \equiv \langle d, u\rangle$. We therefore have that i must have created an id token with the issuer d and the identity u. CHECK_ID_TOKEN can be called in Line 28 in Algorithm C.4 and in Line 43 in Algorithm C.5. We distinguish between these two cases.

- CHECK_ID_TOKEN *was called in Line 28 in Algorithm C.4:* When the function is called in this line, there must have been an HTTPS request reference with the string `TOKEN` (cf. generic web server model, Algorithm A.12). Such a reference is only created in Line 16 of Algorithm C.6. With Lemma 22 we know that this HTTPS request was sent to the token endpoint (path `/token`) of i (because the issuer, stored in the login session record,

is i). Since the token endpoint returned an id token of the form described above, and i is honest, there must have been a record in i, say v, with $v[\texttt{subject}] \equiv u$. In the request to the token endpoint, r must have sent a nonce c such that $v[\texttt{code}] \equiv c$. This request, as already mentioned, must have been sent in Line 16 of Algorithm C.6. This means, that there must have been an HTTPS request to i containing the session id lsid as a cookie, c, and state. Such a request can only be the request m as shown above, hence there must have been the HTTPS response m_{redir} containing the values c and state. Recall that we have the record v as shown above in the state of i. Such a record is only created in i if $\mathsf{authenticated}_\rho^{Q''}(b, r, u, i, \mathit{lsid})$. Therefore, $u = u'$ in this case.

– CHECK_ID_TOKEN *was called in Line 43 in Algorithm C.5:* In this case, the id token must have been contained in m and m_{redir} as above. Such an id token is only sent out in m_{redir} by i if $\mathsf{authenticated}_\rho^{Q''}(b, r, u, i, \mathit{lsid})$. Therefore, $u = u'$ in every case. □

Proof of Session Integrity for Authorization

To prove that every OIDC web system with web attackers is secure w.r.t. session integrity for authorization, we assume that there exists an OIDC web system with web attackers which is not secure w.r.t. session integrity for authorization and lead this to a contradiction.

Assumption 6. There is a OIDC web system $\mathcal{OIDCWS}^w$ with web attackers, a run ρ of $\mathcal{OIDCWS}^w$, a processing step Q in ρ with $Q = (S, E, N) \to (S', E', N')$ (for some S, S', E, E', N, N') a browser b that is honest in S, an OP $i \in \mathsf{OP}$, an identity u that is owned by b, an RP $r \in \mathsf{RP}$ that is honest in S, a nonce lsid, with (1) $\mathsf{usedAuthorization}_\rho^Q(b, r, i, \mathit{lsid})$ and there exists no processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, r, \mathit{lsid})$, or (2) i is honest in S and $\mathsf{actsOnUsersBehalf}_\rho^Q(b, r, u, i, \mathit{lsid})$ and there exists no processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, r, u, i, \mathit{lsid})$.

Lemma 31. Assumption 6 is a contradiction.

Proof. **(1)** We have that $\mathsf{usedAuthorization}_\rho^Q(b, r, i, \mathit{lsid})$. With Definition 81 we have that r sent out the access token belonging to lsid to i. This can only happen when the function USE_ACCESS_TOKEN (Algorithm C.7) was called with lsid. This function is called in Line 46 of Algorithm C.5 and in Line 27 of Algorithm C.4.

In both cases, there must have been a request, say m, to r containing a cookie with the session id lsid. In the former case, this is the request that is processed in the same processing step as calling the function USE_ACCESS_TOKEN. In the latter case, there must have been an HTTPS request reference with the string `TOKEN` (cf. generic web server model, Algorithm A.12). Such a reference is only created in Line 16 of Algorithm C.6. To get to this point in the algorithm, a request as described above must have been received. Since we have web attackers (and no network attacker), it is easy to see that this request must have been sent by b. With the same

reasoning as in the proof for session integrity for authentication, we now have that there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, r, \mathit{lsid})$.

(2) We have that i is honest and $\mathsf{actsOnUsersBehalf}_\rho^{Q}(b, r, u, i, \mathit{lsid})$. From (1) we know that r must have received a request m from b containing a cookie with the session id lsid. Therefore, we know that m_{redir} exists just as in the proof for Lemma 30 (2). As in that proof, we have that $\mathsf{authenticated}_\rho^{Q''}(b, r, u', i, \mathit{lsid})$ for some identity u'. We therefore need to show that $u = u'$.

Because of $\mathsf{actsOnUsersBehalf}_\rho^{Q}(b, r, u, i, \mathit{lsid})$, r must have called USE_ACCESS_TOKEN with some access token t (Algorithm C.7). We further have that there is a term g such that $g \in^{\langle\rangle} S(i).\mathtt{records}$ with $t \in^{\langle\rangle} g[\mathtt{access_tokens}]$ and $g[\mathtt{subject}] \equiv u$.

USE_ACCESS_TOKEN can be called in Line 27 in Algorithm C.4 and in Line 46 in Algorithm C.5. We now distinguish between these two cases.

- USE_ACCESS_TOKEN *was called in Line 27 in Algorithm C.4:* When the function is called in this line, there must have been an HTTPS request reference with the string `TOKEN` (cf. generic web server model, Algorithm A.12). Such a reference is only created in Line 16 of Algorithm C.6. With Lemma 22 we know that this HTTPS request was sent to the token endpoint (path `/token`) of i (because the issuer, stored in the login session record, is i). Since the token endpoint returned the access token t, and i is honest, there must have been a record in i, say v, with $v[\mathtt{subject}] \equiv u$. In the request to the token endpoint, r must have sent a nonce c such that $v[\mathtt{code}] \equiv c$. This request, as already mentioned, must have been sent in Line 16 of Algorithm C.6. This means, that there must have been an HTTPS request to i containing the session id lsid as a cookie, c, and state. Such a request can only be the request m as shown above, hence there must have been the HTTPS response m_{redir} containing the values c and state. Recall that we have the record v as shown above in the state of i. Such a record is only created in i if $\mathsf{authenticated}_\rho^{Q''}(b, r, u, i, \mathit{lsid})$. Therefore, $u = u'$ in this case.

- USE_ACCESS_TOKEN *was called in Line 46 in Algorithm C.5:* In this case, the access token t must have been contained in m and m_{redir} as above. This access token is only sent out in m_{redir} by i if $\mathsf{authenticated}_\rho^{Q''}(b, r, u, i, \mathit{lsid})$. Therefore, $u = u'$ in every case. □

C.3.4. Proof of Theorem 2

With Lemmas 27, 28, 30, and 31, Theorem 2 follows immediately. ■

Bibliography

[AF01] M. Abadi and C. Fournet. "Mobile Values, New Names, and Secure Communication". In: *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL 2001)*. ACM Press, 2001, pp. 104–115.

[Akh+10] Devdatta Akhawe et al. "Towards a Formal Foundation of Web Security". In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*. IEEE Computer Society, 2010, pp. 290–304.

[Akh+14] Devdatta Akhawe et al. "Clickjacking Revisited: A Perceptual View of UI Security". In: *8th USENIX Workshop on Offensive Technologies, WOOT '14, San Diego, CA, USA, August 19, 2014*. USENIX Association, 2014.

[Akh+16] Devdatta Akhawe et al., eds. *Subresource Integrity - W3C Recommendation*. Jun. 23, 2016. June 23, 2016. URL: http://www.w3.org/TR/2016/REC-SRI-20160623/.

[Ann14] Anne van Kesteren, ed. *Cross-Origin Resource Sharing - W3C Recommendation*. Jan. 16, 2014. URL: http://www.w3.org/TR/2014/REC-cors-20140116/.

[Arm+08] Alessandro Armando et al. "Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-based Single Sign-on for Google Apps". In: *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*. Ed. by Vitaly Shmatikov. ACM, 2008, pp. 1–10.

[Arm+13] Alessandro Armando et al. "An authentication flaw in browser-based Single Sign-On protocols: Impact and remediations". In: *Computers & Security* 33 (2013), pp. 41–58.

[Bai+13] Guangdong Bai et al. "AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations". In: *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.

[Bal+18] Dirk Balfanz et al. *Web Authentication: An API for accessing Public Key Credentials Level 1 - W3C Candidate Recommendation*. Mar. 20, 2018. URL: https://www.w3.org/TR/2018/CR-webauthn-20180320/.

[Ban+13] Chetan Bansal et al. "Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage". In: *Principles of Security and Trust - Second International Conference, POST 2013*. Ed. by David A. Basin and John C. Mitchell. Vol. 7796. Lecture Notes in Computer Science. Springer, 2013, pp. 126–146.

[Ban+14] Chetan Bansal et al. "Discovering Concrete Attacks on Website Authorization by Formal Analysis". In: *Journal of Computer Security* 22.4 (2014), pp. 601–657.

[Bau+15] Lujo Bauer et al. "Run-time Monitoring and Formal Analysis of Information Flows in Chromium". In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8–11, 2015.* The Internet Society, 2015.

[BBM12] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. "Discovering Concrete Attacks on Website Authorization by Formal Analysis". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012.* Ed. by Stephen Chong. IEEE Computer Society, 2012, pp. 247–262.

[BCG12] Egon Börger, Antonio Cisternino, and Vincenzo Gervasi. "Contribution to a Rigorous Analysis of Web Application Frameworks". In: *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012.* Ed. by John Derrick et al. Vol. 7321. Lecture Notes in Computer Science. Springer, 2012, pp. 1–20.

[Ber+17] Robin Berjon et al., eds. *HTML5.2 - W3C Recommendation.* Dec. 14, 2017. URL: https://www.w3.org/TR/2017/REC-html52-20171214/.

[Ber+18] Adam Bergkvist et al. *WebRTC 1.0: Real-time Communication Between Browsers - W3C Candidate Recommendation.* June 21, 2018. URL: https://www.w3.org/TR/2018/CR-webrtc-20180621/.

[Ber89] Tim Berners-Lee. *Information Management: A Proposal.* 1989. URL: https://www.w3.org/History/1989/proposal.html.

[BJM08a] Adam Barth, Collin Jackson, and John C. Mitchell. "Robust defenses for cross-site request forgery". In: *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008.* ACM, 2008, pp. 75–88.

[BJM08b] Adam Barth, Collin Jackson, and John C. Mitchell. "Securing Frame Communication in Browsers". In: *Proceedings of the 17th USENIX Security Symposium, July 28–August 1, 2008, San Jose, CA, USA.* USENIX Association, 2008, pp. 17–30.

[Bla01] B. Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules". In: *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14).* IEEE Computer Society, 2001, pp. 82–96.

[BLZ18] John Bradley, Torsten Lodderstedt, and Hans Zandbelt. *Encoding claims in the OAuth 2 state parameter using a JWT – draft-bradley-oauth-jwt-encoded-state-08.* IETF. Jan. 2018. Jan. 5, 2018. URL: https://tools.ietf.org/html/draft-bradley-oauth-jwt-encoded-state-08.

[BP10] Aaron Bohannon and Benjamin C. Pierce. “Featherweight Firefox: formalizing the core of a web browser”. In: *Proceedings of the 2010 USENIX conference on Web application development*. USENIX Association, 2010, pp. 11–11.

[Bra12] John Bradley. *The problem with OAuth for Authentication*. Blog post. Jan. 2012. Jan. 28, 2012. URL: http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html.

[BW17] Adam Barth and Mike West. *Cookies: HTTP State Management Mechanism*. Internet-Draft draft-ietf-httpbis-rfc6265bis-02. Work in Progress. Internet Engineering Task Force, Aug. 2017. URL: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-02.

[Cam+18] Brian Campbell et al. *OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens*. Internet-Draft draft-ietf-oauth-mtls-09. Work in Progress. Internet Engineering Task Force, June 2018. URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-mtls-09.

[Che+14] Eric Y. Chen et al. “OAuth Demystified for Mobile Application Developers”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS ’14*. 2014, pp. 892–903.

[CJR11] Suresh Chari, Charanjit S. Jutla, and Arnab Roy. “Universally Composable Security Analysis of OAuth v2.0”. In: *IACR Cryptology ePrint Archive* 2011 (2011).

[DY83] D. Dolev and A.C. Yao. “On the Security of Public-Key Protocols”. In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208.

[Eck10] Peter Eckersley. “How Unique Is Your Web Browser?” In: *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21–23, 2010. Proceedings*. Vol. 6205. Lecture Notes in Computer Science. Springer, 2010, pp. 1–18.

[ES17] Jochen Eisinger and Emily Stark. *Referrer Policy - W3C Candidate Recommendation*. Jan. 26, 2017. URL: https://www.w3.org/TR/2017/CR-referrer-policy-20170126/.

[Fet11] D. Fett. “Formalizing Security Aspects of the Web Platform in Alloy”. Diplomarbeit. Lehrstuhl für Informationssicherheit und Kryptografie, Universität Trier, 2011.

[Fetch] whatwg.org. *Fetch*. Ed. by Anne van Kesteren. URL: http://fetch.spec.whatwg.org/.

[FKS14] Daniel Fett, Ralf Küsters, and Guido Schmitz. “An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System”. In: *35th IEEE Symposium on Security and Privacy (S&P 2014)*. Technical report available at https://arxiv.org/abs/1403.1866. IEEE Computer Society, 2014,

pp. 673–688. URL: https://sec.uni-stuttgart.de/_media/publications/FettKuestersSchmitz-SP-2014.pdf.

[FKS15a] Daniel Fett, Ralf Küsters, and Guido Schmitz. "Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web". In: *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21–25, 2015, Proceedings, Part I*. Vol. 9326. Lecture Notes in Computer Science. Technical report available at https://arxiv.org/abs/1411.7210. Springer, 2015, pp. 43–65. URL: https://sec.uni-stuttgart.de/_media/publications/FettKuestersSchmitz-ESORICS-2015.pdf.

[FKS15b] Daniel Fett, Ralf Küsters, and Guido Schmitz. "SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12–16, 2015*. Technical report available at https://arxiv.org/abs/1508.01719. ACM, 2015, pp. 1358–1369. URL: https://sec.uni-stuttgart.de/_media/publications/FettKuestersSchmitz-CCS-spresso-2015.pdf.

[FKS16] Daniel Fett, Ralf Küsters, and Guido Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *Proceedings of the 23nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. Technical report available at https://arxiv.org/abs/1601.01229. ACM, 2016, pp. 1204–1215. URL: https://sec.uni-stuttgart.de/_media/publications/FettKuestersSchmitz-CCS-2016.pdf.

[FKS17] Daniel Fett, Ralf Küsters, and Guido Schmitz. "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines". In: *IEEE 30th Computer Security Foundations Symposium (CSF 2017)*. Technical report available at https://arxiv.org/abs/1704.08539. IEEE Computer Society, 2017. URL: https://sec.uni-stuttgart.de/_media/publications/FettKuestersSchmitz-CSF-2017.pdf.

[GGS] Google. *How Google Search Works*. Retrieved 2018-01-28. URL: https://www.google.com/intl/ALL/search/howsearchworks/crawling-indexing/.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7–8, 2016, Proceedings*. Vol. 9721. Lecture Notes in Computer Science. Springer, 2016, pp. 300–321.

[Gro+12] Willem De Groef et al. "FlowFox: a web browser with flexible and precise information flow control". In: *the ACM Conference on Computer and Communications*

Security, CCS'12, Raleigh, NC, USA, October 16–18, 2012. ACM, 2012, pp. 748–759.

[Guh+11] Arjun Guha et al. "Verified Security for Browser Extensions". In: *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22–25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011, pp. 115–130.

[HBS16] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. "Information-flow security for JavaScript and its APIs". In: *Journal of Computer Security* 24.2 (2016), pp. 181–234.

[Hic15] Ian Hickson, ed. *HTML5 Web Messaging - W3C Recommendation*. May 19, 2015. URL: http://www.w3.org/TR/2015/REC-webmessaging-20150519/.

[Hom14] Egor Homakov. *How I hacked Github again*. Feb. 2014. URL: http://homakov.blogspot.de/2014/02/how-i-hacked-github-again.html.

[Hua+12] Lin-Shung Huang et al. "Clickjacking: Attacks and Defenses". In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8–10, 2012*. USENIX Association, 2012, pp. 413–428.

[Ian16] Ian Hickson, ed. *Web Storage - W3C Recommendation*. Apr. 19, 2016. URL: http://www.w3.org/TR/2016/REC-webstorage-20160419/.

[IET09] IETF OAuth Working Group. *OAuth Security Advisory: 2009.1 – A session fixation attack against the OAuth Request Token approval flow (OAuth Core 1.0 Section 6) has been discovered*. Apr. 2009. URL: https://oauth.net/advisories/2009-1/.

[IET18] IETF Rtcweb Working Group. *Rtcweb Status Pages*. 2018. URL: https://tools.ietf.org/wg/rtcweb/.

[Jac+07] Collin Jackson et al. "Protecting browsers from DNS rebinding attacks". In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28–31, 2007*. ACM, 2007, pp. 421–431.

[Jac02] Daniel Jackson. "Alloy: A New Technology for Software Modelling". In: *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002*. Ed. by Joost-Pieter Katoen and Perdita Stevens. Vol. 2280. Lecture Notes in Computer Science. Springer, 2002, p. 20.

[JBS16] Mike Jones, John Bradley, and Nat Sakimura. *OAuth 2.0 Mix-Up Mitigation – draft-ietf-oauth-mix-up-mitigation-01*. July 6, 2016. URL: https://tools.ietf.org/html/draft-ietf-oauth-mix-up-mitigation-01.

[Jon+] M. Jones et al. *OAuth 2.0 Token Binding - draft-ietf-oauth-token-binding-07*. URL: https://www.ietf.org/id/draft-ietf-oauth-token-binding-07.txt.

[Kar+07] Chris Karlof et al. "Dynamic pharming attacks and locked same-origin policies for web browsers". In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007*. Ed. by Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson. ACM, 2007, pp. 58–71.

[Ker07] Florian Kerschbaum. "Simple Cross-Site Attack Prevention". In: *Third International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm 2007*. IEEE Computer Society, 2007, pp. 464–472.

[Kum12] Apurva Kumar. "Using automated model analysis for reasoning about security of web protocols". In: *28th Annual Computer Security Applications Conference, ACSAC 2012*. Ed. by Robert H'obbes' Zakon. ACM, 2012, pp. 289–298.

[Kum14] Apurva Kumar. "A Lightweight Formal Approach for Analyzing Security of Web Protocols". In: *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17–19, 2014. Proceedings*. Vol. 8688. Lecture Notes in Computer Science. Springer, 2014, pp. 192–211.

[LM14] Wanpeng Li and Chris J. Mitchell. "Security issues in OAuth 2.0 SSO implementations". In: *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*. 2014, pp. 529–541.

[LM16] Wanpeng Li and Chris J. Mitchell. "Analysing the Security of Google's Implementation of OpenID Connect". In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Vol. 9721. 2016, pp. 357–376.

[Lod+18] T. Lodderstedt et al. *OAuth 2.0 Security Best Current Practice*. May 2018. URL: https://tools.ietf.org/html/draft-ietf-oauth-security-topics.

[Mla+16] Vladislav Mladenov et al. "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect". In: *CoRR* abs/1508.04324v2 (2016). URL: http://arxiv.org/abs/1508.04324v2.

[OAuthD] *oauth.com, Differences Between OAuth 1 and 2*. URL: https://www.oauth.com/oauth2-servers/differences-between-oauth-1-2/.

[Ope15] Open Web Application Security Project (OWASP). *Session Fixation*. Dec. 1, 2015. URL: https://www.owasp.org/index.php/Session_Fixation.

[Ope17] Open Web Application Security Project (OWASP). *Unvalidated Redirects and Forwards Cheat Sheet*. Nov. 9, 2017. URL: https://www.owasp.org/index.php/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.

[Ope18] Open Web Application Security Project (OWASP). *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet*. Mar. 2, 2018. URL: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

[Pai+11] S. Pai et al. “Formal Verification of OAuth 2.0 Using Alloy Framework”. In: *CSNT '11 Proceedings of the 2011 International Conference on Communication Systems and Network Technologies*. Proceedings of the International Conference on Communication Systems and Network Technologies, 2011, pp. 655–659.

[Pay] *PayPal Developer Documentation*. URL: https://developer.paypal.com/.

[Pel+16] Giancarlo Pellegrino et al. “Uses and Abuses of Server-Side Requests”. In: *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19–21, 2016, Proceedings*. Vol. 9854. Lecture Notes in Computer Science. Springer, 2016, pp. 393–414.

[Pop+18] Andrey Popov et al. *Token Binding over HTTP*. Internet-Draft. Work in Progress. Internet Engineering Task Force, June 2018. URL: https://datatracker.ietf.org/doc/html/draft-ietf-tokbind-https-17.

[RFC1034] P.V. Mockapetris. *Domain names - concepts and facilities*. RFC 1034 (Internet Standard). RFC. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936, 8020. Fremont, CA, USA: RFC Editor, Nov. 1987. DOI: 10.17487/RFC1034. URL: https://www.rfc-editor.org/rfc/rfc1034.txt.

[RFC1035] P.V. Mockapetris. *Domain names - implementation and specification*. RFC 1035 (Internet Standard). RFC. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604, 7766. Fremont, CA, USA: RFC Editor, Nov. 1987. DOI: 10.17487/RFC1035. URL: https://www.rfc-editor.org/rfc/rfc1035.txt.

[RFC3986] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (Internet Standard). RFC. Updated by RFCs 6874, 7320. Fremont, CA, USA: RFC Editor, Jan. 2005. DOI: 10.17487/RFC3986. URL: https://www.rfc-editor.org/rfc/rfc3986.txt.

[RFC6265] A. Barth. *HTTP State Management Mechanism*. RFC 6265 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Apr. 2011. DOI: 10.17487/RFC6265. URL: https://www.rfc-editor.org/rfc/rfc6265.txt.

[RFC6454] A. Barth. *The Web Origin Concept*. RFC 6454 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Dec. 2011. DOI: 10.17487/RFC6454. URL: https://www.rfc-editor.org/rfc/rfc6454.txt.

[RFC6455] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455 (Proposed Standard). RFC. Updated by RFCs 7936, 8307. Fremont, CA, USA: RFC Editor, Dec. 2011. DOI: 10.17487/RFC6455. URL: https://www.rfc-editor.org/rfc/rfc6455.txt.

[RFC6749] D. Hardt (Ed.) *The OAuth 2.0 Authorization Framework*. RFC 6749 (Proposed Standard). RFC. Updated by RFC 8252. Fremont, CA, USA: RFC Editor, Oct. 2012. DOI: 10.17487/RFC6749. URL: https://www.rfc-editor.org/rfc/rfc6749.txt.

[RFC6750] M. Jones and D. Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC 6750 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Oct. 2012. DOI: 10.17487/RFC6750. URL: https://www.rfc-editor.org/rfc/rfc6750.txt.

[RFC6797] J. Hodges, C. Jackson, and A. Barth. *HTTP Strict Transport Security (HSTS)*. RFC 6797 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Nov. 2012. DOI: 10.17487/RFC6797. URL: https://www.rfc-editor.org/rfc/rfc6797.txt.

[RFC6819] T. Lodderstedt (Ed.), M. McGloin, and P. Hunt. *OAuth 2.0 Threat Model and Security Considerations*. RFC 6819 (Informational). RFC. Fremont, CA, USA: RFC Editor, Jan. 2013. DOI: 10.17487/RFC6819. URL: https://www.rfc-editor.org/rfc/rfc6819.txt.

[RFC7033] P. Jones et al. *WebFinger*. RFC 7033 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Sept. 2013. DOI: 10.17487/RFC7033. URL: https://www.rfc-editor.org/rfc/rfc7033.txt.

[RFC7230] R. Fielding (Ed.) and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7230. URL: https://www.rfc-editor.org/rfc/rfc7230.txt.

[RFC7231] R. Fielding (Ed.) and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7231. URL: https://www.rfc-editor.org/rfc/rfc7231.txt.

[RFC7232] R. Fielding (Ed.) and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*. RFC 7232 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7232. URL: https://www.rfc-editor.org/rfc/rfc7232.txt.

[RFC7233] R. Fielding (Ed.), Y. Lafon (Ed.), and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Range Requests*. RFC 7233 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7233. URL: https://www.rfc-editor.org/rfc/rfc7233.txt.

[RFC7234] R. Fielding (Ed.), M. Nottingham (Ed.), and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Caching*. RFC 7234 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7234. URL: https://www.rfc-editor.org/rfc/rfc7234.txt.

[RFC7235] R. Fielding (Ed.) and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Authentication*. RFC 7235 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7235. URL: https://www.rfc-editor.org/rfc/rfc7235.txt.

[RFC7515] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Signature (JWS)*. RFC 7515 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, May 2015. DOI: 10.17487/RFC7515. URL: https://www.rfc-editor.org/rfc/rfc7515.txt.

[RFC7519] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519 (Proposed Standard). RFC. Updated by RFC 7797. Fremont, CA, USA: RFC Editor, May 2015. DOI: 10.17487/RFC7519. URL: https://www.rfc-editor.org/rfc/rfc7519.txt.

[RFC7523] M. Jones, B. Campbell, and C. Mortimore. *JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants*. RFC 7523 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, May 2015. DOI: 10.17487/RFC7523. URL: https://www.rfc-editor.org/rfc/rfc7523.txt.

[RFC7591] J. Richer (Ed.) et al. *OAuth 2.0 Dynamic Client Registration Protocol*. RFC 7591 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, July 2015. DOI: 10.17487/RFC7591. URL: https://www.rfc-editor.org/rfc/rfc7591.txt.

[RFC7617] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Sept. 2015. DOI: 10.17487/RFC7617. URL: https://www.rfc-editor.org/rfc/rfc7617.txt.

[RFC7636] N. Sakimura (Ed.), J. Bradley, and N. Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Sept. 2015. DOI: 10.17487/RFC7636. URL: https://www.rfc-editor.org/rfc/rfc7636.txt.

[RFC7662] J. Richer (Ed.) *OAuth 2.0 Token Introspection*. RFC 7662 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Oct. 2015. DOI: 10.17487/RFC7662. URL: https://www.rfc-editor.org/rfc/rfc7662.txt.

[RFC768] J. Postel. *User Datagram Protocol*. RFC 768 (Internet Standard). RFC. Fremont, CA, USA: RFC Editor, Aug. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/rfc/rfc768.txt.

[RFC791] J. Postel. *Internet Protocol*. RFC 791 (Internet Standard). RFC. Updated by RFCs 1349, 2474, 6864. Fremont, CA, USA: RFC Editor, Sept. 1981. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/rfc/rfc791.txt.

[RFC793] J. Postel. *Transmission Control Protocol*. RFC 793 (Internet Standard). RFC. Updated by RFCs 1122, 3168, 6093, 6528. Fremont, CA, USA: RFC Editor, Sept. 1981. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/rfc/rfc793.txt.

[Sak+14a] N. Sakimura et al. *OpenID Connect Core 1.0 incorporating errata set 1*. OpenID Foundation. Nov. 8, 2014. URL: http://openid.net/specs/openid-connect-core-1_0.html.

[Sak+14b] N. Sakimura et al. *OpenID Connect Discovery 1.0 incorporating errata set 1*. OpenID Foundation. Nov. 8, 2014. URL: http://openid.net/specs/openid-connect-discovery-1_0.html.

[SB12] San-Tsai Sun and Konstantin Beznosov. "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems". In: *ACM Conference on Computer and Communications Security, CCS'12*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM, 2012, pp. 378–390.

[SBJ14] N. Sakimura, J. Bradley, and M. Jones. *OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1*. OpenID Foundation. Nov. 8, 2014. URL: http://openid.net/specs/openid-connect-registration-1_0.html.

[Sch14] Justin Schuh. *Chromium Blog: The Final Countdown for NPAPI*. Nov. 2014. URL: https://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html.

[Sel14] Jose Selvi. "Bypassing HTTP Strict Transport Security". In: *Blackhat (Europe) 2014*. 2014.

[She+15] Ethan Shernan et al. "More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9–10, 2015, Proceedings*. Vol. 9148. Lecture Notes in Computer Science. Springer, 2015, pp. 239–260.

[Sim] SimilarTech. *Facebook Connect Market Share and Web Usage Statistics. Last visited Nov. 7, 2015*. URL: https://www.similartech.com/technologies/facebook-connect.

[SM14] Mohamed Shehab and Fadi Mohsen. "Towards Enhancing the Security of OAuth Implementations in Smart Phones". In: *2014 IEEE International Conference on Mobile Services*. Institute of Electrical & Electronics Engineers (IEEE), June 2014. DOI: 10.1109/mobserv.2014.15. URL: http://dx.doi.org/10.1109/mobserv.2014.15.

[Sme15] Benjamin Smedberg. *NPAPI Plugins in Firefox - Future Releases*. Oct. 2015. URL: https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/.

[Sol+10] Ashkan Soltani et al. "Flash Cookies and Privacy". In: *Intelligent Information Privacy Management, Papers from the 2010 AAAI Spring Symposium, Technical Report SS-10-05, Stanford, California, USA, March 22–24, 2010*. AAAI, 2010.

[SS13] Sooel Son and Vitaly Shmatikov. "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites". In: *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[SSN18] Nat Sakimura, Anoop Saxena, and Anthony Nadalin. *Financial API (FAPI) WG - OpenID*. OpenID Foundation. 2018. URL: http://openid.net/wg/fapi/.

[STSPre] Chromium Project. *HSTS Preload Submission*. URL: https://hstspreload.appspot.com/.

[Tsc16] Hannes Tschofenig. *OAuth Security Advisory: Authorization Server Mix-Up*. OAuth Working Group mailing list. Jan. 2016. URL: https://www.ietf.org/mail-archive/web/oauth/current/msg15336.html.

[Wan+11] Rui Wang et al. "How to Shop for Free Online - Security Analysis of Cashier-as-a-Service Based Web Stores". In: *32nd IEEE Symposium on Security and Privacy, S&P 2011*. IEEE Computer Society, 2011, pp. 465–480.

[Wan+13] Rui Wang et al. "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization". In: *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14–16, 2013*. USENIX Association, 2013, pp. 399–314.

[WS] whatwg.org. *HTML Living Standard, Web sockets*. July 28, 2018. URL: https://html.spec.whatwg.org/multipage/web-sockets.html.

[Yan+16] Ronghai Yang et al. "Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30–June 3, 2016*. ACM, 2016, pp. 651–662.

[Yos+09] Sachiko Yoshihama et al. "Information-Flow-Based Access Control for Web Browsers". In: *IEICE Transactions* 92-D.5 (2009), pp. 836–850.

[Zhe+15] Xiaofeng Zheng et al. "Cookies Lack Integrity: Real-World Implications". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 707–721. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/zheng.

Academic Curriculum and Publications

Academic Curriculum

January 2017 – August 2018	**University of Stuttgart, Germany.** Ph.D. student at the Institute of Information Security. Supervisor: Prof. Dr. Ralf Küsters
April 2011 – December 2016	**University of Trier, Germany.** Ph.D. student at the Chair of Information Security and Cryptography. Supervisor: Prof. Dr. Ralf Küsters
October 2005 – March 2011	**University of Trier, Germany.** Diploma in Computer Science. Thesis title: *Formalizing Security Aspects of the Web Platform in Alloy* Supervisor: Prof. Dr. Ralf Küsters

Publications

- Daniel Fett, Ralf Küsters, and Guido Schmitz. "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines". In: *IEEE 30th Computer Security Foundations Symposium (CSF 2017).* Technical report available at https://arxiv.org/abs/1704.08539. IEEE Computer Society, 2017. URL: https://sec.uni-stuttgart.de/_media/publications/FettKuestersSchmitz-CSF-2017.pdf

- Daniel Fett and Guido Schmitz. "Pi and More - eine Veranstaltungsreihe rund um ‚kleine Computer'". In: *46. Jahrestagung der Gesellschaft für Informatik, Informatik 2016, 26.–30. September 2016, Klagenfurt, Österreich.* Vol. P-259. LNI. GI, 2016, pp. 1195–1196

- Daniel Fett, Ralf Küsters, and Guido Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *Proceedings of the 23nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2016).* Technical report available at https://arxiv.org/abs/1601.01229. ACM, 2016, pp. 1204–1215. URL: https://sec.uni-stuttgart.de/_media/publications/FettKuestersSchmitz-CCS-2016.pdf

- Daniel Fett, Ralf Küsters, and Guido Schmitz. “SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12–16, 2015.* Technical report available at https://arxiv.org/abs/1508.01719. ACM, 2015, pp. 1358–1369. URL: https://sec.uni-stuttgart.de/_media/publications/FettKuestersSchmitz-CCS-spresso-2015.pdf
- Daniel Fett, Ralf Küsters, and Guido Schmitz. “Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web”. In: *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21–25, 2015, Proceedings, Part I.* vol. 9326. Lecture Notes in Computer Science. Technical report available at https://arxiv.org/abs/1411.7210. Springer, 2015, pp. 43–65. URL: https://sec.uni-stuttgart.de/_media/publications/FettKuestersSchmitz-ESORICS-2015.pdf
- Daniel Fett, Ralf Küsters, and Guido Schmitz. “An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System”. In: *35th IEEE Symposium on Security and Privacy (S&P 2014).* Technical report available at https://arxiv.org/abs/1403.1866. IEEE Computer Society, 2014, pp. 673–688. URL: https://sec.uni-stuttgart.de/_media/publications/FettKuestersSchmitz-SP-2014.pdf

Version: 3df4631622b24608fc48310b9aa559b5447d177e