Institut für Informationssicherheit

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Eine prototypische Protokollimplementierung des OAuth 2.0 Protokolls mit Demonstration von Angriffen

Michael Erdemann

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Ralf Küsters
Betreuer/in:	Pedram Hosseyni, M.Sc.

Beginn am:	14. April 2020
Beendet am:	12. November 2020

Kurzfassung

OAuth 2.0 [15] ist ein bekannter Standard zur Autorisierung, der einem in leicht veränderter Form in vielen Situationen des heutigen Lebens begegnet. Dies können Dienste sein, welche auf Daten des Nutzers zugreifen müssen, um diese zu verarbeiten und für andere Zwecke zu nutzen. Ein Beispiel wären Kalenderdienste, welche Zugriff auf Termine benötigen, damit diese ohne Aufwand des Nutzers übernommen werden können. Da durch ein Aushebeln des Prozesses möglicherweise sensible Daten an einen Angreifer gelangen könnten, ist die Sicherheit ein zentraler Aspekt von OAuth 2.0. Dennoch sind mehrere Angriffe bekannt geworden, die die Sicherheitsvorkehrungen umgehen können. Ziel der Arbeit war es diese Angriffe aufzugreifen und anschaulich darzustellen, damit diese besser verstanden werden können. Dazu wurde eine Testumgebung in Python erschaffen, in denen OAuth 2.0 und OpenID Connect Flows implementiert wurden mit entsprechenden Erweiterungen zur Erhöhung der Sicherheit. Mit dieser können unterschiedliche Angriffe auf OAuth 2.0 simuliert und der dabei entstehende Nachrichtenfluss protokolliert werden. Teilweise wurden auch die entsprechenden Fehlerbehebungen zu Angriffen ergänzt. In dieser Ausarbeitung sollen die Grundlagen von OAuth 2.0 behandelt und die Architekur vorgestellt, die bei den Angriffen verwendet werden. Auch sollen die Art der Implementierung aufgegriffen und diskutiert werden.

Inhaltsverzeichnis

1 Einleitung **11**

2 Grundlagen **13**
2.1 Rollen . . . 13
2.2 Endpunkte . . . 15
2.3 Tokens und Parameter . . . 16
2.4 OAuth 2.0 . . . 18
2.5 OpenID Connect . . . 23

3 Angriffe **25**
3.1 IDP Mix Up Attack . . . 25
3.2 307 Redirect Attack . . . 28
3.3 PKCE Downgrade Attack . . . 30
3.4 Cuckoo's Token Attack . . . 32
3.5 State Leak Attack . . . 34

4 Realisierung **37**
4.1 Ausgangsprojekt . . . 37
4.2 Designentscheidungen . . . 37
4.3 Stand der Technik . . . 41

5 Verwandte Arbeiten **47**

6 Zusammenfassung und Ausblick **49**

Literaturverzeichnis **51**

Abbildungsverzeichnis

2.1 Client Credentials Flow 18
2.2 Resource Owner Password Credentials Flow 19
2.3 Implicit Flow 20
2.4 Authorization Code Flow 21
2.5 Ablaufende Schritte des OpenID Connect Authorization Code Flow 24

3.1 IDP Mix Up Attack 26
3.2 307 Redirect Attack 28
3.3 PKCE Downgrade Attack 30
3.4 Cuckoo's Token Attack 33
3.5 State Leak Attack 35

4.1 Menü der Webseite 44
4.2 Nachrichtenverlauf in der Webseite 45
4.3 Download Bereich auf der Webseite 45

Glossar

Auth-Code Authorization Code: Weitere Informationen siehe Abschnitt 2.3. 15

BasicAuth Methode zur Authentifizierung gegenüber eines Webservers mithilfe von Benutzername und Passwort. 15

CSRF Cross-Site Request Forgery. 17

Django Django ist ein weit verbreitetes Webframework in Python. 37

HTTP Hypertext Transfer Protocol. 29

JWT JSON Web Token. 17

mTLS Mutual TLS authentication. 23

OAuth 2.0 OAuth 2.0 ist ein offener Protokollstandard für die Autorisierung und der Nachfolger von OAuth 1.0. Weitere Informationen siehe Abschnitt 2.4. 3

OpenID Connect OAuth 2.0 Abwandlung zur sicheren Authentifizierung. Weitere Informationen siehe Abschnitt 2.5. 3

PAR Pushed Authorization Requests. 37

PKCE Proof Key for Code Exchange. 21

PKI Public Key Infrastructure. 37

RAR Rich Authorization Request. 37

TLS Transport Layer Security. 17

1 Einleitung

Vor allem in dem letzten Jahrzehnt hat die Digitalisierung Einzug gehalten und die Vernetzung zwischen Diensten ist stetig gewachsen. Immer mehr Menschen nutzen digitale Dienste und die Zahl dieser wächst kontinuierlich. Es ist nicht verwunderlich, dass durch die hohe Zahl an Anwendungen, die sich jeweils auf ein spezifisches Aufgabengebiet konzentrieren, auch der Wunsch zur Verknüpfung dieser Dienste aufgekommen ist. Damit werden die Anwendungen umfangreicher und können dem Nutzer mehr Funktionen anbieten. Unter anderem ist es möglich, dass Anwendungen auf Daten eines anderen Dienstes zugreifen können und der Nutzer seine Daten somit nicht bei mehreren Diensten speichern muss. Diese müssen selber dafür auch keine Infrastruktur entwickeln und anbieten.

Für diese Verbindung zwischen den einzelnen Diensten ist es wichtig, dass die benötigten Daten ausgetauscht werden können. Dabei sollen jedoch keine Informationen geleakt werden, sodass persönliche oder geheime Daten nicht an Angreifer gelangen können. OAuth 2.0 [15] hat sich dabei als eine sichere Variante etabliert, welche auch einfacher in der Verwendung ist als sein Vorgänger. Es unterstützt mehrere Flows, welche unterschiedliche Szenarien einer Autorisierung abdecken. Für die Entwicklung des Standards wurden insbesondere sicherheitsbezogene Aspekte berücksichtigt, dennoch ist es durch seine weite Verbreitung auch Ziel diverser Angriffe. Zum Teil wurden solche Angriffe schon verübt, z. B. auf Facebook [1], andere sind nur in Papern veröffentlicht worden [7]. Deshalb wurde versucht, solche Angriffe mithilfe weiterer Schritte zu unterbinden. Trotz mehreren Erweiterungen des Standards zur Erhöhung der Sicherheit ist ein Abgreifen der Daten unter bestimmten Bedingungen wohl immer noch möglich [1].

Da die Sicherheit der Autorisierung ein Kernbestandteil von OAuth 2.0 ist und eine Behebung dieser Sicherheitslücken essenziell ist, sollten in dieser Arbeit diese Angriffe aufgegriffen und analysiert werden. Schon in vielen Arbeiten wurde das Thema OAuth 2.0 untersucht und auch Angriffe beschrieben, welche ein hohes Sicherheitsrisiko auf bestimmte Flows des Standards bergen. Dennoch wurden bislang die Angriffe nur einzeln untersucht und besonders die theoretischen Details betrachtet. Ein Tool, welches diese vereint und einen Überblick über die zum Teil grundverschiedenen Angriffe gibt und dabei auf konkrete Umsetzungen setzt, gibt es bislang nicht. Durch eine ansprechende Bedienoberfläche und ein Nachrichten Logging, können sich auch Personen, die sich wenig mit den theoretischen Details beschäftigen, ein besseres Bild von dem Angriff machen und diesen mitsamt seinen Auswirkungen begreifen. Deshalb war das Ziel des Projektes, einen Teil der Angriffe in der Praxis umzusetzen und den Nachrichtenfluss zu protokollieren, damit eine Veranschaulichung der Problematik gegeben ist. Dazu wurden die wichtigesten Flows von OAuth 2.0 umgesetzt und um manche Erweiterungen ergänzt.

[1] Die Aussage geht von der direkten Implementierung nach dem OAuth 2.0 Standard [12] aus. In der Praxis zu findende Umsetzungen haben meist zusätzliche Sicherheitsebenen.

Die Grundlagen von OAuth 2.0 und OpenID Connect mit deren Flows, die für diese Implementierung wichtig sind, werden in dieser Ausarbeitung in Kapitel 2 näher betrachtet. Im darauffolgenden Kapitel 3 sollen die im Projekt integrierten Angriffe vorgestellt werden. Darin werden die Voraussetzungen, die Abläufe und die möglichen Fixes für die einzelnen Angriffe geklärt. In Kapitel 4 wird die konkrete Realisierung des Projektes analysiert. Weitere in diesem Kontext relevante Projekte werden in Kapitel 5 vorgestellt.

2 Grundlagen

Bevor das im Rahmen dieser Arbeit entwickelte Demonstrationstool und der Weg der Realisierung vorgestellt werden, sollen im folgenden Kapitel grundlegende Begriffe erläutert werden. Dafür werden zunächst die einzelnen Rollen in Abschnitt 2.1 vorgestellt und auch die verschiedenen Endpunkte werden in Abschnitt 2.2 näher betrachtet. In gleicher Weise ist es wichtig, die gesendeten Tokens und Parameter, welche eine entscheidende Rolle einnehmen, näher zu beleuchten (siehe dazu Abschnitt 2.3). In Abschnitt 2.4 wird auf den Standard OAuth 2.0 [12] eingegangen mit den enthaltenen Flows und Erweiterungen. Danach wird in Abschnitt 2.5 auch der Authentifizierungsstandard OpenID Connect [22] erläutert.

Damit die nachfolgenden Abschnitte besser im Kontext eingeordnet werden können, wird hier die grundsätzliche Konstellation von OAuth 2.0 kurz zusammengefasst:

Es gibt einen Resource Owner, welcher Resourcen in Form von Daten besitzt und diese bei einem Server speichert. Bei diesem Server, der im Folgenden Authorization Server heißt, hat der Nutzer selber einen Account und die Daten des Nutzers werden bei einem weiteren Server, dem Resource Server, gespeichert. Daneben gibt es noch einen weiteren Dienst, der sogenannte Client, welcher autorisiert werden soll, um Zugriff auf die Daten des Nutzers zu bekommen. Bei OpenID Connect findet statt einer Autorisierung eine Authentifizierung statt. Ein Nutzer soll sich bei dem Client anmelden und dafür einen bestehenden Account beim Authorization Server verwenden. Der grundsätzliche Ablauf ähnelt dabei stark dem von OAuth.

2.1 Rollen

Im Folgenden werden die im RFC 6749 [12] vorkommenden Parteien, die in den OAuth 2.0 Flows eine Rolle spielen, eingeführt.

Resource Owner

Der Resource Owner wird im Folgenden teilweise auch nur Nutzer genannt, obwohl dieser nicht notwendigerweise eine Person darstellt. Der Nutzer besitzt selber ein Konto mit Anmeldeinformationen beim Authorization Server. Über dieses Konto hat der Nutzer Zugriff auf bestimmte eigene Daten bzw. Ressourcen, weshalb der Nutzer auch eigentlich *resource owner* genannt wird. Das Interesse des Nutzers ist es, einen Client bei einem Server zu autorisieren für bestimmte Dienste, damit dieser Tätigkeiten für den Nutzer ausführen kann [12].

Browser

Der Browser dient häufig als Interaktionsschnittstelle zwischen dem Nutzer und dem Client bzw. Authorization Server, weshalb dieser auch *User Agent* genannt wird. Über den Browser startet der Nutzer den Autorisierungsvorgang und führt weitere Schritte aus. Weiterleitungen, die vom Authorization Server oder Client gestartet worden sind, werden im Browser ausgeführt und ermöglichen einen fließenden Wechsel der Kommunikationspartner und einen Austausch der dabei entstehenden Werte [12].

Client

Der Client ist derjenige, welchem Zugriff auf bestimmte Daten des Resource Owners erteilt werden soll. In den meisten Flows handelt es sich dabei um externe Daten eines Resource Owner, es können aber auch Daten des Clients selber sein. In diesem Fall nimmt der Dienst neben der Rolle des Clients auch die des Resource Owners ein. Der Client bietet meist verschiedene Authorization Server an, bei denen dieser autorisiert werden kann. Je nach Anwendungsfall und Implementierung unterstützt der Client unterschiedliche Sicherheitsvorkehrungen und Standards, die bei den OAuth 2.0 Flows Anwendung finden. Clients werden mithilfe einer Client ID unterschieden, welche von dem Authorization Server ausgestellt wird. Dadurch ist diese auch nur bezüglich eines Authorization Servers eindeutig. Die Client ID ist dabei keine geheime Information [12].

Authorization Server

Der Authorization Server wird für jeden Flow benötigt. Der Resource Owner ist dabei mit einem Konto beim Server registriert, welches meist mit Benutzerdaten geschützt ist. Für die verschiedenen Flows ist es wichtig, dass der Client beim Authorization Server registriert ist. Dabei wird eine Client ID vom Server an den Client vergeben und die Redirection URI des Clients an den Server übertragen. Passwörter in Form von Client Secrets werden teilweise für eine Authentifizierung gebildet und mit dem Client ausgetauscht. Damit hängen die Sicherheitsvorkehrungen und Standards bei den Flows nicht nur vom Client, sondern auch von dem Server ab [12].

Resource Server

Der Resource Server wird meist nur am Schluss eines Flows benötigt. Dieser Server bietet den Resource Endpunkt an, durch den der Client Zugriff auf die Daten des Nutzers oder denen des Clients erlangen kann. Teilweise ist der Resource Server mit dem im vorherigen Absatz definierten Authorization Server identisch, kann aber auch davon eigenständig sein. Dies kann unter anderem sein, wenn der Authorization Server eine Kooperation mit einem anderen Unternehmen hat, welches die Daten des Servers verwalten soll [12].

2.2 Endpunkte

Zum Durchführen von OAuth 2.0 oder OpenID Connect werden verschiedene Endpunkte benötigt. In manchen Flows sind viele davon verpflichtend zu nutzen, aber auch optionale Endpunkte werden angeboten. Die Aufgaben der einzelnen Endpunkte sollen im Folgenden geklärt werden.

Authorization Endpunkt

Zu diesem Endpunkt als Teil des Authorization Servers (siehe Abschnitt 2.1) wird der Nutzer oder vielmehr dessen Browser vom Client direkt nach dem Starten des entsprechenden Flows weitergeleitet. Bei diesem findet die erste Kommunikation zwischen dem Resource Owner und dem gewählten Authorization Server statt. Die Aufgabe dieser Schnittstelle ist es, dass der Nutzer sich bei dem Server anmelden kann, um sich zu authentifizieren und den Autorisierungsvorgang zu bestätigen. In der Regel werden für die Nutzerauthentifizierung die Anmeldedaten (z. B. bestehend aus Benutzername und Passwort) des Resource Owners benötigt. Auf das genaue Vorgehen wird aber nicht in dem OAuth 2.0 Standard [12] eingegangen und wird dem Authorization Server überlassen. Weiterhin startet dieser Endpunkt die Weiterleitung zurück zu dem Client. Dabei wird entweder direkt das Access Token oder der Authorization Code (Auth-Code) erstellt und mitgegeben. Dies hängt von dem *response_type* Parameter ab, welcher für den Auth-Code den Wert „code“ oder für das Access Token den Wert „token“ annehmen kann [12].

Token Endpunkt

Der Token Endpunkt stellt die zentrale Schnittstelle des Authorization Servers (siehe Abschnitt 2.1) dar, welche die Aufgabe hat, den Client zu verifizieren (optional), ein Access Token zu generieren und dieses anschließend zurückzugeben. Der Client wird mithilfe der zwischen dem Authorization Server und dem Client abgestimmten Client ID identifiziert und falls möglich authentifiziert. Die Authentifizierung wird in der Regel mit BasicAuth realisiert, kann aber auch auf andere Arten erfolgen. Weitere Überprüfungen und Sicherheitsaspekte unterscheiden sich nach dem verwendeten OAuth 2.0 Flow. Teilweise wird zusätzlich der Resource Owner authentifiziert, weshalb die Anmeldedaten des Nutzers der Anfrage beigefügt werden müssen, oder es wird der vom Authorization Endpunkt ausgestellte Auth-Code mitgeschickt und beim Token Endpunkt überprüft. Der Token Endpunkt bildet damit eine weitere wichtige Zwischenstelle für die Sicherheit des Standards, damit die Daten nicht von anderen abgefragt werden können. Denn erst das ausgestellte Access Token (siehe Abschnitt 2.3) fungiert als Schlüssel für die Ressourcen, welcher in dem Endpunkt nur kontrolliert herausgegeben werden soll [12].

Redirection Endpunkt

Der Redirection Endpunkt ist beim Client zu finden. Er dient als Schnittstelle, an den der Authorization Server nach einer erfolgreichen Nutzerauthentifizierung beim Authorization Endpunkt den Browser weiterleitet und dabei wichtige Parameter, u. a. das Access Token oder den Auth-Code, an

den Client sendet. Die Adresse des Endpunktes wird *redirection URI* genannt und wird in der Regel dem Authorization Server bei der Registrierung des Clients beim Server übermittelt. Der Wert kann auch in der Anfrage an den Authorization Endpunkt mitangegeben werden [12].

Resource Endpunkt

Im Gegensatz zu den vorherigen Endpunkten befindet sich der Resource Endpunkt nicht zwingend bei dem Authorization Server, sondern bei dem Resource Server (siehe Abschnitt 2.1). Der Resource Endpunkt ist neben dem Token Endpunkt ein weiterer essentieller Teil von OAuth 2.0 und OpenID Connect. Denn durch den Token Endpunkt wurde bislang nur ein Token zurückgegeben, welches selber keinen Informationsgehalt für den Client liefert. Erst bei dem Resource Endpunkt kann das Access Token eingelöst werden, um Zugriff auf die Ressourcen zu erlangen. Zu jedem Authorization Server kennt der Client die Adresse des Resource Endpunkts, damit dieser zielgerichtet Anfragen ausführen kann. Details zu der Herkunft und Speicherung dieser Information sind dabei im OAuth 2.0 Standard [12] nicht näher erläutert und können variieren [12].

UserInfo Endpunkt

Der UserInfo Endpunkt stellt eine Besonderheit der hier dargestellten Endpunkte dar. Denn dieser findet sich nur in OpenID Connect und ähnelt dem Resource Endpunkt. Über diese Schnittstelle kann der Client weitere Informationen über den Nutzer erfragen. Dafür wird ähnlich zum Resource Endpunkt ein Access Token zur Verifizierung benötigt. Die Daten, die abgefragt werden können, werden anfangs über den Scope bestimmt und können z. B. die E-Mail Adresse oder die Telefonnummer sein [22].

2.3 Tokens und Parameter

Für die Kommunikation zwischen den einzelnen Rollen und Endpunkten spielen spezifische Tokens oder Parameter eine wichtige Rolle, da diese entweder die Sicherheit erhöhen oder zum Informationsaustausch beitragen sollen.

Access Token

Das Access Token ist in allen Flows in OAuth 2.0 und OpenID Connect zu finden als „Schlüssel“ zu den Ressourcen. Es ist das Kernelement, auf dem OAuth 2.0 aufbaut. In den meisten Flows muss das Token über den Token Endpunkt (siehe Abschnitt 2.2) erworben werden, kann aber auch direkt in der Weiterleitung vom Authorization Endpunkt (siehe Abschnitt 2.2) an den Client als Fragment vorliegen. In beiden Fällen ist für die Herausgabe des Access Tokens eine Verifizierung nötig, entweder des Nutzers oder des Clients [12].

Authorization Code (Auth-Code)

Der Auth-Code findet nur in ausgewählten Flows Verwendung. Der Wert stellt einen Zwischenschritt zur Generierung des Access Tokens und schließlich für den Zugriff auf die geschützten Daten dar. Dies soll der erweiterten Sicherheit dienen, da dadurch weitere Überprüfungen bezüglich der Validierung des Clients möglich werden. Des Weiteren wird sonst das Access Token über den Browser weitergeleitet, wodurch dies vom Nutzer und anderen Instanzen mit Zugriff auf den Browser gelesen werden könnte. Durch die Verwendung des Auth-Codes verschwindet somit ein etwaiges Sicherheitsrisiko. Für das Erlangen des Access Tokens wird dafür der Token Endpunkt (siehe Abschnitt 2.2) erweitert, sodass dieser nicht nur den Client validiert (optional), sondern auch den in der Anfrage angegebenen Auth-Code prüft [12].

ID Token

Das ID Token wird nur bei OpenID Connect ausgegeben. Es ist kein Ersatz für das Access Token, da dieses nur zum sicheren Informationsaustausch genutzt wird. In dem ID Token werden allgemeine Informationen des Benutzers und zur Authentifizierung, u. a. Ausstellungs- und Zielort, Gültigkeit und Details des letzten Logins, gespeichert. Diese Informationen werden auch Claims genannt. Das ID Token besteht aus 3 Abschnitten, die mit einem Punkt voneinander getrennt sind. Der erste Teil stellt den Header dar, welcher den verwendeten Signieralgorithmus angibt. Eine Signierung ist aber nicht Pflicht, da dieses Token über Transport Layer Security (TLS) übertragen wird. Der 2. Teil enthält die gesamten Claims. Im dritten Teil ist die Signatur gespeichert. Das gesamte Token liegt dabei als JSON Web Token (JWT) vor [14][22].

Scope

Mit dem Scope soll festgelegt werden, auf welche Daten der Client Zugriff bekommen will. Diese Angabe wird vom Client getätigt und sollte bei dem Authorization Endpunkt des Authorization Servers bei der Bestätigung der Autorisierung auch für den Nutzer angezeigt werden [12].

State

Der State Parameter kann von dem Client erstellt und bei der Weiterleitung zu dem Authorization Server mitgegeben werden. Dies ist zwar keine Pflicht, hat aber eine besondere Aufgabe, da dieser Cross-Site Request Forgery (CSRF) Angriffe verhindern soll. CSRF Angriffe zielen darauf ab gefälschte Nachrichten von außerhalb an bestimmte Endpunkte des Systems zu schicken. Ziel der Attacke ist es, dass die Instanz, auf die der Angriff zielt, die Nachricht als eine Echte erkennt und verarbeitet. Bei OAuth 2.0 können die Angreifer z. B. Nachrichten des Authorization Servers fälschen, sodass der Client ein anderes Access Token oder Auth-Code bekommt, welches mit den Angreifer Ressourcen verknüpft ist und nicht mit denen des Resource Owner. Zur Verhinderung des Szenarios muss der Authorization Server, wenn dieser den State vom Client erhält, diesen Wert bei der Weiterleitung zurück zu dem Client Endpunkt beifügen. Da außer dem Client und dem Authorization Server keiner den State kennen sollte und eine sichere TLS Verbindung genützt wird, kann ein Außenstehender den State Wert bei den Nachrichten nicht korrekt angeben, sodass diese

Nachrichten an den Client Endpunkt vom Client abgelehnt werden. Dennoch ist ein Angriff bekannt geworden, welcher diesen Wert abgreifen kann und somit die Sicherheit des Verfahrens untergräbt (siehe Abschnitt 3.5) [12].

2.4 OAuth 2.0

OAuth 2.0 [15] ist der 2012 erschienene Nachfolger des OAuth 1.0 Protokolls [11]. Es stellt eine Möglichkeit zur Autorisierung von Drittanbieteranwendungen (Clients) dar, damit diese im Auftrag vom Resource Owner Zugriff auf deren Ressourcen bei einem Authorization Server erlangen können. Im Gegensatz zu früheren Ansätzen bei der Autorisierung, bei denen zum Teil direkt die Anmeldedaten mit dem Client geteilt wurden, lag der Fokus bei OAuth 2.0 stark auf der Entwicklung eines sicheren Verfahrens zur Autorisierung. Dazu wurde unter anderem das Access Token eingeführt, welches in Abschnitt 2.3 beschrieben ist. Damit wird eine Speicherung der Anmeldedaten des Resource Owners beim Client nicht mehr benötigt. Entwickelt wurde OAuth 2.0 von der IETF OAuth Working Group. Der Standard wird in RFC 6749 [12] eingeführt und näher beschrieben.

2.4.1 Flows

Im OAuth 2.0 Standard [12] werden mehrere Wege, welche auch als *Flow* oder *Grant (Type)* bezeichnet werden, vorgestellt. Diese unterscheiden sich dabei teilweise sehr stark in Bezug auf die Anforderungen und dem Sicherheitslevel. Alle Abbildungen der Flows wurden mit dem in dieser Arbeit entwickelten Program selbst erstellt unter Zuhilfenahme von Annex Language [8] (siehe dazu Abschnitt 4.3.2).

Client Credentials Flow

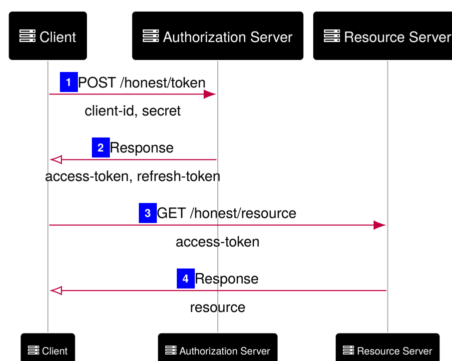

Abbildung 2.1: Client Credentials Flow

Dies ist die einfachste mögliche Art der Autorisierung mit OAuth 2.0. Denn im Gegensatz zu den anderen Flows wird keine Nutzerinteraktion benötigt und der Flow kann vom Client direkt gestartet werden, da der Client in diesem Flow selbst die Rolle des Resource Owners übernimmt. Wie der Name des Flows schon vorwegnimmt, werden nur die *Client Credentials* d. h. die Anmeldeinformationen des Clients benötigt [12].

Der zugehörige Flow wird in Abbildung 2.1 illustriert. In Schritt 1 kann der Client direkt eine Anfrage an den Authorization Server schicken und sich dabei authentifizieren. Wenn der Client gültige Werte verwendet und auch beim Authorization Server bekannt ist, wird ein Access-Token generiert und dem Client in Schritt 2 zusammen mit anderen Informationen übermittelt. Durch Nutzung dieses Tokens kann der Client Zugriff auf seine Ressourcen beim Authorization Server bekommen. Dieser Austausch Access Token gegen Resourcen wird in Schritt 3 und Schritt 4 gezeigt [12].

Der *Client Credentials Grant* eignet sich nicht für den Zugriff auf Daten eines Benutzers, denn dazu werden nutzerspezifische Informationen allein der Zuordnung zum entsprechenden Nutzer wegen zwingend benötigt. Dennoch kann dieser Grant nützlich sein, um den Zugriff auf die eigenen Daten des Clients zu steuern [12].

Resource Owner Password Credentials Flow

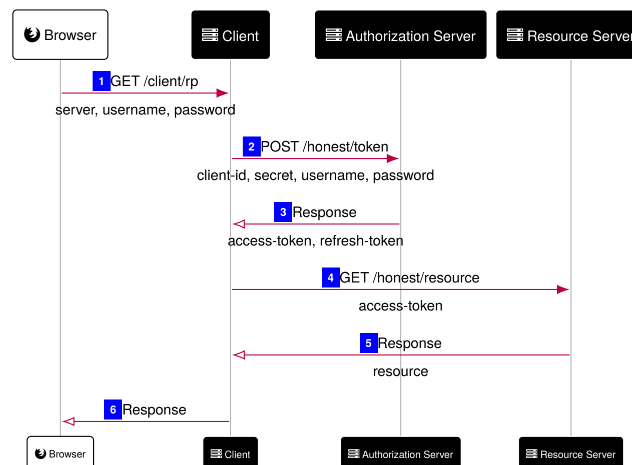

Abbildung 2.2: Resource Owner Password Credentials Flow

Der Resource Owner Password Credentials Flow, zu sehen in Abbildung 2.2, erweitert den Client Credentials Flow um Nutzerinformationen beim Authorization Server. Diese können variieren, sind aber in der Regel der Benutzername und das zugehörige Passwort. Damit sind die Voraussetzungen geschaffen, um Zugriff auf nutzerspezifische Daten beim Authorization Server zu erhalten. Damit dies bewerkstelligt werden kann, wird die denkbar einfachste Variante gewählt. Der Nutzer schickt seine Anmeldedaten vom Authorization Server direkt an den Client, wie in Schritt 1 zu sehen ist. Die weiteren Schritte ähneln denen des Client Credentials Flows. Doch im Gegensatz dazu wird nach der ersten Anfrage an den Authorization Server in Schritt 2 nicht nur der Client, sondern

auch der Nutzer mithilfe der Anmeldedaten authentifiziert. Mit dem erhaltenen Access-Token kann nun der Client wieder Zugriff auf die Ressourcen beim Resource Server in Schritt [4] und Schritt [5] bekommen, welche aber diesmal Nutzerdaten beim Authorization Server widerspiegeln [12].

Diese Einfachheit des Vorgehens speziell für den Nutzer, hat aber auch seinen Preis. Denn der Client kennt damit die Anmeldeinformationen des Nutzers und könnte diese ausnützen oder weitergeben. Dadurch, dass die Nutzerdaten aber in ein Access Token umgewandelt werden und nur diese Information für den Zugriff auf den Resource Server notwendig ist, werden die Anmeldedaten des Nutzers nur einmal gebraucht und müssen nicht notwendigerweise gespeichert werden. Dennoch ist diese Variante weniger verbreitet und ist aufgrund der benötigten Vertrauensbasis zu dem Client nur für die Clients vorgesehen, die als Anwendung lokal laufen z. B. als Teil des Betriebssystems [12].

Implicit Flow

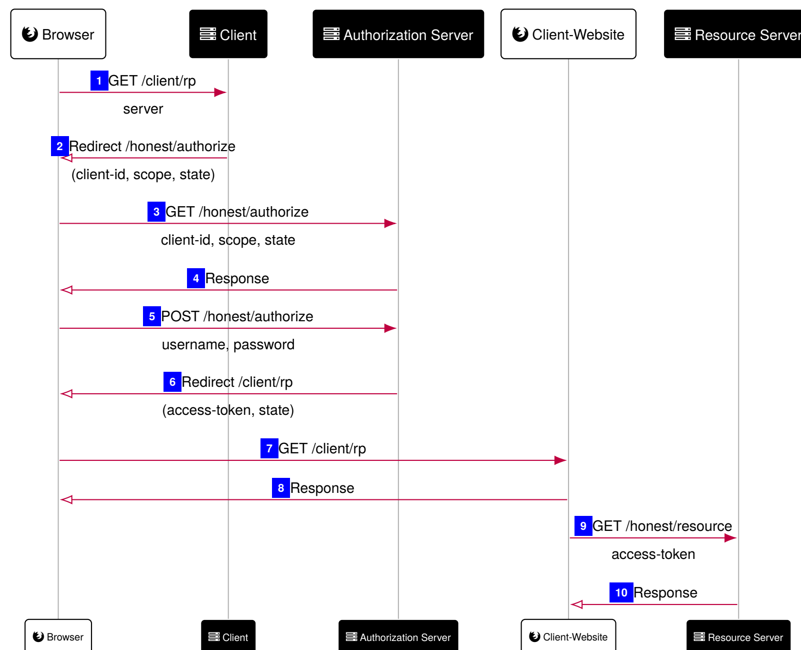

Abbildung 2.3: Implicit Flow

Mit dem Implicit Flow wird der Autorisierungsprozess im Gegensatz zu den vorherigen Varianten deutlich komplexer. Denn der Client soll hierbei keine Anmeldedaten des Nutzers beim Authorization Server bekommen. Deshalb wird der Nutzer mithilfe des Browsers vom Client zum Authorization Server weitergeleitet. Dies wird in Schritten [2] und [3] von Abbildung 2.3 gezeigt. In Schritt [5] meldet sich der Nutzer dort mit seinen Anmeldedaten direkt bei dem Authorization Server an und kann dem Client den Zugriff auf seine Daten erlauben. Wenn der authentifizierte Nutzer diesem zustimmt, wird dieser mit einer festgelegten *redirection URI* (siehe Abschnitt 2.2) zu dem Client weitergeleitet (zu sehen in den Schritten [6] und [7]). Bei dieser Prozedur wird das Access Token direkt generiert und der URI beigefügt. Der Implicit Flow wurde optimiert für webbasierte Clients, weshalb das Access Token als Fragment in der URI vorliegt. Clients ohne eine Browser Integration

können somit nicht ohne Weiteres auf dieses Token zugreifen. Das Einlösen des Access Tokens, um Zugriff auf die Nutzerdaten zu erhalten, ist mit denen der anderen Varianten identisch und in den Schritten 9 und 10 dargestellt [12].

Obwohl der Nutzer authentifiziert wird und der Client darüber keine Informationen erlangt, birgt diese Methode dennoch Risiken. Denn es findet keine Authentifizierung des Clients statt. Dadurch ist es nicht gesichert, dass nur der echte Client das Access Token erhalten kann. Zudem wird das Access Token über den Browser verschickt, sodass sowohl der Nutzer als auch Anwendungen mit Zugang zu dem Browser das Token auslesen könnten [12].

Authorization Code Flow

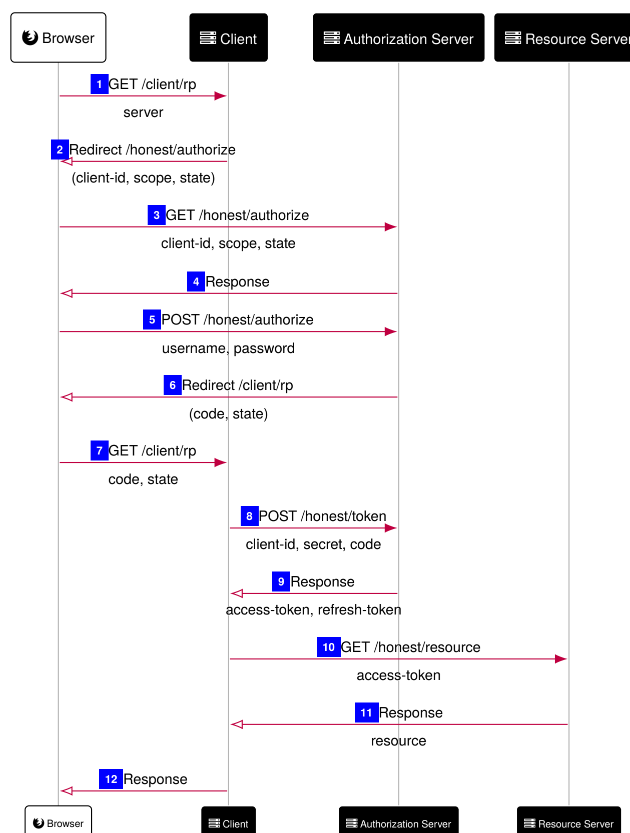

Abbildung 2.4: Authorization Code Flow

Dies ist der komplexeste Weg der hier dargestellten Flows, bietet aber auch viele Vorteile. Denn es kann, falls es möglich ist, neben einer Nutzer- auch eine Client Authentifizierung beim Authorization Server stattfinden. Des Weiteren wird im Vergleich zum Implicit Flow aus dem vorherigen Abschnitt, das Access Token nicht über den Browser geschickt und auch kein Fragment zur Übertragung des Parameters eingesetzt. Damit entfallen mögliche Sicherheitsrisiken. Daneben besteht bei diesem Flow die Möglichkeit, die Sicherheit durch Erweiterungen z. B. Proof Key for Code Exchange (PKCE) zu steigern.

Dazu sind aber im Vergleich zu dem Implicit Flow zusätzliche Schritte nötig, die in Abbildung 2.4 sichtbar werden. Dennoch ähnelt das Vorgehen stark dieser zuvor vorgestellten Variante. Die ersten Schritte bis Schritt 5 sind identisch. Doch nach der Authentifizierung des Nutzers auf der Server-Seite wird nun nicht direkt das Access Token ausgestellt, sondern der Auth-Code (siehe Abschnitt 2.3). Dieser wird als Parameter bei der Weiterleitung zurück zu dem Client in den Schritten 6 und 7 beigefügt. Durch das Beifügen dieses Parameters kann der Client nun in Schritt 8 über den Token Endpunkt des Authorization Servers ein Access Token anfragen. Bei der Anfrage kann die Client Authentifizierung stattfinden, indem ein Authentifizierungsheader beigefügt wird. Im Hintergrund wird, wenn die optionale Verifizierung des Clients erfolgreich war, das Access Token bei dem Authorization Server erstellt und der Antwort hinzugefügt (Schritt 9). Nun kann dieses wieder, wie in den Schritten 10 und 11 illustriert, für einen Zugriff auf die Nutzerdaten verwendet werden.

Der Schritt über den Auth-Code erscheint zunächst unnötig und kompliziert, doch damit wird sichergestellt, dass nur der vom Nutzer gewählte Client das Access Token und damit den Zugriff auf die sensiblen Daten bekommen kann [12].

2.4.2 Erweiterungen

Im Folgenden werden zwei wichtige Erweiterungen von OAuth 2.0 vorgestellt.

Proof Key for Code Exchange (PKCE)

PKCE [21] stellt eine wichtige Erweiterung für OAuth 2.0 dar. Dieser Standard ermöglicht es, die Sicherheit von OAuth 2.0 weiter zu steigern, indem der Auth-Code mit einem bestimmten Wert verknüpft wird. Dieser Wert wird auch *Code Challenge* genannt und wird von dem Client generiert und der Weiterleitung zu dem Authorization Server beigefügt. Für die Erstellung dieser Code Challenge liegt dem Client ein zunächst geheimer Ursprungswert, der *Code Verifier* vor, der selber zufällig gewählt sein kann und eine Nonce darstellt. Der Code Verifier entspricht entweder direkt der Code Challenge oder er wird noch mit einer ausgewählten Methode vorher gehasht. Die gewählte Hashfunktion oder der Wert „plain“, falls nicht gehasht wurde, muss bei der Weiterleitung zum Authorization Server ebenfalls mitgegeben werden. Dies entspricht in Abbildung 2.4 den Schritten 2 und 3, bei denen als Parameter die Werte *Code Challenge* und *Code Challenge Method* mitangegeben werden müssen. Damit der Client den Auth-Code bei dem Token Endpunkt des Authorization Servers einlösen kann, wird nun dieser Code Verifier benötigt und muss entsprechend der Anfrage hinzugefügt werden in Schritt 8. Der Authorization Server prüft nun die Gültigkeit, indem dieser den Code Verifier mit der gewählten Hashfunktion hasht und diesen mit der Code Challenge vergleicht. Wenn diese nicht übereinstimmen, wird der Flow abgebrochen. Damit wird verhindert, dass fremde Clients, die den Auth-Code abgreifen konnten, diesen zum Erzeugen des Access Tokens und damit für den Zugriff auf fremde Daten nutzen können. Ein solcher Angriff wird direkt in [21] beschrieben. Bei diesem Szenario ist es möglich, dass der Angreifer die Weiterleitung vom Authorization Endpunkt inklusive dem Auth-Code abfangen kann, da keine TLS Verbindung genutzt wurde, z. B. bei einer Kommunikation zwischen Anwendungen innerhalb des Client Betriebssystems. Mit dem gewonnenen Auth-Code kann der Angreifer nun das Access Token bei dem Token Endpunkt erhalten. Dieser muss dafür die Client ID und das Client Secret, falls es vorliegt, kennen, welches aber möglich ist zu bekommen, da alle nativen App Client Instanzen die

gleiche Client ID besitzen und das Client Secret bei Binäranwendungen nicht sicher vorliegt. Trotz der erweiterten Sicherheit gibt es dennoch Angriffe, die diese Vorkehrung untergraben können (siehe Abschnitt 3.3).

Mutual TLS authentication (mTLS)

Mit mTLS [3] wurde das Verifizierungsmodell für den Client im Vergleich zu BasicAuth komplett umgebaut. Statt zwischen dem Client und dem Authorization Server nur ein Client Secret auszutauschen, welches als Passwort dient, soll nun ein Zertifikat den Client ausweisen. Dieses muss statt dem Client Secret in Schritt 8 an den Token Endpunkt geschickt werden. Daneben wird das dort ausgestellte Access Token mit diesem Zertifikat gekoppelt, sodass dieses vom Resource Server verifiziert werden kann. Dieses Vorgehen ist in den meisten Fällen von der Authentifizierung des Clients entkoppelt und kann z. B. mit dem Hash des Zertifikates erfolgen. Das Zertifikat muss dann nicht nur bei der Anfrage an den Token Endpunkt beigefügt werden, sondern auch bei einem Zugriff auf den Resource Server mit dem Access Token in Schritt 10. Damit soll sichergestellt werden, dass nur der richtige Client auf diese vertraulichen Daten zugreifen kann, selbst wenn das Access Token von einem Angreifer gestohlen wurde. Auch für diese Erweiterung wurden schon Möglichkeiten gefunden, diese Barrikaden unter gewissen Umständen zu umgehen und damit erneut ein Sicherheitsrisiko zu erzeugen (siehe Abschnitt 3.4).

2.5 OpenID Connect

OpenID Connect [16] stellt eine angepasste Form von OAuth 2.0 für die Authentifizierung von Nutzern dar. Damit ist es möglich, dass Nutzer, die bereits beim Authorization Server ein Konto besitzen, auch beim Client angemeldet werden können. Dazu ist es nötig, dass der Client notwendige Informationen über den Nutzer bekommt, welche neben sicherheitsrelevanten Details wie der Zeitpunkt der letzten Anmeldung auch personenbezogene Daten wie die (E-Mail-) Adresse des Nutzers sein können. Dazu wurden neue Konzepte zur Übertragung der Informationen entwickelt. Zum einen wurde dafür das in Abschnitt 2.3 vorgestellte ID-Token eingeführt. Außerdem entstand ein neuer UserInfo Endpunkt (siehe Abschnitt 2.2) beim Authorization Server, bei denen weitere nutzerspezifische Daten abgerufen werden können [14] [16].

2.5.1 Authorization Code Flow

In Abbildung 2.5 wird der Authorization Code Flow von OpenID Connect gezeigt. Dieser ähnelt nicht nur vom Namen her dem Authorization Code Flow von OAuth 2.0, sondern erweitert eben diesen um OpenID Connect Funktionen. Bis zum Schritt 8 entspricht der Ablauf direkt der Variante von OAuth 2.0. Einziger Unterschied ist, dass der Scope, der in 1 mitgeschickt wird, jedes Mal mit „openid“ beginnt, sodass der Authorization Server die Absicht des Clients zur Authentifizierung mit OpenID Connect erkennen kann. Doch nach der Autorisierung des Clients beim Token Endpunkt wird nicht nur das Access Token, sondern auch das ID Token erstellt. Dieses wird daraufhin in Schritt 9 dem Client in der Antwort beigefügt. Der Client sollte nach Erhalt des Tokens die Signatur und die enthaltenen Daten davon prüfen und bei Unstimmigkeiten diesen Vorgang abbrechen. Falls die Daten des ID Tokens korrekt sind, kann der Client den Flow fortsetzen. Er hat die Möglichkeit,

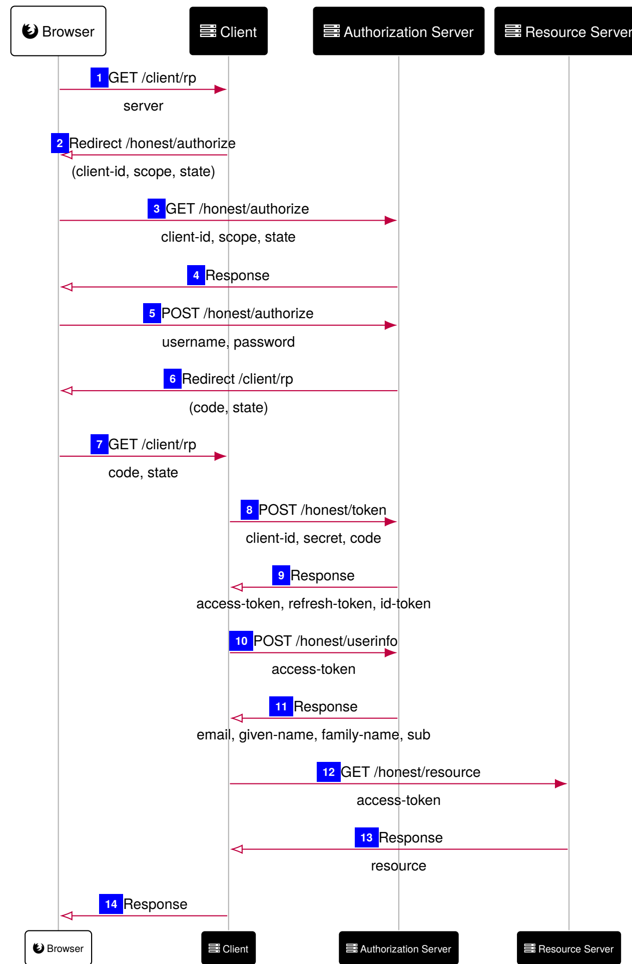

Abbildung 2.5: Ablaufende Schritte des OpenID Connect Authorization Code Flow

weitere Daten beim Authorization Server anzufragen. Dazu schickt der Client eine POST Request an den User Info Endpunkt des Authorization Servers, bei dem er das erhaltene Access Token angibt. Dies dient als Sicherheitsvorkehrung. Als Antwort werden die Claims zurückgegeben. Welche Daten dies sind, kann der Client steuern, indem er diese in dem Scope angibt. Diese Prozedur ist in den Schritten 10 und 11 dargestellt. Analog zu OAuth 2.0 hat der Client in den Schritten 12 und 13 die Möglichkeit, mit dem Access Token Zugang zu Ressourcen auf dem Resource Server zu erhalten [16].

3 Angriffe

In diesem Kapitel sollen eine Auswahl der bereits bekannten Angriffe auf den Authorization Code Flow (siehe Abschnitt 2.4.1) von OAuth 2.0 näher betrachtet werden, welche auch im Rahmen des entwickeltem Demo Tools (siehe Kapitel 4) implementiert worden sind. Manche dieser Angriffe können auch durch kleinere Anpassungen auf den Implicit Flow (siehe Abschnitt 2.4.1) angewendet werden, welches in dieser Arbeit aber nicht näher behandelt wird. Es werden im Folgenden die vom Angreifer kontrollierten Authorization Server als *Angreifer Server* und die Restlichen als *ehrliche Server* bezeichnet. Alle Abbildungen der Flows wurden mit dem in dieser Arbeit entwickelten Program selbst erstellt [1] unter Zuhilfenahme von Annex Language [8] (siehe dazu Abschnitt 4.3.2).

3.1 IDP Mix Up Attack

Der *IDP Mix Up Attack* [7] hat das Ziel, dass der Angreifer auf die Nutzerdaten bei dem ehrlichen Server zugreifen kann. Um dies zu bewerkstelligen, benötigt der Angreifer ein von dem ehrlichen Server ausgestelltes Access Token. Dies ist aber schwer zu bekommen, da entweder das Token direkt gestohlen werden muss oder das Token mithilfe des Auth-Code angefragt werden muss, welches dem Angreifer auch nicht ohne Weiteres vorliegt. Da beide Werte auch über eine gesicherte TLS Verbindung an den Client oder Server geschickt werden, ist ein Abfangen dieser Werte praktisch unmöglich und der einzige Weg ist es, dass der Client diese Daten direkt an den Angreifer sendet statt an den richtigen Server. Diese Idee scheint zunächst naiv und schwer umzusetzen, aber die nötigen Schritte des Angreifers sind dafür überschaubar. Wobei für diese Methode mehrere Voraussetzungen geschaffen werden müssen.

3.1.1 Voraussetzungen

In der Standardvariante ist ein Netzwerkangreifer nötig, welcher Nachrichten abfangen und modifizieren kann. Daneben gibt es aber auch eine Variante, die ohne Eingreifen in den Nachrichtenfluss auskommt, dafür aber andere Voraussetzungen verlangt (siehe Abschnitt 3.1.4). Es werden zwei Authorization Server betrachtet, von denen einer dem Angreifer gehört oder von diesem korrumpiert und unter seiner Kontrolle ist. Der andere Authorization Server wird normalerweise nicht vom Angreifer kontrolliert. Der Client ist bei beiden Authorization Servern registriert, wobei sich die Client ID bei den Servern unterscheiden können. Eine stärkere Einschränkung, die aber nach dem OAuth 2.0 Standard [12] möglich ist, ist es, dass der ehrliche Server dem Client kein Client Secret zugeordnet hat, wodurch die Client Verifizierung mit dem Secret beim Token Endpunkt

[1] Teilweise wurden die Flows nachträglich gekürzt und bei Abbildung 3.4, die Bezeichnung „Browser“ zu „Attacker Browser“ nachträglich abgeändert.

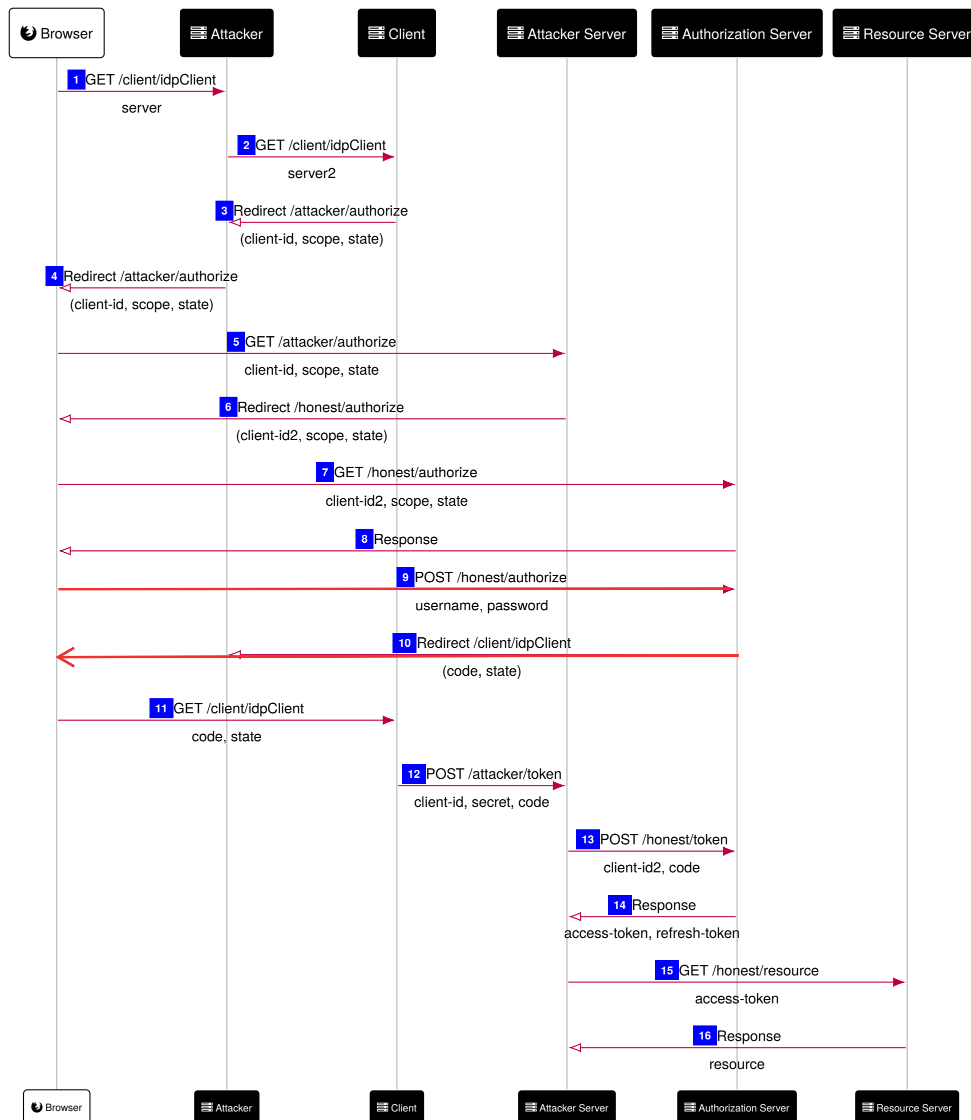

Abbildung 3.1: IDP Mix Up Attack

wegfällt. Für den Angriff in der Standard Variante nehmen wir an, dass der Resource Owner sich bei dem Client anmeldet und dabei den ehrlichen Server auswählt für die Autorisierung. Damit der Angreifer Einfluss auf den ausgewählten Server nehmen kann, ist es wichtig, dass der Schritt 2 nicht über eine TLS Verbindung geschieht, welches auch für OAuth 2.0 keine Voraussetzung ist. Der Angreifer Server benötigt auch die Client ID in Bezug auf den ehrlichen Server. Da dies aber als eine öffentliche und keine geschützte Information vorliegt, ist ein Erlangen davon möglich. Die Client ID kann unter anderem direkt aus der Query ausgelesen werden, nachdem ein Flow mit dem Client und dem ehrlichen Server vom Angreifer gestartet wurde [7].

3.1.2 Ablauf

Der gesamte Ablauf ist in Abbildung 3.1 zu sehen. Damit dieser Angriff funktioniert, muss schon in Schritt [1] in den Authorization Code Flow eingegriffen werden. Denn in diesem Schritt übermittelt der Nutzer dem Client den gewünschten Authorization Server, über den die Autorisierung stattfinden soll. Da es bei diesem Angriff essenziell ist, dass der Client davon ausgeht, dass der Nutzer den Angreifer Server ausgewählt hat, muss diese Nachricht direkt manipuliert werden. Diese Änderung wird in diesem Fall mittels eines Angreifers im Netzwerk realisiert. Dieser fängt die Nachricht in Schritt [1] ab und schickt die manipulierte Nachricht in Schritt [2] weiter an den Client. Da der Client davon ausgeht, dass der Nutzer eine Autorisierung bei dem Angreifer Server wünscht, schickt der Client die Nachrichten für den Authorization Server direkt an den Angreifer. Dadurch wird auch die Weiterleitung zu eben diesem Angreifer Server eingeleitet (zu sehen in den Schritten [3] und [4]) [7].

Damit der Angriff nicht direkt für den Nutzer sichtbar ist, soll er zu dem ausgewählten ehrlichen Server weitergeleitet werden. Um dies zu verwirklichen, wird entweder die Antwort wieder abgefangen und manipuliert, oder wie in der Abbildung 3.1 in den Schritten [5] und [6] zu sehen ist, zuerst zu dem Angreifer Server weitergeleitet, welcher einen dann im Browser direkt zu dem ehrlichen Server weiterleitet. Der Vorteil von dieser Variante ist es, dass kein weiterer Eingriff nötig ist und diese Weiterleitung so schnell erfolgt, sodass der Nutzer davon nichts direkt mitbekommt. Für die Weiterleitung muss neben der geänderten Serveradresse auch die Client ID angepasst werden, da diese sich meist von Server zu Server unterscheidet. Die Schritte [7] bis [11] entsprechen den Schritten [3] bis [7] des Authorization Code Flows (siehe Abbildung 2.4). Der Nutzer authentifiziert sich bei dem ehrlichen Server und bestätigt die Autorisierung. Der daraufhin erstellte Auth-Code wird an den ursprünglichen Client weitergeleitet. Eine Modifizierung durch den Angreifer ist nicht nötig, da der ehrliche Server die Redirection URI des Clients kennt und die Client ID passend für den Authorization Server anfangs geändert wurde [7].

Die vom Client gestartete POST Request an den Token Endpunkt in Schritt [12] entspricht der üblichen Prozedur im Authorization Code Flow. Das Problem an dieser Nachricht ist die Zieladresse, die wieder der Angreifer Server darstellt. Dadurch gelangen verschiedene Parameter an den Angreifer, darunter der Auth-Code. Die Authentifizierung des Clients kann für eine weitere Verwendung nicht genutzt werden, da sich die Authentifizierungsdetails zwischen den Authorization Servern unterscheiden. Dadurch ist es nicht möglich, dass der Angreifer das Client Secret zu dem ehrlichen Server erfährt und somit ist dieser Angriff nur bei Authorization Servern möglich, die kein Client Secret erwarten (siehe Abschnitt 3.1.1). Mit dem gewonnen Auth-Code aus der Anfrage ist es möglich, dass der Angreifer selber den Token Endpunkt des ehrlichen Servers aufruft und dabei diesen Auth-Code Wert beifügt (Schritt [13]). Neben der Client ID ist kein weiterer Authentifizierungswert des Clients nötig. Da der Auth-Code Wert selber vom ehrlichen Server erstellt wurde, akzeptiert dieser. Das in der Antwort in Schritt [14] enthaltene Access Token kann nun direkt beim Resource Server eingelöst werden und der Angreifer bekommt in den Schritten [15] und [16] Zugriff auf die Daten des Nutzers [7].

3.1.3 Fix

Um diesen Angriff vorzubeugen, ist kein ganz neuer Flow vonnöten. Sondern mit einem einzigen zusätzlichen Parameter in einer Anfrage wird der Angriff unmöglich. Dazu wird in Schritt 10 und damit auch in Schritt 11 von der Abbildung 3.1 der Ausgangsserver in einem Parameter, der in dem Beispiel *iss* heißen soll, der Anfrage beigefügt. Der Client muss diesen Wert nur noch mit dem vom Nutzer ausgewählten Authorization Server vergleichen. Wenn diese nicht übereinstimmen, ist ein mutmaßlicher Angriff erkannt worden und der Client sollte den weiteren Flow abbrechen. Auch der in Abschnitt 3.1.2 geschilderte Angriff wird über den *iss* Parameter erkannt, da der Authorization Server, der die Weiterleitung durchführt, der ehrliche Server ist und der Client aber den Angreifer Server als Ursprungsort erwartet. Eine Eingreifen des Angreifers mit dem Ziel, das Token zu modifizieren, ist praktisch unmöglich, da diese Verbindung mit TLS sicher ist [7].

3.1.4 Varianten

Neben dem Ablauf in Abschnitt 3.1.2 gibt es auch eine andere Variante, die weniger Voraussetzungen efordert. Es werden unter anderem keine Manipulationen der Nachrichten benötigt, wodurch der Angreifer weniger mächtig sein muss. Für den Angriff muss davon ausgegangen werden, dass der Nutzer einen Fehler macht und selber in Schritt 1 den Angreifer Server auswählt. Danach entspricht der weitere Flow dem von Abschnitt 3.1.2 [7].

3.2 307 Redirect Attack

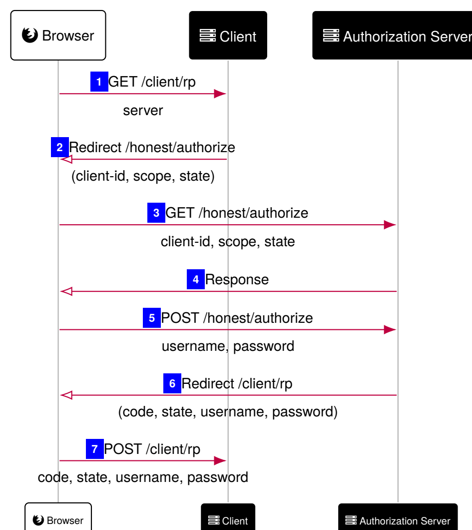

Abbildung 3.2: 307 Redirect Attack

Der *307 Redirect Attack* [7] hat nicht direkt das Ziel, an die Ressourcen des Nutzers zu kommen. Vielmehr sollen die Anmeldedaten des Benutzers bei dem Authorization Server in Erfahrung gebracht werden. Dazu wird eine bestimmte Reihenfolge der Aktionen beim Authorization Server

und den Eigenschaften des 307 Hypertext Transfer Protocol (HTTP) Status Codes ausgenützt, sodass die Benutzerdaten direkt an den Client geschickt werden. Denn beim 307 Redirect werden nicht nur Informationen zur aktuellen Anfrage mitgegeben, sondern auch Daten aus der vorherigen Anfrage. Und dies stellt in diesem Fall die Authentifizierung des Nutzers bei dem Authorization Server dar. Dadurch werden unbeabsichtigt die Benutzerdaten meist bestehend aus Nutzername und Passwort bei der Weiterleitung mitgeschickt. Da der Client für diesen Angriff als vom Angreifer korrumpiert angesehen werden muss, werden diese Daten von diesem aus der Anfrage extrahiert und für einen späteren Missbrauch gespeichert.

3.2.1 Voraussetzungen

Bei dem 307 Redirect Attack ist die Abfolge der Schritte beim Authorization Server und die Art der Weiterleitung vom Authorization Server zu dem Client entscheidend. Damit dieser Angriff Wirkung zeigt, muss der Authorization Server nach der Benutzerauthentifizierung direkt den Nutzer zurück an den Client weiterleiten mithilfe des 307 HTTP Status Codes. Im OAuth 2.0 Standard Hardt et al. [12] wurde die Art der Weiterleitung nicht weiter eingeschränkt und als ein „Implementierungsdetail“ bezeichnet, womit auch die Nutzung des 307 HTTP Status Codes vorkommen kann. Im Gegensatz zum IDP Mix Up Attack soll nun der Client vom Angreifer kontrolliert werden [7].

3.2.2 Ablauf

Die Übersicht der ablaufenden Schritte sind in Abbildung 3.2 dargestellt. Der eigentliche Angriff besitzt keine abweichende Struktur als der normale Authorization Code Flow. Es wird nur eine bestimmte Art der Weiterleitung verlangt. Der Ablauf von dem Start der Autorisierung in Schritt 1 bis zur Benutzerauthentifizierung in Schritt 5 entspricht exakt denen des Authorization Code Flow (Abbildung 2.5). Nach der erfolgreichen Anmeldung wird der Auth-Code generiert und der Browser mit diesem Parameter wieder zurück an den Client Endpunkt weitergeleitet. Dies erfolgte in den bisher vorgestellten Varianten mit dem 303 Status Code, doch diesmal wird der 307 Code verwendet. Wie auch in den Schritten 6 und Schritt 7 zu sehen sind, wird der Nutzername und das Passwort in der Anfrage mitgeschickt. Dies geschieht, da in der vorherigen Anfrage eben diese Werte an den Authorization Server geschickt wurden und nach dem HTTP 307 Status Code die vorherige Anfrage in der jetzigen enthalten ist mitsamt deren Parametern. Durch die Art der Weiterleitung wird auch vom Browser in Schritt 7 eine POST Anfrage an die weitergeleitete Adresse durchgeführt statt einer GET Anfrage. Aus dieser Anfrage erfährt nun der Client die Authentifizierungswerte des Nutzers. Da wir annehmen, dass der Client vom Angreifer kontrolliert wird, können diese mitgesendeten Informationen für die Zwecke des Angreifers missbraucht werden [7].

3.2.3 Fix

Die einfachste Möglichkeit, dieses Problem zu umgehen, ist es, wenn das 307 Protokoll beim Weiterleiten nicht verwendet wird. Falls dies nicht möglich ist, sollte zumindest die Benutzer Authentifizierung nicht unmittelbar davor stattfinden, sodass die vorherige Anfrage keine sensiblen Daten enthält [7].

3.3 PKCE Downgrade Attack

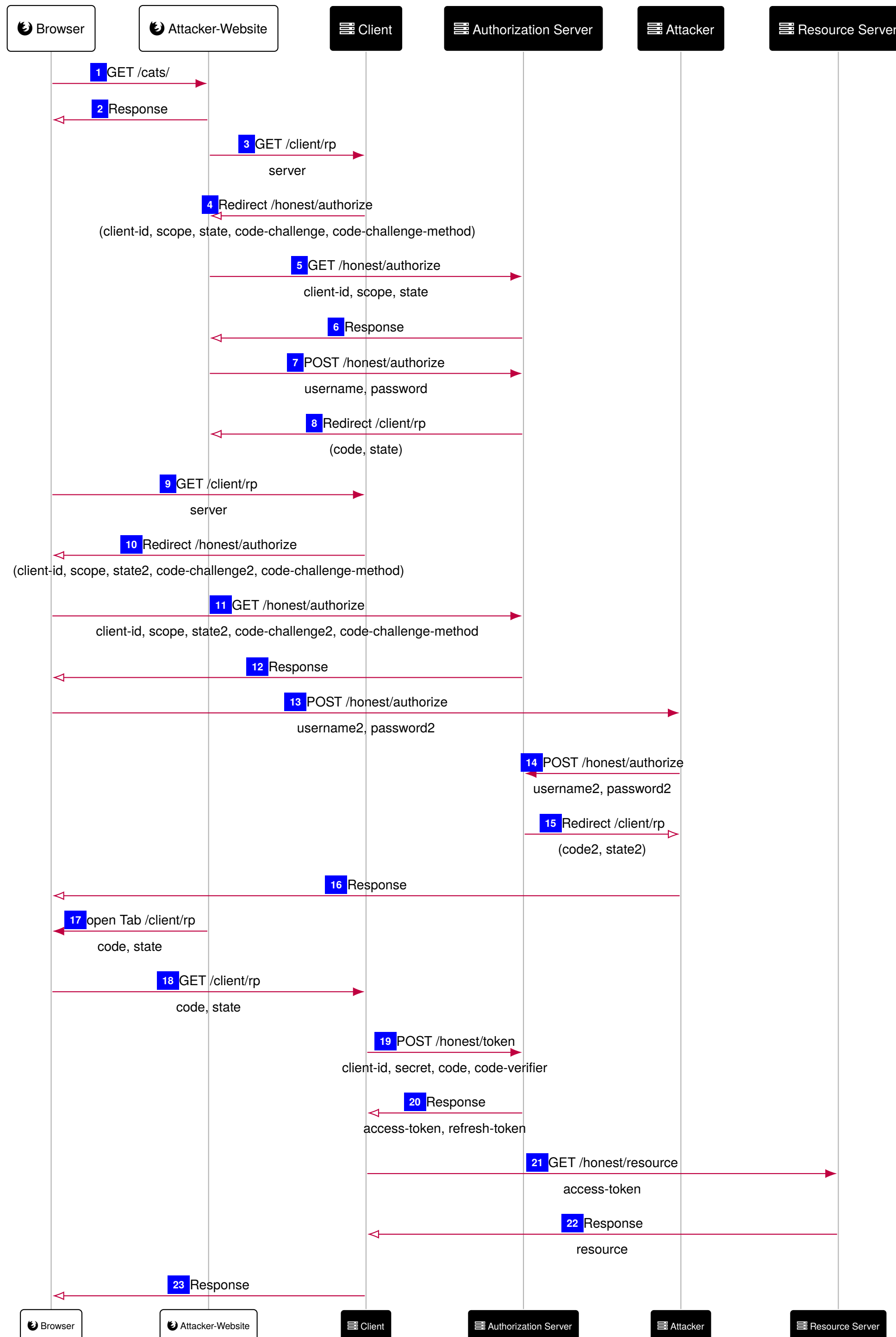

Abbildung 3.3: PKCE Downgrade Attack

Die Besonderheit des *PKCE Downgrade Attack* [5] wird schon aus dem Namen deutlich. Der Angriff zielt auf die PKCE Erweiterung, die die Sicherheit stark erhöhen sollte. Dennoch müssen Einschränkungen beachtet werden, sodass dieser Angriff nicht direkt PKCE aushebelt, sondern auf den Authorization Server abzielt (siehe Abschnitt 3.3.1). Da dieser Angriff zu den CSRF Angriffen gehört, ist das Ziel des Angreifers, über den Browser Anfragen versteckt an den Nutzer zu senden, ohne diesem über den Client Zugriff auf die eigenen Ressourcen, sondern denen des Angreifers zu geben. Dies können dabei potenziell gefährliche Daten sein oder der Nutzer könnte damit dem Angreifer Daten übermitteln. Um dies zu bewerkstelligen, wird häufig eine Webseite verwendet, die der Angreifer kontrolliert.

3.3.1 Voraussetzungen

Für diesen Angriff ist es essenziell, dass der verwendete Authorization Server PKCE nur optional anbietet, sodass die erweiterte Überprüfung des Codes nur stattfindet, wenn bei der Weiterleitung zu dem Authorization Server eine Code Challenge mitgeschickt wurde. Das heißt, der Auth-Code Wert muss nicht mit einer Challenge verknüpft werden. Damit der Angreifer Anfragen durchführen kann und im Browser des Nutzers Seiten öffnen kann, eignet sich eine Website, die der Angreifer kontrolliert. Es ist wichtig, dass der Nutzer diese Seite aufruft. Dies kann aus Versehen sein, könnte aber auch eine bewusste Entscheidung sein. Die Webseite müsste auch exakt den gleichen Authorization Server und Client für ihre Anfragen verwenden, damit dieser Angriff funktioniert. Dieses Szenario könnte aber bei vielen Aufrufen schnell eintreffen. Daneben wäre es von Vorteil, wenn der Angreifer Nachrichten zwischen dem Authorization Server und dem Browser des Nutzers als Netzwerkangreifer abfangen kann, sodass diese nicht beim Empfänger ankommen [5].

3.3.2 Ablauf

Die Prozedur für diesen Angriff ist in Abbildung 3.3 veranschaulicht. Diese unterscheidet sich zu den vorher vorgestellten Angriffen erheblich, da für diesen Angriff mehrere Flows ablaufen. Der Angriff startet in diesem Beispiel mit dem Aufruf der Webseite des Angreifers (siehe Schritt 1). Der erste Flow wird hier von der Webseite aus gestartet. Diese führt daraufhin die bekannten Schritte aus dem Authorization Code Flow in Abbildung 2.5 durch. Dabei werden aber Änderungen vorgenommen: Der Schritt 4 soll den Angreifer zum Authorization Endpunkt des Authorization Servers weiterleiten. Da PKCE verwendet wird, fügt der Client in dieser Anfrage eine Code Challenge und eine Code Challenge Methode bei. Damit der Auth-Code nicht mit diesen Werten verknüpft wird, entfernt der Angreifer zunächst diese aus der Anfrage und führt anschließend die Weiterleitung in Schritt 5 ohne diese Werte aus. Bei den weiteren Schritten (5 bis 8) wird der Flow weiter fortgesetzt. Zu beachten ist, dass sich der Angreifer bei Schritt 7 mit seinem eigenen Account anmeldet und somit die persönlichen Anmeldedaten verwendet. Die Adresse der Weiterleitung aus Schritt 8 wird von der Webseite gespeichert und nicht direkt durchgeführt, sodass eine spätere Weiterleitung zu dieser möglich ist [5].

Der Nutzer führt daraufhin selber den Authorization Code Flow durch. Nachdem der Nutzer diesen bei demselben Client mit dem gelichen ausgewählten Authorization Server in Schritt 9 gestartet hat, wird wieder eine Weiterleitung zu dem Authorization Server veranlasst. Die enthaltenen PKCE Parameter in der Form als Code Challenge und Code Challenge Methode bleiben diesmal unangetastet. Auch die restlichen Schritte werden nicht manipuliert. Nur in Schritt 15 wird

eine weitere Weiterleitung verhindert. Dazu kann diese entweder direkt gelöscht werden, sodass der Zieladressat davon nichts erfährt, oder die Weiterleitung muss zu einer normalen Antwort umgewandelt werden (siehe Schritt 16), wodurch der Browser auch keine Weiterleitung einleitet [5].

Die Webseite, die nur ein bestimmtes Zeitfenster abwartet, verwendet danach die gespeicherte Adresse der Weiterleitung, indem diese damit die Seite im Browser öffnet z. B. in einem neuen Tab (siehe Schritt 17). In den folgenden Schritten 18 bis 23 werden nun die bislang fehlenden Schritte vom Client durchgeführt. Bei dieser Prozedur greift der Angreifer nicht ein und entsprechen somit denen des normalen Authorization Code Flows. Da der Nutzer eine Autorisierung bei diesem Client veranlasst hat und sonst keine Antwort erhalten hat, wird dieser Aufruf der Webseite vom Nutzer als seiner interpretiert und akzeptiert. Dennoch stammt dieser Auth-Code Wert vom Angreifer, wodurch der Nutzer unwissend auf die Ressourcen des Angreifers zugreift [5].

3.3.3 Fix

Eine Möglichkeit, dem Problem entgegenzutreten, ist es, dass der Authorization Server eine optionale Verwendung des PKCE Standards nicht mehr unterstützt, sondern PKCE nur noch verpflichtend anbietet. Daneben könnte das Problem behoben werden, wenn der Authorization Server beim Token Endpunkt prüft, ob ein Code Verifier mitgeschickt wurde. Wenn der Auth-Code Wert selbst ohne eine Code Challenge erstellt wurde, soll bei vorhandenen Code Verifier der Flow abgebrochen werden.

3.4 Cuckoo’s Token Attack

Der *Cuckoo’s Token Attack* [6] stellt ein weiteres Problem bei vermeintlich sicheren Erweiterungen dar. Diesmal soll mTLS betrachtet werden, welches das Access Token an den Client binden soll, z. B. verwirklicht durch das Mitschicken eines Zertifikates an den Token- und Ressource Endpunkt. Dadurch soll gewährleistet sein, dass der Angreifer keine Informationen gewinnen kann, selbst wenn dieser das Access Token bekommen konnte, z. B. durch Phishing. Es ist richtig, dass nur der Client das Access Token verwenden kann, aber es wird nicht verhindert, dass der Client dem Angreifer beim Einlösen zur Hand gehen kann.

3.4.1 Voraussetzungen

Für diesen Angriff wird angenommen, dass ein vom Angreifer kontrollierter Authorization Server bei dem Client bekannt ist. Beim Client soll dieser Authorization Server mit demselben Resource Server verknüpft sein wie beim ehrlichen Server. Diese Bedingung ist möglich, da die Resource Server von den eigentlichen Authorization Servern getrennt sein können und jeder Authorization Server sich den vom Client verwendeten Resource Server aussuchen kann. Weitere Voraussetzungen sind nicht nötig [6].

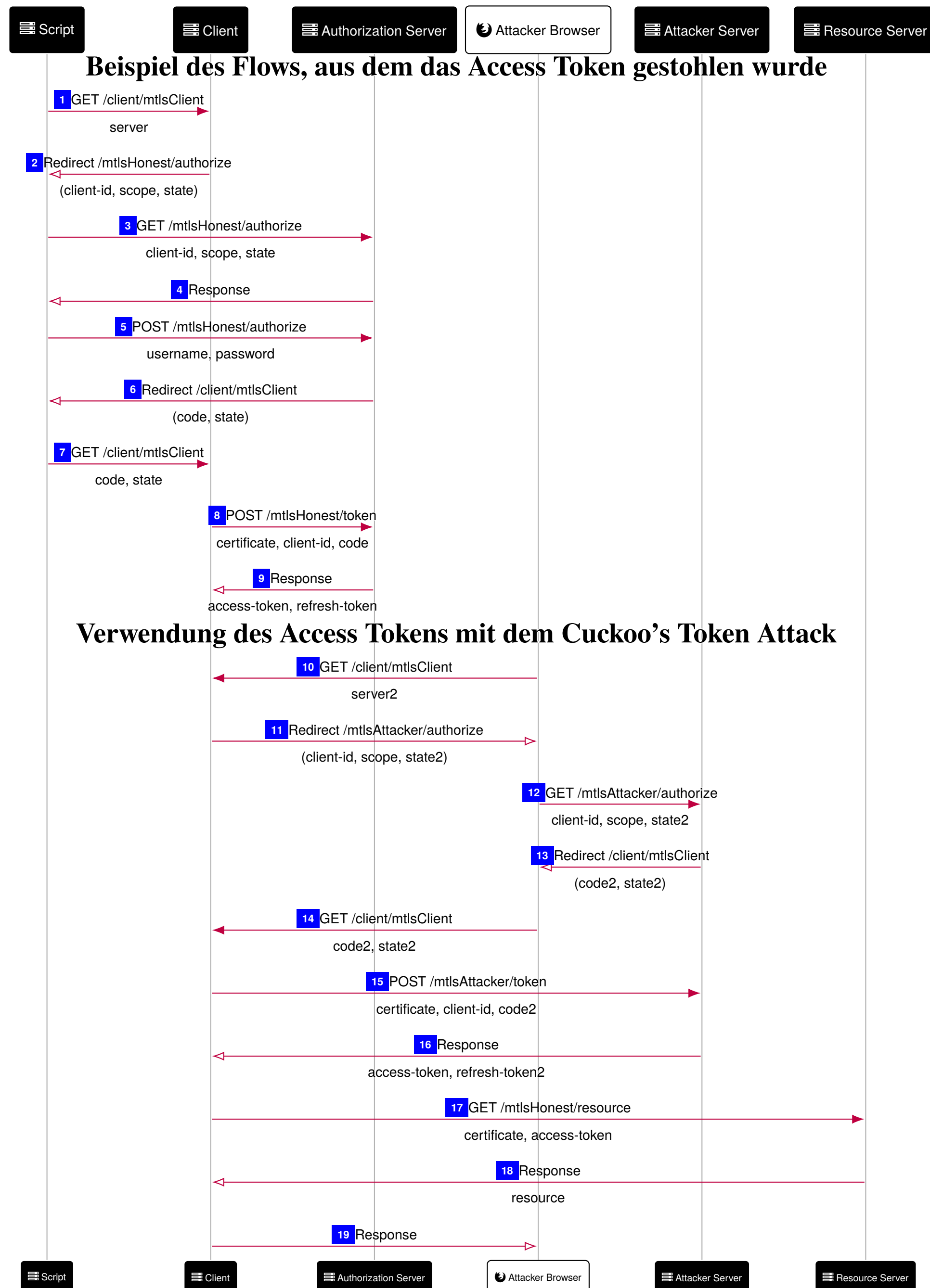

Abbildung 3.4: Cuckoo’s Token Attack

3.4.2 Ablauf

Im Folgenden soll der in Abbildung 3.4 beschriebene Ablauf erläutert werden. Dieser gliedert sich in zwei getrennte Vorgänge, von denen nur der zweite den wirklichen Angriff darstellt. Bis zu Schritt 9 ist ein exemplarischer Flow abgebildet, aus dem das Access Token für den Cuckoo’s

Token Attack gestohlen wurde. Es stellt den normalen Authorization Code Flow dar, bis darauf, dass der Flow in Schritt 9, in welchem das Access Token übermittelt wird, abgebrochen wird. Dies soll den Diebstahl des Tokens repräsentieren, sodass das Token nicht eingelöst wird. Dieses Token wird von dem Angreifer gespeichert und da es mit mTLS erstellt wurde, ist es für den Angreifer direkt nicht nutzbar. Deshalb soll dieses Token nun im zweiten Teil bei demselben Client eingelöst werden, indem dieses dem Client untergejubelt wird [6].

Der Angreifer startet dazu in Schritt 10 einen neuen Flow mit demselben Client, aber wählt dabei den eigenen Angreifer Server aus. Die Weiterleitung zu diesem Server in Schritt 11 kann entweder direkt ignoriert werden, oder der Angreifer Server führt, ohne weitere Anfragen zu benötigen, die Weiterleitung zurück zu dem Client aus. Diese zweite Variante ist in den Schritten 12 und 13 zu sehen. Die Schritte zur Authentifizierung beim Authorization Server können übersprungen werden, da der Angreifer Server den Angreifer kennt und diese Prozedur für den Angriff unwichtig ist. Alle Parameter, die zur Weiterleitung notwendig sind, sind entweder aus der anfänglichen Anfrage bekannt oder können zufällig gebildet werden. Der benötigte Auth-Code Wert kann zufällig bestimmt werden, da dieser für den Angreifer keine Rolle spielt und nur obligatorisch mitgegeben werden muss. Nachdem die Weiterleitung zum Client in Schritt 14 erfolgt ist, schickt dieser eine Anfrage an den Token Endpunkt des Authorization Servers (siehe Schritt 15). Für die Antwort muss der Angreifer Server nicht den Auth-Code prüfen oder den Client verifizieren. Er kann direkt das gestohlene Access Token mitsamt anderen möglicherweise gefälschten Parametern in Schritt 16 zum Client schicken. Dieser löst das Access Token dann bei dem Resource Server des Angreifer Servers ein, welcher dem des ehrlichen Servers entspricht. Dieser Aufruf ist in der Abbildung in Schritt 17 zu sehen. Da der Client der gleiche ist und ein gültiges Zertifikat bei der Anfrage mitschickt, akzeptiert der Resource Server und gibt den Zugriff auf die Nutzerdaten in Schritt 18 frei [6].

3.4.3 Fix

Für diesen Angriff entspricht die Lösung im Prinzip der Idee vom Fix des IDP Mix Up Attack (siehe Abschnitt 3.1.3). Auch diesmal soll ein weiterer Parameter hinzugefügt werden, welcher den verwendeten Authorization Server identifizieren soll. Im Gegensatz zur vorherigen Anwendung wird dies nun vom Client in Schritt 17 hinzugefügt. Diesen Wert prüft der Resource Server, indem er den angegebenen Authorization Server mit dem zugehörigen Server der Ressource vergleicht. Wenn dieser sich unterscheidet, sollte der weitere Flow abgebrochen werden [6].

3.5 State Leak Attack

Der *State Leak Attack* [7] beruht wieder auf dem Standardablauf des Authorization Code Flows ohne weitere Erweiterungen (siehe Abschnitt 2.4.1). Bei den bisherigen Angriffen waren der Auth-Code, das Access Token oder die Anmeldedaten des Nutzers das zentrale Element, welches gestohlen wird. In diesem Angriff ist der Hauptpunkt, wie der Name schon andeutet, der State. Dieser hat im Gegensatz zu dem Auth-Code den Vorteil, dass dieser nach dem OAuth 2.0 Standard [12] mehrfach einsetzbar sein darf. Der Angriff kann unter anderem gestartet werden, indem ein Link auf der

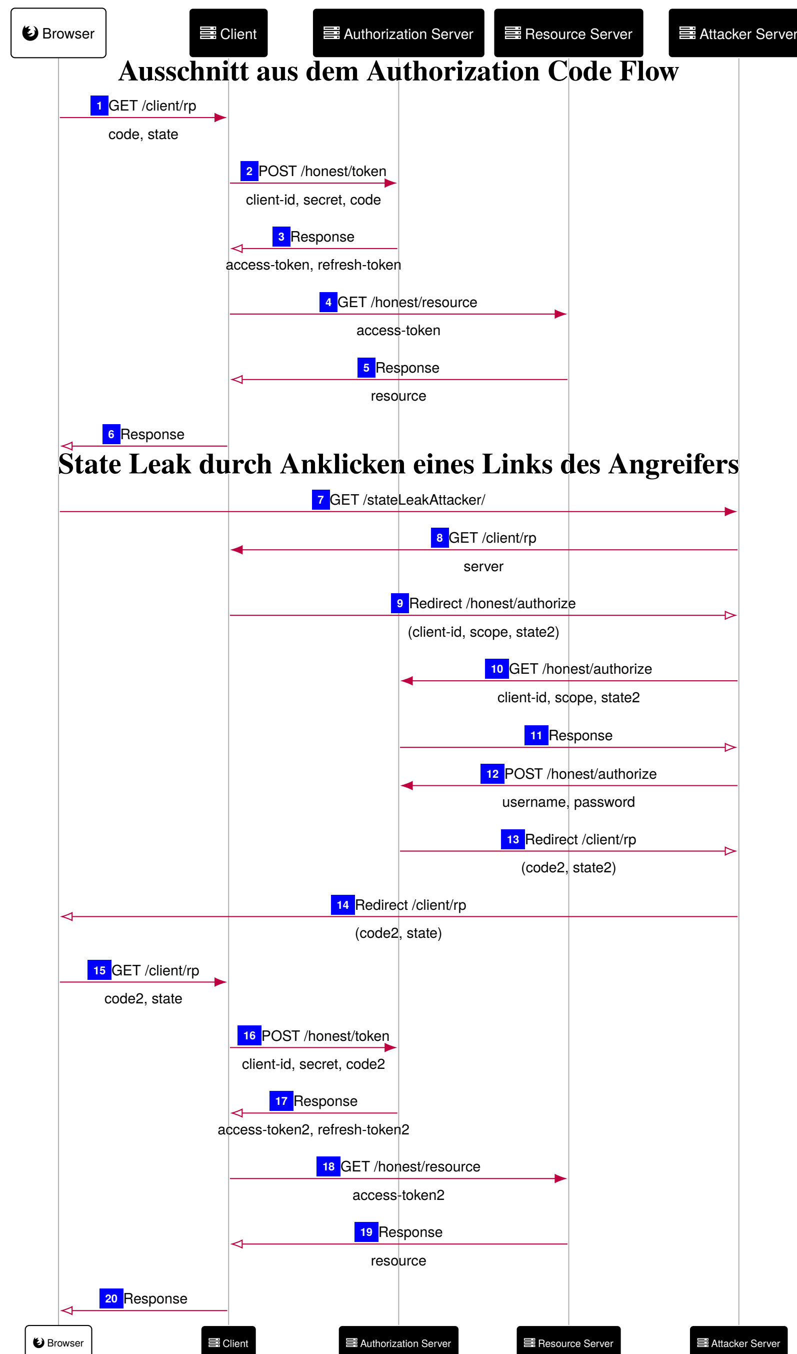

Abbildung 3.5: State Leak Attack

Webseite des Clients z. B. in der Werbung versteckt ist und dieser nach einem durchgeführten Autorisierungsvorgang betätigt wird. Ziel des Angriffes ist es, ähnlich zum PKCE Downgrade Attack (siehe Abschnitt 3.3), dem Nutzer Ressourcen des Angreifers unterzujubeln.

3.5.1 Voraussetzungen

Für diesen Angriff wird angenommen, dass der Angreifer bestimmte Ressourcen und Verlinkungen auf der Seite des Clients hat, welche unter anderem Werbebanner sein können. Diese sind mit einer Angreifer Seite verbunden, welche ein Skript ausführt. Daneben wird ein bestimmtes Client Verhalten benötigt. Dieses sollte den gleichen State in mehreren Anfragen erlauben. Dies ist in dem OAuth 2.0 Standard [12] auch nicht weiter eingegrenzt. Weiterhin soll eine erneute Anfrage an den Client mit dem gleichen State die alte Session überschreiben [7].

3.5.2 Ablauf

In den Schritten 1 bis 6 in Abbildung 3.5 sind die letzten Schritte des Authorization Code Flows dargestellt (siehe Abbildung 2.5). Dies soll den Beginn des Angriffes und den Kontext, weshalb dieser Angriff so gefährlich ist, aufzeigen. Wenn der Nutzer, nachdem dieser den Flow beim Client durchgeführt hat, die Werbung mit dem Link des Angreifers anklickt, wird eine Anfrage an diesen geschickt. Diese soll im Schritt 7 abgebildet sein. Meistens wird ein HTTP Referer mitgeschickt, welche die Query der vorigen Anfrage beinhaltet. Da dies der Schritt 1 ist und in dieser die Adresse der Weiterleitung an den Client mitsamt der Daten der Query enthalten ist, kann der Auth-Code und der State vom Angreifer extrahiert und verwendet werden. Da der Auth-Code sich nur einmal einsetzen lässt, kann für den Angriff nur der extrahierte State verwendet werden [7].

Der Angreifer führt zunächst in den Schritten 8 bis 13 den üblichen Flow des Authorization Code Grant bis zur Weiterleitung zurück zum Client aus. Der verwendete Client muss dabei dem gleichen entsprechen, der auch beim vorherigen Flow verwendet wurde. Die Authentifizierung in Schritt 12 erfolgt mit den Benutzerdaten des Angreifers. Die Weiterleitung aus Schritt 13 mit dem neu erstellten Auth-Code wird vom Angreifer modifiziert, sodass der neue State mit dem Gestohlenen getauscht wird. Diese modifizierte Weiterleitung wird nun in Schritt 14 an den Browser geschickt, sodass erneut mit demselben State die restlichen Schritte beim Client ablaufen. Dadurch, dass ein neuer Auth-Code verwendet wird, ist eine erneute Durchführung dieser Schritte erst möglich. Durch diese Durchführung und einen erneuten Ressourcen Zugriff wird die alte Session überschrieben. Damit ist in dieser der Angreifer angemeldet und der Nutzer greift bei Aktionen mit dem Client auf die Daten des Angreifers zu [7].

3.5.3 Fix

Der Client sollte bei dem State die Sicherheitsvorkehrungen, die schon bei dem Auth-Code eingesetzt werden, übernehmen, sodass derselbe State nur einmal akzeptiert wird. Damit ist ein Abgreifen dieser Werte aus einer alten Anfrage nutzlos und der Angriff kann nicht funktionieren [7].

4 Realisierung

In diesem Kapitel wird ein Überblick über die Entwicklung des Projektes vom Ausgangspunkt, welcher in Abschnitt 4.1 beschrieben wird, bis zum jetzigen Stand in Abschnitt 4.3 gegeben. Dafür werden die wichtigsten Designentscheidungen für das Projekt in Abschnitt 4.2 erläutert.

4.1 Ausgangsprojekt

Das im Rahmen der Bachelorarbeit entwickelte Programm baut auf *proto6749* [9] auf. Viele grundlegende Technologien und Flows von OAuth 2.0 waren darin vollständig oder in Ansätzen implementiert. Zu diesen Flows zählten der „authorization code flow", „implicit flow" und der „client credentials flow". Auch mTLS, OAuth 2.0 Rich Authorization Request (RAR) und Pushed Authorization Requests (PAR) waren schon in den Grundzügen implementiert. Es gab Testfälle zu den Flows, mit denen diese stark vereinfacht durchgeführt werden konnten. Dabei diente nur die Konsole als Interaktionsmöglichkeit. Auch Browser Aktionen wurden über Links realisiert, die in der Konsole ausgegeben wurden oder als Eingabe dienten. Ein Testfall für den „authorization code grant" und eine Realisierung des Flows „ressource-owner-password-credential grant" fehlten. Der Prototyp wurde hauptsächlich in Python geschrieben und für die Realisierung des Authorization Servers wurde Django [23] verwendet. Unter Zuhilfenahme des Admin Panels von Django war es möglich, Anpassungen an den OAuth 2.0 Partys durchzuführen. Spezifische Sicherheitsvorkehrungen, u. a. *PKCE* oder *Access Token-Binding*, waren beim Authorization Server einstellbar. Beim Client waren unterschiedliche Authentifizierungsmethoden, darunter BasicAuth oder Public Key Infrastructure (PKI) Zertifikate, vorhanden.

4.2 Designentscheidungen

Während der Entwicklung des Prototypen mussten viele Designentscheidungen und Kompromisse getroffen werden. Die Wichtigsten und Zentralsten werden im Folgenden genauer erklärt.

4.2.1 Nutzermanagment

Bei der Realisierung des Resource Owner Password Credentials Flows (siehe Abschnitt 2.4.1) musste über das Nutzermanagement entschieden werden. Für diesen Flow war es nötig, dass der Client die Nutzerdaten, genauer die Kombination aus Benutzername und Passwort, erhält und dem Authorization Server zur Prüfung übermittelt. Da fest eincodierte Benutzerdaten keine Flexibilität boten, musste eine persistente Art der Nutzerverwaltung gefunden werden. Da eine selbstständige Implementierung aufwendig wäre und Django selber ein Nutzermanagement bietet, welches den

Zugriff auf das Admin-Panel regelt, fiel die Entscheidung auf Django. Testweise wurde das „Django OAuth Toolkit“ [10] eingebunden, da dieses denselben Flow auch implementiert hatte mit der Verwendung von Django Nutzer. Da dieses aber sonst keine nennenswerten Vorteile bot und das Problem mit der Authentifizierung gelöst werden konnte, wurde diese Idee verworfen. Damit das Nutzermanagement etwas realistischer wird, wurden auch an der Datenbank Änderungen vorgenommen, sodass nun jedem Authorization Server spezifische (Django-)Nutzer zugewiesen werden können, die beim Server registriert sind. Die Nutzernamen und Passwörter sind dabei aus Gründen der Einfachheit pro Nutzer identisch, unabhängig vom verwendeten Authorization Server. Wenn konsequent für jeden Authorization Server eigene Benutzer erstellt und zugewiesen werden und keine Mehrfachbelegung der Nutzer stattfindet, lassen sich trotzdem reale Bedingungen herstellen. Auch bei dem Anmeldevorgang wurden Änderungen durchgeführt, unter anderem wurde beim Authorization Endpunkt eine Login-Oberfläche erstellt, mit dem man sich beim Authorization Server anmelden kann. Ohne eine solche Oberfläche musste man sich sonst direkt beim Admin-Panel anmelden, um den Authorization Endpunkt nutzen zu können.

4.2.2 Client Endpunkt

Bei der Entwicklung des *Authorization Code* Testfalls wurden schnell die Nachteile der Repräsentation des Clients als Python Skript deutlich (siehe Abschnitt 4.1). Auch wenn eine Interaktion mit dem Browser über Umwege funktioniert hatte, stieß die Repräsentation schnell an seine Grenzen. Die Weiterleitung zum Authorization Endpunkt des Authorization Servers war noch ohne größere Umwege möglich, indem das Skript ein Fenster im Browser öffnete mit der entsprechenden Seite. Aber bei der Weiterleitung vom Authorization Server zurück zum Client über die Redirection URI kam man auf ein Problem, da der Client diese nicht empfangen konnte. Zuerst war es eine Lösung, im Skript einen kleinen Server laufen zu lassen, welcher auf diese Nachricht wartete. Da dies aber keine dauerhafte Lösung darstellte, sollte diese zur Vereinfachung weiter geändert werden. Eine Integration in Django stellte sich schnell als beste Lösung dar, da Django schon allein zur Funktionalität der Authorization Server benötigt wird und eine Erweiterung einfach ist. Für den Client musste dafür ein eigener Endpunkt erstellt werden, der unter „http://127.0.0.1:8000/client/clientID“ erreichbar ist. Um den *Authorization Code Flow* zu starten, muss dafür nur ein HTTP-Request an den Client Endpunkt gesendet werden mit gegebenenfalls weiteren Parametern. Die Redirection URI entspricht ebenfalls der URL des Client Endpunktes.

4.2.3 Proxy

Bei den meisten Angriffen ist es unabdingbar, dass Nachrichten abgefangen und gegebenenfalls modifiziert werden können. Um dies zu bewerkstelligen und einen Netzwerkangreifer zu simulieren, sollte ein Proxy eingesetzt werden. Dieser ist momentan über Django realisiert, doch es gab davor mehrere Alternativen. Am Anfang wurde *Burp Suite* [2] dazu verwendet. Dieser Proxy ist mächtig und Nachrichten lassen sich damit leicht manipulieren. Schnell zeigten sich aber die Nachteile der kostenlosen Edition. Besonders die fehlende Speicherfunktion, welche für dieses Projekt essenziell ist, führte zu einer erneuten Suche nach Alternativen.

Da *OWASP ZAP* [24] Nachrichten auch manipulieren lässt in einem (Python-)Skript und durch eine Open Source Lizenz auch leichter einsetzbar ist, fiel die Wahl darauf. Die nötigen Nachrichten Manipulationen ließen sich schnell damit umsetzen und der Proxy leicht in den Browser integrieren.

Der Nachteil generell der Proxys im Vergleich zu Netzwerkangreifern ist es natürlich, dass diese als Empfänger explizit ausgewählt werden müssen im Browser oder bei Anfragen in Python. Deshalb stellt diese Wahl auch nur eine Vereinfachung dieser Vorgänge dar. Im späteren Verlauf des Projektes sollten die Nachrichten protokolliert werden, damit es möglich war, den Nachrichtenverlauf grafisch umzusetzen. Der *OWASP ZAP* Proxy stellte sich dabei als Nachteil heraus, da dieser nicht mit Django kommunizieren konnte. Dadurch können zwar die Nachrichten, die aus den Django-Endpunkten entspringen oder die dort empfangen werden, protokolliert werden, aber bei Datenmanipulationen beim Proxy können entweder nur die Anfangsdaten oder die modifizierten Daten empfangen werden. Diese Änderung der Daten ist aber entscheidend, um den Angriffsverlauf nachzuverfolgen. Um dieses Problem zu beheben, wäre es das sinnvollste, den Proxy mit Django direkt umzusetzen und die Nachrichten beim Durchleiten ohne Umwege zu loggen. Anfangs war die einzige Idee, einen Endpunkt zu definieren mit einem bestimmten Pattern in der URL (in diesem Fall „proxy/“). Das hatte aber den Nachteil, dass die Nachrichten vom Browser nicht an diesen Proxy Endpunkt gehen. Dadurch war es nötig, zusätzlich *OWASP ZAP* zu verwenden, welches nur den Präfix „proxy/“ zu der URL hinzufügen musste, sodass die Nachricht an den Proxy Endpunkt gelangt. Bei dem Django Proxy war es unter Zuhilfenahme der *Django Revproxy* [18] Bibliothek einfach möglich Nachrichten und ihre Antwort zu manipulieren. Dabei wurde auch das Präfix „proxy/“ entfernt, sodass die Nachricht danach zum richtigen Endpunkt weitergeleitet wurden.

Da die Verwendung von mehreren Proxys kompliziert und unnötig war, sollte eine Alternative gefunden werden. Am Schluss wurde eine Möglichkeit gefunden, nur den Django Proxy zu verwenden, welcher ohne eine spezielle URL erreichbar ist. Dazu empfängt der Proxy Endpunkt zunächst alle Nachrichten an den localhost, auch wenn diese an genau spezifizierte Endpunkte gehen würden. Damit der Proxy die Nachrichten beim Weiterleiten nicht an sich selber weiterleitet in einer unendlichen Rekursion, ist es nötig, dass an den URL Pfad ein Suffix angehängt wird, welches in dem entwickelten Tool ein „!“ darstellt. Dafür müssen zunächst bei Verwendung des Proxys die Endpunkte des URL Listeners um dieses Suffix ergänzt werden und bei einer Deaktivierung des Proxys diese wieder entfernt werden. Es gibt auch die Möglichkeit, ohne ein Suffix auszukommen, indem der URL Listener des Proxys abwechselnd hinzugefügt und entfernt wird, doch dies führte zu Problemen bei parallelen Aufgaben und Flows. Durch die Integration in Django funktioniert das Logging sehr gut und nur ein Proxy wird benötigt, der zudem in das Projekt direkt eingegliedert ist und nicht separat integriert werden muss. Eine Auswahl des Proxys im Browser ist dadurch auch nicht nötig und möglich.

4.2.4 Logging & YAML Erstellung

Durch die Integration des Proxys in Django stand einem übergreifendem Logging der Nachrichten nichts mehr im Wege. Da aber nur die Anfrage und nicht die Antwort berücksichtigt und neben den HTTP Methoden und den Zieladressen keine Informationen aus den Headern und dem Body mitberücksichtigt wurden, reichte das Standard Logging Verhalten von Django nicht aus. Deshalb wurde die Middleware *django-request-logging 0.7.2* [17] integriert, sodass neben den Anfragen auch stets die Antworten geloggt wurden mitsamt den Header- und gegebenenfalls den Body Informationen. Dennoch fehlten die Details zu den Weiterleitungen, die bislang nur als Antwort gewertet wurden. Durch einen manuellen Logging Aufruf konnte dieses Problem aber gelöst werden. Daneben wurde damit auch der Proxy bei der Protokollierung kenntlich gemacht, welcher ansonsten nicht von dem gewollt adressierten Endpunkt zu unterscheiden wäre. Für die Erstellung der YAML

Datei, welche den Nachrichtenfluss darstellen soll, gab es verschiedene mögliche Ansätze. Zum einen könnte die Erstellung zeitgleich mit dem Logging erfolgen. Zum anderen ist es einfacher, die Informationen erst zu sammeln und am Schluss als Ganzes in eine YAML umzuwandeln. Bei beiden war es aber ein offenes Problem zu wissen, wann das Logging starten und wann es beendet werden sollte. Am Schluss wurde eine Variante implementiert, die sich beiden Konzepten bedient. Es werden die Informationen direkt verarbeitet und gefiltert, aber erst am Schluss in eine YAMl Datei umgewandelt. Anfangs war die Initialisierung, Erstellung der YAML und das Resetten der Protokollierung der Nachrichten ein und derselbe Aufruf über eine GET Anfrage an den Logger Endpunkt. Im späteren Verlauf wurde dies entkoppelt. Nun wird über eine DELETE Request an den Logger Endpunkt der Logger initialisiert oder resettet, sodass alle gespeicherten Nachrichten gelöscht werden. Die Speicherung erfolgt über einen POST Request. Bei der Speicherung der Nachrichten werden bislang die HTTP Methode, die URL, wichtige Parameterwerte und die Partys, welche den Start und das Ziel der Anfrage sind, berücksichtigt. Bei Weiterleitungen wird auch die URL der Zieladresse verwendet. Weitere Informationen zu der Erstellung der YAML Dateien sind im Abschnitt 4.3.2 zu finden.

4.2.5 Angreifer Repäsentation

Der Aufbau des Angreifersystems kann über unterschiedliche Ansätze realisiert werden. Zum einen kann jeder Angreifer Server oder Client außerhalb von Django separat implementiert und eingerichtet werden. Dies würde aber zu einem aufwendigen und komplizierten Aufbau mit weiteren Folgeproblemen führen. Die andere Möglichkeit ist es, vorhandene Authorization Server und Client Architekturen weiterzuverwenden und diese nur als Angreifer zu kennzeichnen, um Verhaltensänderungen mithilfe einer If-Else Klausel darzustellen. Diese stellt eine einfachere Möglichkeit dar und es erfordert keine neuen Strukturen bei den Endpunkten. Die Art der Kennzeichnung kann auch auf unterschiedlichen Arten erfolgen. Zum einen könnte über das Admin Panel ein Client als Angreifer ausgewählt werden. Die andere Möglichkeit ist es, diesen über den Namen zu kennzeichnen. Dies ist deutlich eingeschränkter als die vorherige Alternative, da nur feste Authorization Server bzw. Clients als Angreifer fungieren. Dafür ist es aber sehr verständlich, dass der Authorization Server „attacker“ der Angreifer ist und der Server „honest“ nicht. Da das Ziel des Projektes auf eine gute Verständlichkeit der Angriffe abzielt und die Einschränkungen nicht ins Gewicht fallen, wurde die Namenskennzeichnung der Angreifer gewählt.

4.2.6 Ressource Endpunkt

Im Ausgangsprojekt (siehe Abschnitt 4.1) war der Ressource Endpunkt fest eincodiert, sodass dieser immer den eigenen Authorization Server verwendete. Da dies aber nicht der Realität entsprach, wurde dies in dem Projekt geändert. Durch die Änderung kann dieser Endpunkt pro Authorization Server über das Admin Panel definiert werden. Der Client greift auf diese Information zu, um das Token beim richtigen Endpunkt einzulösen. Auch dieses Vorgehen entspricht nur einer vereinfachten Version der Realität. Zwar wird damit korrekt wiedergegeben, dass ein Authorization Server einen externen Ressource Endpunkt haben kann oder dass sich mehrere Server einen Endpunkt teilen können. Dennoch kennt der Client zu jedem Authorization Server einen oder sogar mehrere Ressourcen Endpunkte. Wie dies in der Realität verwirklicht wird, ist in OAuth 2.0 nicht näher erläutert und ist out of scope. Doch mit diesem Ansatz ist dies nicht möglich, da ein Authorization

Server genau einen festgelegten Resource Server hat und dieser nicht für jeden Client verschieden sein kann. Auch wird die Adresse beim Authorization Server und nicht beim Client gespeichert. Da diese Implementierung aber deutlich einfacher ist und dem Zweck gerecht wird, wurde es bei dieser vereinfachten Implementierung belassen.

4.2.7 Weboberfläche

Zum Ende des Projektes wurde der Logging Endpunkt um eine Weboberfläche erweitert. Diese sollte den Nutzen haben, dass bevor die YAML-Datei erstellt wird, noch Kommentare hinzugefügt werden können, um bestimmte Vorgänge zu separieren und näher zu beschreiben. Es gäbe auch die Alternative, dass bestimmte Vorgänge implizit kommentiert werden. Dies müsste aber alles im Code deklariert werden und ist eine sehr eingeschränkte Variante. Dagegen bietet diese Oberfläche eine deutlich flexiblere Interaktionsmöglichkeit mit dem Nutzer. Einen weiteren positiven Nebeneffekt hat diese Oberfläche, da der Nachrichtenfluss direkt ersichtlich ist und damit überprüft werden kann, ob das Logging funktioniert hat und ein Nutzer einen Überblick darüber bekommt, ohne die YAML Datei erst auszuwerten. Eine Ausgabe der YAML Datei direkt bei der Oberfläche wurde auch noch ergänzt, womit diese Dateien einfacher heruntergeladen und an einen gewünschten Ort gespeichert werden können. Da bislang das Starten der Flows und der Angriffe nur über die Konsole oder in einer Entwicklungsumgebung möglich war und bestimmte Parameter im Code gesetzt werden mussten, lag eine intuitive Steueroberfläche nahe. Da die Logging Weboberfläche schon existierte, bot sich an, diese entsprechend zu erweitern, sodass die Parteien ausgewählt werden können und alle zentralen Parameter und Eigenschaften eingestellt werden können. Somit entstand eine Weboberfläche, die zusammen mit dem Admin Panel von Django alle Aufgaben vereint und das Arbeiten mit dem Tool erleichtert. Mehr Details zu dieser Oberfläche ist auch im Abschnitt 4.3.3 zu finden.

4.3 Stand der Technik

In diesen Abschnitten soll der aktuelle Stand des Projektes separiert in den Backend (siehe Abschnitt 4.3.1) und den Frontend Teil in Abschnitt 4.3.3 vorgestellt werden. Daneben wurde ein weiterer Abschnitt zu der Erzeugung der YAML Dateien in Abschnitt 4.3.2 geschaffen. Während im Backend besonders die verwendeten Technologien im Vordergrund stehen, wird beim Frontend Abschnitt auch die grafische Benutzeroberfläche behandelt.

4.3.1 Backend

Der Hauptteil des Projektes ist im Backend zu finden, welcher hauptsächlich in Python programmiert wurde. Für die Implementierung der OAuth 2.0 Funktionalitäten mit den nötigen Endpunkten war ein Webframework essenziell. Für diese Aufgabe wurde Django [23] gewählt und in das Projekt integriert. Sowohl die Authorization Server als auch die Clients nutzen Django für ihre Schnittstellen und Funktionalitäten. Jeder dieser Endpunkte besitzt dabei eine spezifische URL, die in einer separaten Klasse definiert sind. Die einzelnen Authorization Server oder Clients werden in einer PostgreSQL Datenbank gespeichert und können über das allgemeine Admin Panel von Django angesehen, erstellt und modifiziert werden.

Analog zum Ausgangsprojekt (siehe Abschnitt 4.1) werden die Authorization Server durch den Servernamen, repräsentiert durch die ID, unterschieden. Spezifische Sicherheitsvorkehrungen u. a. *PKCE* oder *Access Token-Binding* können direkt beim Authorization Server festgelegt werden. Die Server können anschließend über eine spezifische URL („http://127.0.0.1:8000/servername“ oder („https://localhost/servername“) erreicht werden. Die Clients, spezifiziert durch ihre Client ID, haben zugewiesene Authorization Server. Zu beachten ist, dass ein Client mehrere Authorization Server haben kann, welche aber in den meisten Fällen zu dem Client unterschiedliche Client ID verbinden. Das heißt, dass derselbe Client mehrmals im Admin Panel spezifiziert werden kann, aber mit unterschiedlicher ID. Des Weiteren unterstützt der Client verschiedene Authentifizierungsmethoden. Dabei wird unter anderem unterschieden, ob BasicAuth verwendet wird oder PKI Zertifikate. Die vom Authorization Server verwendete Redirection URI kann auch bei dem Client auf die gleiche Weise wie der Ressource Endpunkt (siehe Abschnitt 4.2.6) im Admin-Panel bestimmt werden. Bei den Flows wird automatisch eine Session erstellt, die die beteiligten Partys, Eigenschaften, Werte und Tokens speichert. Diese lassen sich im Admin Panel auch einsehen.

Die Funktionalitäten der einzelnen Endpunkte wurden in mehrere Klassen separiert, wodurch nur zusammengehörige Endpunkte, z. B. welche, die nur zum Authorization Server oder Client gehören, in einer Klasse sind. Dabei wurden auch Schnittstellen für die Webseite und den Proxy geschaffen. Alle in Kapitel 3 beschriebenen Angriffe wurden in diesem Projekt realisiert. Dazu sind Skripte zu den Angriffen bei den normalen Endpunkten von OAuth 2.0, getrennt von dem Standard Flow durch Abfragen der Client oder Server Namen (siehe Abschnitt 4.2.5), oder extra eingerichtete Endpunkte des Angreifers zu finden. Neben den Hauptfunktionalitäten gibt es auch Klassen, die für die Authentifizierung des Clients und andere Sicherheitsaspekte verantwortlich sind. Auch Hilfsklassen und Methoden sind im Projekt zu finden. Um bestimmte Funktionen abzudecken, wurden auch mehrere Bibliotheken eingebunden. Zu nennen ist unter anderem *Django Revproxy* [18], welches die Implementierung eines Proxys in Django unterstützt. Damit war es möglich, Nachrichten und deren Antworten zu manipulieren, ohne externe Proxys zu verwenden. Daneben wurde auch *PyYAML* [20] genutzt. Diese Bibliothek half bei der Erstellung der YAML Objekte und Dateien. Mit der *django-request-logging* [17] Bibliothek konnte auch das Logging erweitert werden, sodass auch Antworten und Header protokolliert wurden. Weiterhin ermöglichte die *PKCE* [19] Bibliothek eine einfache Erstellung des Code Verifiers und der Code Challenge (siehe dazu Abschnitt 2.4.2).

4.3.2 Erzeugung der YAML Dateien

In diesem Kapitel soll das Verfahren zur Erzeugung der YAML Dateien näher betrachtet werden, welche anschließend mithilfe von Annex Language [8] in Nachrichtenverlaufsschemas in LaTeX [13] umgewandelt werden können. Für die Erzeugung ist besonders das Logging der Nachrichten wichtig (siehe dazu Abschnitt 4.2.4). Die Vorarbeit für die YAML-Dateien wird während des Logging Prozesses betrieben. Es werden zunächst die irrelevanten Nachrichten herausgefiltert und die restlichen kategorisiert, sodass eine spezifische Behandlung möglich ist. Es wird dabei anhand bestimmter Merkmale der Logging Nachricht zwischen Anfragen, Antworten, Weiterleitungen und Header- und Body- Informationen unterschieden. Durch die Unterscheidung ist es auch möglich geworden, bestimmte Nachrichtenbestandteile aus der gesamten Logging Nachricht zu extrahieren. Diese Informationen werden dann in einem Array gebündelt gespeichert. Bei der Speicherung von Anfragen werden neben der HTTP-Methode, der Zieladresse und den Query Werten auch die

Rolle des Absenders und des Empfängers erfasst. Dies kann z. B. ein Nutzer, Authorization Server oder auch der Angreifer sein. Da die Erkennung des Absenders recht schwer ist, werden bei den Anfragen teilweise Informationen in den Header hinzugefügt, welche eine eindeutige Zuordnung des Absenders ermöglichen. Damit man zu der Anfrage auch die zusammengehörige Antwort oder Weiterleitung finden kann, wird die Thread Nummer genutzt, welche mit erfasst und gespeichert wird. Für die Antwort wird zunächst nur die Thread Nummer gespeichert. Dagegen muss bei der Weiterleitung auch die gewünschte URL mitsamt deren Query Werten gespeichert werden. Da bei einer Weiterleitung auch eine Antwort protokolliert wird, muss diese nicht betrachtet werden.

Wenn Header oder Body Informationen eintreffen, werden diese der zuletzt gespeicherten Nachricht hinzugefügt. Teilweise werden dabei Werte durch einen verständlicheren Begriff ersetzt z. B. bei dem Authentifizierungsheader oder dem Zertifikat. Bei der Speicherung der Query, Header und Body Werte wird besonders vorgegangen. Diese Daten werden vorher bearbeitet, indem zuerst die ungewünschten Werte herausgefiltert und dann in die gewünschte Form gebracht werden. Daraus entsteht anschließend ein Tuple aus zwei Strings, welche die einzelnen Informationen mit Komma getrennt beinhalten. Der Grund der mehrfachen Strings ist die Abwägung zwischen Ausführlichkeit und Lesbarkeit. Damit der Benutzer des Programms das passende Schema je nach Anwendungsfall nutzen kann, werden direkt zwei Versionen erstellt. Diese unterscheiden sich nur in der Darstellung der Parameter. Während der eine die Key Value Paare beinhaltet, soll der andere nur die Keys verwenden. Damit eine Unterscheidung zwischen verschiedenen Werten im zweiten Fall trotzdem möglich ist, werden diese nummeriert, falls Keys mit verschieden Werten belegt werden. Durch die verschiedenen Versionen in Bezug auf die Parameterliste werden schlussendlich auch mehrere YAML Dateien erstellt.

Wenn eine Erstellung der YAML Dateien vom Nutzer gewünscht wurde, werden die gesammelten Informationen verarbeitet. Dafür wird durch die gespeicherten Nachrichten iteriert und mithilfe der verfügbaren Informationen passende YAML Objekte erstellt. Durch die Kennzeichnung der Nachrichten mit der Thread Nummer können auch die Antworten und Weiterleitungen in den Objekten eine Referenz auf die passende Anfrage erhalten. Dies ist für die Erstellung der YAML Dateien essentiell. Für die Rollen stehen entsprechende Parteien zur Verfügung. Kommentare, die über die Webseite vom Nutzer ergänzt wurden, werden ebenso berücksichtigt und in das passende YAML Objekt transferiert. Der ebenfalls in der Webseite vom Nutzer angegebene ID Wert wird ebenso genutzt und bei der Initialisierung der YAML Objekte beigefügt. Damit ist es möglich später in der LaTeX Version auf einzelne Schritte zu verweisen. Zur Erstellung der YAML Datei müssen neben der Auflistung der YAML Objekte auch die Erstellung und die Zerstörung der einzelnen Parteien aufgeführt werden.

4.3.3 Frontend

Die Webseite ist in 3 Teile unterteilt. In dem ersten Reiter, welcher „Menu“ heißt und in Abbildung 4.1 zu sehen ist, können die einzelnen Flows und Angriffe gestartet werden. Dabei können auch weitere Einstellungen getroffen werden. Bei den Authorization Servern und Clients wird beim Laden der Seite der aktuelle Stand geholt und als Drop Down Liste zur Verfügung gestellt. In Abbildung 4.2 ist der zweite Reiter mit dem Titel „Message Flow“ zu sehen, welcher eine Übersicht über den Nachrichtenverlauf, der protokolliert wurde, gibt. Dies ist schon eine gefilterte Ansicht und stimmt mit den Nachrichten überein, die anschließend als YAML Datei gespeichert

Abbildung 4.1: Menü der Webseite

werden. Durch einen Tooltip können die wichtigen Parameter mitsamt deren Werten pro Nachricht eingesehen werden. Damit die Liste den aktuellen Stand zeigt, muss diese aktualisiert werden. Dazu gibt es einen Button, aber auch durch Wechseln des Reiters auf das Logging wird diese Liste aktualisiert. Daneben können in diesem Fenster Kommentare hinzugefügt werden, welche sich zwischen den Nachrichten einfügen lassen. Diese können auch gelöscht und mit anderen Kommentaren getauscht werden. Anschließend kann man dies über einen Button in eine YAML Datei umwandeln lassen. Diese Dateien können dann im letzten Reiter „Download" heruntergeladen werden (siehe Abbildung 4.3). Es wird immer nur der letzte Stand der Dateien zum Download angeboten. Das letzte Bearbeitungsdatum wird dabei auch angezeigt.

Neben dieser Webseite wurden auch eine Login Oberfläche und die Webseite des Angreifers, welche ein Skript enthält, um den PKCE Downgrade Attack (siehe Abschnitt 3.3) durchzuführen, erstellt. Zuletzt entstand auch eine eigene Client Webseite, auf die der Nutzer nur am Ende des Flows weitergeleitet wird. Über diese war es möglich, Fragmente aus der URL auszulesen und somit den Implicit Flow (siehe Abschnitt 2.4.1) auszuführen. Diese Webseite enthält auch einen Werbebanner, der den Einstieg für den State Leak Attack (siehe Abschnitt 3.5) liefert.

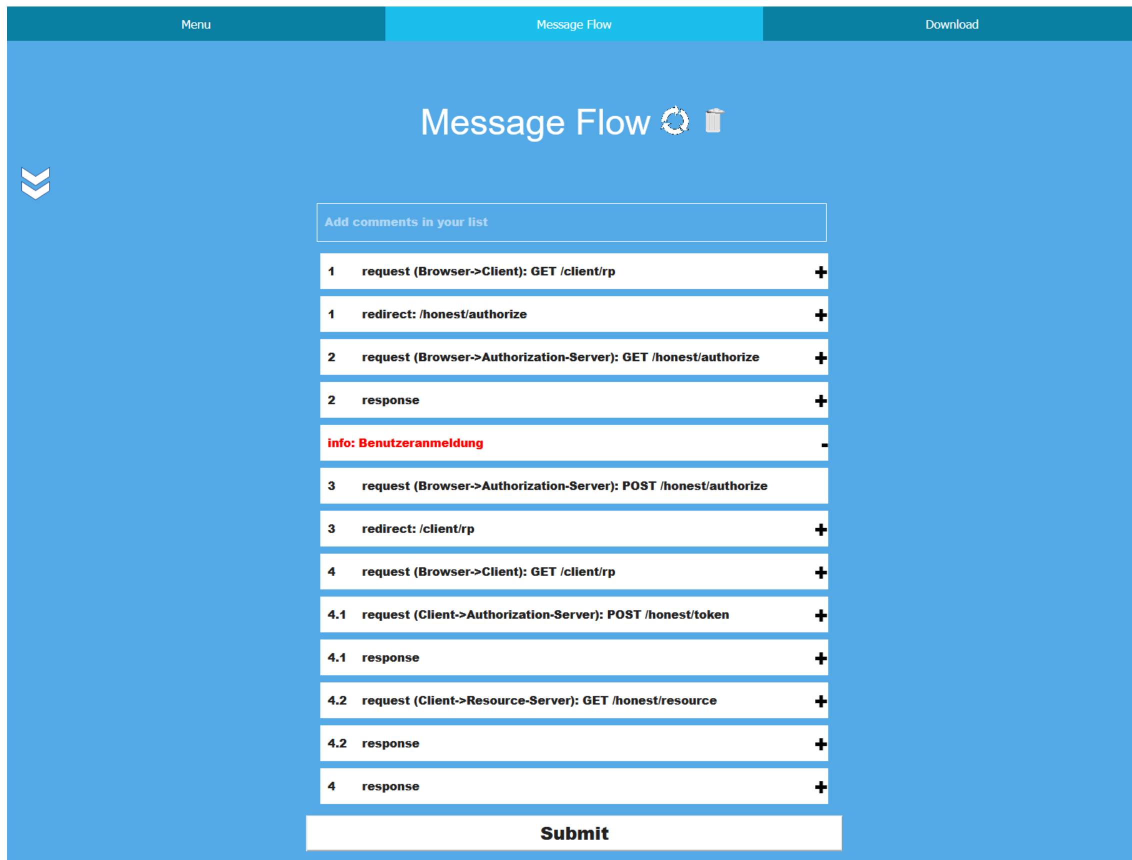

Abbildung 4.2: Nachrichtenverlauf in der Webseite

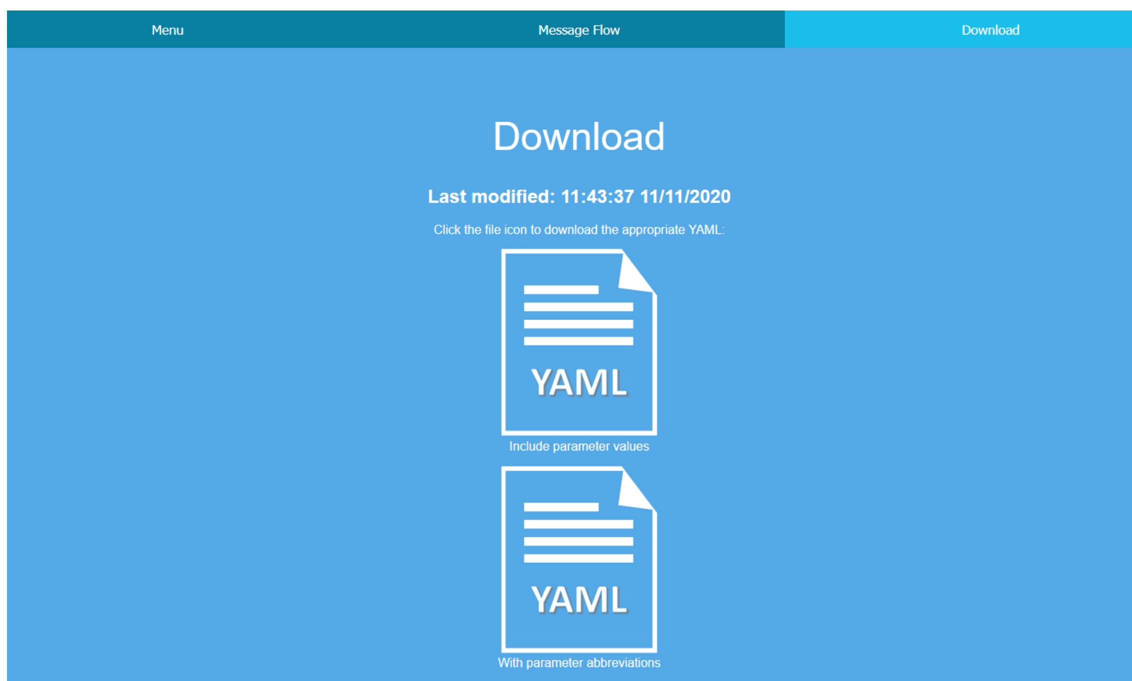

Abbildung 4.3: Download Bereich auf der Webseite

5 Verwandte Arbeiten

Programme, die zur Demonstration von Angriffen auf OAuth 2.0 und damit auf eine bessere Verständlichkeit dieser Attacken abzielen, wurden in Rahmen bisheriger Arbeiten nicht erstellt und untersucht, wodurch sich diese Arbeit von anderen im selben Themenbereich abgrenzt. In [4] ist besonders hervorzuheben, da diese auch eine starke Praxisorientierung aufweist. In dieser Arbeit wurde gesondert OpenID Connect betrachtet und OAuth 2.0 nur am Rande erwähnt. Der Fokus stellte dabei die Entwicklung eines Tools dar, welches eine IdP, welche dem Authorization Server in dieser Arbeit entspricht, im Kontext von OpenID Connect simulieren sollte und dabei erlaubt, Nachrichten zu analysieren und zu manipulieren. Dabei konnte das Verhalten der IdP in den einzelnen Phasen verändert werden. Die Anwendung war allgemeiner gehalten als die aus diesem Projekt, da damit verschiedene Angriffe dargestellt werden können, welche aber nicht direkt implementiert waren. Das bedeutet, dass es nur ein Gerüst an Werkzeugen und Möglichkeiten, in OpenID Connect einzugreifen, bot, um damit die Sicherheit von Systemen zu testen. Obwohl in der Arbeit auf Angriffe hingewiesen wurde und die möglichen Quellen von Manipulationen dargestellt wurden, sind keine Implementierungen von konkreten Angriffe oder Angriffsvektoren vorgenommen worden. Denn das Ziel dieser Arbeit lag auf der Überprüfung der Sicherheit von Systemen, indem der Nutzer Manipulationen der Nachrichten durchführt, um Angriffe zu simulieren.

6 Zusammenfassung und Ausblick

Das realisierte Projekt beschäftigte sich mit der Demonstration von Angriffen auf OAuth 2.0. Dazu wurden die nötigen Flows mitsamt wichtigen Erweiterungen integriert, Angriffe verwirklicht und eine intuitive Weboberfläche erschaffen. Mit dieser können unterschiedliche Angriffe auf OAuth 2.0 gestartet und protokolliert werden. Damit grenzt sich das Projekt von anderen Arbeiten ab, welche nur einzelne Angriffe betrachten und kein zentrales Tool zur Demonstration dieser in der Praxis aufweisen. Damit sollen auch Personen, die sich bislang nicht mit der Materie auseinandergesetzt haben, ein Verständnis für bestimmte Sicherheitsrisiken in Bezug auf den Einsatz von OAuth 2.0 gewinnen. In dieser schriftlichen Ausarbeitung sollten die integrierten Flows von OAuth 2.0 und von OpenID Connect vorgestellt und ihre Anfälligkeit für bestimmte Angriffe deutlich gemacht werden. Daneben wurden Details zu der Realisierung des Projektes beleuchtet und auch dargelegt, welche Kompromisse getroffen werden mussten.

Ausblick

Diese Arbeit deckt nur einen Teil der möglichen Angriffe ab. Indem weitere Angriffe realisiert werden und auch mögliche Verbesserungen und Erweiterungen von OAuth 2.0 implementiert werden, könnte das Projekt auch in Zukunft relevant und hilfreich sein. Der Fokus der Arbeit lag insbesondere auf OAuth 2.0 und nicht auf OpenID Connect. Durch weitere realisierte Flows und Angriffe, insbesondere mit einer stärkeren Abdeckung von OpenID Connect, könnte die Relevanz des Projektes noch gesteigert werden.

Literaturverzeichnis

[1] Ars Technica. *50 million Facebook accounts breached by access-token-harvesting attack*. 2018. URL: https://arstechnica.com/information-technology/2018/09/50-million-facebook-accounts-breached-by-an-access-token-harvesting-attack/.

[2] *Burp Suite - Application Security Testing Software*. URL: https://portswigger.net/burp.

[3] B. Campbell, J. Bradley, N. Sakimura, T. Lodderstedt. „OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens“. In: *Internet-Draft draft-ietf-oauth-mtls-09, Internet Engineering Task Force* (2018).

[4] Christian Mainka. „Developing a Security Analysis Tool for OpenID-based SingleSign-On Systems“. Bachelor Arbeit. Ruhr-Universität Bochum, 2013. URL: https://www.nds.ruhr-uni-bochum.de/media/ei/arbeiten/2014/12/04/OpenIDAttacker.pdf.

[5] Daniel Fett. „PKCE vs. Nonce: Equivalent or Not?“ In: (2020). Hrsg. von Daniel Fett. URL: https://danielfett.de/2020/05/16/pkce-vs-nonce-equivalent-or-not/.

[6] D. Fett, P. Hosseyni, R. Kuesters. *An Extensive Formal Security Analysis of the OpenID Financial-grade API*. 2019. URL: http://arxiv.org/pdf/1901.11520v1.

[7] D. Fett, R. Kuesters, G. Schmitz. *A Comprehensive Formal Security Analysis of OAuth 2.0*. 2016. URL: http://arxiv.org/pdf/1601.01229v4.

[8] GitHub. *danielfett/annexlang*. URL: https://github.com/danielfett/annexlang.

[9] GitHub. *danielfett/proto6749*. URL: https://github.com/danielfett/proto6749/commit/a22321ea22d1073a1adad127dfcd518991d4d245.

[10] GitHub. *jazzband/django-oauth-toolkit*. URL: https://github.com/jazzband/django-oauth-toolkit.

[11] E. Hammer-Lahav, D. Recordon, D. Hardt. *The oauth 1.0 protocol*. Techn. Ber. RFC 5849, April, 2010.

[12] D. Hardt et al. *The OAuth 2.0 authorization framework*. Techn. Ber. RFC 6749, October, 2012.

[13] *LaTeX - A document preparation system*. URL: https://www.latex-project.org/.

[14] T. Lodderstedt. „OpenID Connect: Login mit OAuth, Teil 1 – Grundlagen“. In: *heise Online* (2014). URL: https://www.heise.de/developer/artikel/OpenID-Connect-Login-mit-OAuth-Teil-1-Grundlagen-2218446.html?seite=all.

[15] *OAuth 2.0 — OAuth*. URL: https://oauth.net/2/.

[16] *OpenID Connect | OpenID*. URL: https://openid.net/connect/.

[17] PyPI. *django-request-logging*. URL: https://pypi.org/project/django-request-logging/.

[18] PyPI. *django-revproxy*. URL: https://pypi.org/project/django-revproxy/.

[19] PyPI. *pkce*. URL: https://pypi.org/project/pkce/.

[20] PyPI. *PyYAML*. URL: https://pypi.org/project/PyYAML/.

[21] N. Sakimura, J. Bradley, N. Agarwal. *Proof key for code exchange by oauth public clients*. 2015.

[22] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, C. Mortimore. „Openid connect core 1.0“. In: *The OpenID Foundation* (2014). URL: https://openid.net/specs/openid-connect-core-1_0.html.

[23] *The Web framework for perfectionists with deadlines | Django*. URL: https://www.djangoproject.com/.

[24] *The ZAP Homepage*. URL: https://www.zaproxy.org/.

Alle URLs wurden zuletzt am 11. 11. 2020 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift