Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

XSS in Issue Tracking Systems

Moritz Hildebrand

Course of Study:	Informatik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Dr. Guido Schmitz

Commenced:	06.10.2020
Completed:	06.04.2021

Abstract

Today, virtually every software project, especially in a collaborative and distributed setting, is managed through an issue tracking system (ITS). As developers rely heavily on ITSs, the risk of cyberattacks and their associated impact increases. An interesting particularity of ITSs is that, compared to conventional web applications, the attack surface is extended through additional input interfaces such as email or version control systems (VCSs). This bachelor thesis develops a methodology to test ITSs for Cross-site scripting (XSS) vulnerabilities via these ITS-specific input interfaces. Exemplarily, we implement the developed methodology for the input interfaces email and Git and test it on the three open-source ITSs Redmine, MantisBT, and Trac.

Abstract (German)

Issue-Tracking-Systeme helfen Softwareentwicklern Projekte zu organisieren und Fortschritte nachzuvollziehen. Vor allem im Rahmen dezentraler und kollaborativer Entwicklung sind Issue-Tracking-Systeme ein unverzichtbares Werkzeug. Entsprechend groß ist die Gefahr durch Cyberangriffe. Zusätzlich zur Weboberfläche, erlauben Issue-Tracking-Systeme Eingaben über, für Webapplikationen unübliche, Schnittstellen wie E-Mail oder Versionsverwaltungssysteme. In dieser Bachelorarbeit entwickeln wir eine Methodik, um Issue-Tracking-Systeme auf Cross-Site-Scripting (XSS)-Schwachstellen über diese Eingabeschnittstellen zu testen. Exemplarisch implementieren wir die entwickelte Methodik für die Eingabeschnittstellen E-Mail und Git und testen die drei Open Source Issue-Tracking-Systeme Redmine, MantisBT und Trac damit.

Contents

List of Figures

List of Tables

Listings

Acronyms

API application programming interface. 15

CORS Cross-Origin Resource Sharing. 30

CSP content security policy. 15, 30

CSRF cross-site request forgery. 32

CVE Common Vulnerabilities and Exposures. 23

CVSS Common Vulnerability Scoring System. 31

DBMS database management system. 16

DOM Document Object Model. 19

ERB Embedded Ruby. 53

HTML Hypertext Markup Language. 15

HTTP Hypertext Transfer Protocol. 19

IDE integrated development environment. 28

ITS issue tracking system. 5, 15, 16

MantisBT Mantis Bug Tracker. 15

OWASP Open Web Application Security Project. 15

pentester penetration tester. 15

POC proof of concept. 28

RCE remote code execution. 32

SOP Same-Origin-Policy. 30

URL Uniform Resource Locator. 22

VCS version control system. 15

w3af Web Application Attack and Audit Framework. 47

W3C World Wide Web Consortium. 34

XSS cross-site scripting. 5, 15, 19

ZAP Zed Attack Proxy. 37

1 Introduction

Issue tracking systems (ITSs) belong to the essential tools for software development. Developers, especially in collaborative and distributed settings, use ITSs for organizing projects of any size. Among the big commercial players are GitHub, Jira, and Bitbucket. However, open-source issue-trackers like Redmine, *Mantis Bug Tracker (MantisBT)* and Trac are in the top-tier as well. ITSs keep track of project progress and offer the capability to manage and maintain issues via multiple input interfaces. The range of supported input interfaces varies among different ITSs, with web, email, *version control systems (VCSs)* and REST *application programming interfaces (APIs)* being the most common ones. To improve the user experience GitHub and co. facilitate and automate as many workflow processes as possible and thus take on even more importance. With the stronger getting reliance on ITSs the danger of cyberattacks and the associated damage grows [Byr20; CVE19; New18]. For this reason, providers step up the investment in cybersecurity and conduct bug bounty programs [Git; Sof]. Due to the wide variety of input interfaces in ITSs, they offer a large attack surface for injection attacks. One type of injection attack that is of particular interest to us and this thesis focuses on is *cross-site scripting (XSS)*. XSS belongs to the most common types of security bugs found in web applications and occupies a spot in the *Open Web Application Security Project (OWASP)* Top Ten since decades [OTT17]. XSS refers to the injection of client-side code to compromise user interactions with a web application. A typical XSS attack is to submit JavaScript code inside `<script>` tags via an input form to manipulate the *Hypertext Markup Language (HTML)* code of the website in such a way that the browser executes malicious JavaScript when the victim accesses the site. An attacker can compromise the interaction between user and web application with an XSS exploit to hand, often leading to account takeover through stolen cookies or passwords. When it comes to web application security common countermeasures against XSS attacks are input filtering, encoding and *content security policy (CSP)*, but still *penetration testers (pentesters)* and security researchers continue to disclose XSS bugs in web applications every day. The hope is to detect security holes before adversaries can exploit them for malicious purposes. Hence, much research has already gone into web application vulnerability detection and serves as the foundation for many well-established tools and strategies. However, compared to standard web user interfaces, comparatively little research went into how pentesters can use unconventional input interfaces like email or VCSs to disclose XSS vulnerabilities. Thus it is interesting to investigate for attack vectors in those domains. Within this thesis, we design a methodology to systematically analyze ITSs using ITS-specific input interfaces. To evaluate the applicability of our approach, we apply it to the three open-source ITSs Redmine, MantisBT and Trac. In the next section, we give an introduction to ITSs in general and provide an overview of the three testees.

System	**Implementation language**	**Back end**	**Input interfaces**
Redmine	Ruby	MS SQL Server, MySQL, PostgreSQL, SQLite	Web, Email, VCS,[1] REST
MantisBT	PHP	MySQL, PostgreSQL, MS SQL, Oracle, DB2	Web, Email, VCS,[2] REST, SOAP[3]
Trac	Python	MySQL, PostgreSQL, SQLite	Web, Email, VCS[4]

Table 1.1: Comparison of Redmine, MantisBT and Trac regarding implementation information. A more detailed overview is available at Wikipedia [Wik20a; Wik20b] and the respective documentation sites [Man19; Reda; Traa].

1.1 Issue tracking systems (ITSs)

An ITS is a software application for managing and maintaining issues. Issues are the electronic form of a business request, an incident report, or a request in the form of a revision request, an information request, or a feature request. Issues can be tracked via customizable criteria, most commonly by priority, status, or assignee. Usually, ITSs support multiple interfaces to users for creating and managing issues. Typical interfaces are web-based or VCSs like Git. Modern information technology enterprises heavily rely on some form of ITSs to organize their projects, especially in widely distributed settings. The main advantage of ITSs is that the documentation process is simplified. Therefore, the people involved can focus on resolving issues, and progression is easy to comprehend for anyone. Besides, in large cooperative settings, ITSs enjoy great popularity among individual developers due to project management ease.

Table 1.1 contains a testee-wise listing of the respective input interfaces and implementation languages. All three test ITSs are published under open source licenses and provide a plugin API. ITSs vary in implementation language and *database management system (DBMS)* back end, but offer similar features. Project leaders can assign users different customizable roles with corresponding permissions for different projects. Users can then create issues, label them, and sort them by tags such as priority, assignee, or status. Most web interfaces are feature-rich in the displayed domain and offer the possibilities of file uploads and markdown comments. In most cases, ITSs support hosting multiple projects and sub-projects within one installation for which users can have different roles. Maintainers can configure roles, privileges, and workflows as required.

1.1.1 Redmine

Jean-Philippe Lang launched Redmine in 2006 as an open-source project under the GNU General Public License version 2 (GPLv2). Redmine is written using the Ruby on Rails framework and supports different database back ends (see Table 1.1). The source code is hosted on the Redmine

[1] SVN, CVS, Git, Mercurial and Bazaar

[2] Git, SVN, Mercurial

[3] Support for mobile devices

[4] Git, SVN, Perforce, Mercurial, Darcs, Bazaar, monotone

website and is mirrored at GitHub [Redc; Redd]. Since its launch, Redmine has attracted several popular projects for its use. Among them are the Ruby language, Lighttpd, and TYPO3 Forge [Rede]. Redmine provides multiple documentation and reporting features like an integrated wiki, discussion forums, news blogs, email integration, calendars, Gantt Charts, RSS, export to PDF and Excel/CSV functionality, as well as notification interfaces for Atom and Twitter. Over time the Redmine community has developed numerous plugins to improve usability and functionality. For example, source code integration is realized through plugins as well.

1.1.2 MantisBT

The initial MantisBT release was in the year 2000 by its original author Kenzaburo Ito. MantisBT is written using the PHP programming language and supports multiple DBMSs as back ends (see Table 1.1). MantisBT is published under the GNU General Public License (GPL) and its source code is hosted at GitHub [Manb]. Similar to Redmine, MantisBT provides notification interfaces for email, RSS, and Twitter as well as documentation features like roadmaps, graphing, and wikis. The web user interface is written using plain HTML. MantisBT latest version has no built-in source code integration feature, as it was removed in version 1.3.x. However, source code integration is available through the MantisBT Source Integration plugin authored by John Reese [Dav10]. Like most alternatives, MantisBT supports the management of multiple projects and issue tracking customization.

1.1.3 Trac

Edgewall Software released Trac in 2006 under the New Berkeley Software Distribution License (New BSD). The source code is available at the Trac website and mirrored at GitHub [Trab; Trac]. Trac is written in Python and supports SQLite, PostgreSQL and MySQL as DBMS (Table 1.1). Trac contains a web-based interface for repository (SVN or Git) access, a bug tracker, and a wiki. Trac follows a minimalistic approach for issue tracking. Nevertheless, popular open-source projects like FFmpeg, FileZilla, and HTTPS Everywhere use it [Trae]. To provide more functionality, the project administrator can install plugins. Popular plugins and Trac itself can be installed via the packet managers Advanced Packaging Tool (APT) and `pip`.

1.2 Contributions

XSS is a widespread attack on the web. Using XSS, an attacker can gain complete control over a user account, allowing them to steal confidential information and impersonate the user. The focus of this bachelor thesis lies on developing a strategy to systematically audit ITSs for XSS vulnerabilities. In addition to the web interface, ITSs provide various other input interfaces that introduce potential security risks. Throughout this bachelor thesis, we investigate if and how ITS penetration tests differ from traditional XSS security assessment approaches. We especially emphasize non-standard input methods, which are typical for issue-tracking web applications. The first input interface we chose to apply our designed approach to is email. Further, we have great interest in VCS interfaces that allow files and version control information to be entered into a system that in some way is then represented in the web front-end of the ITS. As VCS representative, we chose Git. In order to cover as many areas of a web application as possible, it is helpful to automate processes as much as possible or at least to support manual tasks with tools. In the area of tooling, this thesis addresses the question of how vulnerability detection in the area of unconventional XSS attack vectors can be automated. The practical element of this thesis is applying the developed approach to three open-source ITSs, namely Redmine, MantisBT and Trac. To make our approach applicable to arbitrary ITSs, including proprietary systems, we intentionally design it easy to apply and generic.

1.3 Thesis Structure

The remaining thesis is structured as follows. The XSS and defenses chapter provides required explanations to comprehend the work around this thesis, mainly XSS fundamentals. The approach chapter contains the main contribution of our work and describes our strategy on analyzing ITSs for XSS vulnerabilities. We discuss the challenges the approach faces and present our implementation applied to email and Git input interfaces. In the results chapter, we explain our test setup and report our findings. Last, the conclusion and future work chapter serves as a summary and contains an outlook on possible further research.

2 XSS and Defenses

The XSS and defenses chapter provides a basis of knowledge necessary to comprehend the proceeding work. The section begins by introducing the concept of XSS subdivided into the three XSS types, namely reflected, stored and *Document Object Model (DOM)*-based XSS. Further, we present common XSS attack vectors, contexts in which XSS arises and talk about the exploitability of XSS vulnerabilities as well as countermeasures. In the subsequent section, we present related research that deals with XSS detection and mitigation techniques.

2.1 Cross-site scripting (XSS)

XSS is one of the most popular web application vulnerabilities. XSS refers to an injection-based security vulnerability that undermines the integrity of interactions between users and web applications. The basic idea is that information from a context in which it is not trustworthy is inserted into another context in which it is classified as trustworthy. An attacker provides maliciously crafted input to a system in order to trigger unintended behavior. XSS attacks are based on manipulating a vulnerable website to return malicious JavaScript to its users. As the browser executes the malicious code, the attacker can compromise the user interaction with the web application. Thereby, every input field is an attack vector, and XSS can occur even in different applications running on the same domain.

2.1.1 XSS Types

This subsection is dedicated to the three XSS types. We explain reflected, stored, and DOM-based XSS respectively and provide basic examples for each.

Reflected XSS

Reflected XSS arises when web applications insecurely include data from a request in the response. Therefore, by controlling the request data, an attacker can inject HTML code into the response. Situations in which the possibility to use harmless HTML tags, such as `<h1>` or `<p>`, is desired do exist, but generally user controlled tags are a sensitive security risk. Using `<script>` tags, it is possible to execute JavaScript code from an HTML source. Figure 2.1 shows an example, where data from a search query is included in the *Hypertext Transfer Protocol (HTTP)* response. Suppose a simplistic website that offers to search its contents and returns the results as in Figure 2.1a. When searching for “asdf”, the web application returns an HTML site saying “0 search results for asdf”. The responsible server-side code can look like the code snippet shown in Figure 2.1b. Apparently the user controlled search input is reflected in the immediate response inside `<div>` tags. To exploit

```
...
<div>0 search results for 'asdf'</div>
...
```

(a) Part of HTML response for search query “asdf”.

```
<?php
    ...
    $query = $_GET['query'];
    echo "<div>$num_results search results for '$query'</div>"
    ...
?>
```

(b) Vulnerable PHP code snippet.

```
...
<div>0 search results for '<script>alert()</script>'</div>
...
```

(c) Part of HTML response for search query “`<script>alert()</script>`”.

Figure 2.1: Reflected XSS example - The search string is contained in the HTTP response.

this vulnerability, an attacker can search for the string “`<script>alert()</script>`” and trigger an alert pop up in the browser when the response is loaded. The response of the XSS query contains “0 search results for <script>alert()</script>” (Figure 2.1c). In this example, browsers can not differentiate between HTML source code and user injected HTML tags.

Reflected XSS is the easiest XSS type to test for. A pentester needs to identify all input vectors on a web application and check whether the response contains some reflection of the input. The detection of reflected input data can easily be automated by supplying a unique identifiable string to every entry point and searching the HTTP response for it. When the supplied string is located somewhere in the response, an input reflection is present, and scans can automatically probe different XSS payloads. Sometimes input data is reflected more than once in a response. However, identifying reflected data does not imply a XSS vulnerability. Countermeasures can still prevent an attacker from exploiting the website, more on that in Section 2.1.4.

Stored XSS

The principle of stored XSS is similar to reflected XSS with the distinction that the vulnerable web application includes input data in a later HTTP response unsafely. As the name suggests, the potentially malicious input is stored in the application and is part of a later response, possibly in another context. A typical example for stored XSS is a simple social network site, which allows account creation with custom usernames. For simplicity, we assume the site does not implement any countermeasures against XSS attacks. When the user visits the profile page of another user, the username is displayed. In this scenario, user-controlled data, the username, is

Sink	**Description**
`eval()`	The `eval()` function evaluates JavaScript code represented as a string. [1]
`Document.write()`	"The `Document.write()` method writes a string of text to a document stream opened by `document.open()`." [Mozb]
`Document.writeln()`	Writes a text string, followed by a newline character, into a document.
`Document.domain`	`Document.domain` accesses the domain of the current document. [2]
`Document.location`	`Document.location` accesses the location object. The browser will redirect to any site it is assigned. XSS can be achieved via the JavaScript pseudo-protocol feature. [3]
`Element.innerHTML`	The `innerHTML` element property retrieves or sets the HTML contained in an element.
`Element.outerHTML`	The `outerHTML` element property retrieves or sets the HTML element itself.
`Element.insertAdjacentHTML`	The `insertAdjacentHTML` element property interprets the specified text as HTML and inserts resulting nodes at the specified position in the DOM.
`Element.onevent`	The `onevent` element property stands for a variety of event handlers, that specify how elements react to events. Common event handlers are `onclick`, `onload` and `onerror`

Table 2.1: Common sinks which can lead to XSS.

included in HTTP responses unchanged. To exploit this security vulnerability, the adversary can create a user account and abuse the username to inject JavaScript code. The mentioned test payload `<script>alert()</script>` as username is sufficient to spawn an alert popup in every browser that visits the profile page. Detecting stored XSS is more complex than detecting reflected XSS. To scan for stored XSS, we must identify user-controlled data, which is returned to the user in a later response. This is why automated scans check for stored XSS in multiple phases. First, the web application is crawled to identify all input vectors. Then the tool supplies input to the entry points found in the previous phase before the application is crawled again to find out where user-controllable data is returned. When user-controllable data is found, tools can supply different payloads in an attempt to find an exploit.

DOM XSS

DOM-based XSS is a client-side vulnerability in web applications. In this context, DOM refers to the API that defines the structure of HTML documents in browsers. Thus, the DOM specifies how HTML elements in browsers can be found, added, modified or deleted. A DOM XSS vulnerability exists, if attacker-controllable data is included in unsafe dynamic code execution. Referring to DOM XSS, objects or functions that support dynamic code execution are called sinks. A selection of common sinks which can lead to DOM XSS is listed in Table 2.1.

[1] The Mozilla documentation advises to avoid the usage of `eval()` [Mozd].

[2] `Document.domain` is deprecated [Moza].

[3] The JavaScript pseudo-protocol feature is explained in Section 2.1.2.

```
...
var searchParams = new URLSearchParams(document.location.search);
eval('var tax = ' + searchParams.get('income') + '* 0.1;');
...
```

(a) DOM XSS vulnerable JavaScript source code.

$\underbrace{\text{https://domain.com/vulnerable.html}}_{\text{Example URL}}$ `?income=alert()`

(b) Exploit for vulnerable code from Figure 2.2a. The string `alert()` is assigned to the `income` URL parameter.

Figure 2.2: DOM XSS example with URL parameter as source and `eval()` as sink.

If an attacker can control data that is passed into a sink, they can use this knowledge to execute arbitrary code. An example of DOM-based XSS is provided in Figure 2.2. Figure 2.2a shows the vulnerable JavaScript code. The code makes use of the `eval` function to calculate "`var tax = income * 0.1`", where `income` is a *Uniform Resource Locator (URL)* parameter. Accordingly, the adversary sets the value of `income` to `alert()`, which when passed to the `eval` function creates an alert popup. Since the attacker-controlled variable is directly inserted into a JavaScript context, there is no need to surround the payload with HTML `<script>` tags. The resulting exploit URL is shown in Figure 2.2b.

DOM-based XSS usually involves code flow analysis and is not as trivial to detect as reflected or stored XSS. To detect DOM XSS sources, which eventually are passed to a sink, need to be identified. Usually, the reversed approach is more straightforward; searching for sinks and figuring out which sources can be used to control the input.

2.1.2 Contexts

After detecting user-controllable output on a web application, the next step is to identify its context. For a successful XSS attack, the web application must necessarily interpret parts of user input as executable code. Code execution often requires breaking out of a context and changing into another one. Attacker-controllable data can appear in various contexts such as HTML tags and attributes or inside JavaScript code. In the subsequent paragraphs we consider common XSS contexts, with examples for each in Table 2.2. In the examples, we assume that no precautions to prevent XSS attacks are taken.

HTML Text

Should controllable data appear within HTML text on a vulnerable website, the attacker can inject arbitrary HTML code including `<script>` tags. Browsers execute vanilla JavaScript in between `<script>` tags as soon as the browser loads the page containing it, given no script blocker, e.g., browser add-on prevents execution. In addition, other tags such as the `<img>` tag can be abused for XSS attacks. In case of the `<img>` tag, attackers often make use of event handlers like `onerror` or `onload` in combination with the JavaScript pseudo-protocol. The URL specification requires that every URL needs to have a scheme at the beginning, see Figure 2.3. Modern browsers support a wide range of pseudo-schemes to implement additional features that, concerning the JavaScript pseudo-protocol, allow for code execution. A resulting XSS payload could look somewhat like `<img src="" onerror=:alert()"/>`. Note that the `src` attribute in the `<img>` tag is intentionally left empty, to trigger the `onerror` event handler. When the browser cannot locate the image resource, the `onerror` event handler is triggered and executes the JavaScript specified using the pseudo-scheme.

HTML Tag Attributes

XSS occurring in HTML attribute values can be exploited similarly to the HTML tag context, the only difference being that the attacker-controllable data appears inside an HTML tag attribute. Therefore, if the attribute itself is not exploitable, we need to close the attribute value with double quotes (") and insert another attribute or even close the HTML tag to inject a new one. Both possibilities can be seen in Table 2.2. HTML tag attributes which in combination with the JavaScript pseudo-protocol can be exploited are for example `href`, `src` and `action` along with various HTML event handlers.

JavaScript Code & Template Literals

Occasionally user input is directly included in JavaScript computations. Should the XSS context be existing JavaScript code in an HTTP response, many situations can arise that require diverse techniques to perform XSS. Suppose the scenario in Table 2.2, where user data appears inside a JavaScript string. Given that none of the input characters are encoded, we can terminate the string with a single quote ('), end the prior statement with a semicolon (`;`), and append arbitrary JavaScript code. To not break the JavaScript code, it is necessary to take care of the remaining part of the original statement. The most basic solution for this is to comment out the rest of the line. Other ways to use variables inside strings are JavaScript template literals that allow for embedded expressions. Template literals are enclosed by backticks (`) and use the `${<expression>}` syntax to evaluate expressions within a string. Hence, plain JavaScript code can be used as an exploit.

2.1.3 Attack Vectors

Frequently XSS payloads are supplied via URL parameters or input fields. In this subsection, we present a selection of XSS attack vectors. Typically, XSS attacks are limited to web interfaces, as these represent the only means of interaction between user and application. Regarding issue-tracking, we are not limited to the web interface solely. Additional input interfaces like email, VCSs or file uploads represent additional attack vectors. Table 2.3 lists a real-world example with according *Common Vulnerabilities and Exposures (CVE)* for each introduced attack vector.

Context	**Example exploit**
HTML tags	`<h1><script>alert()</script></h1>`
HTML tag attributes	`<img src="" onload="javascript:alert();"/>`
HTML tag attributes	`<input value=""/><script>alert()</script><input value=""></input>`
JavaScript code	`...` `var input = ''; alert();//';` `...`
JavaScript template literals	`...` ``var input = `${alert();}`;`` `...`

Table 2.2: XSS contexts and example exploits. The attacker supplied input is marked in red. See [Porb] for reference.

CVE	**Attack vector**	**Description**
CVE-2014-9607	URL parameter	"Cross-site scripting (XSS) vulnerability in remotereporter/load_logfiles.php in Netsweeper 4.0.3 and 4.0.4 allows remote attackers to inject arbitrary web script or HTML via the url parameter." [NIS14]
CVE-2020-9016	HTTP header	"Dolibarr 11.0 allows XSS via the joinfiles, topic, or code parameter, or the HTTP Referer header." [The20b]
CVE-2020-9447	File upload	"[. . .] XSS (cross-site scripting) vulnerability in GwtUpload 1.0.3 in the file upload functionality. Someone can upload a file with a malicious filename, which contains JavaScript code, which would result in XSS. [. . .]" [The20c]
CVE-2019-12311	File upload	"Sandline Centraleyezer (On Premises) allows Unrestricted File Upload leading to Stored XSS." [NIS19]
CVE-2020-15562	Email	"An issue was discovered in Roundcube Webmail before 1.2.11, 1.3.x before 1.3.14, and 1.4.x before 1.4.7. It allows XSS via a crafted HTML e-mail message, as demonstrated by a JavaScript payload in the xmlns (aka XML namespace) attribute of a HEAD element when an SVG element exists." [The20a]
CVE-2018-14605	VCS	"An issue was discovered in GitLab Community and Enterprise Edition before 10.8.7, 11.0.x before 11.0.5, and 11.1.x before 11.1.2. XSS can occur in the branch name during a Web IDE file commit." [CVE18]

Table 2.3: Example list of disclosed XSS exploits sorted by attack vector.

$$\underbrace{\textit{https}}_{\text{scheme}}\text{ : // }\overbrace{\text{domain.com}}^{\text{host}}\underbrace{\text{/resource/path}}_{\text{path}}\text{ ? }\overbrace{\text{param1=123\&param2=abc}}^{\text{parameters}}\text{ \# }\underbrace{\text{fragment}}_{\text{fragment}}$$

Figure 2.3: Example URL with URL parameters: Scheme - the technical method by which the resource is addressed (e.g. https, ftp, Git). Host - IP/domain of the host. Path - the path to the resource, can also be an remote path. Parameters - URL parameters separated by &. Fragment - can reference a part of the resource (e.g. HTML anchors). ?, # - Separators between path, parameters and fragment.

Character	#	&	(	)	<	>	?
URL encoding	`%23`	`%26`	`%28`	`%29`	`%3C`	`%3E`	`%3F`

Table 2.4: Some special characters and their according URL encoding.

XSS via URL Parameters

The usual way to access a website is to open a URL in a web browser, which already comprises a potential XSS vulnerability. A URL consists of several parts (see Figure 2.3). The first part is the scheme specifying the access method, for a website the scheme is `http` or `https`, followed by "`://`". The next part is the host, usually a domain name or IP address. To navigate through a website after the host path to a resource can be supplied. If there is any additional information to transmit, the URL can contain parameters. A URL parameter is a key-value pair and separated from the preceding part by a question mark. Multiple URL parameters are separated via the ampersand symbol (`&`). After the parameters, appended by the number sign (`#`), a fragment can be provided. The fragment can reference a part of the resource, typically an anchor in an HTML page. In reality, URLs can be even more complicated, like containing login credentials, but for the purpose of this chapter, a simplified version is sufficient. An example URL is broken down in Figure 2.3.

When a website reflects the value of a URL parameter in its code, an attacker can provide JavaScript code surrounded by `<script>` tags as parameter and cause XSS. As the web server delivers the HTML site to a browser, the browser can not distinguish between legitimate HTML and the tag from the URL parameter (see Section 2.1.1). An example exploit for this attack can be found in Figure 2.4a. In this scenario, accessing the URL from Figure 2.4a triggers an alert popup. At this point, there is one more mechanism worth mentioning in the process of accessing a website. If characters are to be transmitted as values, which have a special meaning in the context of a URL, they are encoded using the so-called URL encoding. Basically certain characters are substituted by their percentage representation (see Table 2.4). Figure 2.4b shows the URL with an encoded parameter. URL encoding can in some scenarios be used to evade filter-based defense mechanisms.

$$\mathit{https}\text{://domain.com/path?search=}\underbrace{\texttt{<script>alert()</script>}}_{\text{payload}}$$

(a) Example URL with payload as parameter.

$$\mathit{https}\text{://domain.com/path?search=}\underbrace{\text{\%3Cscript\%3Ealert\%28\%29\%3C\%2Fscript\%3E}}_{\text{URL encoded payload}}$$

(b) Encoded URL from Figure 2.4a

Figure 2.4: Example URLs: Figure 2.4a shows an example of how an XSS payload can be passed as URL parameter. Figure 2.4b shows the encoded URL from Figure 2.4a.

XSS via HTTP Headers

When a website is required to send large amounts of data - images, for example - or confidential information to a server, the `GET` method is not ideal because all the data transferred is written openly in the address bar. In these cases, the `POST` method is appropriate. It does not write the URL parameters into the URL, but appends them to the HTTP header. `POST` requests are often used in connection with input forms. An example header can be seen in Listing 2.1. `POST` indicates the HTTP method, `/test` the resource path and `HTTP/1.1` the HTTP version. The minimal example contains 3 header fields, namely `Host`, `Content-Type` and `Content-Length` (see [Hen20] for reference). The last line contains the appended key-value pairs. In general, all HTTP header fields can be used to exploit an XSS vulnerability. In the case of CVE-2020-9016 (see Table 2.3) a field named `topic` is used to exploit an XSS vulnerability in Dolibarr 11.0, a management software.

```
POST /test HTTP/1.1
Host: foo.example
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

field1=value1&field2=value2
```

Listing 2.1: HTTP `POST` header example from the MDN web docs [Moze].

Similar to XSS via URL parameters, when data from an HTTP `POST` request is handled unsafely, XSS arises. A standard example is comment fields on websites, which enable stored XSS. The attacker posts a comment containing an XSS payload, which is delivered to every user visiting the site. This attack is a slightly different variation of the stored XSS attack described in Section 2.1.1.

XSS via File Uploads

A common functionality in ITSs is the file upload feature. Issue creation often comes along with the possibility of file attachments, which bears a security risk. Not only the filename can contain an XSS payload (e.g. `<script>alert()</script>.png`), also the file itself can. Suppose a web application offers a personal picture gallery and displays metadata like resolution and artist. Without proper XSS protection, users can craft a malicious file containing an XSS payload in the artist metadata entry and upload it. The result of this scenario is a stored XSS vulnerability. Security vulnerabilities originating from file uploads are a reason for segregating the file uploads to a dedicated domain. Hosting different parts of a website on separate domains is a form of damage control if a vulnerability exists. More countermeasures and mitigation strategies are described in Section 2.1.4.

CVE-2020-9447 is an example of an XSS vulnerability due to improper filename handling. The exploitability for CVE-2020-9447 is fairly low, since the victim must be tricked into uploading a file with XSS payload in the filename (see Self-XSS in Section 2.1.5). For CVE-2019–12311 this is not the case, an XSS payload contained in an uploaded HTML file is being executed when a report file is downloaded. CVE-2019–12311 also requires the victim to interact with the application, but the interaction is arguably less suspicious.

XSS via Email

There exist all sorts of web services that process emails somehow, which creates another XSS attack vector. The example chosen here is the Roundcube webmail software. Roundcube allows its users to manage their emails through a web interface and supports formatting and semantic markup capabilities of HTML emails. When displaying HTML emails, it is desirable to render only "good" HTML. Any element which can be abused for code execution can be exploited in a harmful manner. CVE-2020-15562 demonstrates how improper sanitation in HTML emails can lead to XSS and how a single disregarded edge case can be dangerous. Between version 1.2 and 1.4.7, Roundcube Webmail XML namespace attributes of `<svg>` tags in received emails were not sanitized appropriately. In the sanitizing process, the `xmlns` attribute of the root element was taken and appended to the `<svg>` tag without modification. The PHP snipped in Listing 2.2 shows the responsible code, the `foreach` loop, beginning in line 3, iterates over all non-default XML namespaces and adds them to the `<svg>` tag. The `<svg>` tag later becomes part of the rendered HTML email. To exploit that the `xmlns` attribute is not sanitized in any way, an attacker can terminate the `xmlns` attribute with an HTML-encoded quotation mark (`&quot;`) and append another attribute. For JavaScript code execution an `onload` event listener is sufficient. The full payload is shown in Figure 2.5a, as root element a `<head>` tag is used. When Roundcube now receives an email with the payload, it processes the HTML and renders an `<svg>` tag with an `onload` event listener that executes `alert(document.domain)` (see Figure 2.5b). A more in-depth description of CVE-2020-15562 and how to exploit the vulnerability can be found at [And20].

```
if ($tagName == 'svg') {
    $xpath = new DOMXPath($node->ownerDocument);
    foreach ($xpath->query('namespace::*') as $ns) {
        if ($ns->nodeName != 'xmlns:xml') {
            $dump .= ' ' . $ns->nodeName . '="' . $ns->nodeValue . '"';
        }
    }
}
```

Listing (2.2) Vulnerable JavaScript code from Roundcube Webmail which led to CVE-2020-15562 [Ale20].

```
<head xmlns="&quot; onload=&quot;alert(document.domain)"><svg></svg></head>
```

(a) proof of concept (POC) payload of CVE-2020-15562 [And20].

```
<svg xmlns=""onload=(document.domain)"/>
```

(b) Sanitized payload after being processed [And20].

Figure 2.5: Vulnerable code snipped and payload of CVE-2020-15562 [The20a].

XSS via VCSs

The number of disclosed XSS vulnerabilities caused by VCSs is comparably low since only a tiny proportion of web applications supports such features. Nevertheless, XSS via VCSs exists. An easy-to-understand real-world example is CVE-2018-14605. In CVE-2018-14605 a branch name within a Git repository is used to achieve XSS in the GitLab web *integrated development environment (IDE)*. The following steps will reproduce the *POC* in GitLab version <11.2 [Fra18].

1. Create a new branch containing the XSS payload and push it to the remote GitLab repository.
2. Make a change to the branch in the Web IDE and press the commit button twice.

The problem was that the branch name was not being sanitized correctly in the commit-popup. Within the commit-popup, the branch name is being displayed, which can consist of plain HTML. Therefore, the branch name '`<img/src='x'/onerror=alert(document.domain)>`' causes the website to try to load the specified image, resulting in the popup seen in Figure 2.6. Since no image from source 'x' can be loaded, the `onerror` event handler is triggered and creates an alert popup on the page.

2.1.4 Countermeasures

In the previous sections, we have disregarded defense measures against XSS attacks for simplicity. There are two critical stages in web systems where one can take action and implement defenses to prevent XSS attacks. The first critical moment, when countermeasures become effective, is when user data is fed into the system. Potential malicious user input can be detected and blocked from being stored or returned to users. The second instance where user-controllable data can be sanitized is when data is output from the web application immediately before the data is delivered to users. Before responding to requests, precautions should be taken to avoid harmful data being delivered to users.

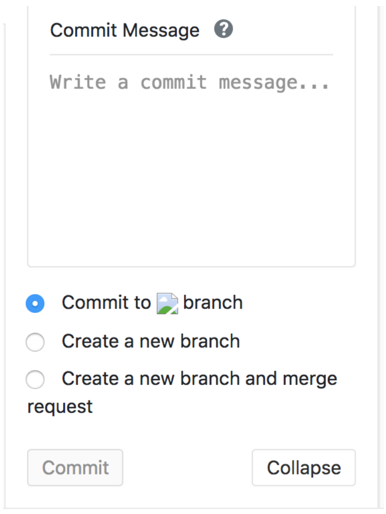


Figure 2.6: CVE-2020-15562 commit-popup [Fra18].

Input Validation

Input validation describes the process of ensuring that user data matches the intended domain of legitimate input. There are generally two approaches when it comes to input validation, namely blacklisting and whitelisting. The blacklisting approach addresses input that is explicitly forbidden, allowing input that is known to be harmful to be rejected. Whitelisting, on the other hand, only allows input that is explicitly permitted. Input that does not satisfy predefined constraints gets rejected. Typical tasks for whitelisting are data typing, data size, range validation, and restricting character sets for inputs. Usually, both approaches, black- and whitelisting, make us of regular expressions to validate user data and are used in conjunction. Standard security solutions feature custom black-/whitelists responsible for blocking potentially dangerous HTML tags, attributes, and JavaScript expressions. In general, whitelist validation is to be preferred since it has less severe consequences to unintentionally prohibit a legitimate use case than accidentally missing an exploitable use case. By no means does that imply that the whitelisting approach is foolproof; whitelisting restricts the attacker more if done correctly.

Input/Output Sanitization

Harmful input should be rejected via input validation, but in order not to prohibit intended use cases, there must be another line of defense that protects against attacks using arguably legitimate input. Therefore, input sanitization describes the process of filtering and encoding malicious input from incoming user data so that it is represented safely. Predefined rules, possibly implemented as regular expressions, determine when data is to be encoded. Encoding, especially escaping, must be applied context-aware, exemplified values inside HTML entities must be HTML encoded, while data inside JavaScript strings should be Unicode-escaped. Sanitization is usually implemented for both directions, incoming and outgoing data flows. User-controllable data should always be encoded directly before being delivered to the user to prevent XSS. When malicious data is adequately sanitized, the receiving browser will not confuse it with original source code and therefore not execute it. Similar to data validation, sanitization is no trivial task either and can become very complex. A single disregarded edge case can be devastating. For instance, the escaping of double quotes (") with a backslash (\) in an HTML context can be avoided by supplying a leading backslash (\"). The inserted escaping backslash will be escaped by the attacker-supplied backslash (\\") and cancel out the countermeasure.

Data-flow Analysis

Data-flow analysis is the static code analysis of an application, which examines between which parts of a program data is passed to figure out which dependencies result from it. Data-flow analysis primarily uses control flow graphs to observe how the data flows through the application and changes at different nodes. In terms of security, developers use data-flow analysis to trace where protective measures are required. Typically, the source code is examined for data-flow analysis, but limited insight into the data flow can be obtained even without source code. For example, pentesters can feed an application with data and see whether and where the data is output again. Security researchers can use techniques alike to target specific input vectors, i.e., for fuzzing.

Content security policy (CSP)

CSP is a security concept to mitigate the harm of injection vulnerabilities in web pages and was initially designed by the Mozilla Foundation. The way CSP works is that developers define policies for resource types or scopes that explicitly restrict the contexts in which scripts are allowed to be executed and from where content may be applied. For not explicitly specified contexts, a default policy should be defined. To enable CSP developers must configure the website to return the `Content-Security-Policy` HTTP header. For example, to restrict a site's content to its origin, one can specify a default policy and set the content attribute to `self` like so: `Content-Security-Policy: default-src 'self'.` In this example, we exclude subdomains. The goal of CSP is to provide developers with the control over which contexts allow for code execution, privilege control, attack reporting mechanisms, and mitigate the risk of injection attacks. CSP in no way eliminates the need for input validation, and sanitization, CSP is more of a second defense line to restrict exploitation should a vulnerability arise.

Browser Defenses

The last possibility to stop XSS attacks is in the web browser itself. An attack is only successful if the browser executes the payload and does not prevent it by, for example, script blockers. Browser developers are aware of the dangers of insecure web clients and implement various security mechanisms to protect their users. Among the most important security standards implemented in browsers are *Same-Origin-Policy (SOP)*, CSP, *Cross-Origin Resource Sharing (CORS)*, cookie security according to RFC 6265 "HTTP state management mechanism"[RFC626511] and measures concerning request or response headers. The SOP is a security concept that prohibits client-side scripting languages such as JavaScript and Cascading Style Sheets from accessing objects that originate from another web page whose location does not correspond to origin. SOP is the resource restriction mechanism that applies in cases where CSP is not enabled. CSP can be seen as a more dynamic implementation of SOP but is only effective if the browser supports it. CORS is an extension to the XMLHttpRequest API enabling a website to communicate cross-origin. That means that client-side JavaScript can request objects from a URL on a different domain conflicting with the SOP. Therefore, it is especially important that CORS is implemented securely [Ann21]. To make cross-origin requests more secure CORS ensures that the receiving server must explicitly allow the request. Further, browsers employ even more security measures, likewise mentioned above, and are subject to excessive security checks. For comparison, the Google Chrome web browser is continuously fuzzed with 15,000 cores, and Microsoft fuzzes Microsoft Edge during product development with the computing power of 670 machine-years [Eri17].

Penetration Testing

Penetration tests differ from the protective measures mentioned so far in that the focus is not on avoiding security vulnerabilities but on finding them. A penetration test is characterized by the idea that a person or a team attempts to attack the system under test within the specified scope and uses the knowledge gained to improve security. One form of penetration testing that is gaining popularity is publicly advertised bug bounty programs that reward security researchers for responsibly disclosing vulnerabilities they find. From a developer's point of view, the security of web applications is a cost-benefit estimate. The highest possible security level is desirable but is tied to the available resources. For this reason, bug bounty programs are a profitable model since security-relevant parts of the development can be outsourced, and payment is only made if a security vulnerability is actually discovered. Many organizations publicize the vulnerabilities found in bug bounty programs after being fixed, which serves as a guide for further penetration testing and shows how effective bug bounty programs are.

2.1.5 Exploitability

The traditional POC for an XSS vulnerability is to trigger a popup using the `alert()` function. Besides the confirmation that arbitrary JavaScript can be executed, the alert popup has another advantage, called with `document.domain`, the popup displays the domain on which the XSS occurs. This information can be helpful since not every XSS vulnerability occurs on the same domain as the input is supplied to (see Section 2.1.1). When determining the severity of XSS vulnerabilities, two questions must be considered. First, the question of in which context the exploitation of the vulnerability is possible, e.g., whether user interaction is required or limited by other factors. Secondly, one must ask the question of which actions the attacker can perform by exploiting the vulnerability. Usually, other aspects like attack complexity are considered as well, a score between 0 and 10 is calculated according to the *Common Vulnerability Scoring System (CVSS)*, and vulnerabilities are classified into five severity categories, none, low, medium, high, and critical [CVSS]. For this chapter, we focus on the exploitability of XSS vulnerabilities and their limitations. The most common applications of XSS are actions that lead to or equate to an account takeover. XSS exploits are often used to steal passwords and session cookies, which can be as easy as making a `GET` request to an attacker-controlled domain and sending the `document.cookie` value. Limitations of such attacks are, for example, that the victim must be logged into their account and server-side precautions like binding sessions to IP addresses. Further, some XSS exploits require user interaction, like CVE-2020-9447 described in Section 2.1.3 or clicking a link in case of reflected XSS. XSS exploits that heavily depend on social engineering are called Self-XSS vulnerabilities. Self-XSS might require the victim to execute malicious code or exposing sensitive information to the attacker. In many scenarios XSS exploits are used to enable other attacks like phishing. In case a website is secured against *cross-site request forgery (CSRF)* attacks with CSRF tokens, an attacker can use a XSS to access the tokens. For phishing attacks, XSS is used to redirect a victim to an attacker-controlled clone of the website to steal login information. Since modern browsers prohibit websites from accessing files, *remote code execution (RCE)* would require a 0-day exploit for the browser.

2.2 Related Work

Since the 1990s XSS has been a subject in web application security [PC 11]. Developers and security researchers have been researching in the area of identification and protection for years. This section covers the research, which forms the foundation for this thesis.

2.2.1 XSS Detection

Application development best practices are in place to minimize the risk of vulnerabilities, but dangerous bugs still happen. Therefore, it is necessary to test both during and after development to identify and fix potential vulnerabilities. The following section discusses relevant research that has been done in the area of XSS vulnerability scanning.

Web Security Testing Guide

In the fourth version of the Web Security Testing Guide [Eli20] published by the OWASP Foundation, Elie Saad, Matteo Meucci, and Rick Mitchell address the issue of testing for XSS. The Web Security Testing Guide is part of the OWASP Testing Project "[. . .] to help people understand the what, why, when, where, and how of testing web applications" [Eli20]. Accordingly, chapter 4.7 covers the testing of input validation, including reflected (section 4.7.1) and stored (section 4.7.2) XSS. The guide introduces the different XSS variants and provides a structured black-box testing procedure. For reflected XSS the suggested procedure is to detect input vectors, analyze them and check the impact. The authors provide examples for vulnerabilities and XSS filter bypasses. Stored XSS has a similar approach, the steps are identify all input possibilities (HTML forms, file upload, etc.) and analyze the relevant HTML code including dynamic testing. Further, the guide recommends multiple tools to support web application testing. Regarding gray- and white-box testing, the guide recommends the same approach for black-box testing using static analysis. The Web Security Testing Guide is also a great resource regarding many other web application vulnerabilities, which can be helpful in case different attacks must be combined to exploit a weakness.

LigRE: Reverse-Engineering of Control and DataFlow Models for Black-Box XSS Detection & KameleonFuzz:Evolutionary Fuzzing for Black-Box XSS Detection

Duchene et al. (2013) have developed a black-box detection tool for XSS vulnerabilities named LigRE [DRRG13]. The tool operates in 2 stages. It reverse-engineers the web application in terms of control flow inference and data flow annotation to create a control flow model. In the second stage, the fuzzing utility utilizes the generated control flow model for constrained fuzzing. Using the control flow model, LigRE fuzzes potentially vulnerable components of the web application with malicious payloads. In the follow-up paper, Duchene et al. (2014) enhance the LigRE workflow by creating KameleonFuzz, an evolutionary fuzzing tool [Fab14]. KameleonFuzz implements a generic fuzzing approach. It takes an input sequence generated by LigRE and creates more input sequences according to a custom grammar. The evaluation suggests that the combination LigRE and KameleonFuzz detect more true unique XSS vulnerabilities than popular competitors like w3af or Skipfish.

A Web Second-Order Vulnerabilities Detection Method

Liu et al. (2018) presented a method for second-order web vulnerability detection to improve the detection rate of classical black-box testing [LW18]. The proposed approach can be separated into 3 phases. In the first iteration, the algorithm crawls the web application to identify input vectors and inserts unique identifiable strings. In the next phase, the algorithm crawls the website for the second time to find all reflections of the supplied inputs. The last phase is to fuzz the web application efficiently since phases 1 and 2 reveal which input vector is responsible for each output vector.

25 Million Flows Later - Large-scale Detection of DOM-based XSS

In their paper, Lekies et al. present an approach for large-scale automatic DOM-XSS detection [LSJ13]. For automatic DOM-XSS detection, Lekies et al. enhanced the open-source Chromium web browser with taint-tracking capabilities. The modifications include extensions to known XSS sinks like `eval` to report to the user when user data is entered into a sink. To automate the vulnerability verification process, Lekies et al. implemented an exploit generation framework that generates XSS context-specific payloads and fuzzes the application accordingly. Besides the 6167 unique vulnerabilities discovered across 480 web applications, the main contribution of this paper is the automated DOM-based XSS detection integration into web browsers.

Cross-Site Scripting Detection Based on an Enhanced Genetic Algorithm

Hydara et al. (2015) present another generic fuzzing algorithm to detect XSS vulnerabilities in PHP web applications [HMZA15]. The presented approach can be broken down into two components, a utility that creates a control flow graph of the application under test and the second component being a generic fuzzer. In this paper, the authors only consider white-box testing. The first component analyzes the source code to create the control flow graph. After that, the generic fuzzing algorithm generates XSS payloads involving the steps shown below (see [HMZA15]) and then fuzzes the web application under test.

- Step 1: Create an initial population of candidate solutions
- Step 2: Compute the fitness value of each candidate
- Step 3: Select all the candidates that have their fitness values above or on a threshold
- Step 4: Make changes to each of the selected candidates using genetic operators, e.g., crossover and mutation
- Step 5: Repeat from step 2 until a solution is reached or an exit criterion is met

Hydara et al. found actual XSS vulnerabilities with a mixed false positive rate in the selected test applications through automated generic fuzzing.

2.2.2 Mitigation

XSS is one of the most widespread attacks on web applications, and until today no approach solves all loopholes in practice. For this reason there are many guides like “Client-side form validation” [Mozc] and “Validating Input” [Wor] with best practices to avoid XSS vulnerabilities. The OWASP Foundation released in 2010 the OWASP Secure Coding Practices Quick Reference Guide, which contains a checklist to review for potential security risks [The10]. Additionally the OWASP Foundation provides an “Input Validation Cheat Sheet” which covers the topic of input validation [The]. The German Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik) published a catalog of measures and best practices for the prevention of typical vulnerabilities in web applications, including a section (2.1 M100-2.4 M130) relevant for XSS protection [Sec06]. In the year 2000, the Software Engineering Institute of Carnegie Mellon University published a collection of advisories for web developers addressing “Malicious HTML Tags Embedded in Client Web Requests” in chapter 2 (CA-2000-02) [Car00]. Best practices include whitelist filtering, which should be preferred to blacklist filtering, manipulated input should be rejected, and error messages to users should not contain any information about their cause. In the preceding subsections, we present related work regarding XSS mitigation.

Content Security Policy Level 3

On October 15, 2018, the Web Application Security Working Group of the *World Wide Web Consortium (W3C)* released a working draft of the CSP specification level 3. CSP is “[. . .] a tool which developers can use to lock down their applications in various ways, mitigating the risk of content injection vulnerabilities such as cross-site scripting and reducing the privilege with which their applications execute.” [Wor18]. By restricting which resources an attacker can fetch or execute by exploiting an XSS vulnerability, the magnitude of possible damage can be degraded. The CSP is “[. . .] intended to become a W3C Recommendation [.]” [Wor18]. It and also mitigates damage of attacks other than XSS like dangling markup attacks or clickjacking [Pora].

Browser’s Defenses Against Reflected Cross-site Scripting Attacks

In their paper B. Mewara et al. examine 3 of the most widely used web browsers for security with respect to XSS attacks [MBG14]. First of all, the authors inspect the Internet Explorer XSS filter and address its limitations like content injection directly into JavaScript and allowance for some dangerous HTTP headers. The second testee is the XSS Auditor for Chrome, which has been removed in release 78 due to widely known bypasses [Chr19]. Further, the authors looked at XSS-Me for Firefox and concluded that neither of the examined web browsers could defend against all possible exploits.

3 Approach

This chapter covers the core of this thesis, our approach to scan for XSS vulnerabilities in ITSs. We start with an overview of a general penetration testing approach of web applications and then focus on the additional input interfaces that ITSs bring with them. We discuss which approaches we considered and which we decided on, then we provide an insight into how we implemented the approach. A legal permit is a mandatory requirement for penetration testing. Since we are only concerned with the technical aspects, we do not go into any detail on legal aspects.

3.1 General Method

Because of the amount of work that has gone into penetration testing, there is a correspondingly large amount of guidance from experts on how to go about vulnerability scanning. In this section, we represent the consensus of experts in the field, inspired in part by the earlier mentioned Web Application Security Testing Guide from the OWASP Foundation [Eli20]. Since this thesis has a particular focus on XSS, we will only cover XSS relevant aspects.

3.1.1 Information Gathering

To perform an effective vulnerability scan, it helps to have some understanding of the system under test. An excellent first step is to familiarize oneself with the application while creating a sitemap. Enumerating the web application and identifying possible attack vectors, be it automated or by hand, is a necessity that must be undertaken to conduct a methodical vulnerability scan. Another helpful point of reference is to look at publicly disclosed vulnerabilities that have already been found and fixed in the system under test. The work of other pentesters on the same system can serve as an indicator of which components to focus on during the search. Sometimes security fixes introduce new weaknesses, and other times, fixes reflect which components have already been tested thoroughly, suggesting the idea that other components are less secure. Within the same phase, it is essential to get an overview of how the application works. In order to test the web application in-depth, the roles of the identified attack vectors must be understood. A complete understanding of large software projects is in the scope of a penetration test unrealistic, especially for black-box tests, so the pentester should create a model of the relevant parts of the application in his head and amend it as the test progresses. For the purpose of XSS vulnerabilities, it is advantageous to understand the data flow of user input within the application. In cases where the application source code is available, the necessary knowledge can be obtained with static analysis. For black-box testing, the results of automated security assessment tools like within Burp Suite or the *Zed Attack Proxy (ZAP)* are adjuvant. Once this initial recon phase has been completed, one should be well equipped to search specifically for vulnerabilities.

3.1.2 Probe Attack Vectors

With enough information gathered, it is time to focus on the actual testing. In this phase, the attack surface is systematically processed in order to test as many parts of the application as possible for as many types of security vulnerabilities as possible. Different types of security vulnerabilities typically require a different manner of testing. Not only do the payloads differ, but also the components of the web application to be attacked vary among vulnerability types. Regarding XSS vulnerabilities, we recommend an iterative approach in which the input interfaces are processed one by one. Reflected XSS can be tested for comparatively well in an automated way. A test like the one described in Section 2.1.1 works fine to find vulnerabilities or identify input fields worth manual testing. Testing for stored XSS generally requires more effort but can also be supported with tools. In both cases, we are interested in reflected input data which we then gradually analyze. Fundamental questions to be answered are: What kind of input/output validation or sanitization is implemented and where? What are the possibilities to bypass the input/output validation? Which characters are available, i.e., are not encoded? With these questions in mind, we take educated guesses and try to exploit the potential XSS weakness, often progressing with trial and error. Within this procedure lies a decision to be made, deciding when to stop testing a particular entry point and when to focus on another attack vector. In the case that a vulnerability is found, the same input option could contain a second vulnerability. If no vulnerability is found during testing, one must come to an end at some point. Here, the pentester must weigh up whether they should invest further resources. From our perspective, testing should continue until the pentester considers the chance of a security vulnerability to be too low. That can happen quite quickly, such as when a widely used library is used for escaping or only when the pentester has applied its complete repertoire of skills to the potential vulnerability. That is not to say that widely used libraries should not be scrutinized. Often software that is considered secure is misused, but the time invested should be distributed as intelligently as possible, i.e., attack vectors with high vulnerability certainty first.

3.2 Unconventional Input Interfaces

One requirement of a qualitative security assessment test is completeness. If specific input options are not checked for security, this is a risk. One inconvenience regarding unconventional input interfaces is that standard security assessment tools can only make inputs that can also be made via web interfaces. While the tools can inspect changes done using different input interfaces, they cannot provide the input. This thesis aims to design a methodology that allows security assessment tools to make input via unconventional input interfaces. In the following, we will discuss the approach we have chosen and how we implemented it.

3.2.1 Approaches

Before choosing a particular approach for testing ITS-specific input interfaces for XSS vulnerabilities, we need to determine what requirements we demand from such an approach. The most apparent requirement with the highest priority is functionality. The approach should be easily implementable and assist pentesters. Also, we have two further requirements for our strategy. First, it should be generally applicable. The approach should be transferable to all relevant input interfaces with the

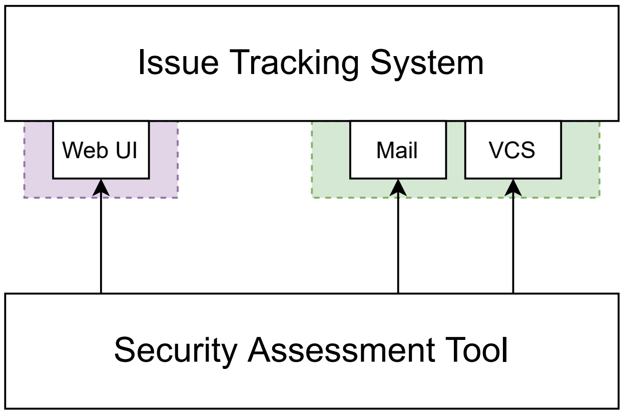


Figure 3.1: Visualization of our ITS testing approach. The preexisting user interface is indicated by the purple color. The interfaces we plan to create are indicated by the green color.

least possible effort. Second, we do not want to reinvent the wheel; tools that have already been developed should not have to be rewritten for ITS-specific input interfaces. Functionality related to ITS-specific input interfaces should be integrated into the already existing security assessment tools.

Given the above requirements, we considered two main approaches. The first approach is to extend already existing tools with the desired functionality, e.g., through plugins. The second approach is to implement web interfaces for the corresponding input interfaces. The idea behind the first approach is to design a test concept for relevant input interfaces, which then has to be integrated into the desired security assessment tools. However, this approach has serious disadvantages. Every tool that is to be used must be modified. The know-how for extending the tools is required, which can become quite complex when no extensibility API is provided. Even if it is realistic to develop and maintain plugins for popular security assessment tools, a large proportion of the available tools are left out. In addition, we would have to rely on an API for proprietary tools. The second approach overcomes the disadvantages of the first one. Since the security assessment tools are designed for testing web applications, the web interfaces we plan to develop can likewise be tested without problems. The tools do not require to be extended. Furthermore, the web interfaces can easily be adapted for different ITSs. In the case of Git as an input interface, one web interface should be configurable for different ITSs without changes in its source code. Another benefit is that no knowledge of how the security assessment tools work internally is necessary. It is just a matter of creating a pure interface. However, one hurdle in designing the web interfaces is not to limit the attack surface. While the interface itself should not contain security holes, malicious input should be passed through without modification. Also, no input possibility should be neglected. After comparing the two approaches and their benefits, we decided to use the web interface approach.

Figure 3.1 shows our vision of how testing works with our method. At the bottom of the figure, a security assessment tool is drawn as a representative of any tool. This tool tests the ITS via the regular web interface (left in the figure) as usual in penetration tests. On the right-hand side, two examples of web interfaces we plan to create can be seen. As the arrows indicate, the security assessment tool also uses the additional web interfaces for email and VCSs.

3.3 Implementation

This chapter explains our web interface implementation for ITS-specific input interfaces as explained above. Since this thesis focuses mainly on strategy and is not intended to be a complete penetration test, we do not implement a web interface for every input interface, but only for a selection that we consider most interesting. Web interfaces for input interfaces that are not covered can be developed analogously. The approach should be universally applicable. Of interest to us are interfaces that are often used for issue tracking and are unusual for conventional web applications. First of all, we look at email as an interface and which particularities have to be considered. Next, we will create a web interface for Git. Repository hosting is a common feature of popular ITSs, and Git is probably the most common VCS. The basic structure for our web interfaces is always the same. As a programming language, we have chosen Python. We use the web framework Flask [Flask] and the templating language Jinja [Jinja] to set up a web server. We provide the functionality of the web interfaces to the security assessment tools using the python modules WTForms [WTForms], and Flask WTF [FlaskWTF]. Each web interface should contain one or more forms through which the input interface-specific options are set and submitted. That allows security assessment tools to supply input via `GET` or `POST` requests to, for example, a Git repository. Due to the different input interfaces of different ITSs, we design the interfaces individually. We can also imagine one interface for all input interfaces, where input options can be enabled and disabled, but since this thesis is mainly intended to serve as a POC, we chose the simple approach. The fact that we test the input interfaces iteratively also makes a combined web interface unnecessary. Regarding the attack variants, we assume an attacker who has write permission for test projects, i.e., the attacker can create and modify issues. Furthermore, we design interfaces that can be configured for individual projects in ITSs. In case the pentester has access to an instance of the ITS that is used exclusively for their tests, they can also use web interfaces that dynamically test in all hosted projects, e.g., with access to the ITS database. However, this approach only makes sense if a dedicated test environment is available. Otherwise, genuine hosted projects would be attacked and flooded with test payloads. For this reason, our web interfaces are configurable to provide access to predefined projects only. In the following, we describe our implementation of the web interfaces in more detail.

3.3.1 Mail Interface

The principle of our mail interface is simple. We use an HTML form to specify sender, recipient, subject, and other options. When the form is submitted, the interface sends a corresponding email. First, we define which input fields the form of our web interface contains. There are four main input vectors, the sender, the recipient, the subject, and the body that contains the actual message. Typically, the pentester only needs a sender email address, so it is not mandatory to set this in the form. However, the pentester may want to use different email accounts, e.g., to test accounts with different permissions. For this reason, we have decided to include the sender in our form. Since we only want to allow specific (previously configured) senders, we use a drop-down list (selectable field) for the sender with the email addresses as choices. We use drop-down lists every time we want to limit the input options for a field. Generally, we do not want to limit input, but there are input fields where a restriction makes sense, e.g., the sender/recipient email addresses should not be modifiable. Otherwise, no email will be sent/received, and the attack will not reach the ITS. In addition to the sender address, our interface needs other email settings to send emails. For this, we use a Python dictionary, which contains any number of email settings and can be passed to the interface. This way, we can configure the email interface easily, e.g., load settings from a JSON file. The dictionary is constructed as follows. It contains the email addresses as keys, and for each key, another dictionary as value. The inner dictionary contains the keys `imap_server`, `imap_port` and `password`, additional settings like SSL can be set if needed. An example of how such a configuration with one email looks can be seen in Listing 3.1. The reason why we use the IMAP login is simple. Email providers limit how many emails a user can send via SMTP within a certain period. However, if we want to automate our tests with tools, this limit is reached quickly and hinders our workflow. We use the IMAP protocol to place the generated emails directly in the mailbox to work around this problem. This way, we are independent of the email sending limit. Typically, ITSs have an email address to which emails are sent to create/modify issues. Therefore, we can set the recipient email directly without an input field or, as in our case, via a drop-down list with the recipient email addresses as selection options.

```
{
    "sender@gmail.com": {
        "imap_server": "imap.gmail.com",
        "imap_port": 993,
        "password": "secure_password"
    }
}
```

Listing 3.1: Example email configuration for the mail interface as Python dictionary.

Next, we look at the subject. We can not generally define whether the subject should be freely choosable or restricted. Some implementations allow arbitrary subjects for issue creation, while in order to edit an issue, the email subject must contain an issue identifier. Therefore we want to give the pentester the possibility to define how far the email subject is restricted. For this we use another Python dictionary, which contains `type`, `name` and optional `choices` as key. `type` specifies whether the field should be a normal text field (`text`) or a drop-down list (`select`), the `name` key holds the name of the field and in case of `select` the key `choices` must return a list with the options to be selected. From this dictionary, our implementation dynamically creates an HTML form. Configuration examples can be seen in Figure 3.2.

Finally, we need to implement the email body field. The email body is mainly a text field that contains arbitrary text messages. However, some ITSs implement a mail handler that looks for key-value pairs in the body to set issue attributes like priority or status. Therefore we need an interface configurable for different key-value pair formats. Our solution to this is to create the body as a text field and to implement the issue attributes in the same way as the subject, i.e., a list of python dictionaries with the keys `type`, `name` and `choices`. That allows us to specify arbitrary input fields that will be appended to the email body and can be restricted as needed. The last missing feature is the possibility to add attachments to the email. We realize email attachments via a standard upload form that web security assessment tools support. In this manner, we have implemented an email web interface that is configurable for different ITSs and can be tested using standard security assessment tools for web applications.

```
{
    "type": "text",
    "name": "subject"
}
```

Listing 3.2: Example: Subject field configuration for arbitrary string input.

```
{
    "type": "select",
    "name": "subject",
    "choices": ["first subject", "second subject", "third subject"]
}
```

Listing 3.3: Example: Subject field configuration for restricted input.

Figure 3.2: Email subject example configurations.

3.3.2 Git Interface

For the Git interface, we must again identify all input vectors that in some way are represented in ITSs web user interfaces. To find out what input vectors exist, we inspected several ITSs, including our three testees. We observed that the Git objects Branch, Tag, and Commit, are used to represent Git repositories in ITSs. For the branches, only the names are relevant input vectors. A Git commit has a hash, a message, a date, and authorship information, although we cannot specify the commit hash ourselves. The authorship information consists of the author and the committer for which name and email are stored, respectively. Furthermore, a commit contains files and file changes, the diff. Standard file formats are usually represented in the ITS web interface in human-readable form. For a Git tag, the relevant inputs are the annotation and the tag message. We, therefore, need to implement functionality that creates branches, commits, and tags. Additionally, we enable our web interface to push changes to a remote in order to enable testing without local access to the relevant repositories. A push is performed after each branch or commit action.

Next, we need to enable the Git interface to execute the commands `git branch`, `git commit` and `git tag` for a Git repository. We decided to create a Git executable which the Git interface calls. A more elegant solution would be to use a native library like `libgit2`, but since we are making changes in the Git source code to get around input restrictions, we create a custom Git executable for simplicity. The restrictions mentioned are that Git only accepts certain characters in authorship information or branch names. For example, authorship information in commits cannot contain the characters `;:,.<>"'\`, which severely limits input related to XSS exploits. We explain the changes made to the Git source code in more detail in Appendix A. With the custom Git version, authorship information can include the disallowed characters. One remaining limitation is caused by the format that Git uses to output authorship information. Git outputs authorship information name first, and then comes the email address. The email address is delimited in between the characters < and >. With our custom Git executable, it is possible to enter arbitrary names/email addresses, but the Git implementation used by the ITS will likely truncate every character after >. Additionally, the outermost < and > characters are normally not displayed in ITS web interfaces. Nevertheless, our changes are an improvement to the official Git implementation (in terms of XSS testing), as more characters are available for input and we are able to input strings like `<img src=""onerror=()"`. Since browsers try to make sense of incorrect HTML syntax, the closing > character is not strictly necessary. We get a similar problem if we want to remove the branch name or tag annotations restriction. Our custom Git version allows us to create illegal branch names or tags, but whether the disallowed characters will be output from an ITS depends on the server-side Git implementation.

Our web interface for Git works as follows. We implement a form for creating new branches, a form for creating new commits, and a form for creating tags. In the forms, the customizable options can be set, and when submitting, the corresponding action is executed in a local Git repository and pushed afterward. In terms of configurability, the Git interface is more straightforward to implement than the mail interface. Since we intend the Git interface to have the repository under test available locally, drop-down lists such as branch name for the commit can be populated easily. Initially, only the local path to the repository and login credentials for the remote must be configured. All other information can be obtained from the repository. In case our interface operates on the same repository as the ITS, and no pushes are needed, we can configure that as well. The branch form has only one text field for the new branch name. The commit and tag forms have text fields for authorship information, such as the name and email address of the author and committer, respectively. Further, both forms include a message field where the user can enter an arbitrary string and a drop-down list for existing branches. Besides, the commit form has an upload form for adding files. Our Git interface works with bare as well as regular repositories. If necessary, we can update or reset the repository manually. When modifying the test repository manually, one has to keep in mind that the interface must be able to push to the specified remote. Merge conflicts must then be resolved manually as well.

3.3.3 Drawbacks

This section addresses the obstacles our implementation faces. We already found a partial solution to the aforementioned disallowed character problem regarding our Git interface, but whether the disallowed characters are output from ITSs depends on the used server-side Git implementation.

Another obstacle is that ITSs usually implement issue creation via email periodically. Commonly a cron task is created that periodically fetches emails and creates or edits issues accordingly. The problem with this way of creating issues is that security assessment tools cannot immediately detect the changes triggered by the email. In the worst case, the email is not processed until a whole time interval after the request. How this waiting time affects automatic tests remains to be seen. In test scenarios in which the pentester has control over the machine on which the ITS is running, additional precautions can be taken to reduce the impact of such delays. The first mitigation we added is to make the interval at which emails are fetched small so that the worst-case delay is reduced. The second idea we have implemented is that the interface only responds when the email has been parsed. This step must be implemented individually for each ITS. Without machine access to the test ITS, this mechanism could be implemented so that the interface crawls the ITS and only sends a response after the specified issue has been created/edited.

Next, we address a problem related to our approach itself. That is, some security assessment tools are not designed to scan multiple domains as a whole. The interfaces we create for testing are not integrated into the ITS under test. If we do not have a dedicated test system, the ITS is only accessible over the internet, while we plan to host our interfaces locally. The resulting problem is that some security assessment tools cannot scan multiple targets together, which implies that the ITS website and our interfaces are scanned one after another. However, to scan correctly, both the ITS and the interfaces must be examined simultaneously. Otherwise, no changes made through the interface can be detected. To work around this problem, we use a reverse proxy in our test setup. The reverse proxy makes the ITS and any number of interfaces accessible on the same port and domain. With this setup, we need to specify only one target. Assuming the reverse proxy is accessible at `http://localhost:80/`, we forward the location `http://localhost:80/` to the ITS and `http://localhost:80/git_interface` to our Git interface. Then we can configure the security assessment tool to scan `http://localhost:80/`. Further, we have to provide the tool with the link to the interface manually. No web crawler would find the interface otherwise since the ITS does not reference it. We also have to make sure that `http://localhost:80/git_interface` does not collide with a URL of the ITS.

4 Results

In the results chapter, we look at the results of our tests. In the following, we give an overview of the tools we used, explain our test setup, and discuss our findings. In the findings section, we discuss how the interfaces we created performed in the tests, and we address a few security-related aspects of the ITSs. Since we focus on the designed method in this thesis, we concentrate in the findings section on the applicability of additional web interfaces for security assessment tests.

4.1 Test Environment

The test environment section is divided into two subsections, the tools section, which deals with the security assessment tools we use, and the system under test section, where we explain how we set up the three ITSs for testing. The test setup includes setting up the ITSs as well as configuring our interfaces.

4.1.1 Tools

In this section, we cover the relevant tools for this thesis. The selection of penetration testing tools for web applications is large and offers various alternatives to the software mentioned here [Off; OWA]. For our purposes, we primarily need tools that, in the best case, autonomously scan websites for vulnerabilities and report the results. That way, we can evaluate whether the security assessment tools can operate in combination with our web interfaces.

Burp Suite

Burp Suite (also called Burp) is a security assessment tool for testing web applications. Burp Suite is written in Java and developed by PortSwigger Security [Burp]. Burp comes in 2 versions: the feature-limited free community edition and a full version that can be purchased after a trial period (professional edition). We mainly use the free community edition for manual testing and use the premium features within the free trial period. Burp Suite includes a web proxy through which any browser traffic can be captured and manipulated. In addition, it comes with valuable utilities and a preconfigured web browser. One of the utilities that Burp offers is its automatic login feature. To test web applications that require the user to be logged in, one can configure login credentials which Burp then uses for authentication. Since the ITSs we test support user management with different access permissions, we use Burp's authentication feature to test with an unrestricted user account. Besides the main functionality, Burp provides within the professional edition an extensibility API which is used to develop various enhancing plugins. Other utilities of Burp are a collection of tools that implement features that have proven to be effective in finding security vulnerabilities in web applications. On their website, PortSwigger Security describe the built-in Burp tools [Burp]:

- Scanner - This is used to automatically scan websites for content and security vulnerabilities.
- Intruder - This allows you to perform customized automated attacks, to carry out all kinds of testing tasks.
- Repeater - This is used to manually modify and reissue individual HTTP requests over and over.
- Collaborator client - This is used to generate Burp Collaborator payloads and monitor for resulting out-of-band interactions.
- Clickbandit - This is used to generate clickjacking exploits against vulnerable applications.
- Sequencer - This is used to analyze the quality of randomness in an application's session tokens.
- Decoder - This lets you transform bits of application data using common encoding and decoding schemes.
- Comparer - This is used to perform a visual comparison of bits of application data to find interesting differences.

We chose Burp Suite as our primary attack tool based on its excellent reputation within the security research community and its valuable features. The 2020 Hacker Report shows that as many as 89% of bug bounty participants on HackerOne use Burp Suite [Hac].

OWASP ZAP

The OWASP ZAP is a free open source web application scanner written in Java and maintained by volunteers [ZAP]. Among other non-commercial tools, the ZAP ranks high, in part due to the respected reputation of the OWASP Foundation. From a functionality point of view, Burp and the ZAP are more or less the same. Both tools are based on a web proxy and allow browser traffic manipulation. In both cases, an integrated web browser and API are available, and the ZAP includes features that enable the same functionality the integrated Burp tools provide. In the end, the decision boils down to personal preference. Typically, the first step, when working with one of these tools, is to create a sitemap. The sitemap can be generated both by manual browsing and automated tools. Usually, a combination of automated and manual crawling is done. Once the sitemap is created, the pentester scans the web app for possible attack vectors and tries to exploit them. Our decision to use both Burp Suite and the ZAP is based on the idea that two pairs of eyes see more than one. Since either of the tools support automated scans, it is no effort to set up both and benefit if one detects a vulnerability the other one oversees. Both tools work out of the box, and the necessary features are easy to operate.

Web Application Attack and Audit Framework (w3af)

The *Web Application Attack and Audit Framework (w3af)* is yet another open-source web application security scanner. It is developed and maintained by Andres Riancho and sponsored by Holm Security [Andb]. The project was started in 2007 and is written in Python. The last w3af release was over five years ago, and due to outdated dependencies, the installation process became complicated on current versions of operating systems. Fortunately, the developers provide a docker container that runs w3af. Currently, the w3af docker image only provides interaction via the command line. For the graphical user interface, we would need to install w3af ourselves. Although the last release is so far back, the basic functionality of w3af is still used and has proven itself over the years. For our tests, we mainly use the scanning and fuzzing capabilities of w3af. For scanning, we can create profiles in w3af that define the behavior of the scan, e.g., we use the predefined `full_audit` profile, which checks among other things for SQL injections and XSS. To be able to configure scans more fine-grained, w3af offers the possibility to activate individual plugins. There are several built-in plugins, the most interesting for us being the XSS plugin. The XSS plugin tests a specified target, a URL, for XSS vulnerabilities, using generated payload mutations. Our decision to use w3af is based on the fact that respected security companies use w3af. For example, w3af is part of the Kali Linux tool assortment [Off]. Another reason to use w3af is the possibility to work in a docker container, which does not require any further setup.

Wfuzz

Wfuzz is an easy-to-use web application fuzzer written in Python [Xav]. Wfuzz can be installed via the `pip` package manager and used from the command line. According to its specification, Wfuzz is based on the simple concept of replacing every reference to the keyword FUZZ with predefined payloads. With Wfuzz HTTP `GET` and `POST` requests can be fuzzed, requiring only the URL, the parameters to be fuzzed, and a word list to be specified. In addition, the results can be easily filtered for HTTP response codes and similar criteria. It can also repeat saved requests exported, for example, from Burp. Wfuzz is a simple alternative to the Burp Intruder and the corresponding ZAP feature. We use Wfuzz because of its ability to generate many requests from the command line quickly.

4.1.2 Systems Under Test

In this section, we evaluate the practical use of our interfaces for XSS vulnerability scanning. For simplicity, we use docker to host each ITS and its dependencies, such as the database. This way, we can restore the system under test to its initial state at any time. Table 4.1 shows the tested ITS versions and docker images. To be up-to-date, we test the latest releases at the time this thesis is written. For the attack scenario, we chose an attacker who has write permission for issues and relevant Git repositories. Since we are primarily testing for XSS, limited user privileges are not of interest. The following setups require more than one docker container, one where the ITS instance runs inside and others hosting, for example, the database. All required docker images along with their setup are described in the respective Docker Hub pages. We provide an exemplary representation of a test setup in Figure 4.1. At the top of the figure, one can see the ITS, which we host in a docker container. Generally, it is best practice to run only one service per container, but

ITS	Version	Docker images
Redmine	4.1.1	sameersbn/redmine:4.1.1-8 [Samb] & redmine:4.1.1 [Redb]
MantisBT	2.24.4	okainov/mantisbt:2.24.4 [Ole]
Trac	1.4.2	mastermindg/trac-ubuntu:latest [Ken]

Table 4.1: Version overview of tested ITSs.

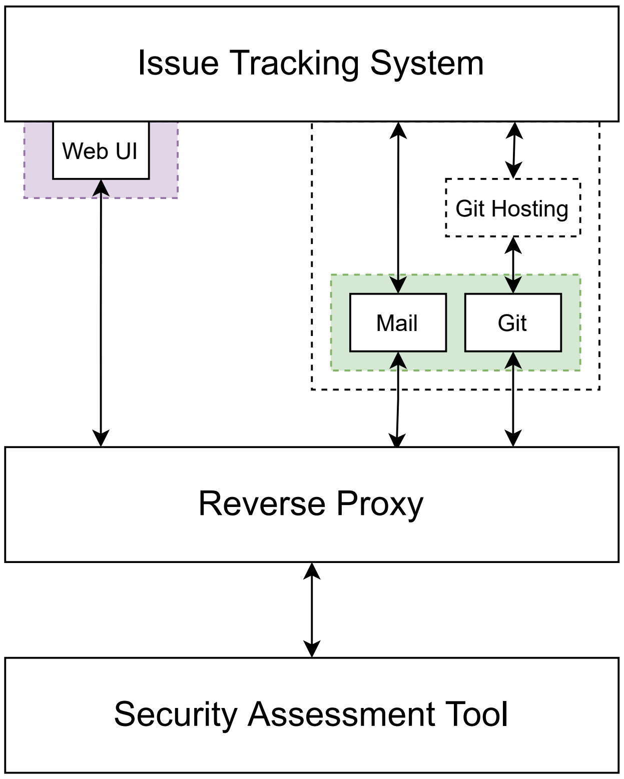


Figure 4.1: The general test setup. The preexisting user interface is indicated by the purple color. Our designed interfaces for mail and Git are indicated by the green color.

since our Git interface needs access to the repository of the ITS, in case we do not have an external Git hosting server, we decided to run the Git interface in the same container. For the case that we need an external Git hosting service, we run a dedicated service. The Git hosting service is located in Figure 4.1 between the ITS and the Git interface. The reverse proxy below the interfaces masks the interfaces as the same web application from the security assessment tool's perspective. At the bottom of the figure remains the security assessment tool that tests the ITS through the reverse proxy.

Reverse Proxy

As we mentioned in the implementation section, we make each ITS and its interfaces available through a reverse proxy. To do so, we use the official NGINX docker image [nginx]. NGINX is a widely-used open-source web server software developed by Igor Sysoev. Since NGINX was developed with the intention to provide a reverse proxy, we can provide access to all ITSs and the respective interfaces via `http://localhost`. We forward the root location (`/`) to the ITS so that relative URLs continue to work. Our mail interface gets the location `http://localhost/mail_interface` and our Git interface gets the location `http://localhost/git_interface`. Annoyingly, we still have to manually ensure that the security assessment tools find the additional interfaces. One way that works for most tools is to create and serve a `sitemap.xml` file that lists links to the interfaces. When a tool is capable of parsing sitemaps, we save manual configuration effort. In addition, we add hyperlinks to the interfaces so that the security assessment tools find the ITS when we specify an interface as the target.

4.1.3 Redmine

For Redmine, we use two test setups, one where we use the official Redmine docker image with source integration and one where we use GitLab to host Git repositories. In general, we recommend using official docker images for testing because, with official images, it can be assumed that the ITS is set up correctly and is up-to-date. However, we decided to use a non-official Redmine docker image as well because it already offers a multi-container docker application for external Git hosting [Sama; Samb]. For the first setup, we extend the official Redmine docker image to run our Git interface inside. For the second setup, we create a separate container for both the mail and the Git interface. With an external Git hosting service, neither interface needs access to the Redmine instance. The mail interface only communicates with the configured mail server, and the Git interface pushes changes only to the Git hosting server. The reverse proxy runs in both setups inside a separate container.

Mail

Here we are interested in the issue creation and editing functionality of Redmine, which can be done via email. The non-official docker image we use supports email reporting out-of-the-box and can be configured in the `docker-compose.yml` file. The docker image already has a cron job that periodically fetches incoming emails and creates or modifies issues accordingly. With the official Redmine docker image, we need to configure the cron job ourselves. In both cases, we set the fetch interval for emails to one minute. Additionally, we specify an email fetch command in the mail interface configuration that is executed after an email was sent. The fetch command simply connects to the Redmine service via ssh and executes Redmine's email fetch utility. With that out of the way, we can configure the web forms of our mail interface. The sender of the email is, in our case, the email address of the attacker, and the recipient is the configured Redmine admin account. For the subject, we have two configurations, one to create an issue and one to edit an issue. For issue creation, the subject is any string, and for editing, we specify the subject in the form "[<issue number>]" as a drop-down list. Redmine issues have preconfigured attributes, like a priority, assigned to, and start date. These issue attributes can be edited via email. To do so, one must specify the name and value for each attribute in the email body. Note that the attributes have types, and therefore the possible input values are attribute dependent. If the input does not comply with the input field's type, the corresponding string (attribute name: attribute value) is added to the description or comment in case of an edit. In order to be able to test as many inputs as possible, we create user-defined fields whose input values are restricted as little as possible. Custom fields can easily be added to our mail interface since we generate the submit form dynamically from a Python dictionary.

Git

We integrated the Git interface for the official Redmine docker image into the container with an extended version of the official `Dockerfile`. With our `Dockerfile`, a test repository is created and configured for the Git interface. Then a Redmine project with a Git repository must be created and linked to the repository inside the container.

For the second setup, the Git interface only needs a local copy of the repository with the configured GitLab remote. Since we host the repositories in a GitLab docker container, we need to create a repository in GitLab, link it in Redmine, and clone it for the Git interface. Note that with this setup, all payloads pass the GitLab instance first, which can act as an additional layer of defense.

4.1.4 MantisBT

In this section, we look at the test setup for MantisBT. MantisBT does not have an official docker image yet, so we use one from Docker Hub (see Table 4.1) and modify it for our purposes. We added needed plugins to the docker image, which can be installed from the MantisBT web interface.

Mail

MantisBT does not support email reporting out-of-the-box and hence requires the `EmailReporting` plugin [Mana]. Once installed, we can configure a mailbox from which incoming emails will be fetched. Mailboxes can be configured for one specific project. To fetch emails periodically, we need to add a cron job. Again we set the time interval to one minute. Additionally, we need to modify the anti-spam configuration of MantisBT. Per default, only ten email events within an hour are allowed. Since we plan to generate more than 10 issues from one email address per hour we set the `$g_antispam_max_event_count` variable in the configuration file (`config_inc.php`) to zero for no limit. The email fetch command for the interface works analogously to our Redmine setup. When a request is sent to the email interface, it connects to the MantisBT service via `ssh` and executes the MantisBT fetch utility. With the `EmailReporting` plugin, the following actions are possible, creating an issue, adding notes to an existing issue, and adding attachments. The issue priority is set to the priority in the email header. Here we do not tamper with the email header, but this could be implemented as well. Since we can not edit issue attributes besides the priority with emails, the web form of our email interface only contains sender, recipient, subject, message body, and an attachment file upload form. Again we need two configurations for the subject field, one where we use a text field to create new issues and one where we use a drop-down list to add notes to an existing issue. For the subject format to address specific issues, different configuration options are available. We use the default format "[<anything> <issue id>]" to identify an issue by its id, where <anything> is an non-empty string. As intended, our email interface works for MantisBT without modifications in the source code. Only the configurations differ compared to Redmine.

Git

MantisBT also offers plugins for VCS integration. The plugins we use are part of the Source Control Integration framework [Joh] that supports a variety of VCSs. First, the base plugin (Source Control Integration) must be installed to enable version controlling with other plugins. In the scope of this thesis, we restrict ourselves only to Git repositories regarding version control. For Git, the Source Control Integration framework offers the integration of GitLab repositories. Local repositories like in our Redmine setup are not supported. For simplicity, we use the same GitLab docker image as for Redmine. GitLab hosted repositories are integrated via their URL similar to Redmine, but additionally, a webhook must be created in GitLab that automatically updates changes. The configuration of our Git interface is relatively simple, it only needs the cloned GitLab repository.

4.1.5 Trac

Like MantisBT, Trac has no official docker image either. The Trac docker container we built is based on a non-official image, with changes that enable us to test the latest Trac version (1.4.2). We install Trac via the package manager `pip` and add plugins and configurations as well as our web interfaces. As of version 1.0, Trac supports the integration of Git repositories out-of-the-box, so we only need to create a repository in the docker container and link it to Trac. We also run our web interface for Git in the docker container and thus get a setup analogous to the one with the official Redmine docker image. For Trac, there is also a plugin called `email2trac` available to create issues via email, but this plugin only supports Trac versions lower than or equal to 1.2 [Dan]. For this reason and because we are limited in time in this thesis, we decided to test only Git as an input interface for Trac. Using an older Trac version in conjunction with the email plugin would be an option for future work.

4.2 Findings

Part of this thesis is to put our approach into practice, for this we test the three ITSs Redmine, MantisBT and Trac with the created interfaces for XSS vulnerabilities. Our findings follow, first we address the practical application of our interfaces and then elaborate on each ITS individually.

4.2.1 Interfaces

In this section, we evaluate the practical use of our interfaces for XSS vulnerability scanning. We later go into more detail about the usefulness of the two interfaces. In terms of usability, we can say that the interfaces work as intended. All used security assessment tools recognize the different text input fields, drop-down lists, and file upload forms as input vectors. Therefore, web interfaces for ITS-specific input interfaces provide the facility to perform automated and manual security scans. However, to perform a vulnerability scan with additional web interfaces, there are a few aspects to consider. Firstly, from our point of view, it is necessary to make the ITS and additional interfaces look like one web application to the security assessment tool, because otherwise tools like w3af scan the different targets individually [Anda]. Our approach, to hide the ITS and interfaces behind a reverse proxy, achieves the desired effect. In some setups, an additional modification of the DNS records is necessary so that the security assessment tool can also resolve absolute URLs that point to the ITS. Secondly, the scopes of the security assessment tools should be configured, if possible, so that the scanner does not search for vulnerabilities in the interfaces. An automated scan with default settings will inspect the drop-down lists in the interfaces. Typically, this behavior is desired, but concerning our interfaces, it means that many attacks lead nowhere, e.g., it is unnecessary to enter XSS payloads as recipients in the email interface. The consequence is that tools test things that definitely do not lead to a vulnerability, and reports from the scans have to be filtered or partially ignored. Because automated scans require hardly any manual effort and are allowed to deliver false positives, we conclude that this problem is negligible. In the following, we will discuss the strengths and limitations of each interface in more detail. Keep in mind that these implementations are only research prototypes intended to serve as a POC.

Mail Interface

The mail interface faces the obstacle that different ITSs and email plugins implement different key-value pair formats to create or edit issues. With the use of a configuration file, we can dynamically create a web form according to the required format. We had to set key-value pairs only in the mail body for our testing purposes, but customizing the interface to allow key-value pairs in the subject is feasible. The main disadvantage of testing with our mail interface is the delay between sending requests to the interface and the email processing in the ITS. The fact that emails are fetched only periodically causes an unavoidable worst-case delay. To reduce this delay, we have decreased the query interval for the test systems to one minute. However, automated scans make hundreds of requests to the mail interface in a few minutes so that even with a one-minute polling interval, the ITS cannot keep up with the parsing of the emails. This problem can be solved with more computing power, but we point out that the processing of emails is a bottleneck. Nevertheless, one-minute delayed emails did not have a severe impact on our vulnerability scans. Changes to data entered via input interfaces without an integrated web interface can only be detected by successive queries, which is usually not done immediately after the request in automated scans. Besides, the delay caused by hundreds of emails is not so significant as to be a deal-breaker; the delay only builds up with a continuous flood of emails. In general, however, this problem cannot be neglected, as it can lead to tools not detecting vulnerabilities that would otherwise be detected. For this reason, when using an email interface like ours, attention must be paid to the extent to which problems can occur and what can be done to counter them. Our approach, to configure an email fetch command for the mail interface, circumvents both the time delay and the bottleneck problem but must be implemented individually.

Git Interface

The Git web interface allows security assessment tools to make inputs through preconfigured Git repositories. Through a custom Git implementation, it is even possible to make inputs that Git itself does not allow. The Git interface can push changes to a remote, and therefore the ITS and the interface can be run on different machines. The only shortcoming of the interface is that, despite the custom Git implementation, some input fields are still restricted. In practical use, the Git interface behaved as intended. The security assessment tools identified and scanned all input fields. Automated scans performed commits and created tags as well as branches. The tools were able to add files of any file format to the repository through the upload form. Due to the unmodified server-side Git implementation, some of the supplied XSS payloads were neglected or only partially displayed in the respective ITS web user interface. For example, some branches and tags were not displayed due to “broken names”, or authorship information was truncated. It is worth mentioning that the unauthorized inputs in the respective Git repositories are also present server-side but are merely not output. To conclude, we consider a web interface for Git such as our implementation to be useful for penetration testing despite restricted input fields.

4.2.2 Issue Tracking Systems

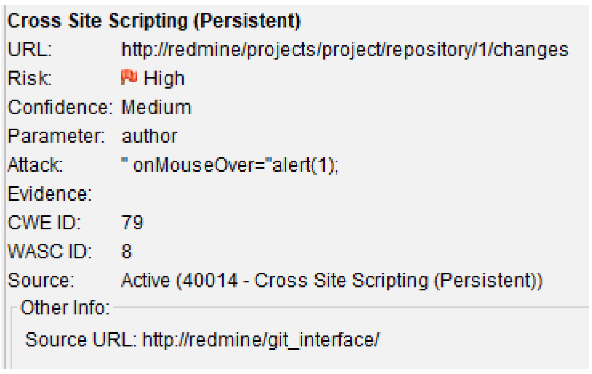
This section deals with the results of the security assessment scans of the three ITSs Redmine, MantisBT and Trac. Since this thesis focuses on the applicability of web interfaces for unconventional input interfaces in penetration tests, we only address the scan results briefly. The extent of three comprehensive penetration tests is beyond the scope of this thesis. In the following, we give an insight into the security-relevant aspects of the implementations and present a showcase tool warning for each ITS.

Redmine

We could not exploit Redmine through email or Git-related inputs based on the warnings generated by the security assessment tools. Redmine parses emails and stores the input unchanged in its database. We can figure this out either by looking at the source code or extracting the database’s data. Storing the input unchanged is nothing unusual since escaping data is context-dependent. Any preprocessing of data could corrupt the data. That means that escaping must happen when the data is output. Redmine uses *Embedded Ruby (ERB)* templates to dynamically generate HTML pages. ERB is a Ruby-based templating language that escapes unsafe data by default. During our tests, we received warnings about possible XSS vulnerabilities, but we could not disclose any vulnerability because the auto escaping functionality prevents us from breaking out of context. An example of a warning generated by ZAP can be seen in Figure 4.2a. The ZAP correctly detected that the author parameter from the `POST` request to create a commit with our Git interface is stored and returned to the user at another location. The warning classifies the potential vulnerability as high risk because the payload `"onMouseOver="alert(1);` is not escaped. The relevant HTML tag can be seen in Listing 4.1. However, this is no XSS vulnerability since the payload is intended to break out of an HTML tag attribute, while in this case, the author name is present as inner text inside a `<p>` tag. To break out of the original HTML context, a payload like `<script>alert()</script>` is required, but the essential characters < and > are correctly encoded by Redmine. Thus, we cannot detect any XSS vulnerability at this point. As a note regarding the warning from Figure 4.2, we would like to point out that the payload could only be entered into Redmine using our custom Git executable. By default Git would not allow the characters `"` and `;` in the author name.

MantisBT

MantisBT relies on its own implementation for the secure data output to users. The MantisBT source code contains sanitization functions used context-dependently to safely output data. Using only automated tests, we could not detect any security flaws in the implementation that could



Cross Site Scripting (Persistent)

URL:	http://redmine/projects/project/repository/1/changes
Risk:	🟥 High
Confidence:	Medium
Parameter:	author
Attack:	" onMouseOver="alert(1);
Evidence:	
CWE ID:	79
WASC ID:	8
Source:	Active (40014 - Cross Site Scripting (Persistent))

Other Info:

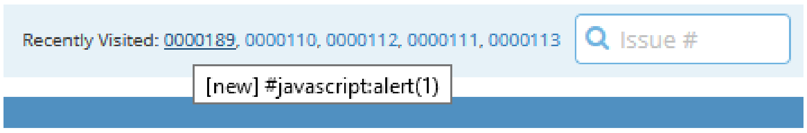
Source URL: http://redmine/git_interface/

(a) Persistent XSS warning for redmine.

```
<td class="comments"><p>" onMouseOver="alert(1);</p></td>
```

Listing 4.1: HTML tag regarding the persistent XSS warning for redmine.

Figure 4.2: Persistent XSS warning generated by the ZAP and corresponding HTML code.



Recently Visited: 0000189, 0000110, 0000112, 0000111, 0000113

Issue #

[new] #javascript:alert(1)

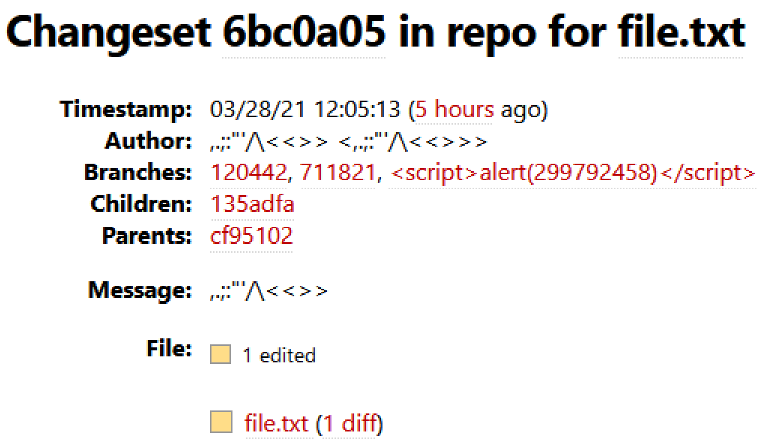
(a) Screenshot of recently visited list in MantisBT.

```
<a href="/view.php?id=189" title="[new] #javascript:alert(1)">0000189</a>
```

Listing 4.2: The HTML anchor tag for a recently visited issue in MantisBT.

Figure 4.3: Exemplification of a reflected XSS warning for MantisBT.

enable XSS. Given that the MantisBT source code is publicly available, it is highly advisable to perform a detailed static analysis of these functions for a proper penetration test. Since our focus lies on the functionality of our web interfaces, we have neglected this part. An illustrative warning generated during automated scans involves the list of recently visited issues. In Figure 4.3 an exemplification can be seen. MantisBT displays a list of links to recently visited issues at the top right of the page. For this example, we selected an issue that Burp Suite created via the mail interface. The used security assessment tools reported that the issue summary is contained in the `title` attribute of the respective HTML anchor tag. The corresponding anchor tag can be seen in Listing 4.2. The `title` attribute consists of two parts, the issue status “`[new]`” and the issue summary “`#javascript:alert(1)`”. To execute XSS at this point, we would have to break out of the tag attribute, but as a matter of fact, the `"` character is escaped correctly. In this case MantisBT uses a function called `string_html_specialchars` to escape the issue summary and handle UTF-8 characters. In Appendix B, we describe the `string_html_specialchars` function in more detail In summary, even though we did not find any weaknesses, both interfaces assisted in scanning MantisBT as intended.



Changeset 6bc0a05 in repo for file.txt

Timestamp:	03/28/21 12:05:13 (5 hours ago)
Author:	,.;:"'/\<<>> <,.;:"'/\<<>>>
Branches:	120442, 711821, <script>alert(299792458)</script>
Children:	135adfa
Parents:	cf95102
Message:	,.;:"'/\<<>>
File:	🟨 1 edited
	🟨 file.txt (1 diff)

Figure 4.4: Git commit changeset from the Trac web user interface.

Trac

Trac was initially developed with the templating language Genshi but was later upgraded to Jinja2 [Trad]. From a security perspective, Jinja2 is a good choice because it auto escapes data user data. An example where auto escaping comes into play is when commit information is output in the repository browser of Trac. For this example, we created a commit using the Git interface to verify which characters get encoded. In Figure 4.4 we provide a screenshot of how Trac outputs commit information to the user. The automated scans we ran produced corresponding warnings. As can be seen, the authorship information, as well as the branch name (`<script>alert(299792458)</script>`) and the commit message contain special characters. All displayed special characters are escaped correctly by Jinja2 depending on the context, so we cannot detect any XSS vulnerability. However, one peculiarity can be observed in Figure 4.4, and that is that the author name contains the characters $<$ and $>$ several times. That suggests that Trac uses a custom implementation to manage Git repositories. In fact, Trac has its own VCS API that contains a low-level wrapper for the system Git executable. More details about the low-level Git wrapper of Trac can be found in Appendix C. Regarding the applicability of our approach to Trac, we can say that our Git interface also worked well here for tool-assisted testing.

5 Conclusion and Future Work

In this bachelor thesis, we designed a methodology to test ITSs for XSS vulnerabilities via unconventional input interfaces such as email or VCSs. Our goals were that the designed methodology is applicable to all ITS-specific input interfaces, that different ITSs can be tested with minimal configuration, and that already existing tools can be used. With the approach to create web interfaces for ITS-specific input interfaces, we have designed a methodology that is not only applicable to ITSs but for arbitrary input interfaces. In the context of this thesis, we have exemplarily created two web interfaces, one for email and one for Git. Both the approach and the two input interfaces brought their unique challenges. To avoid complications with existing security assessment tools, we hid the respective test ITS and the web interfaces behind a reverse proxy. The email interface introduced the problem that ITSs usually fetch emails only periodically. In our opinion, this problem cannot be solved in general, at least not without making the configuration process significantly more complicated. Nevertheless, for our tests and in case a dedicated test environment is available, we have found an arguably simple solution by specifying an optional fetch command. For the Git interface, we created a custom Git executable to allow disallowed input. With the modified Git executable, the deficiency remains that the capability to make use of disallowed input is heavily dependent on the server-side Git implementation. To test the applicability of our methodology, we used it for testing the three open-source ITSs Redmine, MantisBT, and Trac. Our results show that custom web interfaces are helpful for tool-assisted security assessment tests of web applications. The security assessment tools we used recognized the web interfaces as input vectors and used them during their automated vulnerability scans. We conclude that we have succeeded in designing a feasible approach for integrating unconventional input interfaces into the penetration testing of web applications.

In order to test the method designed in this thesis, we have created two research prototypes and proven their applicability. Thus, this thesis provides a basis to extend the presented approach and to refine the implementation. In the following, we address a few things that we consider worthwhile for further research. First, there is the opportunity of creating more web interfaces. We have not covered all ITS-specific input interfaces in this thesis. Applying and evaluating our methodology to other VCSs like SVN would be interesting. Moreover, our research prototypes can be improved. For example, investigating how the email header can be used as an attack vector could help to enhance the email interface. The Git interface can be improved by incorporating a library such as `libgit2` and investigating which other possibilities Git repositories offer as an attack vector. Here we think of developing a polyglot that is recognized by Git as a repository but contains improper input to test for XSS. Such a repository would be universally applicable and could be used to test arbitrary ITSs.

Bibliography

[Ale20] Aleksander Machniak. *Fix cross-site scripting (XSS) via HTML messages with malicious svg/namespace — Commit at GitHub*. Accessed 06.11.2020. 2020. URL: https://github.com/roundcube/roundcubemail/commit/3e8832d029b035e3fcfb4c75839567a9580b4f82 (cit. on p. 28).

[Anda] Andres Riancho (andresriancho). *GitHub issue — Multiple domain names as target*. Accessed 21.03.2021. URL: https://github.com/andresriancho/w3af/issues/2618 (cit. on p. 51).

[Andb] Andres Riancho (andresriancho) and individual contributors. *Web Application Attack and Audit Framework (w3af)*. Accessed 14.03.2021. URL: http://w3af.org/ (cit. on p. 47).

[And20] Andrea Cardaci. *[CVE-2020-15562] Roundcube 1.3.9 — Stored XSS in received emails*. Accessed 08.11.2020. 2020. URL: https://cardaci.xyz/advisories/2020/07/21/roundcube-1.3.9-stored-xss-in-received-emails/ (cit. on pp. 27, 28).

[Ann21] Anne Van Kesteren. *Fetch — Fetch Standard*. Accessed 31.01.2021. 2021. URL: https://fetch.spec.whatwg.org/commit-snapshots/4b5a5b4250e286611cb445b1b1b4c6e7285d1d5c/ (cit. on p. 31).

[Burp] *Burp Suite*. Accessed 21.02.2021. URL: https://portswigger.net/burp (cit. on pp. 45, 46).

[Byr20] Byron Mühlberg. *Malware Attack on GitHub Repositories a Disturbing Development for Open Source Projects*. Accessed 18.10.2020. 16.06.2020. URL: https://www.cpomagazine.com/cyber-security/malware-attack-on-github-repositories-a-disturbing-development-for-open-source-projects/ (cit. on p. 15).

[Car00] Carnegie Mellon University — Software Engineering Institute. “2000 CERT Advisories”. In: (2000). Accessed 30.11.2020. URL: https://resources.sei.cmu.edu/asset_files/WhitePaper/2000_019_001_496188.pdf (cit. on p. 34).

[Chr19] Chrome Developers. *Remove the XSS Auditor from Chrome.* Accessed 20.02.2021 https://www.chromestatus.com/feature/5021976655560704 and https://bugs.chromium.org/p/chromium/issues/detail?id=968591. 2019 (cit. on p. 35).

[CVE18] CVE Details. *CVE-2018-14605*. Accessed 05.11.2020. 2018. URL: https://www.cvedetails.com/cve/CVE-2018-14605/ (cit. on p. 24).

[CVE19] CVE Details. *Security Vulnerabilities — GitLab*. Accessed 18.10.2020. Updated 16.09.2019. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-13074/Gitlab.html (cit. on p. 15).

[CVSS] CVSS Special Interest Group (SIG) members and individual contributers. *Common Vulnerability Scoring System version 3.1: Specification Document*. Accessed 15.11.2020. URL: https://www.first.org/cvss/specification-document (cit. on p. 31).

[Dan] Daniel Lundin and Bas van der Vlies. *email2trac*. Accessed 14.03.2021. URL: https://oss.trac.surfsara.nl/email2trac (cit. on p. 51).

[Dav10] David Hicks. *Source code repository integration with MantisBT 1.3.x*. Accessed 28.10.2020. 31.03.2010. URL: https://mantisbt.org/blog/archives/mantisbt/83 (cit. on p. 17).

[DRRG13] F. Duchène, S. Rawat, J. Richier, R. Groz. “LigRE: Reverse-engineering of control and data flow models for black-box XSS detection”. In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. Oct. 2013, pp. 252–261. DOI: 10.1109/WCRE.2013.6671300 (cit. on p. 33).

[Eli20] Elie Saad, Matteo Meucci, Rick Mitchell. *Web Security Testing Guide — Version 4.1*. Accessed 26.11.2020 https://owasp.org/www-project-web-security-testing-guide/v41/ and https://github.com/OWASP/wstg/releases/download/v4.1/wstg-v4.1.pdf. 2020 (cit. on pp. 32, 37).

[Eri17] Eric Sesterhenn, Berend-Jan Wever, Michele Orrù, Markus Vervier. *Browser Security WhitePaper*. Accessed 31.01.2021. 2017. URL: https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf (cit. on p. 31).

[Fab14] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, Roland Groz. “Kameleon-Fuzz:Evolutionary Fuzzing for Black-Box XSS Detection”. In: Accessed 27.11.2020. 2014. URL: https://www.cs.huji.ac.il/~ai/projects/2014/EvolutionaryXSSDetector/files/original_article.pdf (cit. on p. 33).

[Flask] *Flask*. Accessed 27.02.2021. URL: https://flask.palletsprojects.com/en/1.1.x/ (cit. on p. 40).

[FlaskWTF] *Flask WTF*. Accessed 27.02.2021. URL: https://flask-wtf.readthedocs.io/en/stable/ (cit. on p. 40).

[Fra18] Frans Rosén, James Ritchey, GitLab Team. *Stored XSS in Branch-name when committing to branch from Web IDE — Issue at GitLab*. Accessed 06.11.2020. 2018. URL: https://gitlab.com/gitlab-org/gitlab-foss/-/issues/47793 (cit. on pp. 28, 29).

[Git] GitHub, Inc. *GitHub Security Bug Bounty — GitHub Security*. Accessed 17.10.2020. URL: https://bounty.github.com/ (cit. on p. 15).

[Hac] HackerOne. *The 2020 Hacker Report*. Accessed 10.03.2021. URL: https://www.hackerone.com/resources/reporting/the-2020-hacker-report (cit. on p. 46).

[Hen20] Henrik Nielsen, Jeffrey Mogul, Larry Masinter, Roy Fielding, Jim Gettys, Paul Leach, Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1 — RFC 2616 (Draft Standard)*. Accessed 05.11.2020. Updated 21.01.2020. URL: https://tools.ietf.org/html/rfc2616 (cit. on p. 26).

[HMZA15] I. Hydara, A. B. Md Sultan, H. Zulzalil, N. Admodisastro. "Cross-Site Scripting Detection Based on an Enhanced Genetic Algorithm". In: *Indian Journal of Science and Technology* 8 (Nov. 2015). DOI: 10.17485/ijst/2015/v8i30/86055 (cit. on p. 33).

[Jinja] *Jinja*. Accessed 27.02.2021. URL: https://jinja.palletsprojects.com/en/2.11.x/ (cit. on p. 40).

[Joh] John Reese (jreese) and individual contributors. *Mantis Source Integration*. Accessed 05.03.2021. URL: https://github.com/mantisbt-plugins/source-integration (cit. on p. 51).

[Jun] Junio C. Hamano, Shawn O. Pearce, Linus Torvalds, individual contributors. *Git (v2.31.0-rc2) — Source code*. Accessed 10.03.2021. URL: https://github.com/git/git/tree/13d7ab6b5d7929825b626f050b62a11241ea4945 (cit. on pp. 65–68, 70, 71).

[Ken] Ken (mastermindg). *mastermindg/trac-ubuntu — Docker image*. Accessed 10.03.2021. URL: https://hub.docker.com/r/mastermindg/trac-ubuntu/ (cit. on p. 48).

[LSJ13] S. Lekies, B. Stock, M. Johns. "25 Million Flows Later: Large-Scale Detection of DOM-Based XSS". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer amp; Communications Security*. CCS '13. Berlin, Germany: Association for Computing Machinery, 2013, pp. 1193–1204. ISBN: 9781450324779. DOI: 10.1145/2508859.2516703. URL: https://doi.org/10.1145/2508859.2516703 (cit. on p. 33).

[LW18] M. Liu, B. Wang. "A Web Second-Order Vulnerabilities Detection Method". In: *IEEE Access* 6 (2018), pp. 70983–70988. DOI: 10.1109/ACCESS.2018.2881070 (cit. on p. 33).

[Mana] MantisBT Community. *Mantis Bug Tracker — EmailReporting Plugin*. Accessed 05.03.2021 https://www.mantisbt.org/wiki/doku.php/mantisbt:plugins:emailreporting and https://github.com/mantisbt-plugins/EmailReporting (cit. on p. 50).

[Manb] MantisBT Team. *MantisBT source code repository*. Accessed 31.10.2020. URL: https://github.com/mantisbt/mantisbt (cit. on p. 17).

[Man19] MantisBT Team. *Mantis Feature List*. Accessed 22.10.2020. Updated 15.01.2019. URL: https://www.mantisbt.org/wiki/doku.php/mantisbt:features (cit. on p. 16).

[MBG14] B. Mewara, S. Bairwa, J. Gajrani. "Browser's defenses against reflected cross-site scripting attacks". In: *2014 International Conference on Signal Propagation and Computer Technology (ICSPCT 2014)*. 2014, pp. 662–667. DOI: 10.1109/ICSPCT.2014.6884928 (cit. on p. 35).

[Moza] Mozilla Foundation and individual contributors. *Document.domain — MDN web docs*. Accessed 03.11.2020. URL: https://developer.mozilla.org/de/docs/Web/API/Document/domain (cit. on p. 21).

[Mozb] Mozilla Foundation and individual contributors. *Document.write() — MDN web docs*. Accessed 03.11.2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document/write (cit. on p. 21).

[Mozc] Mozilla Foundation and individual contributors. *MDN Web Docs — Client-side form validation*. Accessed 01.03.2021. URL: https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation (cit. on p. 34).

[Mozd] Mozilla Foundation and individual contributors. *Never use eval()! — MDN web docs*. Accessed 03.11.2020. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval#Never_use_eval! (cit. on p. 21).

[Moze] Mozilla Foundation and individual contributors. *POST — MDN web docs*. Accessed 05.11.2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST (cit. on p. 26).

[New18] L. H. Newman. *GitHub Survived the Biggest DDoS Attack Ever Recorded*. Accessed 18.10.2020. 1.03.2018. URL: https://www.wired.com/story/github-ddos-memcached/ (cit. on p. 15).

[nginx] NGINX Docker Maintainers. *NGINX docker image*. Accessed 15.03.2021. URL: https://hub.docker.com/_/nginx (cit. on p. 48).

[NIS14] NIST — National Institute of Standards and Technology. *CVE-2014-9607*. Accessed 05.11.2020. 2014. URL: https://nvd.nist.gov/vuln/detail/CVE-2014-9607 (cit. on p. 24).

[NIS19] NIST — National Institute of Standards and Technology. *CVE-2019-12311*. Accessed 05.11.2020. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-12311 (cit. on p. 24).

[Off] OffSec Services Limited. *Kali Linux Tools Listing*. Accessed 10.03.2021. URL: https://tools.kali.org/tools-listing (cit. on pp. 45, 47).

[Ole] Oleg Kainov (okainov). *okainov/mantisbt-docker — Dockerfile*. Accessed 05.03.2021 https://hub.docker.com/r/okainov/mantisbt and https://github.com/okainov/mantisbt-docker (cit. on p. 48).

[OTT17] Catalin Cimpanu. *Four Years Later, We Have a New OWASP Top 10*. Accessed 19.10.2020. 21.11.2017. URL: https://www.bleepingcomputer.com/news/security/four-years-later-we-have-a-new-owasp-top-10/ (cit. on p. 15).

[OWA] OWASP Community. *Vulnerability Scanning Tools*. Accessed 10.03.2021. URL: https://owasp.org/www-community/Vulnerability_Scanning_Tools (cit. on p. 45).

[PC 11] PC Plus. *How cross-site scripting attacks work*. Accessed 23.11.2020. 2011. URL: https://www.techradar.com/news/internet/how-cross-site-scripting-attacks-work-1046844 (cit. on p. 32).

[Pora] PortSwigger Ltd. *Content security policy*. Accessed 30.11.2020. URL: https://portswigger.net/web-security/cross-site-scripting/content-security-policy (cit. on p. 34).

[Porb] PortSwigger Ltd. *Cross-site scripting contexts*. Accessed 14.11.2020. URL: https://portswigger.net/web-security/cross-site-scripting/contexts (cit. on p. 24).

[Reda] Redmine Team. *Features*. Accessed 22.10.2020. URL: https://www.redmine.org/projects/redmine/wiki/features (cit. on p. 16).

[Redb] Redmine Team. *Official Redmine Docker image*. Accessed 13.03.2021. URL: https://hub.docker.com/_/redmine (cit. on p. 48).

[Redc] Redmine Team. *Redmine source code repository*. Accessed 31.10.2020. URL: https://www.redmine.org/projects/redmine/repository (cit. on p. 17).

[Redd] Redmine Team. *Redmine source code repository at GitHub*. Accessed 31.10.2020. URL: https://github.com/redmine/redmine (cit. on p. 17).

[Rede] Redmine Team. *Who uses Redmine?* Accessed 28.10.2020. URL: https://www.redmine.org/projects/redmine/wiki/WeAreUsingRedmine (cit. on p. 17).

[RFC626511] David M. Kristol and Lou Montulli. *HTTP State Management Mechanism*. RFC 6265. University of California, Berkeley, Apr. 2011. URL: https://tools.ietf.org/html/rfc6265 (cit. on p. 30).

[Sama] Sameer Naik (sameersbn) and individual contributors. *sameersbn/gitlab — Docker image*. Accessed 01.03.2021. URL: https://hub.docker.com/r/sameersbn/gitlab/ (cit. on p. 49).

[Samb] Sameer Naik (sameersbn) and individual contributors. *sameersbn/redmine — Docker image*. Accessed 01.03.2021. URL: https://hub.docker.com/r/sameersbn/redmine/ (cit. on pp. 48, 49).

[Sec06] SecureNet on behalf of the Bundesamtes für Sicherheitin der Informationstechnik. "Sicherheit von Webanwendungen — Maßnahmenkatalog und Best Practices". In: (2006). Accessed 30.11.2020. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/WebSec/WebSec.pdf?__blob=publicationFile (cit. on p. 34).

[Sof] Software Freedom Conservancy — GitLab. *Bug Bounty Program — HackerOne*. Accessed 17.10.2020. URL: https://hackerone.com/gitlab (cit. on p. 15).

[The] The OWASP Foundation. *Input Validation Cheat Sheet*. Accessed 01.03.2021. URL: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html (cit. on p. 34).

[The10] The OWASP Foundation. "OWASP Secure Coding Practices Quick Reference Guide". In: (2010). Accessed 30.11.2020. URL: https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf (cit. on p. 34).

[The20a] The MITRE Corporation. *CVE-2020-15562*. Accessed 05.11.2020. 2020. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15562 (cit. on pp. 24, 28).

[The20b] The MITRE Corporation. *CVE-2020-9016*. Accessed 05.11.2020. 2020. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9016 (cit. on p. 24).

[The20c] The MITRE Corporation. *CVE-2020-9447*. Accessed 05.11.2020. 2020. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9447 (cit. on p. 24).

[Traa] Trac Team. *Trac Plugins List*. Accessed 26.10.2020. URL: https://trac.edgewall.org/wiki/PluginList (cit. on p. 16).

[Trab] Trac Team. *Trac source code repository*. Accessed 31.10.2020. URL: https://svn.edgewall.org/repos/trac/ (cit. on p. 17).

[Trac] Trac Team. *Trac source code repository at GitHub*. Accessed 31.10.2020. URL: https://github.com/edgewall/trac (cit. on p. 17).

[Trad] Trac Team. *Trac wiki — Switch to the Jinja2 Template Engine*. Accessed 28.03.2021 https://trac.edgewall.org/wiki/TracDev/Proposals/Jinja and https://trac.edgewall.org/wiki/TracDev/HtmlTemplates (cit. on p. 55).

[Trae] Trac Team. *Who Uses Trac?* Accessed 31.10.2020. URL: https://trac.edgewall.org/wiki/TracUsers (cit. on p. 17).

[Wik20a] Wikipedia contributors. *Comparison of issue-tracking systems — Wikipedia, The Free Encyclopedia*. Accessed 17.10.2020. Updated 16.10.2020. URL: https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems (cit. on p. 16).

[Wik20b] Wikipedia contributors. *Comparison of project management software — Wikipedia, The Free Encyclopedia*. Accessed 26.10.2020. Updated 31.07.2020. URL: https://en.wikipedia.org/wiki/Comparison_of_project_management_software (cit. on p. 16).

[Wor] World Wide Web Consortium (W3C). *W3C WAI's Web Accessibility Tutorials — Validating Input*. Accessed 01.03.2021. URL: https://www.w3.org/WAI/tutorials/forms/validation/ (cit. on p. 34).

[Wor18] World Wide Web Consortium (MIT, ERCIM, Keio, Beihang) and Web Application Security Working Group and individual contributors. *Content Security Policy Level 3 — W3C Working Draft*. Accessed 30.11.2020. 2018. URL: https://www.w3.org/TR/CSP3/ (cit. on p. 34).

[WTForms] *WTForms*. Accessed 27.02.2021. URL: https://wtforms.readthedocs.io/en/2.3.x/ (cit. on p. 40).

[Xav] Xavi Mendez (xmendez) and individual contributors. *Wfuzz - The Web Fuzzer*. Accessed 15.03.2021. URL: https://github.com/xmendez/wfuzz (cit. on p. 47).

[ZAP] *OWASP Zed Attack Proxy (ZAP)*. Accessed 21.02.2021. URL: https://www.zaproxy.org/ (cit. on p. 46).

A Custom Git Executable

A.1 Git Authorship Information Restriction

Since our goal is to test the complete attack surface, we have to circumvent the character limitation of the Git commit authorship information. Our approach is to build a custom Git executable that we use in our web interface to create commits. The custom version of Git should limit user input as little as possible. To build a modified Git executable, we use the current source code (v2.31.0-rc2) available on GitHub [Jun]. The modifications we make in the Git source code are intentionally minimalistic; the resulting code is not optimal. Note that our custom Git executable should only be used for testing purposes. We neither consider stability nor security, only functionality.

To represent Git commit authorship information externally, Git uses identifier lines of the form "`name <email> date`", where the date internally consists of a UTC timestamp and an offset. Since Git stores the timestamp not as a string, we can assume that payloads injected at this point will result in an internal error in the ITS. It would be interesting to explore whether the commit date or hash can also be misused as an input vector, but this is beyond the scope of this thesis. A Git commit contains the authorship information of two persons, the author and the committer. The authorship information for a commit can be set both from the command-line and through environment variables. Since Git allows the environment variables to be configured, it is common that the committer is read from environment variables, and the author is passed as a command-line argument.

To remove the character limitation in the authorship information, we have to find out which parts of the source code are responsible for the limitation. Fortunately, all the functions we need to change are in the same source file, namely `ident.c`. Here there are mainly two functions that are responsible for the unwanted behavior. First there is the function `split_ident_line` which splits an identification line into the components name, email and date and second there is the function `fmt_ident` as counterpart. The `fmt_ident` function receives name, email and date and returns the corresponding identification string. When creating commits both functions are called, `split_ident_line` splits the identification line passed from the command line into a data structure which is later passed to `fmt_ident`. First we adjust the `split_ident_line` function. Relevant parts of the original code can be seen in Listing A.1. We have to make sure that the complete string is processed. As mentioned above, the email string is surrounded by the characters < and >, which also causes further problems, as we will see later. When the `split_ident_line` function is passed a string, it truncates everything after the first appearance of the > character. We want to change this behavior. A simple way to make Git also accept characters after > is to set the end of the email address in the data structure. For this we add the line `split->mail_end = split->name_begin + len;` to the function after line 302. Code to parse the following timestamp correctly in case of a corrupt email address is already implemented.

```c
int split_ident_line(struct ident_split *split, const char *line, int len)
{
    ...
    split->name_begin = line;
    for (cp = line; *cp && cp < line + len; cp++)
        if (*cp == '<') {
            split->mail_begin = cp + 1;
            break;
        }
    ...
    for (cp = split->mail_begin; cp < line + len; cp++)
        if (*cp == '>') {
            split->mail_end = cp;
            break;
        }
    ...
    split->mail_end = split->name_begin + len;
    ...
    /*
     * Look from the end-of-line to find the trailing ">" of the mail
     * address, even though we should already know it as split->mail_end.
     * This can help in cases of broken idents with an extra ">" somewhere
     * in the email address.  Note that we are assuming the timestamp will
     * never have a ">" in it.
     *
     * Note that we will always find some ">" before going off the front of
     * the string, because will always hit the split->mail_end closing
     * bracket.
     */
    for (cp = line + len - 1; *cp != '>'; cp--)
        ;
    ...
}
```

Listing A.1: Code snippets from the `split_ident_line` function from the Git source code [Jun]. Additions are colored green.

For the character restriction in the name of the identification line, the function `strbuf_addstr_without_crud` is called from `fmt_ident`. This function copies a string and removes certain characters. The source code is shown in Listing A.2. The disallowed characters are determined with the help of the `crud` function, which returns a non-zero value for an illegal character. The source code of the `crud` function (Listing A.3) reveals that the ASCII characters ., ,, :, ;, <, >, ", \, ' and the ones with ASCII values less than 33 are forbidden. To remove this limitation we can hard-code the return value of `crud` to 0 or comment out the corresponding code lines in `strbuf_addstr_without_crud`. Hard-coding the return value of the `crud` function is possible since it is only used to check for disallowed characters in identification lines. In our opinion, the latter approach, shown in Listing A.2, is cleaner. Further, the switch statement from line 255 to line 258 needs to be eliminated not to strip any characters.

```c
static void strbuf_addstr_without_crud(struct strbuf *sb, const char *src)
{
    size_t i, len;
    unsigned char c;

    /* Remove crud from the beginning.. */
    while ((c = *src) != 0) {
        if (!crud(c))
            break;
        src++;
    }

    /* Remove crud from the end.. */
    len = strlen(src);
    while (len > 0) {
        c = src[len-1];
        if (!crud(c))
            break;
        --len;
    }

    /*
     * Copy the rest to the buffer, but avoid the special
     * characters '\n' '<' and '>' that act as delimiters on
     * an identification line. We can only remove crud, never add it,
     * so 'len' is our maximum.
     */
    strbuf_grow(sb, len);
    for (i = 0; i < len; i++) {
        c = *src++;
        switch (c) {
        case '\n': case '<': case '>':
            continue;
        }
        sb->buf[sb->len++] = c;
    }
    sb->buf[sb->len] = '\0';
}
```

Listing A.2: The `strbuf_addstr_without_crud` function from the Git source code [Jun]. Deletions are colored red.

With that done, we need to make one more change in the source code to be able to create commits with disallowed author names. The `fmt_ident` function contains a check that throws an error in case the name in an identifier line consists only of disallowed characters. We can remove this check, as shown in Listing A.4. It is unlikely that this check will cause problems since XSS payloads do not consist only of the characters disallowed in Git, but we want to include this edge case.

```c
static int crud(unsigned char c)
{
    return  c <= 32  ||
        c == '.' ||
        c == ',' ||
        c == ':' ||
        c == ';' ||
        c == '<' ||
        c == '>' ||
        c == '"' ||
        c == '\\' ||
        c == '\'';
}
```

Listing A.3: The `crud` function from the Git source code [Jun].

```c
const char *fmt_ident(const char *name, const char *email,
              enum want_ident whose_ident, const char *date_str, int flag)
{
    ...
        if (strict  !has_non_crud(name))
            die(_("name consists only of disallowed characters: %s"), name);
    }

    strbuf_reset(ident);
    if (want_name) {
        strbuf_addstr_without_crud(ident, name);
        strbuf_addstr(ident, " <");
    }
    strbuf_addstr_without_crud(ident, email);
    ...
}
```

Listing A.4: Code snippet from the `fmt_ident` function from the Git source code [Jun]. Deletions are colored red.

Using our modified Git version, we can now compile an executable that allows our web interface to create commits with unauthorized characters in the authorship information. We can use almost all forbidden characters with our web interface to set an arbitrary payload with this implementation. However, only almost all forbidden characters are usable. Git uses, as already mentioned, the characters < and > to specify the email address in the identifier line. We can set arbitrary composite author names with our Git executable, but the Git executable that uses the ITS that we do not have access to will treat all characters after the first occurrence of < as an email address and will not return any characters after the first occurrence of >. A side effect of our changes in the Git source code is that we get the remaining characters delivered with the custom executable. The Git implementation used in the ITS will most likely not do that, and thus the ITS will also not output all characters after the first >. An example to illustrate the problem can be seen in Listing A.5. First, we create a commit using the custom executable and use disallowed characters in the author identifier line. Next, we print the `git log` of our executable and the original one. As can be seen, both outputs contain all disallowed characters in the author name, which means our approach works. The difference, however, is that the original Git executable chops of all characters after the first occurrence of >. The choice of the < and > characters to specify the email is highly inconvenient for our purpose, as it prevents us from injecting multiple HTML tags or even just opening and closing a two-part HTML tag. Nevertheless, using an identifier line for a payload like `<img src="..."onerror="alert()">` is possible, even if only as an email address.

```
$ custom-git commit -m "commit msg" --author=".,:;\"\\' <.,:;\"\\'<<>.,:;\"\\'.,:;\"\\'>"

$ custom-git log
commit f30644933583f2646660a4bb65793e29ca9f5f42 (HEAD -> master)
Author: .,:;"\' <.,:;"\'<<>.,:;"\'.,:;"\'>> 1615491339 +0100>
Date:   Thu Mar 11 20:35:39 2021 +0100

    commit msg

$ git log
commit f30644933583f2646660a4bb65793e29ca9f5f42 (HEAD -> master)
Author: .,:;"\' <.,:;"\'<<>
Date:   Thu Mar 11 20:35:39 2021 +0100

    commit msg
```

Listing A.5: A comparison of the `git log` outputs of Git and our custom Git executable. The commit author was set to: `.,:;"\'<.,:;"\'<<>.,:;"\'.,:;"\'>`.

A.2 Git Branch Name Restriction

Besides the input restricted authorship information, we face further input limitations, like for the branch name. ITSs that support VCS integration also provide the ability to select among different branches. Therefore branch names are part of the attack surface. Unfortunately, the input limitation cannot be circumvented with our approach of using a custom Git executable. Git uses a mapping of ASCII values to numbers from zero to five, determining whether characters are allowed in branch names. The mapping is defined in the source code in the `refs.c` file as a `refname_disposition` array, where the number zero represents allowed characters and the numbers one through five represent forbidden or conditionally allowed characters. See Listing A.6 and Table A.1 for reference. To be able to create branch names containing disallowed characters, it is sufficient to set all values in the `refname_disposition` array to zero. However, Git also uses the `refname_disposition` array to return branch names, which means our corrupt branch names will not necessarily be output by the ITS because it does not use our custom Git version.

```
static unsigned char refname_disposition[256] = {
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 2, 1,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 4,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4, 0, 4, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 4, 4
};
```

Listing A.6: `refname_disposition` array from the Git source code [Jun]. See Table A.1 for reference.

Value	Denotation
0	An acceptable character for refs
1	End-of-component
2	., look for a preceding . to reject .. in refs
3	{, look for a preceding @ to reject @{ in refs
4	A bad character: ASCII control characters, and ":", "?", "[", "\", "^", "~", SP, or TAB
5	*, reject unless REFNAME_REFSPEC_PATTERN is set

Table A.1: Denotation of the values in the `refname_disposition` array. See Listing A.6 for reference.

Further, we removed another branch name check from the Git source code to allow white spaces in branch names. The changes had to be done in the `branch.c` file and can be seen in Listing A.7.

```
/*
 * Check if 'name' can be a valid name for a branch; die otherwise.
 * Return 1 if the named branch already exists; return 0 otherwise.
 * Fill ref with the full refname for the branch.
 */
int validate_branchname(const char *name, struct strbuf *ref)
{
    if (strbuf_check_branch_ref(ref, name))
        die(_("'%s'is not a valid branch name."), name);
    strbuf_check_branch_ref(ref, name);
    return ref_exists(ref->buf);
}
```

Listing A.7: The modified `validate_branchname` function from the Git source code [Jun]. Additions are colored green and deletions are colored red.

A.3 Git Tag Annotation Restriction

The Git tag annotation restriction is again straightforward to remove. The restriction is caused by calling the `check_refname_format` function. Only the relevant lines of code in the file `builtin/tag.c` had to be removed like shown in Listing A.8.

```
static int strbuf_check_tag_ref(struct strbuf *sb, const char *name)
{
    if (name[0] == '-')
        return -1;

    strbuf_reset(sb);
    strbuf_addf(sb, "refs/tags/%s", name);

    return 0; return check_refname_format(sb->buf, 0);
}
```

Listing A.8: The modified `strbuf_check_tag_ref` function from the Git source code [Jun].Additions are colored green and deletions are colored red.

B The string_html_specialchars Function (MantisBT)

The warning addressed in Section 4.2.2 under MantisBT is based on the observation that the summary of an issue appears in the `title` attribute of an anchor tag. If the user-controllable issue summary were not encoded properly, this would be an XSS vulnerability. To encode the issue summary, MantisBT uses the `string_html_specialchars` function. The `string_html_specialchars` function takes a string as parameter and returns a HTML safe copy of it. The `string_html_specialchars` function is, as all string sanitization functions of MantisBT, defined inside the `/core/string_api.php` file. Listing B.1 shows the function. Basically the `string_html_specialchars` function uses only two native PHP functions. First, by calling the `htmlspecialchars` function, the function converts special characters in the passed string to HTML codes. This call sets the `ENT_COMPAT` flag, which ensures that double quotes are encoded while single quotes remain unchanged. Note that uncoded double quotes would allow us to break out of the title attribute mentioned above. Furthermore, the `encoding` parameter is set, in this case, UTF-8, which defines the character encoding for a conversion. Then, using the `preg_replace` function, the encoding is partially undone with a regular expression. The `preg_replace` function replaces all occurrences of a specified regular expression with a given string. The regular expression is `/&amp;(#[0-9]+|[a-z]+);/i` and can be translated as follows: The expression starts with `&amp;`, which is the HTML encoding of the `&` symbol. It continues with a `#` followed by an integer or a string from the English alphabet (without a leading `#`). The regular expression ends with a semicolon (`;`). The `i` character at the end specifies that the replacement is case-insensitive. If the regular expression matches a substring, the substring is replaced by itself with the change that `&amp;` becomes `&`. The `string_html_specialchars` function partially undoes the HTML encoding of the ampersand character.

```php
/**
 * Calls htmlspecialchars on the specified string, handling utf8
 * @param string $p_string The string to process.
 * @return string
 */
function string_html_specialchars( $p_string ) {
  # achumakov: @ added to avoid warning output in unsupported codepages
  # e.g. 8859-2, windows-1257, Korean, which are treated as 8859-1.
  # This is VERY important for Eastern European, Baltic and Korean languages
  return preg_replace( '/&amp;(#[0-9]+|[a-z]+);/i', '&$1;', @htmlspecialchars( $p_string,
ENT_COMPAT, 'utf-8' ) );
}
```

Listing B.1: The `string_html_specialchars` from the MantisBT source code.

C The Trac Git Wrapper

Trac provides its own VCS API, located at `trac/versioncontrol`. The access to Git repositories is realized with a wrapper around the system's Git executable. The necessary source code can be found in `tracopt/versioncontrol/git/PyGIT.py`. The `PyGit` module contains the `Storage` class which implements a high-level wrapper for Git and the `GitCore` class which implements a low-level wrapper. The default path to the Git binary is `'git'`, which means the system Git binary is used. To parse a commit there is a helper function called `parse_commit` which returns commit information analogous to the command `git cat-file -p <rev>`. That explains why Trac outputs multiple occurrences of the characters < and > in the author name. `git cat-file -p <rev>`, unlike the commonly used `git show <rev>` command, does not output the authorship information as an identifier line. In Figure C.1, we show an output comparison of the two commands.

```
$ git show 6bc0a05b6dc4411d2fa546eb90433fa79d16782f
commit 6bc0a05b6dc4411d2fa546eb90433fa79d16782f
Author: ,.;:"'/\ <<>
Date:   Sun Mar 28 12:05:13 2021 +0000

    ,.;:"'/\<<>>
```

Listing C.1: Example output of `git show <rev>`.

```
$ git cat-file -p 6bc0a05b6dc4411d2fa546eb90433fa79d16782f
tree 0a44bcc690da87328f9dfeebbc58d7ea3e73de51
parent cf95102fc112fae6efe5493480053035c31e0832
author ,.;:"'/\<<>> <,.;:"'/\<<>>> 1616933113 +0000
committer ,.;:"'/\<<>> <,.;:"'/\<<>>> 1616933113 +0000

,.;:"'/\<<>>
```

Listing C.2: Example output of `git cat-file -p <rev>`.

Figure C.1: Output comparison of `git show` and `git cat-file`. The author name and email was set to `,.;:"'/<>>` for this commit.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 06.04.2021 [illegible]

place, date, signature