BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

IMPACT OF THE APPLICATION OF THE CONTENT-SECURITY-POLICY HEADER ON FIREFOX WEBEXTENSIONS

TESTOVÁNÍ VLIVU APLIKACE HLAVIČKY CONTENT-SECURITY-POLICY NA KÓD VLOŽENÝ ROZŠÍŘENÍMI PROHLÍŽEČE FIREFOX

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **BOHDAN INHLIZIIAN**
AUTOR PRÁCE

SUPERVISOR **Ing. LIBOR POLČÁK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2019/2020

Bachelor's Thesis Specification

Student: **Inhliziian Bohdan**
Programme: Information Technology
Title: **Impact of the Application of the Content-Security-Policy Header on Firefox Webextensions**
Category: Software analysis and testing

Assignment:

1. Study the Content-Security-Policy (CSP) HTTP header, its benefits, and syntax. Describe how the presence of the CSP header influences the JavaScript code injected by webextensions (analyze Firefox bug 1267027, Privacy Badger issue 1793, and JavaScript Restrictor issue 25).
2. Learn how to write Selenium test cases.
3. Design a framework for automatic testing of webextensions downloaded from Addons.Mozilla.org (AMO). The framework will detect errors caused by Firefox bug 1267027, for example, using the report-uri CSP directive.
4. Implement the framework and publicly release the implementation.
5. Test as much of AMO webextensions as possible and provide statistics about the extensions impacted by Firefox bug 1267027.
6. Evaluate the work and propose future improvements.

Recommended literature:

- BASTL Vojtěch. *Automatizace webového prohlížeče*. Brno, 2019. Master's Thesis. Brno University of Technology, Faculty of Information Technology.
- WEST Mike, BARTH Adam, VEDITZ, Dan. Content Security Policy Level 2, W3C Recommendation, 2016.
- MAGLIONE Kris. Page CSP Should Not Apply To Content Inserted By Content Scripts. Bug 1267027, Bugzilla Mozilla.org, available online at https://bugzilla.mozilla.org/show_bug.cgi?id=1267027.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Polčák Libor, Ing., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: May 28, 2020
Approval date: May 5, 2020

Abstract

A four-year-old bug in official Firefox's Bugzilla reported that the Content-Security-Policy response header affects the behavior of browser extensions. The goal of this thesis is to test and analyze all of Firefox's extensions in the official extensions store to learn how many of them are affected by the bug. The work has four phases: download all extensions from the store, create usable web GUI, implement the testing application, execute tests, and evaluate the results. We show that the application of CSP header on a web site may influence about 10% of Firefox web extensions and 29% of extensions recommended by Firefox. The total number of users of all influenced recommended extensions is 11 650 730. Hopefully, this research highlights the problem and pushes Firefox developers to fix the bug.

Abstrakt

Čtyři roky starý bug v oficiální Bugzille prohlížeče Firefox hlásí, že hlavička Content-Security-Policy ovlivňuje chování rozšíření prohlížeče. Cílem této práce je otestovat a analyzovat všechna rozšíření Firefoxu z oficiálního uložiště rozšíření a zjistit, kolik z nich je ovlivněno bugem. Práce má čtyři fáze: stáhnout všechna rozšíření z uložiště, vytvořit použitelné webové GUI, implementovat testovací aplikaci, provést testy a vyhodnotit výsledky. V rámci práce jsme zjistili, že aplikace hlavičky CSP na webu může ovlivnit přibližně 8% rozšíření Firefoxu a 21% rozšíření doporučených Firefoxem. Celkový počet uživatelů všech ovlivněných doporučených rozšíření je 11 650 730. Tento výzkum upozorňuje na problém a nutit tvůrce prohlížeče, aby jej vyřešili a ukazuje jeho rozměr.

Keywords

Browser extension testing, Firefox extensions, bug in Firefox, CSP header, extensions CSP error, CSP reports, Selenium extensions testing.

Klíčová slova

Testování rozšíření prohlížeče, rozšíření Firefoxu, chyba ve Firefoxu, CSP hlavička, chyba CSP ve rozšířeních, chyby CSP, testování rozšíření pomoci Selenium

Reference

INHLIZIIAN, Bohdan. *Impact of the Application of the Content-Security-Policy Header on Firefox Webextensions.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Libor Polčák, Ph.D.

Rozšířený abstrakt

Prohlížeč Mozilla Firefox obsahuje chybu, která byla nahlášena před čtyřmi lety. Tato chyba ovlivňuje chování rozšíření prohlížeče. Hlavička CSP zapnuta na webu může zakázat vkládání skriptů pomocí rozšíření. Pokud content script rozšíření vytvoří element skriptu a poté ho vloží do DOM webové stránky, CSP tuto akci zakáže a způsobí CSP report.

Bakalářská práce si klade za cíl otestovat a analyzovat všechna rozšíření Firefoxu v oficiálním úložišti rozšíření Addons.mozilla.org (AMO) a zjistit kolik z nich je chybou ovlivněno.

Tato práce má širokou cílovou skupinu. Nejprve to jsou vývojáři Firefoxu. Chyba v prohlížeči byla nahlášena před čtyřmi lety a během těchto let to způsobovalo bug reporty o problému v oficiálních bugtrackerech různých webových rozšíření. Všechny tyto reporty vedly dlouhé diskuse o tom, co se děje a jak tento bug obejít. Lidí, kteří tyto reporty vytvořili neví zda se jedna o chybu rozšíření nebo prohlížeče. Řešení nebo obcházení chyby vyžaduje čas a peníze.

Tento výzkum je dále užitečný pro vývojáře webových stránek, kteří chtějí zapnout jejich ochranu pomoci CSP. CSP reporty, způsobené injekcemi skriptů rozšíření způsobují zbytečný šum do webového logů. Tato práce prokázala, že některá rozšíření spouští více než 10 CSP reportů pro každé načtení webové stránky každého návštěvníka. V kontextu webových stránek s miliony návštěvníky, jako je YouTube, je to významné. Každý report navíc vyžaduje šetření správce, což také vyžaduje čas a peníze.

Tato práce se zabývá studiem útoků typu Cross-Site Scripting, jejich hlavními principy a metodami ochrany proti ní. Teoretická část studuje základní metody vývoje rozšíření prohlížeče a reprodukuje popsanou chybu implementací jednoduchého rozšíření, které se snaží vložit skript do testovací webové stránky která je zabezpečena pomoci CSP.

Protože práce musí zajistit testování všech rozšíření z AMO, byl studován automatický testovací nástroj Selenium.

Pro dosažení konečného cílu byl implementován automaticky testovací framework. Protože výzkum má širokou cílovou skupinu, nástroj vyžaduje snadné a srozumitelné uživatelské rozhraní. Framework má dvě hlavní části: webovou aplikaci (GUI) a backendovou aplikaci. Tyto části spolu komunikují prostřednictvím API.

GUI má snadné rozhraní, které lze použít k výběru sady rozšíření a spuštění testů pro ně. Má také funkci reprezentovat výsledky testování pomocí grafů. Tato část frameworku navíc obsahuje skripty, které mají za cíl stáhnout všechny rozšíření z AMO a nahrát je do Amazon Web Services S3 úložiště.

Účelem backend aplikace je obdržet sadu poslaných z GUI rozšíření, stáhnout jejich zdrojové kódy z úložiště, provést testy a vrátit výsledky zpět do GUI.

Tato práce má implementace několika scénářů testování. Všechna rozšíření mají soubor s názvem **manifest.json** obsahující všechna jejich metadata. Nejdůležitějším parametrem v metadatech pro tuto práci je **content_scripts** klič, který obsahuje všechny content skripty rozšíření. Content skripty mohou provádět vkládání zdrojů do webové stránky. Prvním scénářem je provést statickou analýzu content skriptů uvedených v **manifest.json**, aby se našli některé přiznaky vkládání kódu, například některé metody, které vkládají skripty do DOM webové stránky. Druhý testovací scénář provádí Selenium testy na všech rozšířeních z AMO, aby bylo možné detekovat tyto rozšíření které dělají injekce skriptů do webových stránek.

Pomocí frameworku byly provedeny testy na všechna rozšíření z AMO a vyhodnoceny výsledky. Výsledky ukazují, že kolem 10% všech rozšíření má přiznaky injekce skriptu. Firefox má navíc doporučený program rozšíření. Statická analýza ukázala, že 29% doporučených rozšíření má tyto přiznaky. Simulační testy, které instalují rozšíření do instance prohlížeče

a testují jejich chování na webových stránkách zabezpečených pomocí CSP, ukázaly, že 1% všech rozšíření vkládá skripty před načtením webové stránky, bez jakékoliv uživatelských akcí, napřiklad, stisknutí tlačítek nebo vyplnění formuláře. U doporučených rozšíření se tato hodnota zvyšuje na 11%.

Během práce bylo zjištěno, že chyba Firefoxu může majitelům webových stránek pomoct “fingerprintovat” uživatele. Každé rozšíření, které provádí injekce skriptu na webové stránky zabezpečené pomocí CSP, zanechá svůj otisk. Protože všechny reporty přicházejí do webových logů, mohou je vývojáři webových stránek analyzovat a zjistit, jaká rozšíření používají jejich návštěvníci. Tato práce studovala tento problém podrobněji a provedla několik manuálních experimentů s populárními rozšířeními. Výsledkem je, že tato rozšíření lze rozpoznat z CSP reportů, které byli nimi spouštěné.

Byli navrhnute některá budoucí vylepšení k analýze problému “fingerprintovani”. Kromě toho, pracé navrhuje zlepšení stávajícího frameworku, zejména zvýšení jeho výkonu, použitelnosti a přesnosti testování.

Impact of the Application of the Content-Security-Policy Header on Firefox Webextensions

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Libor Polčák.

....
Bohdan Inhliziian
May 28, 2020

Acknowledgements

I would like to thank my supervisor Mr. Ing. Libor Polčák for weekly consultations and discussions.

Contents

1 Introduction **2**

2 Theory **4**
2.1 Browser Security Defenses 4
2.1.1 Same-Origin Policy 4
2.1.2 XSS Attacks 5
2.1.3 Content Security Policy 6
2.2 Page CSP Influences Firefox Webextensions 8
2.3 The Target Audience of This Work 10
2.4 Extension's Manifest.json File 11
2.5 Containerization 12
2.6 Selenium 13

3 Design of Extension Tester Framework **15**
3.1 Testing Scenarios Design 15
3.2 Design of The Main Parts of The Framework 16

4 Implementation of Extension Analyzer Framework **19**
4.1 Data Collecting 19
4.2 Docker Containerization and Architecture of Services 21
4.3 Web Application 22
4.4 Backend Testing Application 25
4.5 Implementation of Testing Scenarios 25

5 Testing and Evaluation **29**
5.1 Results of the Provided Tests and Analysis 29

6 Future Improvements of the Research **34**
6.1 Fingerprinting Problem 34
6.2 Other Improvements 35

7 Conclusion **37**

Bibliography **39**

A List of Script Injection Signs **40**

B Content of Media **41**
B.1 Source Code 41

Chapter 1

Introduction

Nowadays, every modern web browser supports extensions. An extension is a small embedded software (plugin) that brings new features to a browser. Extensions can do different actions, e.g., modify web pages, block advertisements, automatize some manual work on a web site. Many extensions put some resources inside a web page. It can be images, CSS stylesheets, and also a JavaScript code. For example, that hides HTML elements, clicks on the buttons, or does some dynamic interaction with a web page.

Injecting resources such as scripts inside of a browsing web page may be potentially dangerous. Script injection may perform some unexpected actions like stealing user's data, showing spam adds, or others. These vulnerabilities, called Cross-site scripting (XSS), are often used by hackers. All modern browsers support mechanism called **Content Security Policy** (CSP) to protect their users against such attacks. The CSP protection can be set on a web server by web site owners.

So, two mechanisms (resource injecting by extensions and protection against injections) are opposite to each other. The first one uses resource injecting into a web page, but the second one fights against those actions. The CSP should not influence actions performing by an extension to save its behavior. Every extension can be installed only by a user, and every extension asks about permissions that a user has to give before the installation. The user has to agree with the permissions and confirm the installation.

An extension is not an attacker in the CSP model, and the CSP header should not influence extensions functionality. But it is not how the Mozilla Firefox web browser behaves. The browser supports both mechanisms, but it causes the conflict described in the previous paragraph. The CSP header blocks the extension's scripts injection into a web page.

This research tests all extensions from the Addons.mozilla.org (AMO) and tests how CSP headers influence their behavior.

The results are useful for different categories of users. First of all, the work is valuable for people that develop the Mozilla Firefox web browser. The report for the bug was opened four years ago, and it is still open. Some extensions such as "Privacy Badger" or "uBlock Origin" have bug reports opened by developers who think it is an extension's issue, but it is not. These reports have long discussions and workarounds on how to solve or bypass the bug.

Secondly, web site developers can use this work to see how enabled CSP header can make noise to security alert logs and be prepared to it.

Further, if extension developers have found the solution to the problem, they can use this work to compare their extensions with concurrent ones and use the comparison in marketing purposes.

Finally, the bug in Firefox may cause some data protection problems like fingerprinting of users. The research may be useful for people who care about privacy and data protection. After study the thesis, they might decide not to use the Firefox web browser or turn CSP protection off. Section 2.3 of Chapter 2 describes more information about fingerprinting and defines the target audience of this work.

This work requires some information to study the main principles of extensions development and study mechanisms for web site security. Further, this thesis needs to study and reproduce the bug in more detail. Chapter 2 summarize all of the mentioned things and gives theoretical information about web GUI development and Selenium testing.

Chapter 3 describes the design of the final application. It defines all needed pieces of the application and how they need to be related to each other. Besides, it defines what exactly all pieces should do to get to the final goal.

Chapter 4 contains all information about the implementation of the application, and Chapter 5 evaluates the result of the implemented tests on all extensions in the AMO. It shows how many extensions are influenced by the bug.

At last, Chapter 6 studies the problem of how web site owners can fingerprint visitors using the bug in Firefox. Further, it describes some future improvements of this work to increase the application performance and testing accuracy.

Chapter 2

Theory

This research includes work with different tools related to developing and testing browser extensions. Further, it concerns the security of web pages and problems occurring in the Mozilla Firefox browser.

Since different user categories can use this work, for example, extension developers, Firefox developers, or ordinary users, it is essential that the final application needs to be easy to use and easy to deploy. It is necessary to know how to implement a web application GUI, databases, and communication between services with minimum user interaction.

The research requires some theoretical and practical knowledge to get to the final goal. This chapter takes care of the first category and describes things needed to learn to understand the main problems of this work and investigate them.

The subsection 2.1 explains what are XSS attacks, and methods that browsers use to protect web pages against them. Then, the subsection 2.2 describes the core of this research, the bug that needs to be studied and analyzed to understand how many Firefox's extensions are prone to it.

This chapter also concerns theoretical knowledge about the tools used in this work, such as Docker Containerization technology and Selenium.

2.1 Browser Security Defenses

The problem of this work is related to the security of web pages, and the main things needed to know are: what are XSS attacks, how hackers can use web site vulnerabilities to inject malicious resources, and which modern methods help to restrict or eliminate them?

2.1.1 Same-Origin Policy

The **Same-Origin Policy (SOP)** is a browser security restriction that controls scripts from one web page to access data of other web pages. By this policy, JavaScript code can read only the properties of windows and documents that have the same origin as the document that contains the script [3].

Web pages are in the same origin when they have the same *protocol*, *host*, and *port*. Based on that, different web sites (with different URI), have different origins. For example, *site.com* and *site.org* are under different ones. Moreover, a document loaded via the HTTP protocol has a different origin to a document loaded via the HTTPS protocol, even if they come from the same web server [3]. At last, if one host has two different ports, for example,

80 and 81, containing some resources, loading of the resources between services on that ports is prohibited by SOP because of different origins.

A web browser applies SOP only for scripts from different origins, but not for images, videos, or audio [4]. It is because browsers have a **cross-origin** concept to let developers use specific resources from different origins. There are three mechanisms to realize this: **Cross-origin writes** lets using of links to another origins, form submissions, redirections, **Cross-origin reads** allowing the reading of dimensions of an embedded image, actions of an embedded script, and **Cross-origin embedding** letting embedding such resources like `<img>`, CSS stylesheets, `<video>` or `<audio>`, and others[1]. JavaScript, embedded to a web page by the `<script>` tag and `src` attribute, works as well.

For some reason, web sites may need to somehow bypass SOP. For example, giant billboard web sites containing many advertisements for job search, houses rents, discussing forums might have multiple subdomains for each service. It may need to communicate between those subdomains by JavaScript, for example, in the browser's windows. But these actions are restricted by SOP. A web site can change its origin by changing of a subdomain to its superdomain[2].

Further, a server response header may influence and weaken the SOP policy from the server-side. It can be provided by activating Cross-Origin Resource Sharing (CORS) that brings the possibility to define whitelisted domains permitted to read data from the site and bypass the SOP. CORS can be set over Access-Control-Allow-Origin HTTP header by writing domains separated by space.

The Same-Origin Policy is a necessary mechanism that prevents stealing sensitive data or injecting malicious content into a web page. But the weakness of the SOP, especially resources that are allowed by SOP to write data into a page (`<img>`, `<video>`, `<iframe>`, stylesheets), might do actions causing XSS security problems. The next section describes that problems. Moreover, the enabled CORS may let attackers access and read data from a site. The **Content Security Policy** solves the mentioned problems. Section 2.1.3 explains the CSP in more detail.

2.1.2 XSS Attacks

XSS attacks (*Cross-site scripting*) are a type of injection or computer security vulnerability that allows attackers to inject malicious HTML or JavaScript code into a web site. XSS may occur when an application stores untrusted data into its storage without proper validation and escaping (data sanitizing). By this attack, a malefactor can access browser's cookies, session tokens, steal sensitive user data like authorization credentials, provide site defacement, or distribute malware.

For example, consider that a web page has a form to leave a comment below the article. This form has a field *author*, and it does not have any "sanitizing" of input data. All comments are public to all users, and anyone can see it in a web browser. If an attacker comes to a web page and wants to provide an attack, he may write malicious JavaScript code into that field. Code snippet 2.1 shows an example script that may be injected by an attacker. The website stores this script into its database and injects it into the DOM every time an ordinary user comes to the page. It causes an alert message to him.

[1] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[2] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#Changing_origin

```
<script>alert("I am an attacker");</script>
```

Code snippet 2.1: The snippet shows an example script that might be stored into the web site's database by an attacker. This code causes an alert message in the browser whenever a user comes to the web page.

It is a general and not dangerous example, but instead of this, it can be any script that can do anything. For example, it may steal a user's cookies or capture keystrokes on the login page and then send login credentials to the hacker.

XSS may occur in the following types:

- **DOM-based XSS** - Those types of attacks do actions entirely on the client-side, without the server [5].
- **Stored XSS** (*persistent*) - It is the most dangerous type of XSS attack. It may occur when a system stores malicious code on a web server. The injected code is executing when a user requests an origin web page where the code exists.
- **Reflected XSS** (*non-persistent*) - The most popular XSS attack. An attacker embeds a malicious code into a web page by injecting it typically in the URL of the web page. It may occur when web-client sends malicious data to the server, especially in HTTP parameters or HTML forms. Reflected XSS attacks can be carried out by sending spam emails to users that click on prepared by hacker link with the script in the HTTP parameters.

Cross-site scripting is one of the typical computer security problems. XSS attacks take 7th place in the top 10 application security risks [7].

2.1.3 Content Security Policy

HTTP **Content-Security-Policy** (CSP)[3] response header is an additional layer of security for web pages. It helps to detect, prevent, and report XSS and other code injection attacks.

Through CSP header, a web server can prohibit scripts execution from untrusted sources by specifying the domains and ports from which those scripts can come and execute. Applied CSP can restrict resources such as media files, fonts, web workers, images, videos, Java applets, and others.

CSP prohibits resource injection into a web page, but it goes against many of extensions behavior. There are extensions created to modify content of web pages, e.g., add buttons, remove advertisement banners, hide sensitive content. This extension's behavior should not be restricted. Policy enforced on a resource should not interfere with the operation of user-agent features like addons, extensions, or bookmarklets. These kinds of features generally advance the user's priority over page authors [10].

CSP mechanism is still developing. Nowadays, it has two versions: v2[4] (Recommended) and v3[5] (Working draft). All modern browsers almost fully support the second version of the mechanism. Only Firefox supports it partially. The third version is partially supported in Chrome, Firefox and Edge[6].

[3] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy
[4] https://www.w3.org/TR/CSP2/
[5] https://www.w3.org/TR/CSP3/
[6] https://content-security-policy.com/

The CSP header's value contains a list of directives that describes the policy applying for resources and sources those resources are coming from. Some of the essential directives are listed below:

- **image-src** - specifies the policy for images that are loading on a web site.
- **script-src** - specifies the policy for JavaScript code. It also can prevent the executing of inline scripts.
- **style-src** - specifies the policy for CSS styles loading on a web site.
- **default-src** - fallback directive for other ***-src** directives.
- **report-uri** - instructs the user agent to report attempts to violate the Content Security Policy.

One of the significant features of CSP is reporting. All CSP errors triggered on a web page may be reported to web site report logs. It helps to fast react to the malicious actions provided on the web site. CSP header has a directive called **report-uri**. It contains a URL of reporting servers to which all CSP reports come via HTTP POST in JSON format. Code snippet 2.2 shows an example of CSP report sent with help of `report-uri` directive.

```
{
  "csp-report": {
    "document-uri": "https://example.com/page/with/csp",
    "referrer": "https://www.test.com/",
    "violated-directive": "default-src self",
    "original-policy": "default-src self; report-uri https://report.com/store-report",
    "blocked-uri": "http://maliciousscript.com"
  }
}
```

Code snippet 2.2: Example of CSP report in JSON format sent with help of **report-uri** directive.

If a specific line or a specific file can be identified as the cause of the violation (for example, script execution that violates the script-src directive), the user agent **may** add the following keys and values to the violation [11]:

- **source-file** - The URL of the resource where the violation occurred,
- **line-number** - The line number in `source-file` on which the violation occurred.
- **column-number** - The column number in `source-file` on which the violation occurred.

The keys above are optional for a browser only in CSP 2 mechanism. The CSP 3 makes them required[7].

The **source-file** is a URL of the resource where the violation occurred. If a violation has occurred on Chrome's or Firefox's extension, the URL has the following structure:

[7] https://www.w3.org/TR/CSP3/#framework-violation

```
<browserName>-extension://<extensionUID>/<pathToResource>
```

Property `<browserName>` can be "**chrome**" for Chrome or "**moz**" for Firefox.

The next property `<extensionUID>` is 288 bit unique identifier of installed extension. Each installed or reinstalled extension has its UID. The uniqueness of `<extensionUID>` is needed to avoid user fingerprinting.

All resources of an extension such as scripts, images have their path in the directory structure of the extension. It is located in the `<pathToResource>` parameter of the URL.

Browsers use these unique resource URLs for accessing them in the filesystem. For example, browsers use this URL structure also for displaying the logo of an extension. Further, after the installation of "Privacy Badger" extensions into Firefox, the browser opens the extension's home page. This homepage is located on the client's local machine. Since the extension has its unique identifier, the browser uses it and gets the following URL to the page:

```
moz-extension://45cbd003-0307-7f44-ab34-1cfd4df2e5a8/skin/firstRun.html
```

The mentioned URL structure can be used for user fingerprinting. Web sites can detect the presence of a particular extension in visitor's web browser thanks to web accessible resources[8]. By accessing particular URLs, they can know if an extension is installed or not [6]. Sjösten et al. studied this problem. As a result of the study, around 28% (12154 out of 43429) of all Chrome's extensions and around 7% (1003 out of 14896) of Firefox's extensions have accessible resources and are detectable by the users fingerprinting method studied by Sjösten [9].

All modern browsers support the `report-uri` directive. But it is deprecated in CSP 2. CSP 3 replaces it with a directive called **report-to**. As CSP 3 is partially implemented in modern browsers, only Chrome and Edge now support the `report-to` mechanism. The difference between `report-uri` and `report-to` is that the second one allows set multiple groups with multiple endpoints. It means that CSP reports can have different groups. For example, CSP group that reports CSP errors or network group that reports network errors[9]. It also brings priorities of endpoints to distribute the load of servers. After detecting violating behavior, CSP can also trigger *SecurityPolicyViolationEvent* event over the `report-to` directive. An event handler may do anything a developer needs. It can send JSON to a given API or flush cookies and immediately log out the user.

Since Firefox supports only CSP 2 mechanism, this work uses only a `report-uri` directive for extensions analysis and testing.

Further, browser extensions use CSP, and all of them have it set by default. More information about using the CSP in browser extension development describes section 2.4.

2.2 Page CSP Influences Firefox Webextensions

Content Security Policy, described in the previous section, is a secure and flexible configurable layer of additional security for web pages. This section describes what is going wrong with the popular web browser because of CSP and how it restricts extensions developers

[8] https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web_accessible_resources

[9] https://www.yld.io/blog/security-trivia-series-understanding-csps-reporting/

from using some needed functionality. Important to mention that this concerns only CSP header specified on a server-side of a web page, but not extension's CSP.

How Server Specified CSP Header Influences Extension's Behavior in Mozilla Firefox Browser

As was specified in the previous section, CSP header returned by a web server in the HTTP protocol secures users against malicious executing of JavaScript by an attacker. The CSP header must contain a list of the allowed sources for all scripts loading on a web page.

Based on the CSP's official documentation, it can be applied only for resources and scripts on a web site's side, and should not concern scripts executing in browser's extensions. It affects scripts injected by an extension into the web page's DOM by, for example, `document.createElement('script')` JavaScript construction. The scripts have to be executed immediately after the injection[10], regardless of the CSP policy specified by a web server.

Even though the behavior described above is the right one, based on the CSP official documentation, Mozilla Firefox does it in another way. If an extension uses an execution of its content scripts into a web site that uses CSP header to deny all incoming scripts, the header rejects it. It triggers an error. And it is a browser's bug, which this work explores. The bug affects all Firefox's extensions that interact with web pages and use the execution of inline scripts into the DOM.

It causes problems and makes extension development more complicated. It also makes troubles with browser compatibility, and developers need to do overhead work to bypass the bug in Firefox.

The CSP header has the **report-uri** directive. Consider a web server with this option set on its web pages, and the server records all CSP errors to the server's alert log. Then, every user who has installed an extension that injects scripts into a web page triggers a false-positive CSP report or more reports. These reports may make noise to alert logs and spam reporting servers.

This bug was reported several times in official Firefox's Bugzilla and on official extension's issue trackers. Section 2.3 lists some examples of bug reports which give more information about the bug.

Bug Reproduction

This subsection describes the practical reproduction of this bug in Mozilla Firefox. It also explores how Google Chrome behaves in the same situation.

A part of the research is to create a simple web application and Firefox extension to show how CSP on the web page influences the extension.

The testing web site is expanded on an NGINX server combined with PHP-FPM and wrapped into a Docker container. It has one page with some text content on a white background. The HTTP response from the webserver returns the CSP header after a user requests the page. Setting up the header can be done by configuring the `default.conf`

[10] https://developer.chrome.com/extensions/contentSecurityPolicy#interactions

file. The server has the CSP configuration to prohibit all scripts, including inline scripts, by declaring an `add_header` property. Code snippet 2.3 shows how the configuration looks.

```
add_header "Content-Security-Policy" "script-src 'none'";
```

Code snippet 2.3: This snippet shows the server setting that configures the NGINX web server to send the Content Security Policy HTTP response header. The CSP is set to deny all script injections.

The main goal of the developed extension is to change the white background on the testing web page to green by clicking on a button inside of the popup window in the extension.

The extension creates a `<script>` node through the `createElement()` method of the `document` object. Through setting up the `innerHTML` property, extension sets the JavaScript code into the node and then appends the node into the `<body>` element of the testing web page.

As a result, Google Chrome behaves in the right way. If a user clicks on the button in the extension, white background changes the color, and the browser's console not prints any CSP errors. The reverse situation is in Mozilla Firefox. Something blocks changing of the background and console prints following error triggered in file `content.js` on line 4:

```
Content Security Policy: The page's setting blocked the loading of
a resource at inline ("script=src")
```

Based on the test described above, Google Chrome and Mozilla Firefox behave in different ways with the same browser extension installed. As said in the previous section, the right way is Chrome's one. And Firefox extensions need additional workarounds to bypass the bug.

2.3 The Target Audience of This Work

As was said in the introduction part, this research may be useful for different categories of users.

The main category is Firefox developers. The bug described above was reported in Firefox's Bugzilla several times, but the main one is report number 12670275 opened four years ago. It has long discussion about the problem and consolidates all information from other reports. But except issues in the Bugzilla, there are reports for this problem in official extension's GitHub repositories. These reports have long discussions and workarounds on how to solve or bypass the bug. For example, issue number 1793[11] on the GitHub repository of the "Privacy Badger" (PB) extension has a conversation about it. PB is a popular anti-tracking extension that helps to block invisible tracking, for example, trackers on links in social networks.

Moreover, the problem has triggered additional work on services not related to Firefox. The PB's issue has a comment by a developer of the "Report URI"[12] service, which is a monitoring system of CSP and other security features. He has reported: "Right now on *https://report-uri.com* we're constantly adapting our core filter set to remove reports like these so our customers see less noise, but overall it would be better to neutralise this at the source."[13]. Consequently, the bug caused much overhead work for extensions developers

[11] https://github.com/EFForg/privacybadger/issues/1793
[12] https://report-uri.com/
[13] https://github.com/EFForg/privacybadger/issues/1793#issuecomment-401279014

and other services that are not directly related to the browser. It takes time and money to bypass or adapt to this.

Web site developers may also find a piece of useful information in this research. The CSP header sends a CSP report to the site's statistics by using the **report-uri** directive if web page gets unexpected resource injecting. These false-positive reports caused by the Firefox bug may require excess administration or investigations, which also takes money and time.

Further, the work might be useful for extensions developers. They can use the testing application implemented in this research to compare extensions to identify and highlight differences in behavior between concurrent ones. It might be used for marketing purposes if one of the extensions found the solution to the problem and bypassed the bug.

Ordinary Firefox users can use this work to test an extension they want to install. The problem is that users may install an extension and not figure out that the bug influences it because most web sites do not have the CSP header set. The GUI developed in this work can help them to find needed extension and check if it has problems in the Firefox.

This research may be significant for Firefox users who care about privacy and data protection. Web site owners may use the bug in the browser for fingerprinting of site visitors. Section 6.1 gives more information about the study of this problem and description of some conducted experiments.

2.4 Extension's Manifest.json File

All extensions in Mozilla Firefox contain a file with metadata for the extension. This file is called **Manifest.json**[14]. It is a JSON file holding information about extension's version, author, description, browser action, default locale, content scripts, and many more properties. This section describes most related to this work ones.

Content Scripts

The most important key in a `Manifest.json`, related to this work, is the **content_scripts** key containing all of the extension's content scripts. Content scripts are files that run in the context of a web page to read information from the page or somehow changing it. They are using the Document Object Model (DOM) to do that [8].

The restriction of content scripts is that they can not access all of the WebExtension API because their purpose is to run in the context of a web page. They can send messages to extension's background scripts via messaging APIs to let them communicate with the rest of the extension. Since content scripts can communicate with the rest of the scripts in the extension, they may indirectly access full WebExtension API[15].

The key **content_scripts** is a type of array, and each item in the array is an object. Each object has required key **matches** containing a list of URL patterns. Besides, there are two not required keys on the same level as **matches**: **css** and **js**. They are a type of array containing a list of extension's CSS styles or JavaScript content scripts. When a user with installed extension comes on a web site, the extension can inject a content script only if the URL of the web site matches with at least one URL pattern from the **matches** array. Moreover, it can inject only scripts or CSS styles located in **css** or **js** arrays on the same level of **matches** containing the matched URL pattern.

[14] https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json
[15] https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts

Further, the object **content_scripts** might contain the key **run_at**. It has a type of string and can be one of the following: *document_start*, *document_end*, or *document_idle*. Those options correspond with the list of `document.readyState` (*loading*, *interactive*, and *complete*, respectively). The option **run_at** specifies when an extension injects scripts from the array **js**.

Content Security Policy

The CSP policy also exists on the extension's side to eliminate potential cross-site scripting issues. It does not connect to the CSP on the server-side described in the previous section.

All extensions have the CSP policy set by default. Code snippet 2.4 shows it.

```
"content_security_policy": "script-src 'self'; object-src 'self';"
```

Code snippet 2.4: The default CSP header for all Firefox extensions.

This policy eliminates evaluating of strings as JavaScripts, for example, the function `eval()`. Moreover, all inline scripts in tags `<script>` are not executed as well as event handlers like `onclick`, `onchange`, `onmouseover`, and others. At last, the default policy prohibits loading resources like `<script>` and `<object>` from sources that are not local to the extension. Developers need to specify the extension's own **content_security_policy** option in the `manifest.json` file to allow using all of the mentioned things.

2.5 Containerization

A big part of this work implements communication between processes and API calls. Because of that, the practical part of work intensively uses containerization technology provided by Docker[16].

Docker is an open-source containerization technology that allows automation of application deployment. Containerization is a form of operating system virtualization that enables the kernel of an operating system to support isolated instances of user space called **container**. Containers may look like OS inside of another OS. They allow deploying applications in a package with all needed dependencies and resources. From an application's point of view, a container is a real operating system, and all resources from the OS can be used (CPU, folders, networks, and others).

Unlike virtualization technology, containers use the host's kernel of operating systems. Based on that, all containers have to have the same OS kernel like on the host's machine. It can be an advantage because containers need to use fewer resources for running, unlike virtual machines, and also have shortened time for deployment.

By writing simple **Dockerfile** with instructions to execute, Docker can easily install most of the operating systems inside of a container. Besides, the same Dockerfile can have a configuration to install all needed dependencies, and the system can run any of shell commands as well.

This work uses more than one docker container because the application needs at least one docker container for webserver with web application and one with application to run Selenium scripts. There is a tool called **Compose**[17] for defining multiple Docker contain-

[16] https://www.docker.com/why-docker
[17] https://docs.docker.com/compose/

ers at the same time. It has the main configuration file called **docker-composer.yaml** defining all services needed to run. Compose builds and runs the services with one single command **docker-compose up**. The **docker-composer.yaml** file does not need to define all instructions to build single containers. It can refer to a Dockerfile which has all the needed commands.

Besides running multiple containers, Compose can keep a cache of built containers and use the cache when run rebuild command. Thanks to this, Compose rebuilds only containers with provided changes in the configuration. The rest of the data it takes from the cache. It also brings variables and extending of Compose files.

For example, consider a web application that has a web server (Nginx, Apache, Windows server, or others), cache system for its API (Redis, Memcached), and database as persistent data storage (MySql, Postgres, or others). Compose can manage all of these services. It builds each service based on one configuration file **docker-compose.yaml**. The Figure 2.1 visualizes the structure of that example application.

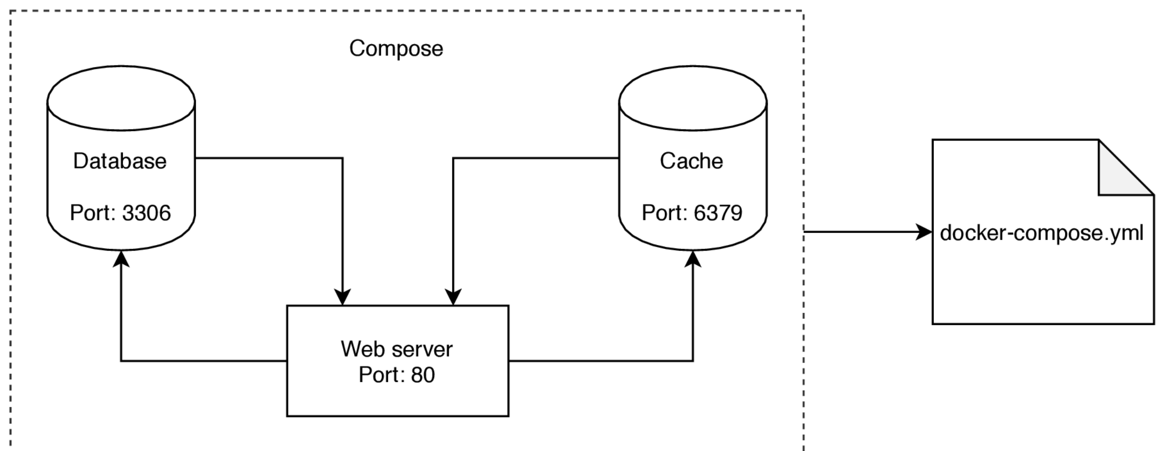

Figure 2.1: Docker Compose architecture of simple example application having a Web server, database, and cache system. All services defined in one **docker-composer.yaml** file.

2.6 Selenium

The main point of this work is to test as many as possible Firefox extensions that might be influenced by the bug described in the previous chapter. Since Mozilla Firefox currently has over 18 000 extensions available in the AMO[18], the problem is to automate that testing. This work is using Selenium Web Browser Automation Project.

Selenium is a tool which provides automation of web browser actions and emulates user-browser interaction. It uses the **WebDriver** interface to execute an action. The interface communicates with browser automation API. Selenium packs standard browser functions and hides their details into a “black box” to allow programmers to write code in a high level without performing complicated workflows.

[18] https://addons.mozilla.org/en-US/firefox/search/?type=extension

Selenium creates and sends a single HTTP request for each Selenium command[19]. Web driver uses HTTP server to get requests from a user and determines flow needed to execute commands in a browser.

The HTTP server listens for requests such as GET, POST, and DELETE [1]. GET requests are useful to get information from a browser. For example, getting text from `<input>` field. On the other hand, it needs POST requests to manipulate with something on the page. The automation of clicking on a button on a web page is a common example.

Usually, Selenium tests contain several commands needed to perform to test a single use-case on the page. Selenium uses sessions to perform stable and persistent communication. Executing of first Selenium command in script generates new session ID related to a single instance of a browser. Then, all commands in the same automation script sending the same session ID in their HTTP requests.

Selenium has the functionality to install an extension into a WebDriver instance by calling the method `install_addon(path/to/addon.xpi)`. This method has the parameter `path` containing the path to the archived source code of an extension. Section 4.4 gives more information about automatic extension installation.

[19] https://seleniumjava.com/2015/09/13/how-does-selenium-webdriver-work/

Chapter 3

Design of Extension Tester Framework

The goal of this work is to study and analyze all of the extensions in the Firefox browser. Since AMO contains over 18000 extensions, the research requires an automatic testing tool. This tool has to be a flexible framework that does everything from data collection to statistic presentation. It has to be able to download all extensions from the AMO, save it in a database, provide tests, analyze results, and show the final statistics.

This chapter takes care of the design of the framework used in this work. It also describes the needed architecture of microservices and the relation between small parts of the application. Further, it describes the database design and relations between all tables in the database.

3.1 Testing Scenarios Design

Mozilla Firefox has many extensions that do different things and work in different ways. Some do interaction with DOM before **onload** event, but some do it after particular action or user interaction. Moreover, some extensions work only for specific web pages and web sites.

There are different testing scenarios created to test all of the extensions in the AMO. This section describes the purpose and design of every scenario.

Manifest.json Analysis

Every extension needs to have a `manifest.json` file containing all necessary information and metadata for the extension. As describes section 2.4, the file contains information about extension's content scripts if the extension has any. The analysis may be used to retrieve content scripts and provide a detailed investigation to detect which extensions use script injection into the DOM of web pages.

Code analysis searches signs of script injecting into the DOM of a web page. The developed application has to get a path of a script, open it, and find signs. It could be, for example, the construction `document.createElement('script')`. A list of all strings that the framework tries to find in a content script is located in Appendix A.

The analysis helps to recognize those extensions, that may have potential risks to be influenced by the CSP. It may be useful for extensions developers, so they might decide not to use specific construction or bypass it somehow.

on_start_test

The described manifest.json analysis may find signs of script injection, but it can not prove the injection. This work needs to have the possibility to provide real tests and get real CSP reports.

This testing scenario is named **on_start_test**. It handles a set of extensions that inject scripts into a web page before the **onload** event occurs. In other words, before a web page finishes loading. Those extensions usually are from the category of hiding advertising banners ("AdBlocker Ultimate") or blocking invisible tracking ("Privacy Badger").

An example can be "AdBlocker Ultimate". This extension removes advertisement banners from the visited web page. If the CSP header denies an injection of the extension's content scripts, Firefox triggers two CSP errors right after the browser gets the response from the server. "AdBlocker Ultimate" of version **2.41**[1] triggers errors in the file preload.js on line 169. From the annotation of the function, which does this execution: "Execute scripts in a page context and cleanup itself when execution completes," it becomes clear that the extension tries to execute content script into the DOM by creating <script> HTML node. In this case, Firefox triggers a false-positive error, which sends a CSP report.

The **on_start_test** works only on the prepared testing web page that has CSP header set to deny script injecting. Selenium tests should navigate browser instance to this web page. Then, the web page generates CSP reports which are stored in the database.

The proposed test handles only general extensions working on all web sites on the Internet. Based on the description of **content_scripts** key in manifest.json, it contains the array **matches** holding a list of URLs on which content scripts should work. It means that there are extensions that work only on specific web sites, such as "YouTube AdBlocker" that works only on YouTube. The application needs to make an extension think that browser navigates to a specific web site, but in reality, it still accesses the testing web page. Todo so, the test needs to change the DNS configuration. For example, the application executes YouTube testing by running **on-start-test-youtube** as well as **on-start-test-twitter** for Twitter.

3.2 Design of The Main Parts of The Framework

Since the framework needs to do different things (collecting data, user interaction, data analysis, presentation), the development of the whole application as one service brings a problem with maintaining. This work uses a decomposition technique that facilitates the development and deployment of the whole application.

The application structure has two significant components that communicate with each other over the REST API: **front-end (GUI)** and **back-end**. The first one does everything connected with user interaction and graphical interface. Users of the framework could have technical knowledge (extensions developers, Firefox developers) as well as ordinary users without technical knowledge, who only want to see the statistic of influences extensions and decide to install it or not. Because of that, it is essential to make the GUI as friendly as possible to the second category of users. On the other hand, the backend does all data analysis and provides testing processes with Selenium. The described decomposition brings the possibility to divide user interaction with a web application on front-end and handling Firefox error logic on back-end onto separate and independent smaller applications. It brings a

[1] https://addons.mozilla.org/en-US/firefox/addon/adblocker-ultimate/versions/

possibility to develop each part independently. Frontend and backend are also divided into several smaller ones, which are described in the next sections.

Front-End (GUI)

The first component is a web interface. The idea for the web GUI is to give a user the possibility to run testing scenarios for a selected set of extensions. Besides, web application shows up users statistics after testing.

The application needs to download the most important extension's information and source code before a user can select it in the GUI and run tests. Since the AMO contains over 18000 extensions, an automatic process is needed to download all of them. The framework has some instruments to it. Scripts go through the AMO and download extensions one by one. The source code of an extension is packed into an archive and stored in data storage.

Local machine is not a good idea to store extensions files. The framework is located inside of a Docker container with the idea to deploy it on any machine. Pulling of about 10GB data every time Docker builds the application is an incorrect way because of possible poor internet connection or memory limit for Docker on the local machine.

The script stores all compressed extensions on Amazon Simple Storage Service (Amazon S3) in the bucket called **firefox-addons-tester**.

The web interface has to be easy and comfortable for users. The GUI should have a selectable list of downloaded extensions and a simple control panel to do actions for the extension. Users should be able to select a specific extension, or set of extensions, run tests, and see the structured result. Since there are a large number of extensions, the web application has to have pagination for all of them.

Database Design

The application does many tests and stores results in the database. This subsection describes the database structure and relation between its entities. Entity Relationship Diagram, which represents the structure of the database, is shown on the Figure 3.1.

The **addons** table is the central table in the database. It contains information about all extensions from the AMO. All rows in the table have properties such as the name of an extension, count of users using the extension, path to the extension's logo to render in the GUI. The `firefox_recommend` property has boolean value and contains information about Firefox's recommendation for the extension. If the value is `true`, Firefox recommends it.

The **addons** table has **One to Many** relation with a table called **csp_reports** containing all CSP reports triggered for specific extension after Selenium tests. The field `test_type` contains string value representing a type of test (`on_start_test`, `on_start_test_youtube` and others).

Further, the table **addons** has **Many to Many** relation with the table **sites** containing a list of sites that are used in statistics for representing on which sites a specific extension has potential CSP report because of the Firefox bug. The pivot table called **addon_site** helps to realize the relation. It also contains additional information about the relation between an extension and a site. The column `content_scripts_count` has information about count of content scripts, declared in extension's `manifest.json` for a specific web site. The next column, called `content_scripts_count_with_signs`, contains the count of content scripts that have signs of script injection in the code. It may mean that the analyzed script injects a script directly into the web page of a specific web site. All signs

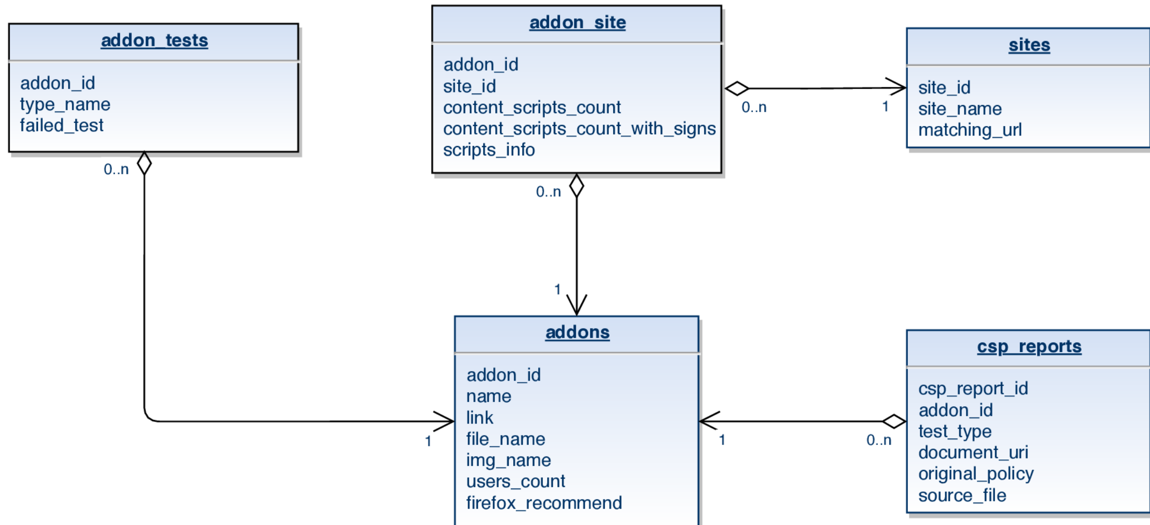

Figure 3.1: Entity Relationship Diagram representing the structure of the database containing information about all extensions, CSP reports, and web sites on which tests should be executed.

are listed in Appendix A. The implementation part of the work gives more information about the analysis.

The last needed table called **addon_tests**. It contains information about the provided tests. The table is connected to the specific extension and has a type of provided test and the column `failed_test`. This column stores boolean information and says if the test was successful or not. It is like a cache that improves performance by not repeating already provided tests.

Testing Backend

The second part of the application is named **backend**. It is an application without a graphical interface. The role of the **backend** is to get requests from the web application (**frontend**), provide Selenium tests or extension's code analysis, and return the result to the GUI.

Testing backend performs all testing scenarios introduced in Section 3.1. Firstly, the `manifest.json` analysis. The backend has to download needed extensions from storage, unpack them, do some preparation with the files and then provide the analysis of the `manifest.json`. Furthermore, it has to find signs of content script injection in extension's content scripts.

Moreover, the backend takes care of the execution of Selenium tests. After the backend gets a request for testing from the GUI, it has to download an extension and then run a Selenium test, which installs the extensions into the browser instance and provides needed testing actions. The DNS faking mechanism, described in section 3.1, also should be realized in the backend.

Chapter 4

Implementation of Extension Analyzer Framework

This chapter explains the implementation part of this work. It concerns architecture things with Docker containerization, explains the way how exactly single services communicate with each other via API, and also, most importantly, how exactly the application manages the testing process in the code.

4.1 Data Collecting

Almost every work related to testing, comparing, and data analysis requires some training or testing dataset. Since this work does not have any relation to machine learning, it does not need to generate or collect as various as possible data set for neural network training. But it requires a fixed set of downloaded extensions from the AMO. All extensions have to be present before the test started. Real-time downloading, when a test already started, is not the correct way because many changes can be submitted to the AMO by extension developers, and many extensions releases can appear during the test. It usually mixes AMO's items, and if an extension is on the 100th page, it can be, for example, on the 102nd page in an hour. It can cause not correct results for every test execution.

This work contains the creating of an automatic extension downloader. The parser is written in PHP language and uses a Simple HTML DOM Parser library[1] for web page scraping. It can extract HTML[2] from a web page and gives an interface to access single tags or elements in HTML by using CSS selectors[3]. It also can manipulate HTML in various ways, such as changing the content or moving DOM elements.

The web application in this work is created on the PHP framework Laravel. The section 4.3 describes the GUI implementation and using Laravel frameworks in more detail. Laravel has a tool named **Artisan** to write its command scripts. It is a command-line interface

[1]Simple HTML DOM Parser library `https://simplehtmldom.sourceforge.io/`
[2]Hypertext Markup Language (HTML) - `https://html.spec.whatwg.org/multipage/`
[3]`https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Selectors`

that helps writing CLI commands. It supports arguments, help section, description of the command, and other functions. Code snippet 4.1 lists two commands for the AMO parsing.

```
php artisan downloader:addons-info
php artisan downloader:addons-files
```

Code snippet 4.1: CLI commands that could be run for parsing the addons.mozilla.org. The first command extracts the primary information about an extension (name, image, and other). The second command only downloads the archived extension's source code.

The first command is going through all pages on the AMO and extracts information about every extension (name, link, users count). It can use parameters like `--start-page` and `--final-page` that define a specific range of pages needed to parse. The script also supports parameter `--download-file`, which triggers the downloading of source files (compressed to **.XPI** format) for each extension.

The second command is only responsible for downloading source files for an extension, which is already scrapped and stored in the database. It is going through the **addons** database table, and based on the link of the extension, downloads the file.

There is a pagination block that paginates all extensions on the AMO by 25 extensions per page. This `<div>` block is placed on the bottom of a page. The script parses this pagination block and extracts the last page number. Then it goes page by page and extracts the most important information for every extension: name of the extension, link to extension's main page on the AMO, icon, count of users using it.

Firefox has a set of extensions that are recommended by the browser. As writes the official Firefox web site, all extensions selected to participate in the Recommended Extensions program[4] are subject to ongoing re-evaluations to ensure they are functionally extremely well, safe, secure, and provide a delightful experience[5]. Every recommended extension has a mark "Recommended".

The script tries to find the mark on an extension's `<div>` block and extracts this information for every single one (1 - recommended, 0 - no information).

Firefox supports few ways how to install an extension from source files on a local machine.

The first way to install an extension from files is to install a temporary extension[6] by loading source code or **.zip** archive. The extension installed in this way is ready to use until the browser works and not restarted, or a user keeps it. This way is not suitable because finding the source files for every extension on the AMO is complicated for the amount of 18 000 extensions. It can be located on GitHub, on an extension's web site, and other resources.

The second way solves the problem described above. Firefox supports the installation of extensions from a **.xpi** file[7]. This file is a compressed installation archive which Mozilla uses in various applications such as Firefox, SeaMonkey or Thunderbird. Firefox uses a component named XPInstall[8] to install extensions archived into **.xpi** archive. Selenium supports it to do it automatically.

An essential part of scrapping from the AMO is to download the extension's source code. The scrapper needs a link to an extension to download it. This link can be easily

[4] https://support.mozilla.org/en-US/kb/recommended-extensions-program
[5] https://blog.mozilla.org/firefox/firefox-recommended-extensions/
[6] https://blog.mozilla.org/addons/2015/12/23/loading-temporary-add-ons/
[7] https://fileinfo.com/extension/xpi
[8] https://developer.mozilla.org/en-US/docs/Archive/Mozilla/XPInstall/Reference

Figure 4.1: Left (green button) - The button that appears on the extension's page in the addons.mozilla.org when a user uses a browser other than Firefox. Right (blue button) - The button that appears on the extension's page in the addons.mozilla.org when a user uses Firefox browser.

copied manually from the "Add to Firefox" button, but it is not that easy for the automatic process. Firefox renders two buttons: for Firefox users and non-Firefox users. The Figure 4.1 shows how the buttons look. The green button does not have a link to the **.xpi** file. Otherwise, the blue one has. The problem is that the green button is the default, and the Simple HTML DOM library cannot define the current browser type when parsing the page.

The script needs to somehow "trick" the Firefox webserver to get the correct button. If the server shows the button depends on the browser type, it needs to know that request was sent from Firefox. Code snippet 4.2 contains the header that should be sent by parser's CURL request to force Mozilla's web site to show proper button on extension's page.

```
User-Agent: Mozilla/5.0 (Macintosh; Intel...) Gecko/20100101 Firefox/75.0
```

Code snippet 4.2: An HTTP header that should be sent to the web server of addons.mozilla.org to force him showing "Add to Firefox" button when using extensions parser.

The scraper generates the name of a file from the name of extension with replacing all unsupported characters. Then, it concatenates string with a hashed link of the addon. It is needed in case if some extensions have the same name. The script uploads file into Simple Storage Service (Amazon S3) into a bucket. The last step is to store the extension's data into the database.

4.2 Docker Containerization and Architecture of Services

This work uses containerization technology with Docker to make deployment more comfortable and more flexible. It gives the possibility to separate logic parts of the application into isolated user spaces (containers). The application uses a tool named Docker Compose to define and run multiple containers that communicate with each other. Theoretical information about Docker and Compose and how containerization works is written in section 2.5.

The entry point is to define services is the **docker-compose.yml** file located in the root folder of the project. It is a configuration file that Compose uses to configure and describe how single services work, which resources they use, and how they relate to each other.

The web application uses PHP language in cooperation with the NGINX[9] web server. For handling dynamic requests on NGINX, the server communicates with PHP-FPM[10]. Further, the application uses a separate container with a tool for compiling CSS and JavaScript. The section 4.3 gives more information about the implementation of the web GUI.

[9] https://www.nginx.com/
[10] https://www.php.net/manual/en/install.fpm.php

Python application, which runs Selenium and test scenarios, is moved to a separate container called **backend**. Implementation of it is described in the section 4.4.

As a result, the whole program, including all smaller parts, is located in one repository. This repository managed by one **docker-compose.yml** file and products six docker containers communicating between each other under one shared network **bridge**:

- **nginx** - Web server
- **app** - PHP-FPM
- **db** - MySQL database
- **redis** - Caching software
- **backend** - Python application for running testing scenarios
- **css-watcher** - JS/CSS Compiler

Figure 4.2 visualizes all of the docker containers, and the structure of Docker Compose used in this work. The instances inside of the "Docker engine" block are single containers listed above, and arrows between them show their relation. For better maintenance of the building, some of the services have definitions in separate Dockerfiles, which the Figure below represents.

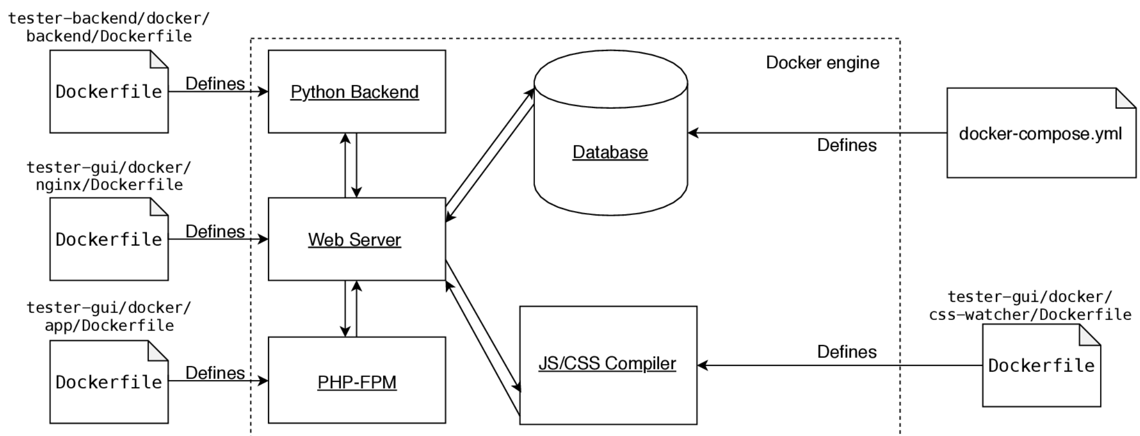

Figure 4.2: Blocks inside of the "Docker engine" represent application components (containers). Arrows show the relation and data flow between them. The Figure also shows which configuration file defines a component.

The next two sections of this chapter handle writing Dockerfiles, building scenarios, and relations between microservices.

4.3 Web Application

As was said before, the web application divides into four Docker containers. The main two of them are NGINX and PHP-FPM. NGINX is a web server that processes incoming HTTP requests.

Since this work concerns testing and processing large amounts of data, the application has to be able to handle a large number of incoming HTTP requests. A complication can be that number of extensions in the AMO is more massive every day.

The web server in this work handles two types of incoming HTTP requests. The first type is HTTP requests, which NGINX obtains to render web page resources like icons for every extension, CSS styles, and JavaScript scripts. Those requests are requests for static data. Another type of request is requests for dynamic data handling. It can be a request for storing a CSP report when it comes to CSP policy errors in Firefox. The Web server handles each report and stores it in the database. It requires specific dynamic handling by PHP.

Even when a client sends a dynamic request to a server running PHP, it is not the first point that contacts with the request. The first point is an HTTP web server. The server has to decide which way to use PHP to handle the request. Whenever the request was received, the web server creates a new process with PHP executed. NGINX can use PHP-FPM via the FastCGI protocol for that case. It means that NGINX does not need to know where PHP is on the server. In that case, PHP is encapsulated to another container with another environment, and the server uses it only for running scripts to handle the incoming dynamic requests.

With using of PHP-FPM, NGINX can handle static HTTP requests (CSS, JavaScript, images) by itself. It raises the speed performance of a web server. PHP-FPM processes only dynamic requests.

The Figure 4.3 visualizes how PHP-FPM works in combination with NGINX.

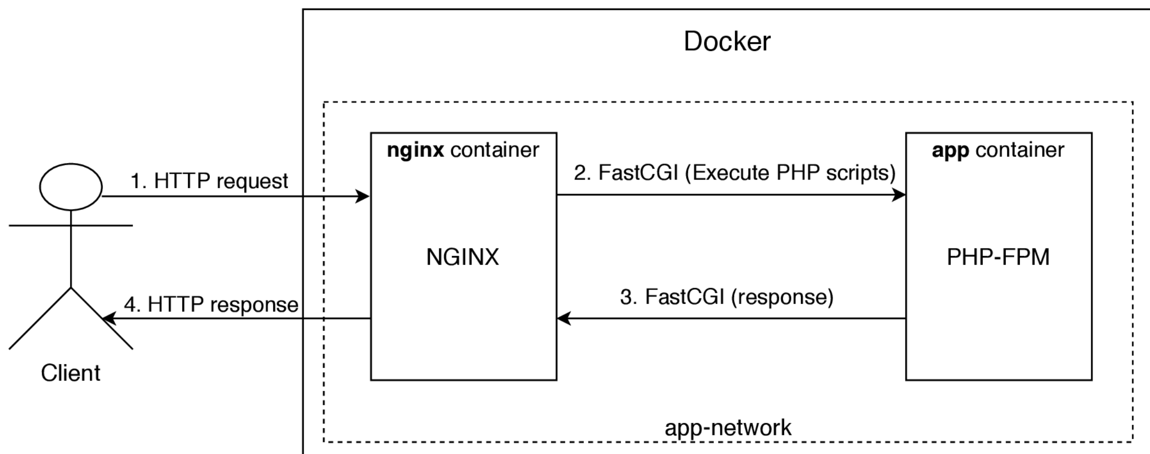

Figure 4.3: The relation between NGINX web server and PHP-FPM encapsulated into separate Docker containers.

The web site has over 18 000 addons stored in the database, and it is not an excellent way to show them all on one page, because of performance issues. A pagination feature can handle it. It is configured value how many addons to be shown on each page. This decision may cause an issue when a user selects addon by clicking on the checkbox and lose it after switching between pages. The application uses the browser's session storage[11] to solve this problem. It saves information about already checked extension, and JavaScript

[11] https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage

fires an event after the extension is checked or unchecked. Then it adds or removes the item from the session storage.

The web application uses a MySQL database to store all the needed data. It is served in a separate docker container with name **„db“**. The section 3.2 explains the database design and the relation between single tables more detailed.

The figure 4.4 shows the main page of the web application.

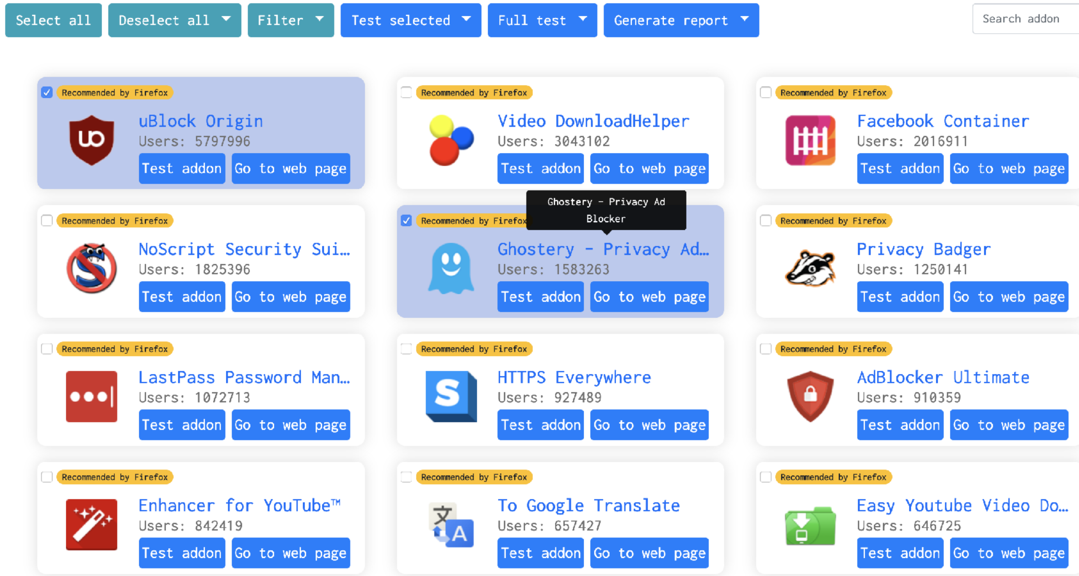

Figure 4.4: The screenshot shows the main page of the web application, which holds all selectable extensions. At the top of the page is located the control panel of the application. The control panel gives access to provide different actions with extensions such as filtering, running testing scenarios, or generating reports based on the tests.

Laravel Mix

The GUI uses SASS language to code CSS styles. SASS is a CSS preprocessor that helps to write complex and robust CSS stylesheets. It gives the possibility to use variables, selectors nesting, and other features. In the end, the SASS file compiles to a standard CSS file to use it in the application.

The web application uses a tool called **Laravel Mix**[12] to compile SASS files and JavaScript. The tool is part of the Laravel PHP framework, and it is a layer on top of Webpack. It gives a simple API for defining Webpack building flow and asset pipeline to compile javascript and SASS.

A part of this work is to create a simple Laravel Mix application. It is moved to a separate Docker container called **css-watcher**. The application requires the installation of NodeJS and NPM. By running the command `npm run watch` after installation in Dockerfile, Webpack watches for defined SASS and JavaScript files and recompile them whenever they are changed and saved.

[12] `https://laravel-mix.com/docs/4.0/basic-example`

4.4 Backend Testing Application

The **backend** service executes and runs testing scenarios that a user triggers through the web application. It is a separate application moved to a separate docker container called **backend**.

The backend is written in Python using the Flask[13] web application framework, a popular Python framework for creating services based on API or for web site development.

The advantage of Flask is that it is a lightweight web framework that allows the creation of APIs for HTTP communication without any additional installations or maintenance. The goal of the **backend** is to get the request for a testing scenario, run a Selenium test, and return a response.

There is a Dockerfile created to build the container, which hosts the application. It uses an **ubuntu:bionic** Docker image from Docker Hub to install the operating system first. After that, Dockerfile defines some applications to install, like Python, PIP, curl, wget, and others.

Selenium uses the **geckodriver** and Firefox of the latest version to run testing scenarios. The driver and browsers are downloaded by using of **wget** application in the Dockerfile.

The application uses the `requirements.txt` file to define all dependencies needed by application in a single **.txt** file, which was created and copied to docker container and immediately parsed by PIP[14] while the build is going through all commands in the Dockerfile. This process installs Flask and Selenium into the container.

The last step is to set some environment variables and run the server in the container.

4.5 Implementation of Testing Scenarios

This section describes the implementation of all of the testing scenarios, which are proposed in Section 3.1. Each scenario has different implementation and different relation between web application and backend, which runs Selenium tests.

The application stores every incoming report into the database in the table **csp_reports**, and it has a relation with the addon from the **addons** table. Information about the type of scenario for each extension is saved in the column **test_type**.

on_start_test

This type of test tries to handle and collect extensions that trigger the CSP report before the **onload** event occurs.

All that is needed to provide this test is to run Selenium tests with pre-installed extension and store triggered CSP report if it appears. Otherwise, if no error is triggered, the extension does not have a related CSP report in the **csp_reports** table, and the extension is considered as correctly working.

A user has to do one of the following to start this test:

- select an extension by marking checkbox ⟶ navigate to control panel on the top of the web GUI ⟶ click on the button “Test selected” ⟶ choose the test name in the dropdown menu

[13] https://palletsprojects.com/p/flask/
[14] https://pypi.org/project/pip/

- navigate to needed extension ⟶ click on the button "Test extension" ⟶ click on the *on_start_test* in the dropdown list.

The following explanation considers using the first way from the item list above.

The first point of the implementation is to send a request for testing from the web application to the backend by using JavaScript with AJAX (Asynchronous JavaScript and XML)[15]. When a user clicks onto the **on_start_test** button, JavaScript gets selected extensions from session storage. Then it iterates through all of them. For each item, JavaScripts creates AJAX request to the backend. The requests for testing carry some needed parameters such as the name of extension's file, extension's identifier, extension's name. These parameters are needed to execute tests on the backend side.

A CSP report, generated by the browser, does not support using custom parameters and does not have any information to tie up the report with the running test. That means that there is no way to figure out from the report what extension is in the testing process when it triggers an error.

All requests for testing **on_start_test** comes to the `/test/on-start-test` endpoint on the backend. After request comes, backend extracts extension's information, especially file name, and then call function `on_start_test_run()`. First of all, the function downloads the extension's file from the AWS S3 bucket and stores it on a local machine. On the next step, it creates a Selenium WebDriver instance that runs a browser on the background and then calls the Selenium function `install_addon()` to install the extension from the downloaded file into the browser. After that, by calling the function `get()`, Selenium navigates the browser to a prepared web page, which has CSP policy set to decline injecting of content scripts into DOM. This may trigger a **on_start_test** CSP report. After the page is loaded, Selenium closes the browser. The backend server returns a positive response to the web application.

The prepared web page for testing is located on the NGINX server and has its endpoint and controller that renders it. The backend sends two parameters to render the testing web page: `test_type` and `addon_id`. Those parameters are handled and validated by the controller, which then generates an HTTP CSP header before the rendering of the page. The controller places the parameters into the URL of the **report-uri** directive to tie up the running test with the report.

All CSP reports from the testing web page described above come back to the web application, which has prepared the `/api/store-csp-reports/{test_type}/{addon_id}` API endpoint to accept them. All incoming reports are handled by the application and saved into the database.

If everything is good, the application moves to the processing of the next extension. The full process is visualized on sequence diagram on the Figure 4.5.

DNS Faking

As was said in Section 3.1, there are extensions that work only on specific web sites. This work aims at testing the **on_start_test** on popular webs on the Internet. So, an extension for a specific web site needs to determine that the browser navigates to the web site where the extension works. But in reality, it still goes to the prepared testing web page. The test script has to change the DNS configuration to do this.

The testing web page is located on the web server in the **nginx** container. When Selenium navigates the browser instance, for example, to *www.youtube.com*, the HTTP

[15] https://api.jquery.com/jquery.ajax/

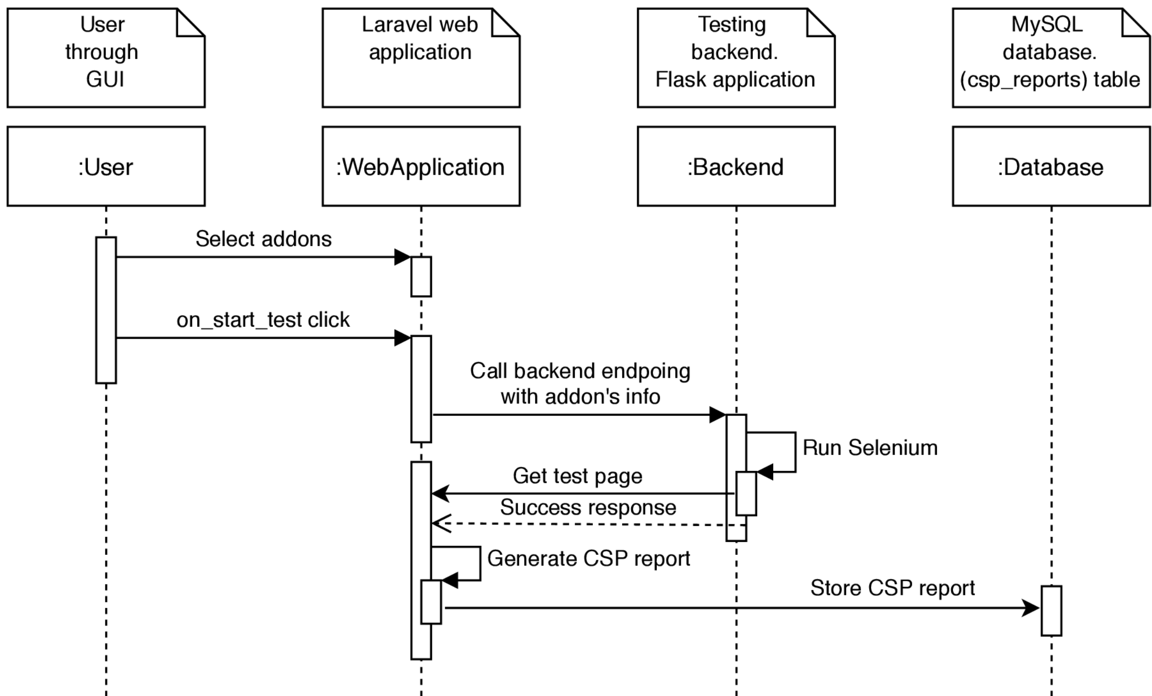

Figure 4.5: Sequence diagram of the **on_start_test** testing scenario. It visualizes data and request from between different actors of the application (User, GUI, backend, database).

request goes to the NGINX. DNS is mapping into the `/etc/hosts` file on the **backend**. After running Docker containers, a shell script gets NGINX's IP address and writes the mapping into the `/etc/hosts` file. Code snippet 4.3 demonstrates an example of domain mapping.

```
<nginx IP address> www.youtube.com
<nginx IP address> twitter.com
<nginx IP address> www.amazon.com
```

Code snippet 4.3: An example of mapping domain names on the right to IP addresses on the left defined in OS's /etc/hosts file.

When Selenium goes to *www.youtube.com*, the `/etc/hosts` configuration redirects it to `<nginx IP address>`, but an extension still performs actions accordingly to YouTube. For example, injects custom scripts.

Analysis of Manifest.json File

The processing of the `manifest.json` file is another type of extension analysis. The file contains information about the extension's content scripts. More information about `manifest.json` is described in the section 2.4.

A user of the web application can run the analysis for every extension from the list. Since all content scripts in a `manifest.json` file have to be related to a specific web site patterns (key **matches**), a user has to choose a web site for which he wants to provide

the analysis. The database table **sites** holds all web site URLs. Before a user starts the analysis, he has to choose web sites from the list on which the analysis provides.

The application takes all extensions and checked web sites from the session storage. Then, for each extension, it sends an HTTP request to *AjaxController* that converts data to a proper format and sends a request to the backend. On the backend's side, there is an `/test/content-scripts-analysis` API endpoint responsible for accepting such requests. After that, the backend downloads extension's file from the S3 bucket and unzips it. Then, it opens the `Manifest.json` file and reads the content via the `json.load()` function.

After analyses for all extensions were provided, there were many errors logged to the application's error log. These errors occur because of not supported by JSON specific symbols in Manifest.json. In the majority, it is "/" symbols at the start of the line. But commentaries with using slashes are not allowed in JSON format [2]. The application preprocesses the file before the opening to solve the problem. If the file still causes errors, the application skips the analysis of the addon.

One of the parameters coming to the backend is **sites_matching**. It contains a set of web sites for matching with the array **matches** in a `manifest.json`. The backend iterates through all sites in a loop, and for each one searches **content_scripts** item with at least one matching URL. The matching provides by using the **fnmatch**[16] Python library. It provides support for Unix shell-style wildcards.

The next step is to provide a code analysis for each content script separately. Then, the application iterates through all scripts from the **js** array and opens every file. It reads each line in the file and tries to find any signs of script injecting into a DOM. For example, it reads a line of code as a string and searches a `.createElement('script')` substring.

The backend returns all extracted information like count of content scripts, script injection signs back to the web application in JSON format. The web application converts it to the proper format and writes into the database in the table **addon_site**. So, the table contains rows with a paired web site and extension, and additional information about content scripts.

[16] https://docs.python.org/3/library/fnmatch.html

Chapter 5

Testing and Evaluation

This work aims to test all extensions from the AMO to detect how the CSP header influences the extension's behavior. The CSP header needs to deny the injecting of content scripts into the page.

This chapter explains the result of the provided tests. At the end of the chapter, there is a section that suggests and explains possible future improvements of the research, such as new test types, new GUI futures, performance raising.

5.1 Results of the Provided Tests and Analysis

This section shows and describes the result of the tests.

First of all, it is important to run the `manifest.json` analysis to mark extensions that may be influenced by the bug. The analysis processes the code of extension's content scripts and tries to find signs of script injection into a web page. Besides, it checks on which web sites there is the most significant number of influenced extensions.

Then, the top web sites from the list are subject to additional testing that discovers how many extensions trigger real CSP errors on a specific web site.

This section evaluates the results and represents them on graphs.

Manifest.json Analysis

This analysis is important to detect extensions that might have potential problems with Firefox's bug. Section 3.1 explains its purpose and design, and section 4.5 describes its implementation.

The `manifest.json` analysis was executed for all extensions from the **addons** database table and for all sites stored in the **sites** table. Figure 5.1 shows its result on a graph representing how many extensions have signs of script injection into the DOM of a web page. The graph shows a list of popular web sites listed on the Y-axis. Each tested web site has its value written in the middle of the related bar. This value means the lower bound of the count of extensions that would trigger false CSP reports if the CSP header was enabled on the web site. Almost all web sites were taken from the list of top 100 most visited websites by search traffic (as of 2020)[1].

It is important to mention that values from Figure 5.1, representing a count of extensions, are related only for the exact URL (protocol, port, domain) on the left side. For example,

[1] https://ahrefs.com/blog/most-visited-websites/

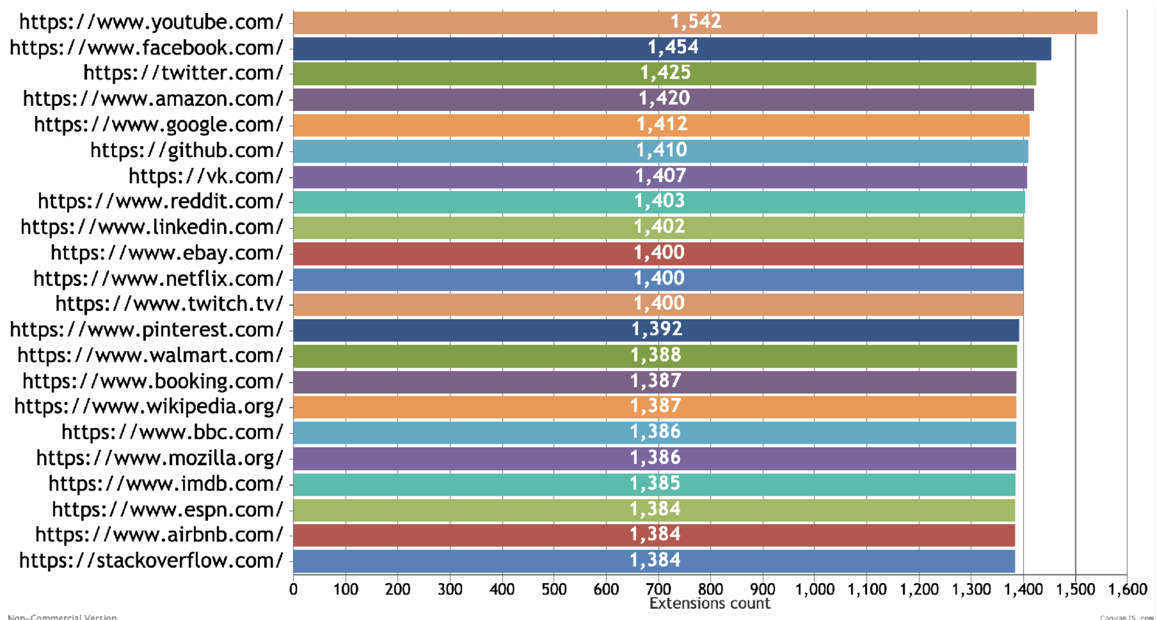

Figure 5.1: The lower bound of count of extensions that have content scripts injection signs in their source code. Each value on the middle of each bar represents the count for the URL, written on the left side, on which the manifest.json analysis for all extensions was executed.

the statistic does not contain extensions that work only on a specific YouTube channel that has a specific URL or only on the **.cz** domain. Therefore, any other URLs require separate analysis and may have a different result.

Some extensions are created for general purposes and do their job for all web pages on the Internet. For example, URL *https://espn.com/* from the graph on Figure 5.1 is a web site of popular cable sports channel ESPN. There are some extensions on the AMO that are created to work only on ESPN's web site, but no one of them triggers CSP error on the testing web page. Therefore, the value 1384 from the graph represents the only count of general extensions working on all web sites.

Each value on the graph consists of two parts: count of general extensions and count of specific extensions working only on the particular URL on the left side of the graph. Figure 5.2 shows a graph that represents the same statistics as on Figure 5.1, but only contains count of specific extensions. Based on the statistic, most of the extensions are developed specifically for YouTube. They can be different advertisements blockers, YouTube themes, or site transformation extensions. Facebook and Twitter take the second and third places, respectively. Since these three sites have more influenced extensions than all of the other tested ones, they need more deeply testing, separately.

In summary, 1384 extensions inject scripts to all web pages, and 490 extensions inject scripts to specific web pages. Hence, 1870 extensions (about 10% of all extensions) in the AMO are influenced by the bug.

As was said in section 4.1, Firefox has a set of recommended by the browser extensions. Figure 5.3 shows the same statistic as in Figure 5.1, but only for recommended extensions.

Based on the provided analysis for recommended extensions, 21 extensions are general, 5 are only for YouTube, and 3 for Google. Hence, together it is 29 extensions. Since the

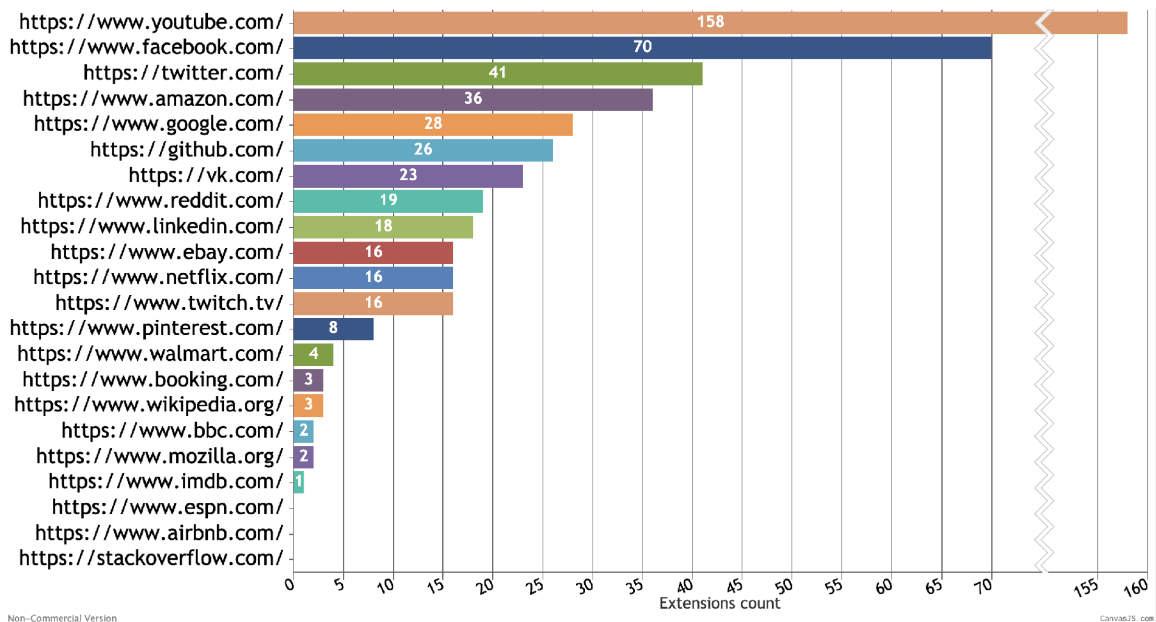

Figure 5.2: The lower bound of count of extensions that have content scripts injection signs in their source code. Each value on the middle of each bar represents the count for the URL, written on the left side, on which the manifest.json analysis for all extensions was executed. All general extensions that work on all web sites on the Internet are filtered out.

AMO has 100 recommended extensions, 29% of them have script injection signs and are potentially influenced by the bug. All of them are on the first four pages of the AMO.

on_start_test

This test carries out real experiments on installed extensions and tries to find a set of extensions that inject scripts and trigger CSP error before the **onload** event.

The `manifest.json` analysis has shown that 1380 general extensions in the AMO have signs of content script injection into a web page. These extensions are for general purposes and perform actions for all web sites on the Internet.

But there are extensions intended only for particular sites. They inject code only if the user's browsing web site URL matches with the extension's preconfigured URL pattern where the extensions should do its work. The analysis has shown that 158 extensions have signs of scripts injecting into YouTube, 70 into Facebook, 41 into Twitter. As was previously said, those extensions require more in-depth analysis.

This test explores extensions for general purposes as well as extensions for specific web sites. Based on the `manifest.json` analysis, there are 3 websites with more influenced extensions than other web sites from the list in the previous section: YouTube, Facebook, and Twitter. The test explores extensions only for these 3 categories and for a general one.

The test does not include extensions that inject a script only after some action on the page. It tests those that do it before the **onload** event. For example, it does not detect an extension that changes the background color of a page after a user clicks on a button.

Figure 5.4 shows the result of the test. After the test, it becomes clear that the result follows the order of web sites shown in Figure 5.2. Most extensions on the AMO trigger CSP errors on YouTube. There are 199 extensions that will cause CSP reports if YouTube

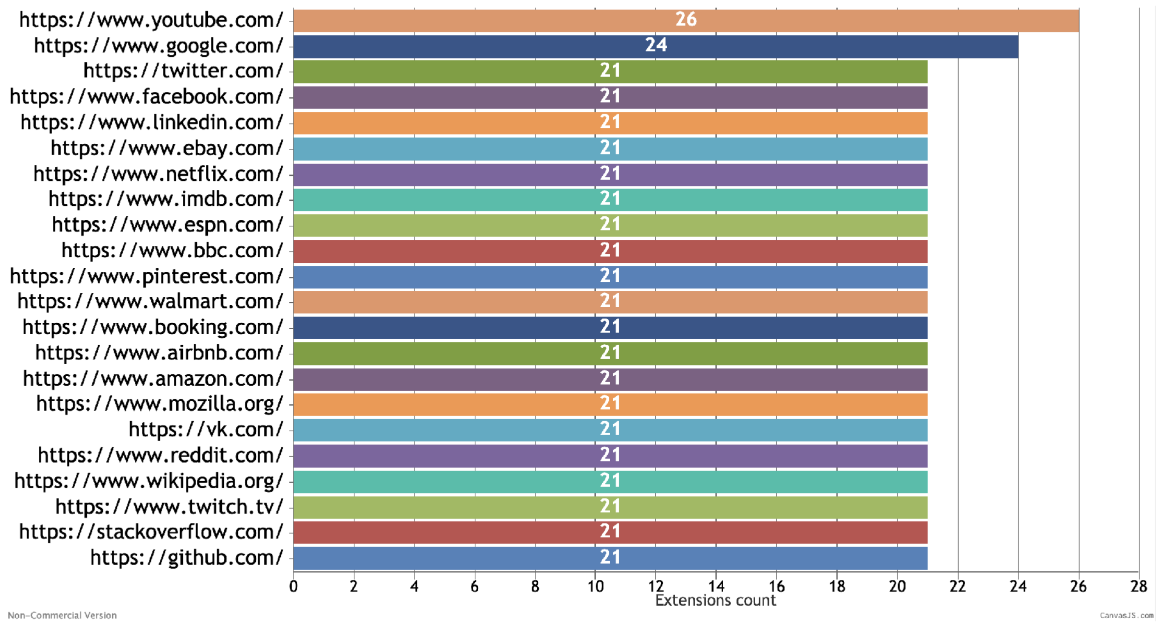

Figure 5.3: The lower bound of count of recommended by Mozilla Firefox extensions that have content scripts injection signs in their source code. Each value on the middle of each bar represents the count for the URL, written on the left side, on which the manifest.json analysis for all extensions was executed.

will decide to protect its web site against content script injecting by enabling the CSP. Facebook and Twitter have 194 and 184 extensions, respectively.

Summarizing all unique extensions created for specific web sites and all general extensions, there are 213 (1%) of them that trigger CSP errors passively (not doing any actions on a web site) on all web sites that have CSP protection enabled.

Moreover, every extension may trigger more than one CSP report. For example, the extension "LastPass Password Manager" triggers one report, but the extension "Emoji by TunisieSMS®" causes 18 reports after the browser navigates to the testing page. It means that every user that has "Emoji by TunisieSMS®" installed triggers 18 false-positive CSP reports, which make excess noise to the reporting log of a web site that has CSP protection enabled. On average, each influenced extension in the AMO causes two reports.

Figure 5.5 shows the same statistic but only for extensions recommended by Firefox. The report shows that the website order is the same as for all extensions, but the count of **on-start-test-twitter** is the same as **on-start-test** and equals 8. It means that all 8 extensions are general. Therefore, AMO does not store recommended extensions that do code injection at the start of page loading only for Twitter. The result represented on Figure 5.5 shows that 11 (11%) of recommended extensions passively trigger CSP errors after testing general ones and extensions created for YouTube, Facebook and Twitter.

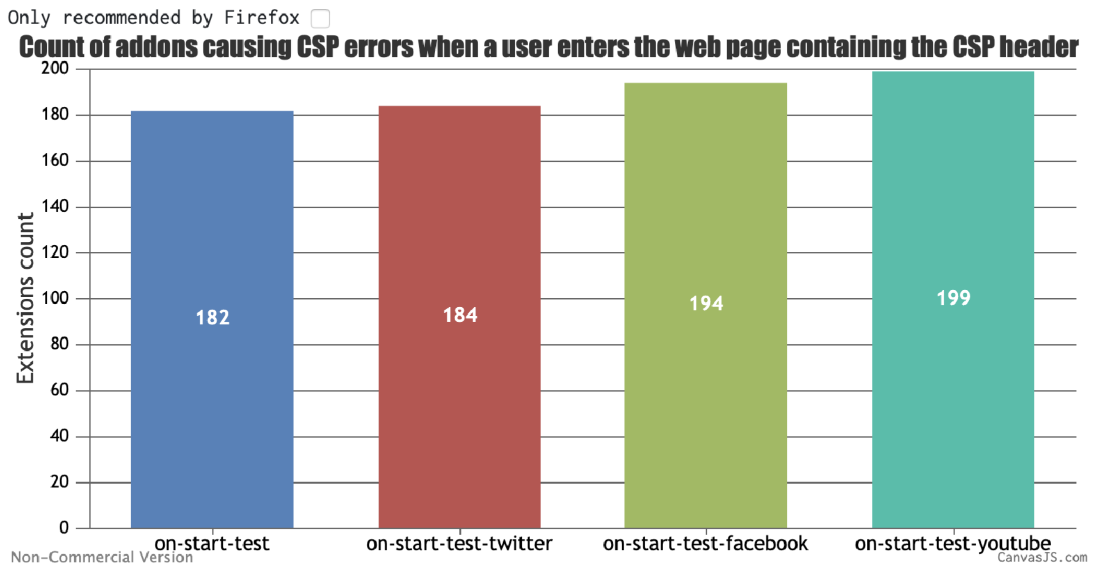

Figure 5.4: Count of Firefox extensions that trigger CSP errors after a browser navigates to a web page with enabled CSP protection. Tested for YouTube, Facebook, Twitter, and also general extensions that work on all web sites on the Internet.

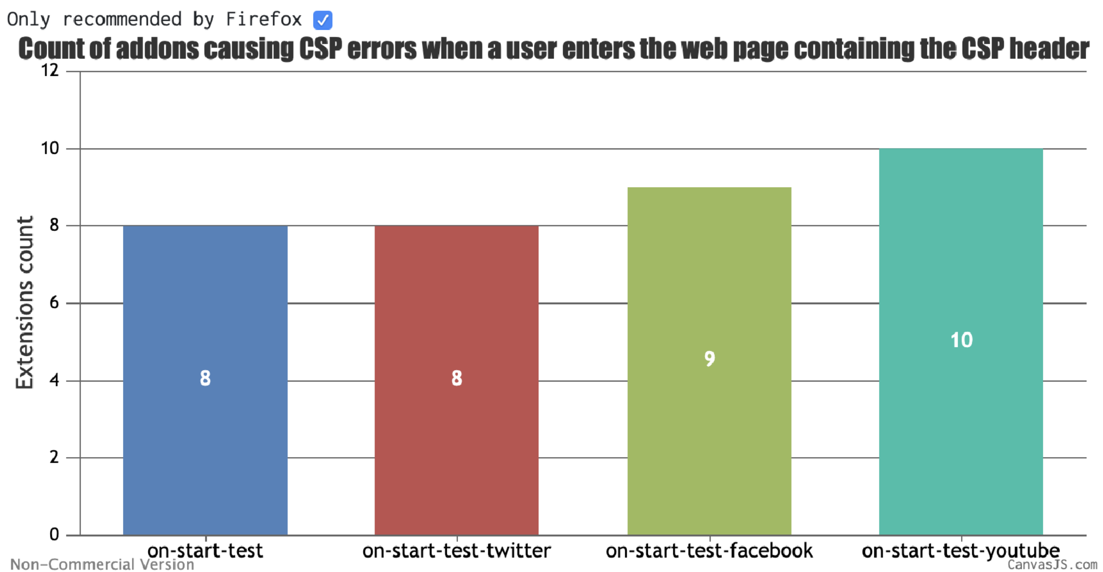

Figure 5.5: Count of recommended by Mozilla Firefox extensions that trigger CSP errors after a browser navigates to a web page with enabled CSP protection. Tested for YouTube, Facebook, Twitter, and also general extensions that work on all web sites on the Internet.

Chapter 6

Future Improvements of the Research

This chapter describes future improvements of this work suggested in Section 6.2. Some of them are improvements of the existing application, but some are additional testings that are needed to find influenced by the bug extensions that did not occur in existed tests. Besides, Section 6.1 describes how the bug in Firefox may be used to fingerprint site visitors and how this work may study it in the future.

6.1 Fingerprinting Problem

This research may be significant for Firefox users who care about privacy and data protection. Besides excess false-positive CSP reports, broken functionality of extensions, the bug in Firefox brings a problem with fingerprinting of web site visitors. If a web site has a CSP protection enabled to deny scripts injecting by extensions, triggered CSP reports may help web site developers to know which extensions site's visitors have installed.

As mentioned in Section 2.1.3, each CSP report generated by the report-to CSP directive provides the server the following:

- **source-file** containing the identifier of the web extension (EUID) and the path of the script in the extension's file hierarchy,
- **line-number** and **column-number**: identifying the position of code violating the CSP in the extension's source code.

Consequently, the bug in Firefox brings new possibilities to users fingerprinting. The UID allows cross-site tracking by designing CSP policy in a way that extensions are inserting script to web pages causing CSP violations. Extension's UID is unique for each extension installation, i.e., every installed extension instance has different EUID. Moreover, the UID changes when the extension is reinstalled. It means that a user with at least 2 web extensions causing CSP reports provides a long-term unique identifier because the tracker can learn that UID of an updated extension changed, and other extensions do not change their UID. As users usually do not update multiple extensions at the same time, the identifier is long-term.

A **column-number** key is a type of non-negative integer number containing information about the column's index in a code that violates the page CSP. It is the position in the code

where JavaScript calls a function that causes the violation. For example, most of the extensions inject scripts into the DOM of a web page using an `element.appendChild(script)` JavaScript construction. A `column-number` of this violation points to the "." symbol.

Table 6.1 contains `source-file`, `line-number` and `column-number` values extracted from CSP reports triggered by some popular Firefox web extensions.

	source-file	line-number	column-number
Privacy Badger	.../js/contentscripts/utils.js	35	9
LastPass Password Manager	.../onloadwff.js	71	798728
AdGuard AdBlocker	.../lib/content-script/preload.js	136	15

Table 6.1: Values of some CSP report keys by which extensions on the left side of the table may be identified. Each extension leave a unique imprint in CSP report by these three keys.

The source code of these extensions was manually studied to find violation constructions on the exact line and column of code. The constructions are listed below:

- Privacy Badger - `parent.insertBefore(script, parent.firstChild);`,
- LastPass Password Manager - `n.appendChild(t)`,
- AdBlocker Ultimate - `parent.appendChild(scriptTag);`

Some extensions use a unique naming structure. So, they are directly identifiable by the path of the script. But many extensions copy or follow some already existing conventions of file structure or file naming. Therefore, multiple extensions share the same script name and path. Many web extensions minify the JavaScript code into one line without whitespaces. As a result, key `line-number` has a value of "1" on all CSP reports caused by extensions with minified JavaScript code. There is a small chance that `column-number` values are the same for the extensions with minified code. Hence, the violating script's path, `line-number`, and `column-number`, in combination, is in practice unique for each extension. A tracker that fingerprint web site visitors can learn the values by monitoring AMO and triggering the violation. The database needs to be updated because the triple can change with a new version of an extension. One of the future improvements of this research is to create that tracker tool and try to identify extensions based on CSP reports in the **csp_reports** database table.

Fingerprinting over CSP reports does not need to use JavaScript like the fingerprinting based on web accessible resources described in Section 2.1.3. Hence, security extensions like "NoScript" do not protect the user from being fingerprinted.

6.2 Other Improvements

This section suggests a few new features that can be added to this work to improve the user experience or raise the accuracy of tests.

Copy of a Testing Web Site

This work uses DNS faking to provide simulate tests on web sites like YouTube, Facebook, or Twitter. Using this method, an extension behaves like the browser has navigated to one

of the mentioned websites, so it performs scrip injecting. But in reality, the DNS redirects the browser to the testing web page. It handles those extensions that inject content scripts without looking to the content of web site.

But some extensions inject resources depending on the content of the page. They may need not only a specific web page for code injection but also a specific content on the page. The content on the faked testing web page, currently used in the test, is different from the original web page. Because of that, an extension might not try an injection. For example, an extension injects a content script only if a web page contains a `<video>` element.

This improvement suggests to fake the content of YouTube, Facebook, or Twitter, or whatever site on which a user wants to provide a test. It can be done by saving the HTML code of a real web site and paste it into the testing web page. It may help to discover more extensions influenced by the bug.

Monkey Testing

The "on-start-test" test found 199 extensions that trigger CSP reports on YouTube. But the static `manifest.json` analysis has detected 1538 extensions. Even if an extension has code injection signs, it does not mean that it injects a content script on the start of the page loading. It may perform an injection after some action that a user does on a web page. For example, after the user clicks on some button.

A monkey test can partially solve the problem. This test can perform chaotic actions on a web site and wait for the right one that triggers a content script injection. Since extensions work on a specific web site, the testing web page should have the right content. So, this improvement requires the implementation of the previous one (Copy of a testing web site).

Improvements of Static Analysis and Application Performance

There are some possible technical improvements to the existed functionality. A big deal while the implementation was to provide parallel testing. It is normal to send from the frontend to the backend of 2-4 asynchronous requests to install and test an extension. But sending more requests causes errors or a very long time to finish the testing. The application needs to deal with it by implementing a cache system and getting more resources on a Docker Machine (RAM, CPU).

Moreover, it is possible to increase the success rate of `manifest.json` analysis. Now, the analyzer simply tries to find a substring in the script's code. But the code can be minified so that all values can be set to variables with one-letter naming. For example, the compressor may link the "document" object to a variable named "d". It is possible to use some existing interpreters and use semantic analysis of the code to discover that links.

Chapter 7

Conclusion

This bachelor thesis has a goal to test all extensions from the addons.mozilla.org (AMO) and show to extension and web site developers possible problems with the Firefox web browser. Firefox developers have to pay attention to this problem. Since this bug is reported four years ago in official Bugzilla, it caused many related bug reports in extensions issue trackers. These reports have long discussions about what is going on, and reports creators think it is a bug in the extension. But it is not.

First of all, this work requires to study the main principles of browser extension development. As the research is related to the CSP protection, information about it is studied to understand how the CSP works and which attacks it denies. Since this work can be used by ordinary users who want to check an extension they want to install, the application needs to have a simple and understandable user interface that requires additional study of GUI development and communication between services via API.

The design of the application requires to know how to build an application structure and how to design communication between microservices. Further, the work needs to design the testing process. Two main processes were designed. The first process is to do `manifest.json` analysis for all extensions in the AMO. It has to show top popular web sites on which most of the extensions trigger CSP errors. The second process is to execute real tests on previously-detected websites by using the Selenium tool. The work has designed a test type named "on-start-test" and derivative test for a specific web site such as **on-start-test-youtube** or **on-start-test-twitter**. These tests should detect those extensions that trigger CSP errors before the `onload` event occurs.

The implementation part of the work starts with the collection of needed data. It is required to create a Mozilla Firefox parser that goes through all pages on the AMO, extracts information about each extension, and then stores it into the database. It also downloads the compressed source code of an extension and uploads it into the AWS S3 bucket. The designed decomposition by microservices was practically realized by using the Docker Compose tool. As a result, the application has two main components that produce five Docker containers. Further, the implementation follows the designed approach of testing processes. Using web GUI, it is possible to run `manifest.json` analysis and "on-start-test" test for a selected set of extensions.

At last and most important, testing and evaluating processes were managed. Firstly, the `manifest.json` analysis was executed on over 18000 extensions. It showed that 1380 extensions have signs of content script execution. These extensions are from the "general" category, which means that they work on all websites on the Internet. Summarizing general extensions and extensions working on specific web sites, the analysis has detected 1870 ones

that have code injection signs in their source code, which is about 10% of all extensions in the AMO. As was investigated, the success of the analysis is 94%.

Further, the analysis showed that YouTube, Facebook, and Twitter are the most popular web sites where most of the extensions trigger CSP errors, more popular than all other 19 tested web sites. These three web sites were tested deeply by the “on-start-test” test. It showed that 199 extensions trigger CSP errors on YouTube, 194 on Facebook, and 184 on Twitter.

Bibliography

[1] Bastl, V. *Automatizace webového prohlížeče* [online]. 2019 [cit. 2020-02-10]. Available at: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=197316.

[2] Crockford, D. *The application/json Media Type for JavaScript Object Notation (JSON)* [online]. 2006 [cit. 2020-04-27]. Available at: https://www.ietf.org/rfc/rfc4627.txt.

[3] Flanagan, D. *JavaScript: The Definitive Guide, 6th Edition*. O'Reilly Media, 2011. ISBN 978-0-596-80552-4.

[4] Kemp, J. *Security on the Web* [online]. 2011 [cit. 2020-02-05]. Available at: https://www.w3.org/2001/tag/2011/02/security-web.html.

[5] Klein, A. *DOM Based Cross Site Scripting or XSS of the Third Kind* [online]. 2005 [cit. 2020-02-02]. Available at: http://www.webappsec.org/projects/articles/071105.shtml.

[6] Laperdrix, P., Bielova, N., Baudry, B. and Avoine, G. *Browser Fingerprinting: A Survey* [online]. 2020 [cit. 2020-05-27]. Available at: https://dl.acm.org/doi/pdf/10.1145/3386040.

[7] OWASP. *The Ten Most Critical Web Application Security Risks* [online]. 2017 [cit. 2020-02-02]. Available at: https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_(en).pdf.pdf#page14.

[8] Robie, J. and Research, T. *What is the Document Object Model?* [online]. 2011 [cit. 2020-03-25]. Available at: https://www.w3.org/TR/WD-DOM/introduction.html.

[9] Sjösten, A., Acker, S. V. and Sabelfeld, A. *Discovering Browser Extensions via Web Accessible Resources* [online]. 2017 [cit. 2020-05-27]. Available at: https://dl.acm.org/doi/pdf/10.1145/3029806.3029820.

[10] West, M. *Content Security Policy Level 3* [online]. 2018 [cit. 2020-01-20]. Available at: https://www.w3.org/TR/CSP3/.

[11] West, M., Barth, A. and Veditz, D. *Content Security Policy Level 2* [online]. 2016 [cit. 2020-01-20]. Available at: https://www.w3.org/TR/CSP2/.

Appendix A

List of Script Injection Signs

This appendix contain a list of script injection signs that the static `manifest.json` analysis tries to find in the extension's source code.

- `injectScript(`
- `insertScript(`
- `appendScript(`
- `insertBefore(script`
- `insertBefore(scrpt`
- `insertBefore( script`
- `insertBefore( scrpt`
- `appendChild(script`
- `appendChild(scrpt`
- `appendChild( script`
- `appendChild( scrpt`
- `.createElement('script')`
- `.createElement("script")`
- `.createElement(script)`
- `.createElement(scrpt)`
- `.createElement( 'script' )`
- `.createElement( "script" )`
- `.createElement( script )`
- `.createElement( scrpt )`

Appendix B

Content of Media

B.1 Source Code

- `tester-backend/` – Python Backend of the application,
- `tester-gui/` – Laravel web application (GUI),
- `test-extension/` – Simple testing extension to reproduce the Firefox's bug,
- `docker-compose.yml` – Configuration file to build all Docker containers,
- `run_clear_app.sh` – Script to build and run the application without data in the database,
- `run_final_app.sh` – Script to build and run the application with all data after tests,
- `mysqldump.sql` – Dump of the database. Used in `run_final_app.sh` script.