Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Computer Science

Browser Extension for Secure Web Development

Master's Thesis

Study program: Open Informatics
Field of study: Cyber Security
Supervisor: doc. Ing. Daniel Novák, Ph.D.

Michal Šípek

Prague 2022

MASTER‘S THESIS ASSIGNMENT

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

I. Personal and study details

Student's name: **Šípek Michal** Personal ID number: **474724**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Cyber Security**

II. Master’s thesis details

Master’s thesis title in English:

Browser Extension for Secure Web Development

Master’s thesis title in Czech:

Bezpečnostní plugin do prohlížeče

Guidelines:

Familiarize yourself with existing desktop tools (Burp Suite, Nikto, Grabber, Zap, jSQL Injection, etc.)
2. Familiarize yourself with the OWASP Top 10 list and OWASP's Web Security Testing Guide
3. Determine what limitations a browser extension poses in terms of automated reconnaissance and vulnerability scanning
4. Analyze the web security browser extension market
5. Design an extension for Chromium-based browsers that:
- can perform basic reconnaissance (detect certain libraries, frameworks, etc.)
- report vulnerable components
- detect certain vulnerabilities (e.g. XSS, CSRF, SQL Injection, ...)
- recommend remediation steps for the findings
6. Implement a proof of concept demonstrating each key part described above

Bibliography / sources:

Qian W., Xiangjun L. - Research and design on Web application vulnerability scanning service -2014 IEEE 5th ICSESS 2014, pp. 671-674, doi: 10.1109/ICSESS.2014.6933657
Esposito, Damiano, et al. - Exploiting the potential of web application vulnerability scanning - ICIMP 2018 13th ICIMP, Spain, July 2018, doi: 10.21256/zhaw-3927
Rennhard, Marc, et al.- Improving the effectiveness of web application vulnerability scanning -International Journal on Advances in Internet Technology 12.1/2 (2019): 12-27, doi: 10.21256/zhaw-17956

Name and workplace of master’s thesis supervisor:

doc. Ing. Daniel Novák, Ph.D. Analysis and Interpretation of Biomedical Data FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **08.02.2022** Deadline for master's thesis submission: __________

Assignment valid until: **30.09.2023**

doc. Ing. Daniel Novák, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

 © VUT v Praze, Design: VUT v Praze, VIC

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

DECLARATION

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ______________________ ____________________

Signature

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, doc. Ing. Daniel Novák, Ph.D., for his guidance and recommendations throughout this project.

Abstrakt

Nedávná globální pandemie ukázala, jak důležitá je přítomnost v online světě a tím i zdůraznila potřebu kyber-bezpečnosti. Tato práce si klade za cíl prozkoumat možnosti prohlížečových doplňků s ohledem na testování bezpečnosti webových aplikací. Dále je cílem navrhnout takový doplněk, který by mohl být používán pro snadné automatizované bezpečnostní testování webových aplikací, a mohl by tak zároveň zvýšit povědomí o běžných bezpečnostních problémech. Nejprve byla provedena revize existujících nástrojů, projektů a informačních zdrojů pro lepší představu o aktuálním stavu testování bezpečnosti webových aplikací. Práce se dále zaměřila na metody detekce problémů, ať už z pohledu relevantních standardů, metod používaných existujícími nástroji, či alternativních metod navržených výzkumníky. Probraná metodologie zahrnuje sběr informací, detekci citlivých dat, detekci zranitelných komponent a aktivní skenovaní zranitelností.

Dále byla provedena analýza Chromium doplňků z hlediska architektury, oprávnění a omezení vzhledem k metodám pro bezpečnostní testování webových aplikací. V důsledku nedávných změn ve specifikacích a implementaci standardů WebSocket a WebRTC byl navržen alternativní způsob skenování portů využívající vyšší oprávnění doplňků. Následně bylo navrhnuto a implementováno proof-of-concept řešení, na kterém byla demonstrována možnost využití prohlížečových doplňků k testování bezpečnosti webových aplikací.

Klíčová slova: webová bezpečnost, bezpečnostní testování, aktivní skenování, pasivní skenování, detekce zranitelností

Překlad názvu: Bezpečnostní plugin do prohlížeče

Abstract

The recent global pandemic highlighted the importance of online presence, as well as the need for good cyber security. This thesis aims to explore and propose a browser extension for automated web security testing that could be easily adopted by developers, raising awareness of various common security risks. Existing tools, projects, and resources were first reviewed to provide an overview of the web security testing field. It was then extended in terms of issue detection methods, including many of the associated standards, approaches used by existing tools, and novel methods proposed by researchers. Explored issues included information gathering, sensitive information detection, vulnerable component detection, and active vulnerability detection.

Chromium extensions were then analyzed in terms of architecture, permissions, and inherent limitations they pose for web security testing. An alternative solution to port scanning from within the extension is proposed, overcoming the recent developments in WebSocket and WebRTC specifications and Chromium's source code. A proof-of-concept extension is then proposed and implemented to demonstrate the viability of the browser extension environment for this purpose.

Keywords: web security, security testing, active scanning, passive scanning, vulnerability detection

List of Figures

2.1 Example use of nikto on a sample website. 15
2.2 Example use of nmap on a sample website. 15
2.3 Example use of traceroute and tracert on www.google.com. 15
2.4 Example use of ZAP on a sample website. 16
2.5 Example use of Wappalyzer on wordpress.com. 17
2.6 Example use of Penetration Testing Kit on wordpress.com. 19

5.1 Dashboard screen of Dolos extension visible in Developer Tools. 52
5.2 History screen of Dolos extension visible in Developer Tools. 53
5.3 Detailed view of a test run within Dolos extension. 54
5.4 Details of a specific finding reported by Dolos. 54

List of Tables

2.1 MITRE's CVE entry of the Heartbleed vulnerability (shortened). 6
2.2 List of enterprise tactics as stated in MITRE ATT&CK version 10 [25]. 7
2.3 NVD's CVE entry of the Heartbleed vulnerability (shortened) [26]. 8
2.4 Summary of attributes allowed in a well-formed CPE name structure [28, p. 11-13]. 9

List of Source Codes

2.6.1 WordPress ruleset definition in Wappalyzer. 18

4.5.1 Chromium's safe headers: services/network/cors/cors_util.cc 36

5.2.1 Manifest for the Dolos extension. 43
5.2.2 IModule interface used for defining new test modules. 46

Contents

1 Introduction **1**
1.1 Goal . . . 1
1.2 Problem statement . . . 1

2 Security Testing **2**
2.1 Introduction . . . 2
2.2 Open Web Application Security Project . . . 2
2.2.1 OWASP Top 10 . . . 2
2.2.2 Web Security Testing Guide . . . 4
2.3 MITRE . . . 6
2.3.1 Common Weakness Enumeration . . . 6
2.3.2 Common Vulnerabilities and Exposures . . . 6
2.3.3 MITRE ATT&CK Framework . . . 6
2.4 National Institute of Standards and Technology . . . 7
2.4.1 National Vulnerability Database . . . 7
2.4.2 Common Platform Enumeration . . . 8
2.5 Selected Weaknesses . . . 8
2.5.1 SQL Injection . . . 9
2.5.2 Cross-Site Scripting . . . 11
2.5.3 Cross-Site Request Forgery . . . 13
2.6 Existing tools . . . 14
2.6.1 Desktop tools . . . 14
2.6.2 Online tools . . . 16
2.6.3 Browser extensions . . . 17

3 Methodology **20**
3.1 Information Gathering . . . 20
3.1.1 Website Model . . . 20
3.1.2 Web Crawling . . . 21
3.1.3 Available APIs . . . 22
3.2 Software Detection . . . 23
3.2.1 Pattern-Based Approach . . . 23
3.2.2 Fingerprinting Approach . . . 24
3.2.3 JavaScript-Specific Approach . . . 24
3.3 Vulnerable Component Detection . . . 25
3.4 Sensitive Information Detection . . . 25
3.4.1 Personal Data . . . 26
3.4.2 Software-Related Data . . . 28
3.4.3 Used Approaches . . . 29
3.5 Vulnerability Detection . . . 30
3.5.1 SQL Injection . . . 30
3.5.2 Cross-Site Scripting . . . 31
3.5.3 Cross-Site Request Forgery . . . 32

4 Chromium Extensions **33**
4.1 Introduction . . . 33
4.2 Distribution . . . 33
4.3 Architecture . . . 34

4.3.1 Extension Manifest . . . 34
4.3.2 Background Scripts . . . 34
4.3.3 User Interface . . . 35
4.4 Permissions . . . 35
4.5 Networking . . . 35
4.5.1 Fetch API . . . 36
4.5.2 WebSocket API . . . 36
4.5.3 WebRTC API . . . 37
4.5.4 Lower-Level Protocols . . . 38
4.6 Storage . . . 38
4.7 Limitations . . . 39
4.7.1 Port Scanning . . . 39
4.7.2 Traceroute . . . 40
4.7.3 Storage . . . 40

5 Extension Design **41**
5.1 Requirements . . . 41
5.2 Architecture . . . 41
5.2.1 Used Technologies . . . 42
5.2.2 Project Structure . . . 42
5.2.3 Website Model . . . 43
5.2.4 Dolos Instance . . . 43
5.2.5 Modular Tests System . . . 46
5.3 Implemented Modules . . . 46
5.3.1 AdminInterfaceEnumerator . . . 46
5.3.2 CDNDetector . . . 47
5.3.3 DefaultCrawler . . . 48
5.3.4 DependencyHashLookup . . . 48
5.3.5 EmailDetector . . . 48
5.3.6 GUIDDetector . . . 49
5.3.7 JSStaticCodeAnalyzer . . . 49
5.3.8 NetworkAddressDetector . . . 50
5.3.9 NVDLookup . . . 50
5.3.10 PortScanner . . . 51
5.3.11 SecureConnectionAnalyzer . . . 51
5.3.12 SQLiDetector . . . 51
5.3.13 XSSDetector . . . 51
5.4 User Interface . . . 52
5.4.1 Dashboard . . . 52
5.4.2 History . . . 53
5.4.3 Settings . . . 53
5.4.4 About . . . 54

6 Conclusion **55**

References **56**

1 | Introduction

1.1 Goal

In the past years, we have seen the rise of multiple different frontend and backend frameworks and libraries. Web developers have a plethora of debugging tools for each of these frameworks at their disposal, usually in the form of a browser extension. These include official debuggers for frontend frameworks, such as React and Angular, SEO analyzers, plugin detectors, and more. Yet there seems to be a relatively low number of extensions that could be used for web security purposes.

The aim of this thesis is to investigate to what extent can browser extensions be used for automated web application security testing. That is, explore existing tools, summarize their approaches to automated detection as well as any alternatives, and determine what limitations may browser extensions pose with respect to automated security testing. Moreover, I shall design and implement a working proof of concept demonstrating some of the key aspects of such tool.

1.2 Problem statement

Since its invention in 1989 [1], the World Wide Web (or *web*) has become a major platform for delivering static content as well as interactive applications to the end users. The modern-day web developer has to choose between various components, such as specific languages or libraries and frameworks for those languages, with which they wish to use to build their application. However, the number of available components is evergrowing, making such decisions more complex while introducing new opportunities for bad actors to breach security of the application.

In recent years, affected by the global COVID-19 pandemic, we have seen the importance of online presence and subsequent expansion of e-commerce among other areas [2]. This made the web an even more lucrative target for hackers. Based on Verizon's 2021 data breach investigations, web applications became the number one attack vector with more than an 80% share in the investigated breaches [3]. However, the issue might be that cybersecurity either may not be company's top priority or it might be difficult to find suitable candidates for those positions. ISACA's 2021 survey revealed that 63% of responding enterprises had vacant cybersecurity positions and 60% had difficulties retaining employed professionals [4]. While companies may mitigate this issue by outsourcing or investing more resources into cybersecurity, there is also another alternative which may be implemented at the same time.

Raising web application security awareness among web developers may offer a suitable option to mitigate the aforementioned issues. There is a wide variety of already available tools for security testing, both paid and free, but their number may be overwhelming and not all of them may seem user-friendly. Moreover, the tools tend to specialize in a limited scope that may not cover the whole web application, making it necessary to use multiple tools in conjunction. For those reasons, I believe a browser extension might be the correct platform to target in this regards, since web developers tend to rely on them and their adoption is simple.

2 | Security Testing

2.1 Introduction

Unlike functional testing which verifies the correctness of implementation, security testing verifies if given implementation is secure [5]. And over the past years, several organizations have developed various resources to help with such testing. In terms of web application security testing, the Open Web Application Security Project Foundation is among the most important. If we consider a broader perspective, then also the National Institute of Standards and Technologies and the MITRE Corporation play a significant role in the cybersecurity space. This chapter aims to review their work that might be relevant to this thesis, as well as analyze existing tools for security testing.

2.2 Open Web Application Security Project

The Open Web Application Security Project Foundation (or the *OWASP Foundation*, alt. *OWASP*), founded in December 2001, aims to improve software security through community-led and open-source software projects and educational resources [6]. These include the *OWASP Top 10* list, *Zed Attack Proxy*, *Security Knowledge Framework*, and more.

2.2.1 OWASP Top 10

The OWASP Foundation publishes and maintains a list of the top web security risks called the *OWASP Top 10*. Its goal is to raise awareness and help developers address the most important issues first. The latest revision was done in 2021 and consists of the following 10 categories [7]:

- A01:2021-Broken Access Control,
- A02:2021-Cryptographic Failures,
- A03:2021-Injection,
- A04:2021-Insecure Design,
- A05:2021-Security Misconfiguration,
- A06:2021-Vulnerable and Outdated Components,
- A07:2021-Identification and Authentication Failures,
- A08:2021-Software and Data Integrity Failures,
- A09:2021-Security Logging and Monitoring Failures,
- A10:2021-Server-Side Request Forgery.

Each of these risks is assigned a specific position based on its importance and/or severity. While there are many other types of security risks, the OWASP Top 10 provides developers with a priority list and concise

summary of how each risk looks in practice and how to prevent it (including references to relevant sections of the Web Security Testing Guide). Adopting its recommendations is, therefore, a good first step towards a more secure web overall, as it raises awareness. In the same spirit, I collected examples for each category and summarized their description in the following paragraphs. While most of them are not covered in the extension design nor the implementation part of this thesis, understanding the real life examples may benefit the reader and inspire future work.

A01:2021-Broken Access Control refers to issues where the application has either insufficient or completely missing authentication or authorization in any part of the application. A practical example from this category would be the DEF CON 29's *Insecure Direct Object Reference* vulnerability reported by Brandon Forbes (also known as *reznok*) in 2021 [8]. In summary, the DEF CON organizers relied on a third-party provider for managing the ticket orders where each order was assigned a sequential numerical ID. If an attendee decided to see their order, the associated URL would have the following format: `https://provider.com/orders/*order id*`. However, there was no authentication happening when accessing the link which meant that reznok was able to access other attendees' order details by simply incrementing or decrementing the order ID.

A02:2021-Cryptographic Failures are cases where data is not encrypted when it should be or when the encryption used is not sufficient or incorrectly implemented (both in terms of the algorithm and its parameters). This was the case of OpenSSL v1.0.1 which occasionally sends out heartbeat requests to notify that the secure connection remains active. Each of these packets contains the length of the encrypted content that follows and then the encrypted content. The receiving machine confirms the connection by sending the data back. The issue was that OpenSSL did not verify the encrypted content had the length that the heartbeat packet declared, meaning that if the attacker specified a higher number of bytes than the content had, the server would return a larger part of its memory than it should have [9]. The additional information sent back could then contain sensitive data such as emails, passwords, or encrypted messages as was the case for Yahoo!.

A03:2021-Injection is possibly one of the most known categories to regular web developers as it encompasses SQL Injection among other weaknesses. More broadly, injections are issues where user-supplied data is not (properly) validated and is subsequently used in database queries, document templates, include directives, etc. One example of SQL Injection was discovered by Orange Tsai in Uber's unsubscribe feature. In order to unsubscribe from their advertisement emails, Uber would include a special link at the end of an email, containing a Base64-encoded JSON: `http://sctrack.email.uber.com.cn/track/unsubscribe.do?p=*B64 string*`. Among other fields, this JSON object contained a field named `user_id` which, if manipulated properly, could be used to perform SQL Injection attack on Uber's application. Orange's initial demonstration was based on changing the value from `"5755"` to `"5755 and sleep(12)=1"` and encoding the JSON string back to Base64. If then used in place of the original `p` parameter value, the request should have taken at least 12 seconds to finish which it did [10, p. 87-89].

A04:2021-Insecure Design is a collection of weaknesses which lie in the logic behind the application processes and architecture design decisions rather than their implementation [7]. One example of insecure design is storing users' credentials in plain text form. Usually, applications do not store the actual passwords but rather their hashes, meaning that, even if leaked, it should be difficult to retrieve the original passwords. Although, it is not impossible. This was the case of Facebook and, as later discovered, Instagram, allowing at least internally to browse millions of passwords [11].

A05:2021-Security Misconfiguration refers to cases where unnecessary features are enabled, default accounts with default passwords are left active, errors lead to exposure of stack traces, security settings do not have secure values, etc. [7] This can include weaknesses such as improper restriction of XML external entity reference which enables adversaries to perform so called XML external entity attacks (or *XXE attacks*) where the attacker exploits the document type definition's (or *DTD*) ability to reference other resources, both local and remote. If not restricted properly, an adversary may access local sensitive files. In 2014, such misconfiguration was discovered in Google's Toolbar button gallery. The gallery allowed developers to upload their own custom buttons which were expressed as XML documents and it would render them inside the gallery. However, it was possible to reference local files and display their contents which allowed the Detectify team to dump server's `/etc/passwd` file, demonstrating the severity of this misconfiguration [10, p. 111-112].

A06:2021-Vulnerable and Outdated Components category encompasses mainly such scenarios where the application uses a component (framework, application, etc.) that is known to contain a security vulnerability. This was the case of the US-based voice over IP (or *VoIP*) provider 8x8 which relies on the F5's BIG-IP software. In January 2020, there was vulnerability disclosure by F5 Networks, CVE-2020-5902, that stated multiple versions of its Traffic Management User Interface have a remote code execution (*or RCE*) vulnerability

present in undisclosed pages. In March 2022, Mehedi Hasan Remon (also known as *remonsec*) discovered and reported through HackerOne that 8x8 used a vulnerable version of said software [12] [13].

A07:2021-Identification and Authentication Failures are cases where the verification of user's identity is missing, is insufficient, can by bypassed, etc. It also covers password requirements, storage, and reset process. As part of this category, it is worth mentioning *credential stuffing attack* which relies on users reusing the same username-password combinations across multiple applications. If these credentials are leaked in a data breach from one application, they could then be tested on other sites as well [14]. This was the case of the international chain of baked goods and coffee named Dunkin' Donuts. In late October 2018, its parent company Dunkin' Brands Inc. noticed signs of a potential credential stuffing attack which could have lead to exposure of sensitive information of their customers. They forced the affected users to change their passwords in response [15].

A08:2021-Software and Data Integrity Failures are issues where the application's components or data are not properly verified both in terms of validity and authenticity. A famous example of such problem, and also referenced in the OWASP Top 10 list, is what is now known as the SolarWinds hack. In 2020, the US-based software development company SolarWinds published a new release of their product Orion, which is a network management tool that runs with high privileges and is used by thousands of organizations, both commercial and government. The problem was that this update was compromised by adversaries and contained malicious code. This lead to a large-scale supply chain attack - a type of attack where the bad actor indirectly compromises various targets by compromising a common tool used by those targets [16].

A09:2021-Security Logging and Monitoring Failures describe cases where the application either improperly logs events (e.g. includes sensitive information) or does not log them at all. Reusing an example mentioned in the OWASP Top 10 list, such was the case of British Airways in 2018. The adversaries were able to gain unauthorized access to their systems and access sensitive personal information of its customers. This breach remained undetected for more than two months and, based on the information stated by the Information Commissioner's Office, was detectable if sufficient logging was in place [17].

A10:2021-Server-Side Request Forgery, a community-voted category, focuses solely on the weakness CWE-918. The logic behind Server-Side Request Forgery (or *SSRF*) is that if an application processes user-supplied URL to fetch the data from given address and it is not properly managed, it can lead to internal network port scanning, sensitive information exposure, and more [18]. One notable example is Google's internal DNS SSRF reported in January 2017 by Julien Ahrens. The issue was found in G Suite Toolbox's dig utility which allowed users to query domain name servers by specifying the domain for which they wished to get DNS records and the IP address of the domain name server from where it should have pull the information. Except that Google did not verify if provided IP address is a private one or not, thus potentially exposing their internal DNS. Enumerating a whole IP range with the Burp Intruder tool, Ahrens was able to query an internal DNS server and brute-force a few internal domains to demonstrate the severity [10, p. 100-104].

2.2.2 Web Security Testing Guide

The Web Security Testing Guide (or *WSTG*) is a comprehensive set of guidelines on how to test a web application from start to finish. Its approach to testing is based on the black box testing model, i.e. it assumes the tester has limited, if any, knowledge of the underlying system and its codebase [19]. The guide is structured into the following twelve main sections:

1. information gathering,
2. configuration and deployment management testing,
3. identity management testing,
4. authentication testing,
5. authorization testing,
6. session management testing,
7. input validation testing,
8. testing for error handling,

9. testing for weak cryptography,
10. business logic testing,
11. client-side testing,
12. API testing.

Information gathering (or reconnaissance) is the initial, and possibly the most crucial, step in the testing process. The tester tries to identify what applications are running on given server, check if there is any sensitive information leaking, what might the underlying architecture look like, etc. When done manually, this could also include techniques like *Google dorking / Google hacking* which is a process of using specially crafted queries for Google Search that may uncover, for instance, if specific file types are present (like *.php*) which may provide hints to the application's architecture. WSTG also suggests to not limit these queries to Google Search only but rather use multiple search engines since they tend to produce different results, allowing for better coverage [19].

Configuration and deployment management testing aims to discover potential issues such as use of default configuration, unreferenced files with sensitive information, use of specific HTTP headers etc. For instance, if the reconnaissance step reported presence of the WordPress content management system (or *CMS*), here a tester would look for its default administration URL `/wp-admin`.

Identity management testing focuses solely on user account life cycle, including account provisioning, and user roles. Note that WSTG is approaching this problem as a black box test, thus it is recommended to collect more information before performing the actual tests. This would include the roles available in the application and their privileges. With the appropriate information, the tester would verify if the specification is consistent with implementation, whether it is possible to switch role by, for instance, cookie manipulation, and if account enumeration is possible.

Authentication testing verifies the strength and consistency of authentication logic. It builds upon several good practices for authentication, such as account lockout after a number of unsuccessful sign in attempts, strong passwords policy, avoiding use of default credentials, and ensuring passwords are transported over a secure channel. The procedure of password recovery, or password reset, is also covered in this step.

Authorization testing has two main focus areas - sufficient privilege checks and OAuth security. The former explores issues such as insecure direct object reference (or *IDOR*), privilege escalation, or being able to bypass authorization completely. The latter verifies the correct use of OAuth client and server.

Session management testing validates that the application is properly handling sessions from initialization to termination. Given HTTP's stateless nature, applications heavily depend on the identification provided by the user, therefore it is essential that this identification is secure. The general focus of these tests is on users not being able to impersonate one another.

Input validation testing is one of the more known steps to web developers. It is based around the idea of never trusting the user's input. The most common examples of attacks exploiting insufficient validation are SQL Injection and Cross-Site Scripting (XSS). The scope is, however, much wider than this, including XML Injection, HTTP request tampering etc.

Testing for error handling is performed to ensure the end user is not receiving sensitive application information when the application fails. While debug information makes developers' work easier when the code is failing, it can also help bad actors to better understand the underlying architecture of the application which is not desired. Thus, the tester would verify that the user is only presented with generic error messages rather than detailed logs or even stack traces.

Testing for weak cryptography is a step where the tester looks for either sensitive information being transferred through an unencrypted channel or with insufficient level of security.

Business logic testing, similarly to input validation testing, centers around the idea that incoming requests cannot be fully trusted and should be validated. This specific step verifies if the application, among other aspects, validates the user's flow through the application. For instance, that a user of an e-commerce site cannot place an order without first entering the shipping details.

Client-side testing ensures that the application cannot exfiltrate any sensitive information from the end user's browser nor can it force the user to perform actions on a different site unknowingly. Exploiting these types of

Field	Value
CVE-ID	CVE-2014-0160
Description	The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information (...)
References	http://www.securityfocus.com/bid/66690, http://www.securityfocus.com/archive/1/534161/100/0/threaded, http://www.us-cert.gov/ncas/alerts/TA14-098A (...)
Assigning CNA	Red Hat, Inc.
Date Record Created	20131203

Table 2.1: MITRE's CVE entry of the Heartbleed vulnerability (shortened).

vulnerabilities requires a specially crafted workload that could be passed in the URL, for instance. That way, the victim only needs to click a malicious link that could have the same origin as the legitimate website but with unexpected hash that would result in Cross-Site Scripting or HTML Injection.

API testing, in the scope of WSTG, is limited to GraphQL and analyzes most common exploits of such APIs such as Denial of Service (or *DoS*), injection attacks, or batching attacks.

2.3 MITRE

The MITRE Corporation is another non-profit organization focusing on, but not exclusively, cybersecurity. Its portfolio contains the following three main projects: Common Weakness Enumeration, Common Vulnerabilities and Exposures, and the ATT&CK Framework, all of which are essential to the cybersecurity field. Historically, MITRE also lead the Common Platform Enumeration project which was then transferred to the National Institute of Standards and Technology and thus will be discussed later in this chapter (see 2.4.2) [20].

2.3.1 Common Weakness Enumeration

Containing more than 900 entries, Common Weakness Enumeration (or *CWE*) is list of common types of weaknesses with potential security consequences [18]. Each entry contains a description, practical examples, mitigation steps, as well as its relation to other weaknesses and concrete vulnerabilities, and more. CWE's scope is not limited to software only but also targets hardware issues.

Among many different database views, the CWE project also includes a mapping to the OWASP Top 10 list (CWE View ID: 1344) and its own Top 25 Most Dangerous Software Weaknesses (CWE View ID: 1337). The general format of a CWE identifier is: `CWE-*number*`, where `*number*` is a non-negative integer. For example: `CWE-89` [18].

2.3.2 Common Vulnerabilities and Exposures

The aim of the Common Vulnerabilities and Exposures (or *CVE*) program is to catalog publicly disclosed security vulnerabilities [13]. Each CVE item consists of a short description, a list of references to the item, the authority that assigned an ID to the found vulnerability (also known as *CVE Numbering Authority* or *CNA*), and the date of submission. The general format of a CVE identifier is: `CVE-*year*-*number*`, where `*year*` is a 4-digit representation of the year and `*number*` is a 4- or more-digit number sequence.

As a continuation of the A02:2021-Cryptographic Failures example (see 2.2.1), we can examine CVE entry of the Heartbleed vulnerability in Table 2.1 [21].

2.3.3 MITRE ATT&CK Framework

The MITRE ATT&CK Framework (or *ATT&CK*) is an online knowledge base containing tactics and techniques utilized by bad actors in the real world used for thread modelling in enterprise environments. It contains

ID	Name	Description
TA0043	Reconnaissance	The adversary is trying to gather information they can use to plan future operations.
TA0042	Resource Development	The adversary is trying to establish resources they can use to support operations.
TA0001	Initial Access	The adversary is trying to get into your network.
TA0002	Execution	The adversary is trying to run malicious code.
TA0003	Persistence	The adversary is trying to maintain their foothold.
TA0004	Privilege Escalation	The adversary is trying to gain higher-level permissions.
TA0005	Defense Evasion	The adversary is trying to avoid being detected.
TA0006	Credential Access	The adversary is trying to steal account names and passwords.
TA0007	Discovery	The adversary is trying to figure out your environment.
TA0008	Lateral Movement	The adversary is trying to move through your environment.
TA0009	Collection	The adversary is trying to gather data of interest to their goal.
TA0011	Command and Control	The adversary is trying to communicate with compromised systems to control them.
TA0010	Exfiltration	The adversary is trying to steal data.
TA0040	Impact	The adversary is trying to manipulate, interrupt, or destroy your systems and data.

Table 2.2: List of enterprise tactics as stated in MITRE ATT&CK version 10 [25].

individual techniques (and their sub-techniques) that an adversary might use against an enterprise environment. ATT&CK then explains how given technique works, how to detect it and possibly mitigate it.

These techniques get grouped into so called tactics [22]. In the latest revision of the ATT&CK Framework (version 10), there are 14 tactics in total, each representing a different goal of the adversary (see Table 2.2). To better demonstrate the structure, let us consider a specific example in a bottom-up approach.

Vulnerability scanning is a *technique* (ID: `T1595.002`) where the adversary tries to gather information about victim's server and/or application to determine if certain vulnerability may be exploited. Per ATT&CK's specification, this technique has no easy *mitigation* but developers should minimize the amount and sensitivity of freely available information. In order to *detect* such attempts, it recommends to monitor for suspicious network traffic, like large number of requests coming from a single source, and its metadata. Vulnerability scanning is a *sub-technique* of a broader active scanning *technique* which is part of the reconnaissance *tactic* [23, 24].

2.4 National Institute of Standards and Technology

Founded in 1901, the National Institute of Standards and Technology (or *NIST*) is one of the oldest physical science laboratories in the United States of America. Among its core competencies is the development and use of standards. In terms of cybersecurity, NIST's main contribution is the National Vulnerability Database project which builds upon the previously mentioned CVE program maintained by MITRE (see 2.3.2).

2.4.1 National Vulnerability Database

The National Vulnerability Database (or *NVD*) is the U.S. government's repository of vulnerability data which is intended for automation. For that purpose, NVD exposes a JSON API then can be queried both with and without an API key - free of charge in both cases. The main advantage of an API key is a higher threshold for the number of request per minute the user can send [26]. For CVE information retrieval, NVD currently exposes two endpoints - one for detailed view of a single entry and one for retrieving a list of entries based on common parameters [27]. These endpoints are:

- `https://services.nvd.nist.gov/rest/json/cve/1.0/` (single CVE),
- `https://services.nvd.nist.gov/rest/json/cves/1.0/` (multiple CVEs).

Field	Value
CVE ID	CVE-2014-0160
CVE dictionary entry	https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
Current Description	The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets (...)
Severity	Base score: 7.5 (CVSS Version 3.x), Base score: 5.0 (CVSS Version 2.0)
Evaluator Impact	CVSS V2 scoring evaluates the impact of the vulnerability on the host where the vulnerability is located. When evaluating the impact of this vulnerability to your organization, take into account the nature of the data that is being protected and act according to your organization's risk acceptance (...)
References	http://advisories.mageia.org/MGASA-2014-0165.html, http://blog.fox-it.com/2014/04/08/openssl-heartbleed-bug-live-blog/, http://cogentdatahub.com/ReleaseNotes.html (...)
Weakness enumeration	CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer)
Affected configurations	cpe:2.3:a:openssl:openssl:*:*:*:*:*:*:*:*, cpe:2.3:a:filezilla-project:filezilla_server:*:*:*:*:*:*:*:*, cpe:2.3:h:siemens:cp_1543-1:-:*:*:*:*:*:*:* (...)
Source	Red Hat, Inc.
NVD published date	04/07/2014
NVD last modified	10/15/2020

Table 2.3: NVD's CVE entry of the Heartbleed vulnerability (shortened) [26].

While NVD is fully synchronized with MITRE's CVE program, the key difference is that NVD includes additional information about each vulnerability. Among other fields, it contains the platform enumeration with which the consumer can easily determine if their version and/or platform is affected by given vulnerability. An example of NVD's entry can be seen in table Table 2.3 which describes the same vulnerability (`CVE-2014-0160`) as Table 2.1.

2.4.2 Common Platform Enumeration

As mentioned previously, the key aspect of NVD is that it associates vulnerabilities with the software (and hardware) which is affected by given vulnerability. In order to provide this mapping, NIST maintains the Common Platform Enumeration (or *CPE*) naming scheme. The latest version, originally published in 2011, is CPE 2.3. The basic building block of this specification is a well-formed CPE name (or *WFN*) which is an abstract structure representing a specific product or a set of products. In its core, WFN represents a set of key-value pairs with specific constraints. The full list of allowed keys (or attributes) is described in Table 2.4. Per NIST's specification, each key may be used at most once and if not specified, it should default to ANY (i.e. "*") [28, p. 8-9].

However, WFN is only an abstract data structure. To transform it into a machine-readable format, NIST defines so called *binding* and *unbinding* procedures for (de)serialization of given WFN. Up to CPE version 2.2, the output of binding was a uniform resource identifier (or *URI*). The subsequent version 2.3 introduced a more relaxed *formatted string binding* with the following format [28, p. 31-32]:

```
cpe:2.3: part : vendor : product : version : update : edition : language : sw_edition
↪  : target_sw : target_hw : other
```

Such string can then be used when querying the National Vulnerability Database through its API to retrieve only those vulnerabilities that affect given platform(s).

2.5 Selected Weaknesses

With hundreds of weaknesses catalogued by the MITRE Corporation (see 2.3.1), it is unrealistic to cover them all. For the purposes of this thesis, I selected several weaknesses from various sources, including the OWASP

Attribute	Description	Value example
Part	Single-letter string identifying the targeted class of platforms - applications ("a"), operating systems ("o"), and hardware devices ("h").	"a"
Vendor	String identifier of the person or organization that manufactured or created given product.	"wordpress"
Product	String representing the most common or title of the product.	"wordpress"
Version	Alphanumeric string representing the specific version of the product.	"5.2.1"
Update	Alphanumeric string representing the specific update, service pack, etc.	"*"
Edition	Legacy attribute intended for the edition name. Should be left as ANY (*) unless specifically required for historical data.	"*"
SW_Edition	String representing how the product is tailor to specific audience or market (the edition name).	"home_premium"
Target_SW	String representing the software environment in which the product is supposed to operate.	"windows"
Target_HW	String representing the instruction set architecture on which the product should operate.	"x64"
Language	Language tag as defined by RFC5646 that corresponds to the displayed (and supported) language of the user interface.	"en-US"
Other	Any vendor- or product-specific identifier that does not fit anywhere else.	"*"

Table 2.4: Summary of attributes allowed in a well-formed CPE name structure [28, p. 11-13].

Top 10, Top 25 Most Dangerous Software Weaknesses, and my own experience. This list of weaknesses is mainly for demonstrative purposes, as the main focus is the feasibility of browser extension platform for security testing purposes.

2.5.1 SQL Injection

Among one of the most known injection attacks is *SQL Injection* (or *SQLi*). SQL Injection is catalogued by MITRE as CWE-89 and is ranked third in OWASP Top 10 for 2021 and ranked first in 2017 [7]. Such attack exploits improper validation of user input which could potentially lead to authentication bypass, sensitive information exposure, and more [18].

To give an example of a SQLi attack, let us assume we have a database table named `users` with the following simplified schema: `(id : UINT32, username : VARCHAR(256), password : VARCHAR(512))`. Let us also assume that, when a visitor inputs their credentials into the login form, the credentials get verified using the following SQL query that is constructed as a string at runtime:

```sql
SELECT id
FROM users
WHERE username = "$username" AND password="$password";
```

Where symbols with the dollar sign prepended get resolved to their respective variable values (such is the case of PHP for instance). The user gets authenticated if and only if this query returns a row.

Clearly, this form of authentication would work if the user just tried to enter their real credentials, assuming their username and password are simple alphanumerical strings. However, since the raw user input gets injected into the query *as provided*, an adversary might modify the query to suit their needs. If they, for instance, enter the following credentials:

```
$username = 'admin" #';
$password = 'does_not_matter';
```

Assuming the targeted database engine resolves the # symbol to a start of a comment, the final query would, effectively, be equivalent to:

```
SELECT id
FROM users
WHERE username = "admin"
```

Therefore, without knowing the real password, the attacker got authenticated as the user `admin`.

However, the approach above is not the only type of SQLi attack. In fact, there are multiple categories of such attacks based on the used technique [29]:

- tautologies,
- illegal or logically incorrect queries,
- union queries,
- piggy-backed queries,
- stored procedures,
- inferences,
- alternate encodings.

Firstly, **tautologies** rely on changing a filtering condition so that it is always satisfied, or a tautology. Consider the following code for verifying and retrieving information about a discount coupon on a fictitious e-commerce site written in PHP:

```
SELECT discount_amount
FROM coupons
WHERE code = "$code"
```

If the adversary was to pass a value of `" or "1"="1`, then they might have been able to get a discount on their purchase. Although, depending on the logic that follows next after this query, it might fail due to returning multiple coupons. Therefore, to extend this example, the attacker might actually prefer to use a payload such as this one `" or 1=1 ORDER BY discount_amount DESC LIMIT 1 #` to get the highest discount available.

However, to get the maximum discount we assumed the adversary had knowledge of the underlying query in order to construct the payload. To procure that information, they may first use **illegal or logically incorrect queries**. Such queries take advantage of the server exposing too many details when a query fails due to improperly handled exceptions or debugging mode being used in production. If they were to submit `"` as the coupon code, the query would fail. And, depending on the server's configuration and the code running the query, they might see the following error:

```
You have an error in your SQL syntax; check the manual that corresponds to your MySQL
↪  server version for the right syntax to use near '""""' at line 3
```

This, in itself, reveals to the attacker that SQL injection may be possible and that it is only a matter of trial and error to determine the correct malicious payload to achieve their goal. Also, this error discloses the database server which may imply certain exploits might be possible.

Once the adversary is able to determine the column name(s) being selected, they could also get an unlimited discount using **a union query**. Union queries take advantage of the `UNION` keyword or logic of many different query languages which allows combination of multiple datasets with the same columns together. Suppose the attacker submits the payload `" UNION SELECT 10000 AS discount_amount #`. The double quotes at the beginning ensure the original query does not return any row (assuming there is no coupon with empty code) and then the adversary creates their own result manually, in this case a `10,000`-unit coupon.

Next, let us extend our example to a scenario where the attacker wants to delete all existing coupons so that no other buyer may use them. This could be achieved using a **piggy-backed query** which exploits the

ability to execute multiple queries within the same request. The malicious payload could, for instance, be `"; DELETE FROM coupons #`. In that case, the database would try to return any coupons with empty code and then it would delete all rows from the coupons table. Note that the individual queries must be separated with a ; (semicolon) symbol. It is also important to consider that, under certain conditions, a piggy-backed query can replace a union query since the malicious command could also be a new SELECT clause as well.

An important feature of modern database systems are so called **stored procedures** which can also be the target of a SQLi attack. A stored procedure is a set of instructions, e.g. queries, defined in the database that can be called repeatedly with certain parameters, which can help with code reusability and maintainability and potentially performance if used properly. An adversary may either take advantage of either vendor-supplied procedures or procedures defined by the developer. In our coupon example, an attack on a stored procedure could happen if the website used a parametrized procedure to retrieve the coupon's discount amount but would essentially use the same query. The SQL query submitted by the PHP backend would then simplify to, for instance, the following:

```
CALL GetCouponDiscount("$code")
```

Making it vulnerable in the same manner as above but indirectly through a stored procedure.

The next category of SQLi attacks, **inferences**, can serve as a fallback option for when the application does not reveal enough information during an execution of an illegal query. The core logic of such attacks is to *infer* information based on various side effects of the executed payload. If, for instance, an adversary would first want to test the possibility of SQL injection in our coupon example, they could submit a payload like `" OR sleep(10) = 1 #`. They would not receive their discount but, if the request took more than 10 seconds to finish, they could tell that SQL injection is actually possible. This approach can be a good way to demonstrate to the owners that their site is vulnerable without actually causing any harm and/or exfiltrating sensitive data.

The last category, **alternate encodings**, is more of an additional step of a SQLi attack rather than an attack type. The goal is to avoid being detected by obfuscating the malicious payload using mainly type conversions. As such, alternate encodings get combined with any of the categories previously described. For instance, the website in our example could monitor the queries being passed to its database to see if it got the sleep command in a query and respond accordingly - do not execute, block the user, etc. Suppose that in our example, the server looks for the keyword DELAY and it is running MS SQL Server. Then, the adversary could submit the following payload:

```
"; DECLARE @q VARCHAR(64) = CONVERT(VARCHAR(64),
↪   0x57414954464F522044454C4159202730303A30303A30322E30303027);
EXEC (@q);
```

Where the hex string translated to ASCII is `WAITFOR DELAY '00:00:02.000'`, meaning that after executing the SELECT query, the database would wait for 2 seconds.

The main recommendation, when it comes to SQL injection, is to use prepared statements instead of blindly injecting unsanitized user input into the query. Such statements clearly define what parts of the query are the input parameters and which are the query code itself. This helps avoid SQL injection and can improve performance for queries that are run multiple times with only the parameters changing, as the query is temporarily saved in the database in a pre-compiled format.

Another appropriate approach which should be used in conjunction with prepared queries is to properly limit database user permissions to the bare minimum needed for it to function. In the context of our coupon example, this would mean that the account used for retrieving coupon information should only be allowed to run SELECT queries. That way, even if SQLi vulnerability is still present, it would minimize the impact the attack could have.

2.5.2 Cross-Site Scripting

Another well known weakness is Cross-Site Scripting (or *XSS*) which is a commonly used term for CWE-79. However, the original or full name of this entry in MITRE's database is actually *Improper Neutralization of*

Input During Web Page Generation which explains where the issue originates. In practice, an XSS exploit tries to inject malicious JavaScript code into a legitimate website which is then interpreted on the client and can redirect the user, exfiltrate cookies, and much more. Such attacks can be categorized into three main groups:

- reflected XSS,
- stored XSS,
- DOM-based XSS.

Each of these types corresponds to a different technique of how to deliver the malicious payload [18].

The first one, **reflected XSS** (or *type 1 XSS*), exploits applications which reflect HTTP request data in their subsequent HTTP response. The usual target for reflection is an HTTP GET parameter which gets sent as a part of the URL. Suppose we have a website `example.com` which has a search field to find articles by keywords. Upon submitting the search query for `xss`, the browser sends an HTTP request that may look like this:

```
GET /search?q=xss HTTP/1.1
Host: example.com
```

And the server would respond with a simple HTML document:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Search Results | Example</title>
        <meta charset="utf-8">
    </head>
    <body>
        <h1>Search results for: "xss"</h1>
        (... more content ...)
    </body>
</html>
```

We can see that the results page displays the original query we used for search. If, however, this query is not properly sanitized, an adversary may supply a specially crafted URL such as this one (URL-encoded form):

```
http://example.com/search?q=%3Cscript%3Ealert(%22XSS%20found%22)%3C/script%3E
```

Visiting this address would display the following document to the victim:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Search Results | Example</title>
        <meta charset="utf-8">
    </head>
    <body>
        <h1>Search results for: "<script>alert("XSS found")</script>"</h1>
        (... more content ...)
    </body>
</html>
```

meaning that the visitor would see a pop-up window saying the words "XSS found."

The second type, **stored XSS** (or *type 2 XSS*), relies on the adversary being able to inject malicious payload into the application's database or other type of data storage so that the application can later send it as part

of HTTP responses repeatedly. Suppose we have a social media platform that allows its users to customize their profile page by adding a headline, a title, or any other type of text on the page. For simplicity, let us assume this piece of text is transformed into HTML using the following PHP snippet:

```
<header>
<h1>Jon Doe</h1>
<h2><?= $userHeadline ?></h2>
</header>
```

Where `$userHeadline` contains the value pulled directly from a MySQL database. Let us assume it contains payload such as this:

```
<script>
    alert("Stored XSS")
</script>
```

In that case, every time a user comes to Jon Doe's profile, they see a popup box with the message `Stored XSS`. This is, for example, similar to what happened to the social media platform MySpace in 2010 with the infamous *Samy Worm* which made anyone visiting Samy Kamkar's profile instantly send a friend request to him [30].

Lastly, there is **DOM-based XSS** (or *type 0 XSS*). Unlike the above two, with this type the injection happens on the client side rather than on the server. With modern websites being more frontend-heavy and dynamic without constant reloading, this type becomes more relevant. To give an example, let us assume we have a single-page application (or *SPA*) with many different categories and sub-categories of articles. Since it is an SPA, the server will always return the same document and the JavaScript part will handle loading of the appropriate (sub)category or article. One solution to achieve this is with so called *hash routing*, which is a technique or an approach to dynamic page addressing by using the part after the # (hash) symbol in a URL for identifying a specific subpage of an SPA, since changes to that part of URL do not trigger a new HTTP request. If such application displayed a breadcrumb navigation by copying everything after the hash symbol from the URL without any sanitization by the client-side code, it could be vulnerable to a DOM-based XSS attack through that.

Depending on the use case, there might be various remediation techniques possible. A good first step is to escape all HTML entities in places where they are unwanted. However, this may not be sufficient in cases where the user-supplied value get interpreted as JavaScript due to, for instance, injection by design. Another solution is to implement a so called Content Security Policy (or *CSP*) which is defined through either an extra HTTP header `Content-Security-Policy` or its HTML meta equivalent. CSP can effectively block the root cause of XSS, i.e. execution of a JavaScript snippet embedded inside an HTML document [31].

2.5.3 Cross-Site Request Forgery

The last weakness selected for this thesis is Cross-Site Request Forgery (or *CSRF*), catalogued as CWE-352. In general, an application that does not employ any mechanism to verify that an HTTP request was intentionally sent by the user could fall victim to CSRF. Exploiting this weakness can vary in impact but, in essence, allows the adversary to perform certain operations on behalf of the victim - the specific actions will depend on the verification of associated requests [18].

Let us consider a badly secured social media platform that allows its users to send each other messages and upon submitting a message, the application's JavaScript code sends an AJAX request to the server. On the server's side, the following verification happens to test if the user is actually logged in:

```php
<?php
session_start();
if (!isset($_SESSION['logged_in'])) {
    header("HTTP/1.1 401 Unauthorized");
    exit;
}
```

If previously authenticated, the user would have a `PHPSESSID` cookie set which would get sent along with the request. However, this also applies if the user visits the attacker's site. In that case, they can launch several "send message" requests on behalf of the user immediately after loading the adversary's website which could be used for a phishing attack, for instance.

There are various ways of remediating this weakness. One popular approach is through so called **CSRF tokens**. If a user visits a website that uses these tokens as protection, they would usually *see* an additional hidden field. This field would contain a randomly generated string that acts as a temporary identifier provided by the server. Upon receiving a request, the server would test if the supplied token was, in fact, generated by itself and only then would it proceed with the action. Since these strings should be unguessable, it would make it practically impossible for the adversaries to perform associated action on the user's behalf - as long as this functionality is properly implemented [10, p. 34]. Another approach could be to utilize the `SameSite` attribute of the associated session cookie(s). That way, the browser would not send the cookie(s) in question if the request originated from a website other than the one where it was created.

2.6 Existing tools

Plethora of tools for security testing already exists, each being specialized to a certain scope, and they are available in different forms or for different platforms. My goal was to explore these tools, try to understand how they work and, most importantly, on what technology, protocol, or mechanism they rely, as this is going to be crucial to determine if similar approach can be recreated in the context of a browser extension. For the purposes of this thesis, I grouped them into three main categories based on their distribution platform - desktop tools, online tools, and browser extensions. For each group, there are several tools listed in no particular order and are either popular on a specific distribution platform (like Chrome Web Store) or recommended by the OWASP Foundation for security testing. Note that these tools represent only a small sample of the popular tools used for security-related testing.

2.6.1 Desktop tools

The first category to explore are the desktop tools, both CLI-based and GUI-based. While many of these tools target mainly Linux / UNIX-based systems, they can also be run on Windows either through a virtual machine or the Windows Subsystem for Linux (or *WSL*) environment.

Nikto is an open-source web server scanner written in Perl by Chris Sullo and David Lodge. Its main purpose is to detect outdated server version, version-specific issues, potentially dangerous files, and more. The architecture is centered around plugins, meaning for given server it will run all of the requested plugins where each one focuses on a specific area or issue. It contains many plugins one might need, but users are free to programatically extend Nikto as well. To the best of my knowledge, Nikto's functionality is based on making many HTTP requests and performing a rule-based analysis to determine presence of certain vulnerabilities or issues [32].

Nmap is an open-source CLI-based network mapping utility first released in 1997 by Gordon Lyon. It aims to scan both large networks, as well as single hosts. For each machine on a network it can determine what ports are open/filtered/closed, what services are running on these ports, brute force SSH credentials, and much more. Nmap works by sending raw IP packets and analyzing the responses - or sometimes the lack of responses to certain packets. To achieve this wide range of functionality, it requires the use of various networking protocols including TCP, UDP, and ICMP [33].

Traceroute (on UNIX systems) or **tracert** (on Windows) is a CLI diagnostic utility used for tracing packet route to a target machine. To achieve this, tracert uses specially crafted ICMP packets with increasing time-to-live (or TTL) value to determine what machines are part of the packet's route. This can be useful when, for instance, trying to detect firewall along the way [34].

SQLmap is yet another open-source tool written in Python that specializes in databases. More specifically, it automates SQL Injection detection and exploitation, and database takeover. It was originally published in 2006 and is currently being maintained by Bernardo Damele Assumpcao Guimaraes and Miroslav Stampar. While it mainly operates over the HTTP protocol, it can also directly connect to specific database management system over TCP if port is specified. Similarly to other tools, it has a set of plugins where each one is intended to

```
└─$ nikto -host localhost
- Nikto v2.1.6
---------------------------------------------------------------------------
+ Target IP:          127.0.0.1
+ Target Hostname:    localhost
+ Target Port:        80
+ Start Time:         2022-04-22 17:10:24 (GMT2)
---------------------------------------------------------------------------
+ Server: Apache/2.4.29 (Ubuntu)
+ The anti-clickjacking X-Frame-Options header is not present.
+ The X-XSS-Protection header is not defined. This header can hint to the user agent to protect against some forms of XSS
+ The X-Content-Type-Options header is not set. This could allow the user agent to render the content of the site in a different fashion to the
 MIME type
+ Root page / redirects to: /pages/homepage.php
+ Apache/2.4.29 appears to be outdated (current is at least Apache/2.4.37). Apache 2.2.34 is the EOL for the 2.x branch.
+ OSVDB-3268: /pages/: Directory indexing found.
+ OSVDB-3092: /pages/: This might be interesting...
+ OSVDB-3268: /Pages/: Directory indexing found.
+ OSVDB-3092: /Pages/: This might be interesting...
+ OSVDB-3092: /phpmyadmin/ChangeLog: phpMyAdmin is for managing MySQL databases, and should be protected or limited to authorized hosts.
+ OSVDB-3268: /static/: Directory indexing found.
+ OSVDB-3233: /icons/README: Apache default file found.
+ Uncommon header 'x-ob_mode' found, with contents: 1
+ /phpmyadmin/: phpMyAdmin directory found
+ OSVDB-3092: /phpmyadmin/README: phpMyAdmin is for managing MySQL databases, and should be protected or limited to authorized hosts.
+ 8491 requests: 0 error(s) and 14 item(s) reported on remote host
+ End Time:           2022-04-22 17:11:20 (GMT2) (56 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

Figure 2.1: Example use of nikto on a sample website.

```
└─$ nmap localhost
Starting Nmap 7.92 ( https://nmap.org ) at 2022-04-22 17:19 CEST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000045s latency).
Not shown: 999 closed tcp ports (conn-refused)
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
```

Figure 2.2: Example use of nmap on a sample website.

```
Windows PowerShell
└─$ traceroute www.google.com
traceroute to www.google.com (142.251.36.132), 30 hops max, 60 byte packets
 1  [illegible] ([illegible].1)  0.490 ms  0.437 ms  0.365 ms
 2  [illegible]8 ([illegible]8)  9.884 ms  10.175 ms  10.160 ms
 3  [illegible].cz ([illegible]7)  28.449 ms  28.436 ms  28.421 ms
 4  * * *
 5  [illegible]1 ([illegible]1)  28.258 ms  28.223 ms  28.197 ms
 6  [illegible].cz ([illegible].5)  28.175 ms  22.237 ms  22.212 ms
 7  * * *
 8  [illegible].net ([illegible]2)  27.615 ms [illegible]4 ([illegible]4)  28.856 ms [illegible]8 ([illegible]8)  28.863 ms

PS C:\> tracert www.google.com

Tracing route to www.google.com [2a00:1450:4014:80b::2004]
over a maximum of 30 hops:

  1     5 ms     9 ms     7 ms  [illegible].cz [[illegible]::1]
  2     *        *        *     Request timed out.
  3     *        *        *     Request timed out.
  4    29 ms     *       24 ms  [illegible]a
  5    39 ms    25 ms    27 ms  [illegible].cz [[illegible]::1]
  6    28 ms    28 ms    27 ms  [illegible]::1
  7    29 ms    27 ms    27 ms  [illegible]a
  8    30 ms    27 ms    27 ms  [illegible]7
  9    26 ms    41 ms    33 ms  [illegible].net [[illegible]4]

Trace complete.
PS C:\>
```

Figure 2.3: Example use of traceroute and tracert on www.google.com.

detect and test vulnerabilities for a different database system - for instance, MySQL or PostgreSQL [35].

Given the high number of websites running the WordPress content management system (or *CMS*), it is worth mentioning **WPScan** - a free CLI tool written in Ruby that specializes in vulnerability scanning solely for said CMS. It tries to detect issues not only in WordPress itself and its configuration, but also installed plugins and themes which tend to be a significant source of vulnerabilities. WPScan also covers a few more areas like directory listing, accessible database dumps, etc. To the best of my knowledge, it operates over HTTP only [36].

Among the GUI-based applications, one important example is OWASP's **Zed Attack Proxy** (or *ZAP*) which is an open-source penetration testing tool available for Windows, Linux, and MacOS. It serves as a vulnerability scanner and a web crawler if needed. ZAP ships with different types of add-ons which provide, among other things, the vulnerability scanning functionality. The main two add-ons in this regard are *Active Scan Rules* and *Passive Scan Rules*. In the case of passive scan, there are currently 29 different rules defined while active scan has 17 rules included. These cover issues such as SQL Injection, Cross-Site Scripting, or sensitive information disclosure, all done via HTTP [37].

The second GUI-based application is **Burp Suite**. It is a collection of various different tools with Burp Proxy

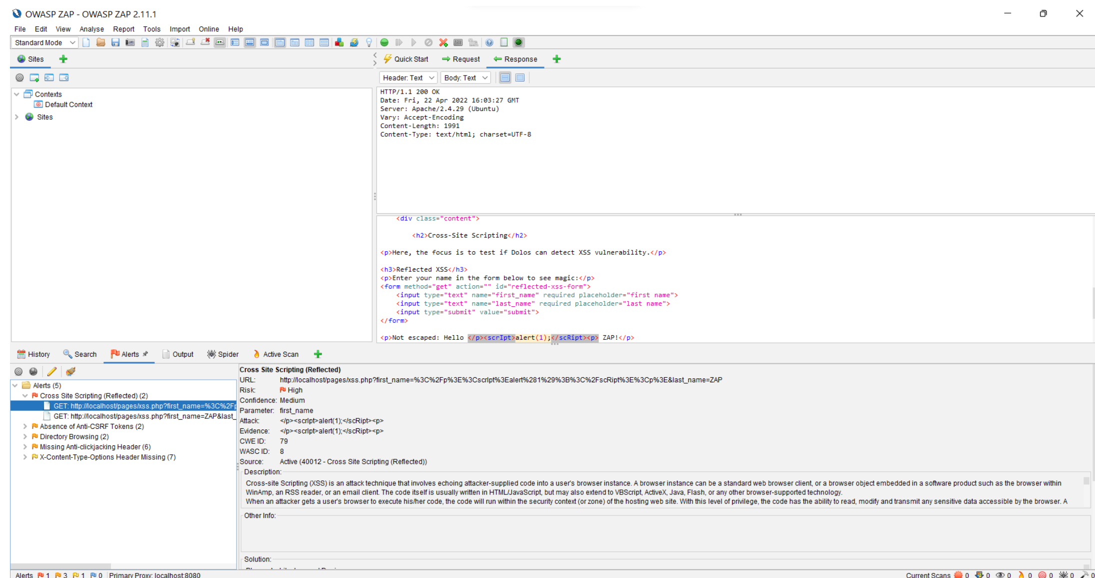

Figure 2.4: Example use of ZAP on a sample website.

at its core. However, it also contains a range of tools for automated testing. Unlike ZAP, Burp Suite has multiple editions, ranging from the free Community Edition to the paid Enterprise Edition. The free version does not contain a web application vulnerability scanner functionality. Apart from this, Burp's and ZAP's capabilities overlap to a high degree. Similarly to ZAP, it also relies only on HTTP [38].

The last GUI-based application, written in Python, is **Web Application Attack and Audit Framework** (or *w3af*). It aims to identify 200+ vulnerabilities, including SQL Injection, Cross-Site Scripting, and Cross-Site Request Forgery. It is intended to be easily extendable. As such, it has a plethora of already existing plugins for various security checks and more can be added. These can categorized as *crawl*, *audit*, and *attack* plugins, where crawl is intended for discovering new URLs and injection points, while audit typically performs injections, and attack plugins perform actual exploitation of found vulnerabilities [39].

2.6.2 Online tools

As an alternative to using multiple desktop tools, developers might select any of the online offerings which, while usually based on paid subscriptions, tend to cover a wider attack surface than a single desktop tool might. From the perspective of this thesis, such tools are not as relevant since they usually do not disclose how they work, however it might be worth mentioning them purely because of their capabilities and to provide the reader with a more complete picture of the available tools.

The first online tool is **Nessus** by Tenable - a comprehensive vulnerability scanner with various plugins for different types of vulnerabilities and flaws to be detected. It covers a wide range of assets, for which it offers pre-built templates of many different types of audits. These templates can then run plugins that provide the user with detection for certain features or issue and associated remediation steps. Plugins are regularly updated and new ones, in response to public vulnerability disclosures, are created [40].

As the next tool, I selected **Snyk**. Snyk specializes in secure development lifecycle by integrating into IDEs, code repositories, CI/CD flows etc. Among its features or tools is also a website vulnerability scanner which performs a passive scan to detect issues like outdated software or insecure headers. Apart from that, its offering consists of open source dependency monitoring, static code analysis, containerized application analysis, and cloud configuration analysis [41].

The last online service to mention is **Detectify**. Its offering is split into two products - Surface Monitoring and Application Scanning. The former is intended for continuous monitoring to detect exposed files, vulnerabilities, and misconfigurations. The latter specifically targets custom web applications and performs regular scanning for vulnerabilities to ensure their security. It also aims to cover issues beyond the OWASP Top 10 list [42].

2.6.3 Browser extensions

Given the aim of this thesis, it is important to also explore existing browser extensions in this area. To better understand their functionality, I tried to swiftly analyze their source code where possible, since several of these tools are actually open-sourced.

Firstly, there is **Wappalyzer** - a tool for identifying the software components used on a website, such as specific content management system, or common JavaScript libraries. The extension itself is open-sourced, though Wappalyzer has a broader offering of paid services as well. In order to detect certain technologies, Wappalyzer contains a set of JSON-expressed definitions where each one corresponds to a specific component and contains a set of rules based on which it should decide. One such definition can be found in code snippet 2.6.3 - for the WordPress CMS. The detection logic is based on comparing the requested HTML document against a set of rule, specifically regular expressions. A notable feature of Wappalyzer's functional logic is that a piece of software can imply other software, such is the case of WordPress which implies PHP and MySQL [43].

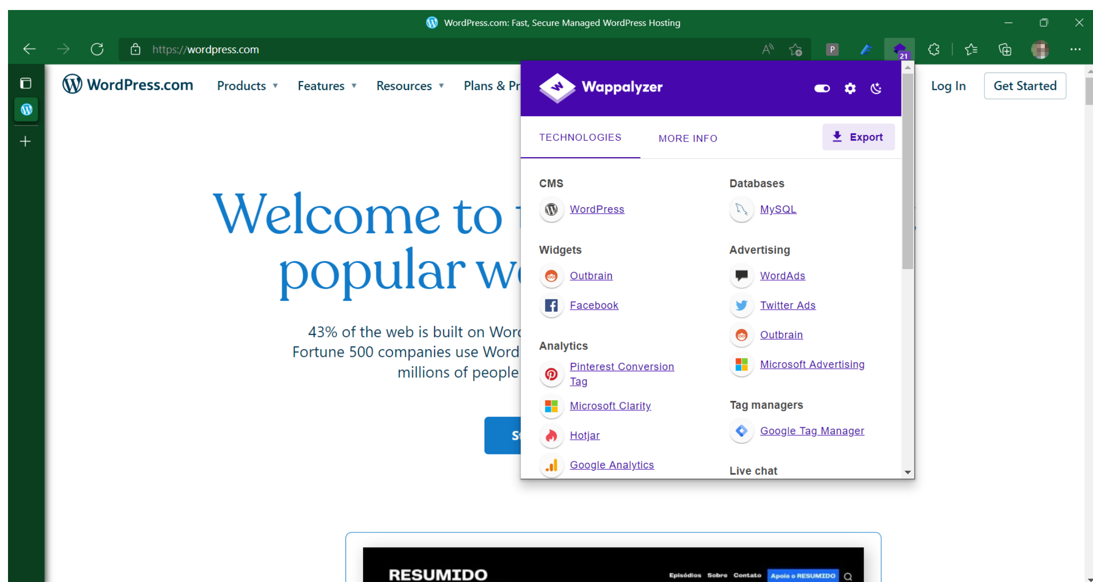

Figure 2.5: Example use of Wappalyzer on wordpress.com.

The second tool to mention is **CSP Scanner**. It is a browser-extension version of the https://cspscanner.com website which analyzes website's content security policy to see how well it can mitigate specific types of attacks, such as Cross Site Scripting (or XSS). Given that CSP is implemented via a special HTTP header, the whole analysis can be done with just HTTP alone [44].

The next tool, **Checkbot**, is a browser extension developed for search engine optimization (or SEO), page speed, and security analysis. While security is not its main focus, it does cover several web security checks, including: Use of HTTPS over HTTP, passwords submitted using POST request via HTTPS, client-side XSS protection. Unlike the the above two tools, CSP Scanner actually crawls the website to test for issues [45].

IP Address and Domain Information, by DNSlytics, is a small extension that acts as a handy frontend for their tools which mostly focus on providing IP and DNS details for any given domain. That being said, the extension itself is only intended as an intermediate between the user and the appropriate APIs. Therefore, to the best of my knowledge, no actual reconnaissance logic should be present in the extension [46].

Bug Magnet is an open-source tool that proves useful when manual testing is desired or required. Its purpose is to provide the tester with a quick way to insert potentially failing values to different types of input fields. Bug Magnet extends browser's context menu when right-clicking an input field or an editable `div` element so that the tester can specify what and where to insert. For each category, it has a set of predefined values that help verify correct processing of special characters, long string, invalid or unexpected characters, etc. [47]

Among the more advanced tools we have the **IP, DNS Security Tools | HackerTarget.com** extension which provides access to Hacker Target's online-hosted versions of various desktop tools. These include traceroute, ping, WhatWeb, and more. That being said, this extension is heavily based on connecting to its APIs which,

```
"WordPress": {
    "cats": [
      1,
      11
    ],
    "cpe": "cpe:/a:wordpress:wordpress",
    "description": "WordPress is a free and open-source content management system
    ↪  written in PHP and paired with a MySQL or MariaDB database. Features include
    ↪  a plugin architecture and a template system.",
    "headers": {
      "X-Pingback": "/xmlrpc\\.php$",
      "link": "rel=\"https://api\\.w\\.org/\""
    },
    "html": [
      "<link rel=[\"']stylesheet[\"'] [^>]+/wp-(?:content|includes)/",
      "<link[^>]+s\\d+\\.wp\\.com"
    ],
    "icon": "WordPress.svg",
    "implies": [
      "PHP",
      "MySQL"
    ],
    "js": {
      "wp_username": ""
    },
    "meta": {
      "generator": "^WordPress(?: ([\\d.]+))?\\;version:\\1",
      "shareaholic:wp_version": ""
    },
    "pricing": [
      "low",
      "recurring",
      "freemium"
    ],
    "saas": true,
    "scriptSrc": [
      "/wp-(?:content|includes)/",
      "wp-embed\\.min\\.js"
    ],
    "website": "https://wordpress.org"
}
```

Source code 2.6.1: WordPress ruleset definition in Wappalyzer.

in this case, means that the free use somewhat limited in terms of number of calls and tools available. Nonetheless, it is still worth mentioning what approaches certain extensions choose [48].

Hack-Tools, by Ludovic Coulon and Riadh Bouchahoua, is an advance tool for penetration testing. Similar to Checkbot, it has a set of predefined payloads a tester might want to use except that these payloads are focused on specific vulnerabilities like SQLi, XSS, and other. It also helps prepare tailored Linux commands based on input parameters so that they can be copy-pasted to the terminal. Among other things, it also integrates a CVE lookup mechanism within the UI. While part of the extension's functionality is embedded in the code itself, it does rely on several RSS data sources which are queried on demand for functionality like listing of `www.exploit-db.com` or several feeds from `www.cxsecurity.com` [49].

The last advanced tool is the **Penetration Testing Kit** extension which, like Hack-Tools, provides the user with a set of malicious payloads ready to be used, as well as software profiling thanks to Wappalyzer's NPM package version. Moreover, it also provides network traffic interception and monitoring functionality. To the best of my knowledge, it relies on evaluating given loaded web page, Chrome's `chrome.devtools.network` API for the interception logic, and creating crafted HTTP requests on demand [50].

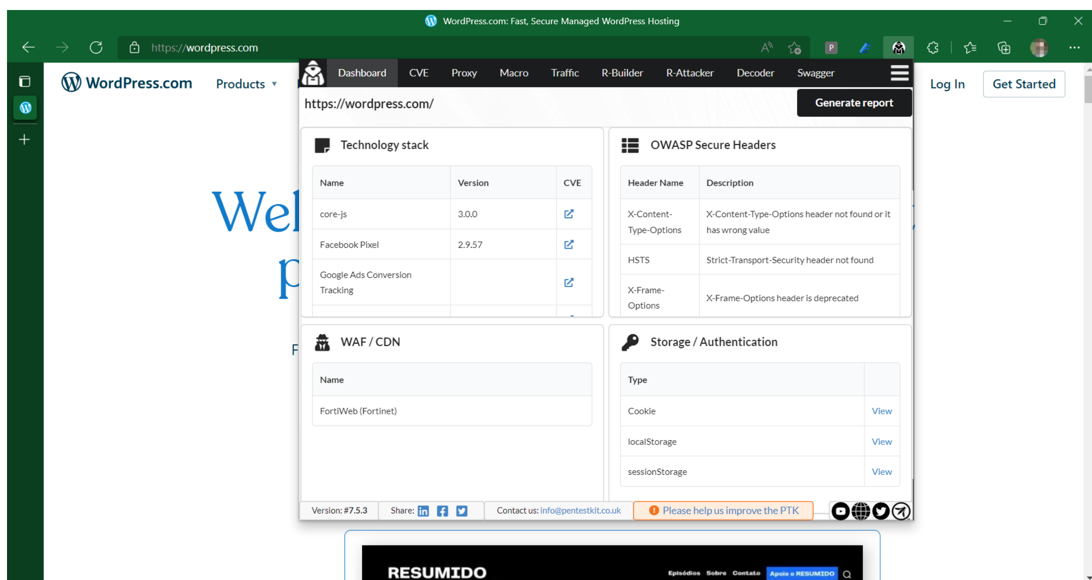

Figure 2.6: Example use of Penetration Testing Kit on wordpress.com.

3 | Methodology

3.1 Information Gathering

Prior to testing for certain vulnerabilities, it is desirable to perform reconnaissance, or information gathering, on the tested website. Combining methodologies learnt from WSTG (see 2.2.2) and ATT&CK (see 2.3.3), the following list of categories, or pieces of information, could be an adequate baseline for reconnaissance:

- application entry points,
- site's topology,
- DNS records,
- digital certificate details,
- open ports,
- used software,
- configuration files,
- leaked sensitive information.

With this information, testers should be able to perform more effective security tests because it can help them better focus their attention towards certain aspects of the website. For instance, if the underlying server seems to be running on an outdated version of nginx, it is worth exploring already known and well-documented vulnerabilities of this version to verify if they can be exploited.

When done manually, testers usually have many tools at their disposal to find relevant information about given website, many of which offer free services when used manually, for example Google Search. In many cases, using these services programmatically via APIs requires paid licenses which may not be a viable option for every tester, nor for the purposes of a browser extension. Taking that into consideration, the rest of this section explores where to source such data without any additional costs.

3.1.1 Website Model

Firstly, it is important to understand the structure (and implementation) of a website. This will become important later, mainly in the Extension Design chapter. In order to avoid unnecessary complexity, however, I limited the following model of a website to the most relevant/meaningful parts.

The term *website* is understood, at a very basic level, as a collection of (web) pages, where a *web page* is an HTML document which may or may not include other dependencies, such as images, JavaScript files, etc. It is hosted on a single server which may also host an associated database system. Therefore, multiple load-balanced servers with firewalls or any other more complicated infrastructure is not considered in this thesis.

An HTML document is the main building block for a single web page. It is an XML document following any of the existing HTML Document Type Definitions (or *DTDs*). It has two main parts or components: a *head* and a *body*. The head serves for storing metadata such as title of the document, character set, site author name,

CSS or JS dependencies, etc. Its contents are not meant to be directly rendered (with the exception of the title to a certain degree). Conversely, the body of an HTML document is intended to be rendered and holds the actual visible content of the page. Though, it should be noted that CSS and JavaScript dependencies can be included in the body as well.

While there are many various tags (i.e. data or component containers) that may be used within an HTML document, I created a list of several tags which will be most relevant for the extension design:

- `<a>` tag (or anchor tag),
- `<meta>` tag,
- `<!-- -->` tag (or comment tag),
- `<form>` tag,
- `<link>` tag,
- `<script>` tag.

First, the **anchor tag** is what helps build website's topology - it links one HTML document to another. It will prove most important for web crawling purposes. Second, the **meta tag** is a great and quick source of additional information about given website. It is not uncommon, for instance, to see content management systems clearly stating their presence as well as version number in one of the meta tags. Moreover, certain systems also contain their own unique meta tags by which they can be easily recognized. Third, the **comment tag**, while not being always being useful, developers may leave certain hints as to how the underlying web application is structured, how it functions, etc. It is generally considered a good practice to remove comments from the client-facing code, as it could reveal sensitive information. Next, there is the **form tag** which indicated a specific part of the web application accepts user input, making it one of the most important tags for which to search, as improper validation/sanitization of user input is often the root cause for various vulnerabilities. Lastly, the **link tag** and the **script tag** are used for including styling or functional dependencies (in both cases) or for declaring a functional piece of code (in the latter case). In either case, knowing what types of dependencies a website uses may reveal already disclosed vulnerabilities, making it much easier for the adversaries to attack given website.

Going back to the infrastructure aspect of a website, a (web) server is understood as either a physical or a virtual machine which hosts certain HTTP-based web application and allows inbound traffic through any of the *well-known* ports, such as port 80 for HTTP traffic or port 443 for HTTPS. It may also have other ports open for communication, for instance to allow remote access to the developer, but their number should be limited as to avoid broadening the attack surface, i.e. the individual places where an attacker might strike.

3.1.2 Web Crawling

As previously hinted, web crawling is an important part of many web security testing tools. The idea of crawling is that a piece of code gradually explores given website, starting from a certain point of origin (e.g. the homepage) and following links on each discovered web page. If we think of a website as a directed graph, where each node is a single page and each edge is a link from one page to another, web crawling is nothing more than graph traversal. With that in mind, a web crawler may implement various strategies, or algorithms, to achieve its goal of exploring a website.

When using a certain strategy (or implementing a new one), there are a few aspects to consider. Firstly, it is the time requirement aspect of each crawl. Unless it happens locally, each GET request will take a certain amount of time possibly in the order of hundreds of milliseconds. If time is of the essence and the crawled website has a large amount of pages, the chosen strategy should minimize the number of visited (i.e. fetched) pages as much as possible / required. This could be done based on detected patterns in, for instance, template-based website.

Among the simplest of solutions is a classical breadth-first search (or *breadth-first traversal*), i.e. traversing the website starting from a certain origin and expanding layer by layer based on all links available on each page. Its main advantage is the simplicity of its implementation, as it only really requires a queue structure

and a set for remembering which pages were already visited. Moreover, it could be considered a thorough approach, as it can systematically crawl the whole website.

Though, BFS's time requirements may not always be desirable. This is especially true for sites that have very few different templates, yet they have a lot of content, such as e-commerce sites. In those cases, skipping certain pages may be preferred. That, however, poses the risk of missing certain vulnerabilities or information exposures. To the best of my knowledge, the reviewed security testing tools tend to use (depth-limited) breadth-first or depth-first search for the crawling functionality.

3.1.3 Available APIs

In certain scenarios, not all information can be procured from the website itself (e.g. during crawling) or it is not practical to do so (e.g. due to time restrictions). To overcome these issues, developers may choose to use certain APIs, both public and private. As previously hinted, in this thesis I shall explore only the free APIs as to avoid any associated cost with running a web security tool.

The first, and probably the most important, data source is **NIST's NVD**, previously described in subsection 2.4.1. With its API, a security testing application may easily procure lists of recently disclosed vulnerabilities, vulnerabilities matching a certain CPE string, etc. This makes it the perfect source for vulnerable component detection which shall be discussed later in this chapter.

There are certain attacks, such as *subdomain takeover*, which require knowledge of existing DNS records of given domain. In the subdomain takeover case, the attacker would look for DNS CNAME / A records pointing to domains that they might be able to claim. This could be the case of various multihosting platforms [10]. To procure such information, developers may access **Google's Public DNS** through its DNS-over-HTTPS (or *DoH*) API. It offers a JSON endpoint with the following format:

```
https://dns.google/resolve?name=example.com&type=ALL
```

This would return all the DNS records (of all types) for the domain `example.com` [51].

If the tested website is centered around displaying user-submitted content to its users, it might also be worth analyzing any external URLs detected within the content. Having malicious URLs on a website may be damaging to the site owner's business and reputation, among other things. For that purpose, developers may choose to use **Google's Safe Browsing APIs**, specifically the Lookup API which returns URL reputation and tests if any of the provided URLs are known sources of threats. The specifics of the API are slightly more complicated than the above DoH which is why I shall exclude them here. Important thing to note is that, while it is free to use for non-commercial purposes, it does require a registered API key [52].

Similar to the Safe Browsing Lookup API, there is also **EmailRep API** provided by Sublime Security, Inc. which provides email reputation details. The use of this API is rather simple, as the requests follow this format:

```
GET https://emailrep.io/{email}
Key: [your api key]
User-Agent: [your app name]
```

Although, as seen in this template, it does require an API key as well. As of April 2022, users may get a free key with 250 queries per month limit and at most 10 queries per day. Apart from reputation, each response also provides details such as on what social media sites the specific email address is used, if it is associated with any credentials leaks, data breaches, etc. [53]

Among the (functionally) more complex APIs is **Mozilla's HTTP Observatory**. To a certain degree, it does provide similar logic to what a certain web security testing tool would do - it analyzes Content Security Policy, looks at correctly set flags in cookies, reviews Cross-Origin Resource Sharing Policy, use of HTTPS over HTTP, etc. While its scope is limited, it does offer relatively fast and free results for HTTP-related issues which could make it a reasonable addition to a testing tool. The API is accessible through its primary endpoint following this format:

```
https://http-observatory.security.mozilla.org/api/v1/analyze?host=www.example.com
```

This specific URL would, when sent as a POST request, initiate an assessment of `www.example.com`. To retrieve the results, a subsequent GET request to the same URL would be necessary [54].

Lastly, there is **Mozilla's TLS Observatory**, a TLS complement to HTTP Observatory. It analyzes TLS certificates associated with given domain and returns all the available information. Unlike its HTTP counterpart, it does not actually perform any sort of tests that would hint as to what are the specific issues. However, it provides enough data for further analysis or to be used in unison with other information [55].

3.2 Software Detection

Having full knowledge of all the components used on a website and their versions can quickly determine what types of attacks might be possible and thus are worth testing. However, unless the tester (or adversary) has access to the source code, this can be difficult to achieve.

In this thesis, a (software) component is understood as any publicly distributed piece of code, both free and paid, that is a functional and reusable unit. For instance, the very popular JavaScript library jQuery would be considered a component, while a custom snippet of code using said library would not. It is also worth distinguishing between frontend (or client-side) components and backend (or server-side) components. The reason is that in the former case, every visitor of a website has full access to their code while in the latter case, we can only observe side effects of their presence. This has a major impact on the certainty with which we can detect them.

3.2.1 Pattern-Based Approach

Among the more popular approaches to software detection is *pattern-based detection*. It is based around the idea that a single piece of software will exhibit the same side effects or traits, such as registering a specific global variable in the case of certain JavaScript libraries or sending a custom HTTP header. This approach can be seen in 2.6.3 where Wappalyzer defines rules (or patterns) for detecting WordPress CMS.

This approach has the benefit of being rather simple to implement, it can be performant, and it can achieve good results. The issue is that it requires a large enough knowledge base prior to detection in order to be effective, and the detectable piece of software must have *unique enough* patterns as to avoid confusion between two different components. In the case of Wappalyzer, this means more than 20 000 lines of configuration files with the patterns that are manually maintained [43].

In order to implement a solution using this method, it is essential to understand where to look for patterns, i.e. what sources of information from the website can be used. The following list is based on recommendations provided in WSTG[19]:

- non-standard meta tags,
- certain standard response headers (e.g. `server`),
- custom response headers,
- response header order,
- 404, 500, and other error page contents,
- cookie names and values,
- HTML comments,
- contents of `robots.txt`,
- existence of certain directories (e.g. `wp-admin`),
- existence of certain files (e.g. `readme.html`),

- JavaScript global variables,
- JavaScript error messages,
- file extensions.

This list is not exhaustive but should provide the reader with an appropriate baseline as to what information may be of interest. Important aspect to consider is that these values are developer-controlled, meaning that they cannot be fully trusted (from an adversary's perspective). For instance, concluding presence of WordPress solely based on `wp-admin` directory being present may lead to false positive if the author previously used WordPress but had replaced it since then, or if they created it on purpose to confuse automated scanners.

The exact way in which those values can be used for detection depends on the specific component - and there are too many. To provide just a few examples based on WSTG's guidance, several popular content management systems, including WordPress, Joomla, and Drupal, include a meta tag with the name `generator` and its content tends to disclose the CMS name, as well as version by default. Alternatively, certain systems tend to use very specific cookie names. Django, a popular Python web application framework, for instance, sets a cookie named `django` and so does CakePHP (cookie: `cakephp`) or Laravel (cookie: `laravel_session`) [19].

3.2.2 Fingerprinting Approach

The pattern-based approach heavily relied on human input for every component and, thus, a significant amount of manual work. While that may provide reliable results under certain circumstances, it might not always be viable. An alternative approach, with higher upfront time requirement but potentially faster extension when onboarding new components, is *fingerprinting*.

The general idea of fingerprinting in this case is very similar to physical fingerprint matching, i.e. for a specific object (in our case a website) we extract a set of features, or a feature vector, we compare this feature vector against an annotated dataset of known feature vectors and we copy the label from the best matching vector. The exact meaning of *feature* and *best matching* will depend on the specific implementation which should be tailored to the use case. Possible features to extract might be CSS style properties, JavaScript functions and variables, or XPaths [56].

This approach might especially be useful for backend component detection, as (usually) the user can only inspect the output of those components rather than their code - which is the case of JavaScript on client-side. Therefore, they must infer the backend components based on their side effects visible to them.

3.2.3 JavaScript-Specific Approach

The modern web is centered around three main languages - HTML, CSS, and JavaScript. Out of these three, JavaScript is the only one intended to convey functionality/interactivity other than form submission and following links. For that reason, I shall limit the frontend component detection to JavaScript libraries only.

In general, any website can source JavaScript code in two ways: directly embedding it in an HTML document or loading it from a separate JavaScript file. Alternatively, the site could dynamically insert additional JavaScript code by performing an asynchronous request from within JavaScript itself (loaded in either of the two ways described above) and then evaluate it However, to the best of my knowledge, this is not a common approach and thus will not be considered.

Let us consider the latter case first - loading a dedicated JavaScript file instead of embedding it in an HTML response has several benefits. Firstly, it helps decrease load time due to static asset caching performed by browsers. Secondly, the file does not need to be hosted by the website owner which can further decrease load time and save storage. And last but not least, loading the file from a 3rd party vendor can also simplify updating dependencies.

Possibly for those reason, it is quite common to see websites using libraries hosted on Content Delivery Networks (or CDNs) such as cdnjs (`cdnjs.com`) or Google Hosted Libraries (`https://developers.google.com/speed/libraries`). An important aspect of CDN-hosted libraries is that they tend to contain the exact library name and version in the URL, making the detection a trivial task. For instance, including jQuery v3.6.0 from Google Hosted Libraries would require the following URL: `https://ajax.googleapis.com/aj`

ax/libs/jquery/3.6.0/jquery.min.js. The key aspect is that the pattern is consistent and, therefore, predictable.

The detection gets slightly complicated if the library is hosted on the website itself. It can be the case that the developer decides to include the library name and version as part of the file name, e.g. `jquery.3.6.0.min.js`. However, they might also omit the version, the real library name, or even intentionally put different name to confuse automated scanners. Due to that possibility it might be better not to rely on the file name alone but also inspect the contents.

One approach to detecting JavaScript libraries based on the file contents is hash comparison. The idea is that a scanner has access to an annotated database of library content hashes, e.g. MD5 or SHA-512. If the scanner then finds an unknown JavaScript dependency, it can hash its content using a certain hashing algorithm and quickly compare the value with all the already known (annotated) hashes. Assuming no two JavaScript libraries have colliding hashes, finding a match between the dependency hash and any of the reference hashes means the scanner found the exact library version. There are two issues with this approach. Firstly, the hash database must be actively maintained and large enough. This problem can be somewhat mitigated if the scanner pulls the data from an specialized resource such as cdn.js which provides APIs to list all of its hosted libraries including their content hashes. Secondly, the JavaScript dependency found on the website must be left unchanged, as adding even a single space character will lead to a significantly different hash value.

To address the hashing issue, the scanner may want to compare the libraries' abstract syntax trees (or *ASTs*) rather then the content hashes. The main benefit of this approach is that it is not prone to failures due to whitespace changes, variable renaming, nor reordering code block. While I am not aware of any existing security testing tools using this approach, it has been proposed for code plagiarism detection by Zhao et al. Using the proposed method, however, poses the potential downside of having a large number of hash value comparison per each library version instead of just one [57]. The exact number will depend on the library's code complexity.

There are, of course, other metrics or aspects of client-side dependencies that may be observed. For instance, an automated detector may analyze copyright notices in the dependencies. For the sake of simplicity, I did not consider additional techniques such as JavaScript bundling which combines multiple files into one, rendering the hash comparison unusable and the AST approach would need to be modified accordingly as to allow for multiple dependencies being detected.

3.3 Vulnerable Component Detection

While application developers themselves can directly introduce weaknesses into their product, such as by not sanitizing user inputs, they are not alone. Nowadays, it is common to build websites from multiple already existing and publicly available components. For instance, this seems to be a rather typical pattern for JavaScript and its package managers, where each package can have other dependencies which potentially broadens the attack surface. Yet, if a single package is compromised or has an exploitable vulnerability, it presents a great opportunity for adversaries to attack many different targets.

From an adversary's point of view, if they were successful in determining the underlying software of the targeted application, they could first look for publicly disclosed vulnerabilities for detected components. This can be achieved using the same tools that the developers or administrators might use - NIST's NVD. The CPE-based search of its API makes a perfect tool for discovering any relevant vulnerabilities. Moreover, it is possible to find working proof of concept code for these vulnerabilities, making it even easier to potentially compromise an application.

3.4 Sensitive Information Detection

One of the goals of an adversary when trying to compromise an application might be to, eventually, access any sensitive information the application might store. This could be login credentials, addresses, or any personal details. However, it is not just the users' data that could be of interest to a malicious actor. Note that there will be a certain overlap with the previously described software detection, as it partially relies on the developer exposing software-related details (or not obfuscating them).

If we only concentrate on the potentially exposed sensitive information, we could split it into two main categories - user data (or personal data) and software-related data. The former case could be described using GDPR's definition of personal data, i.e. *"(...) any information relating to an identified or identifiable natural person (...)"* [58, p. 33]. The latter case is any data related to the application in question which should only be known to the its developers - its internal state, configuration, architecture etc.

The remainder of this section is dedicated to exploring patterns and/or related specification of these types of data and how detection of certain types can be achieved and/or is achieved by existing tools.

3.4.1 Personal Data

While GDPR's definition is quite broad and thus it would be practically impossible to explore it to the full extent, there is only a limited number of attributes or types of personal data that seem to repeat themselves across various websites, each of them fulfilling the criteria to be called personal data by GDPR. Thus, their public disclosure would probably lead to legal liability. I have put together the following list of such attributes or categories based on my own personal experience, in no particular order, that should provide an adequate basis for the purposes of this thesis and any future work:

- real name,
- username,
- password,
- date of birth,
- phone number,
- email address,
- physical address,
- IP/MAC address,
- application-provided identifiers,
- state-issued identifiers,
- credit/debit card details.

Note that this list aims to summarize individual pieces of information rather than more complex structures. This is why, for instance, it does not contain items like order history, or criminal record, etc. Each of these items, or categories, has its own format which may or may not be country-specific. My goal was to analyze common traits of each category that could allow automated detection.

I shall start with the categories that have a clearly defined standard or are at least somewhat standardized, as these are going to be relatively simple to analyze. First one of these is an **email address** as defined by RFC 5322's `addr-spec`. The specification is rather verbose. Therefore, I made a few assumptions to simplify its interpretation. Since the specification is written using Augmented Bacchus-Naur Form [59], I decided to analyze it from the terminals up towards the root to, hopefully, achieve more clarity.

The first assumption is that an email address does not contain so called folding white spaces nor comments, as they seem unlikely to appear outside of actual email communication. The specification for an address is built around so called atoms which are strings of fundamental characters (or atext) - these include alphanumeric characters and special characters like ``!#$%'*+-/=?^_`{|}~``. To allow for dots (`.`), it defines dot atoms [59, p. 11-13].

The second assumption is that an email address does not contain quoted strings which are an extension to the aforementioned atoms or dot atoms to allow for any special characters that would otherwise have some semantic interpretation, such as `<;>` and more. While they are allowed by the specification, they are usually not allowed by email providers and, thus, would introduce unnecessary complexity. Similarly, we shall not consider so called domain literals which are, in essence, almost identical to the quoted string concept [59, p. 13-14, 16–18].

With these assumptions, we can finally describe the address as a string concatenation of a local part which would be a dot atom, the at (`@`) symbol, and the domain part which, again, would be reduced to a dot atom [59, p. 17-18]. Using this knowledge, it is possible to put together a single regular expression for detecting RFC5322-compliant email addresses as such:

```
/
[a-zA-Z0-9!#$%'*+\-\/=?^_`{|}~]+(\.[a-zA-Z0-9!#$%'*+\-\/=?^_`{|}~]+)*
@
[a-zA-Z0-9!#$%'*+\-\/=?^_`{|}~]+(\.[a-zA-Z0-9!#$%'*+\-\/=?^_`{|}~]+)*
/g
```

The regular expression is intentionally split into multiple lines for readability purposes - to emphasize its individual components (`dot-atom "@" dot-atom`). It is, however, important to keep in mind that it only covers a subset of RFC5322-compliant addresses due to the assumptions listed above. For the purposes of this thesis, it is sufficient but it might be worth considering the full specification for cases like network sniffing or traffic analysis. Moreover, it does not account for so called obsolete addressing, as it is defined mainly for backwards compatibility purposes and such addresses should no longer be generated.

Let us continue with **IP and MAC addresses**. The first type to consider is an IPv4 internet address which is defined as a four octet source or destination identifier by the RFC791 specification [60]. While this means that IPv4 address only requires 4 bytes (or 32 bits) of memory, it tends to be presented and stored in the more human-readable format of 4 dot-separated integers between 0 and 255, e.g. `192.168.0.1`. This makes it easily detectable even using regular expressions, such as:

```
/(\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3})/g
```

Although in that case it is important to ensure there are no leading zeros or values outside bounds. While IPv4 addresses are still widely used as identifiers, it is important to also consider its successor IPv6 specified by RFC2460 from 1998. Instead of 32 bits, IPv6 uses 128 bits [61]. Given the length of these addresses, they tend to be represented as colon-separated hexadecimal numbers of 1 to 4 digits, i.e. numbers between 0 and `ffff`. In case there are multiple groups of zeros in series, it is possible to compress them using the `::` syntax which can only be used once and indicates that all of the unspecified blocks contain only zeros. For example, `2001:DB8:0:0:8:800:200C:417A` could be contracted to `2001:DB8::8:800:200C:417A`. An alternative to this notation, which may sometimes be useful, is a combination of IPv4 and IPv6 notations in that the format is `hex:hex:hex:hex:hex:hex:dec.dec.dec.dec` where `hex` is a hexadecimal representation and `dec` is decimal representation [62]. This does complicate, to a certain degree, the detection through a regular expression but it is still possible.

Personal identification number, social security number (or *SSN*), or tax payer identification number are just a few examples of what might be called **state-issued identifiers** which, while country specific, tend to be standardized and considered very sensitive. These are usually several-digit long numbers and can sometimes contain special characters as separators. For instance, SSN in the United States is a 9-digit number that is sometimes separated by dashes into three 3-digit groups [63]. However, given the number of identification number per each country, where each country may specify its own format for these identifiers, I shall not describe them in more detail. It is, nonetheless, important to consider them due to the level of sensitivity.

Phone numbers, while somewhat standardized, vary from country to country. In general, local/national phone numbers tend to be around 9-digit long with spaces in between segments as to make them more readable. For instance, in Czech Republic it is common to see either of the following formats: `NNN NNN NNN` or `NNN NN NN NN`, where `N` represents a single digit. In case of international calls, it is necessary to also include the country code (or country prefix) which identifies the destination country and usually starts with the plus (+) symbol. However, phone numbers may also contain other symbols such as parenthesis and hyphens which mark a *potentially optional* part of the phone number, depending on the caller's location [64]. Similarly to state-issued identifiers, this makes the detection slightly more complicated to achieve because it relies on country-specific preferences and implementation.

Credit/debit card details usually consists of three main values: the (credit) card number, the expiration date, and the card verification code (or *CVC*). From these values, possibly only the card number can be easily detected on its own without high false positive rate. The numbering system for card number is defined by ISO/IEC 7812-1:2017 specification and allows for various number lengths. However, Mastercard and Visa card

holders will be most familiar with the 16-digit format split into four groups of four digits [65]. This would make it possible to, again, use a regular expression to detect card numbers and then do some additional validation to avoid false positives.

All of the above categories or types of data were in some way or another standardized and unique enough as to not cause too many false positives. Their detection was, therefore, possible due to distinct patterns and could relatively easily be achieved using a certain regular expression. This, however, may not always apply to the remaining categories: names, usernames, passwords, dates of birth, physical addresses, and application-provided identifiers. Their detection could, for instance, be done using a certain form of dictionary attack or the detector could be context-aware. The specific methods will be discussed later in this section.

3.4.2 Software-Related Data

Publicly disclosed personal data is, unsurprisingly, a severe problem. However, even if the developers are able to avoid it, they might still disclose their own software-related data which may provide the adversaries with enough details as to simplify software compromise or allow certain types of exploitation. Similarly to personal data, I have created a list of different categories of software-related data based on WSTG, OWASP Top 10, and my personal experience that might provide the reader with an appropriate baseline:

- raw error messages (and stack traces),
- debug outputs,
- API keys,
- credentials,
- verbose comments.

If an application encounters an unexpected input or state, the runtime environment often provides a decent amount of details in **error messages** to help remediate the issue. It can be very useful when debugging but should generally be avoided in production environments, as it can reveal too many details about the implementation. Ideally, a production application should only display generic error messages if an issue is encountered, e.g. instead of displaying MySQL connection error, it should return a general 500 HTTP response page informing the use of unexpected issue and to try again later. From the perspective of automated detection, it is rather simple to implement in that the error messages tend to have a predictable format, though it is usually technology-specific. This means that, for instance, MySQL errors will have different wording than PostgreSQL errors, etc. The challenging part usually is to invoke the error so that it can be detected.

Similarly to raw error messages, developer-defined, or application-defined, **debug outputs** can be extremely helpful while debugging applications but can expose too much information about the application if kept in consumer-facing version. Though, it is not guaranteed to yield any interesting results - unlike error messages which tend to provide quite a lot of details. Their detection will heavily depend on what technology we want to detect/analyze. For instance, it is rather simple to find all debug logs called from within all JavaScript dependencies but it might be harder to detect a single debug text produced by a server-side component that was accidentally passed to the HTML document.

While both of the categories above would usually expose architecture details which may serve as a step towards exploitation or attack, the following category, **API keys**, can be exploited directly. API keys are often associated with paid services, such as Google Search API. Having access to them for free means adversaries can use them for their (possibly malicious) purposes without being associated with them and without any incurred costs. The detection will, again, depend on the actual technology (or rather service) in question, as not all services have the same format of API keys. However, since APIs are associated with specific endpoints, it might be easier to identify for what service the keys are intended. Since the goal of API keys is to make enumeration hard, one could expect them to have the format of GUIDs or specific hashes.

Globally Unique IDentifier (or *GUID*) is a 128-bit long identifier that can guarantee uniqueness without the need for any central issuer authority. Its string representation is well defined by RFC4122 using ABNF notation as follows [66]:

```
UUID                   = time-low "-" time-mid "-"
                         time-high-and-version "-"
                         clock-seq-and-reserved
                         clock-seq-low "-" node
time-low               = 4hexOctet
time-mid               = 2hexOctet
time-high-and-version  = 2hexOctet
clock-seq-and-reserved = hexOctet
clock-seq-low          = hexOctet
node                   = 6hexOctet
hexOctet               = hexDigit hexDigit
hexDigit =
    "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" /
    "a" / "b" / "c" / "d" / "e" / "f" /
    "A" / "B" / "C" / "D" / "E" / "F"
```

An example of such identifier would then be `cd0daff2-1b30-485a-b423-0936ed96ba9d`. Given this representation, they can be easily detected using regular expressions.

Exposed **credentials** can lead to severe consequences. However, automated detection of such data can be difficult if not context-aware. The situation is somewhat similar to user credentials in that they can have very diverse values. There are a few *shortcuts* that can be taken in the case of software-related credentials however. First, certain components use very specific user names for which we might search, e.g. `admin`. Second, instead of looking for values we could search in client-side code for specific keywords such as `password`, `passwd`, `pwd`, `pass`, `key`, etc. There are many potentially interesting keywords for which to search. Although, we should note that some of those may be more prone to false positives than other.

It might be the case that a developer did not leave credentials or any API keys in the code's **comments**. However, comments in general may provide more details about the architecture and/or logic of the application. Moreover, certain comments could suggest the use of a specific piece of software. For instance, specific content management systems may leave certain unchanging comments for the purposes of identifying what component of a page is imported at a specific place in the document. Unfortunately, not every comment is, by definition, useful. In order to assess comment's value or contribution is a slightly more challenging task but could be simplified the same way as credentials detection, i.e. keyword lookup.

3.4.3 Used Approaches

In the previous subsections (Personal Data and Software-Related Data), I tried to look for patterns in various types of data to determine if they might be detectable using regular expressions, as that can be an easy to implement and might potentially yield good results. In this subsection, I shall review what approaches existing tools use, as well as any other proposed methods.

Since OWASP's ZAP closely resembles what might be the ultimate goal of the browser extension, I analyzed its approaches first. The main module to explore in this regard is the Passive Scan Rules add-on (or `pscanrules`). This module implements a few classes intended to detect information disclosure where each class analyzes specific source of leak rather than the type of information leaking. These are:

- debug errors (`InformationDisclosureDebugErrorsScanRule.java`),
- URL (`InformationDisclosureInUrlScanRule.java`),
- `Referrer` header (`InformationDisclosureReferrerScanRule.java`),
- `X-Powered-By` header (`XPoweredByHeaderInfoLeakScanRule.java`),
- `X-Debug-Token` header (`XDebugTokenScanRule.java`),
- comments (`InformationDisclosureSuspiciousCommentsScanRule.java`).

To the best of my knowledge, all of these are using a predefined list of keywords for which ZAP searches in the respective parts of the website (some of which are combined with regular expressions for more targeted search). ZAP is also able to detect private IPv4 addresses as defined by RFC 1918 which is also done through regular expressions. On top of these passive scan rules, ZAP also contains two active scan rules - .htaccess information leak and ELMAH information leak, which are both pattern-based. The former checks if the file is accessible, while the latter looks for keywords [37, 67].

While not a typical sensitive information exposure detector, Wappalyzer does try to infer used technologies based on available bits of information with a relatively high level of confidence. As discussed in section 2.6, it is based on a set of rules which are usually in the form of regular expressions or keywords that are applied to different parts of the website. This is consistent with ZAP's approach to information leak detection.

Possibly similar pattern-based approach can be seen being used by GitHub's Advanced Security secret scanner which stops commits from being pushed to a repository if a software secret is detected [68]. However, to the best of my knowledge, its scope is limited to secrets such as API keys or access tokens which, while product-specific, have a standardized format. Though, it does require large enough knowledge base.

Until now, all of the discussed tools focused on software-related data as described above. In order to detect sensitive information in general, i.e. including personal data, it is possible to use a statistical approach based on information content of a term as proposed by Sanchez et al. The main advantage of their approach is that it can be applied to domain-independent textual data [69]. This might prove useful for content-rich websites. The downside is that it requires a large enough general corpus for training the model and it would be specific to a certain language. As such, it might be a compelling addition to certain desktop tools but not as much for a browser extension.

Lastly, it is also possible to approach this problem using neural networks - specifically, LSTM recurrent neural network. In 2017, IBM filed a patent outlining the use of such networks for context aware sensitive information detection [70]. There are also other approaches such as fuzzy fingerprinting or map reduce [71]. Important consideration here is that these solutions are built on top of natural language processing (*or NLP*) capabilities.

3.5 Vulnerability Detection

One important part of certain security testing tools is vulnerability scanning or detection. However, the specific methods to achieve that vary depending on the type of vulnerability, available resources (such as only client-side code), and how invasive/aggressive the method can be. For instance, certain vulnerabilities can be detected during development using static code analysis tools inside an IDE. In the scope of this thesis, we assume the tool only has access to client-side resources, i.e. anything visible in the browser while traversing the website, including any functional code, and the specific method should have as little side effects and as low impact as possible.

There are two main types of vulnerability detection methods - static analysis and dynamic analysis (or testing). The goal of static testing is to determine possible vulnerabilities based on source code. Given the assumption of only having access to client-side code, this would mean the analysis would be performed on (mostly) HTML and JavaScript code sent to the browser. Conversely, dynamic testing executes analyzed application, tries different input combinations and examines the behavior of the application [72, p. 18-19].

Both types have their advantages and disadvantages which would depend on the specific vulnerabilities that needs to be detected. Since the number of distinct vulnerabilities is rather large, the rest of this section shall be dedicated to the previously listed and described weaknesses (see section 2.5).

3.5.1 SQL Injection

Assuming all database interactions are done through a backend component that accepts user-supplied parameters, dynamic analysis is the only approach to consider in the case of SQL Injection. If access was given to the server-side code as well, static analysis might also be utilized. Though, it would require better understanding of where and how parameters are handled.

The basis for all SQLi detections is to be able to identify all entry points of the tested application. That is, every URL that accepts user input. In practice, this usually means any URL accepting form data or links

passing GET parameters to the server - this can be seen on various sites for content pagination purposes. All of these entry points should then be tested using either of the methods outlined below, as all of them could be susceptible to SQLi.

The first method for automated SQLi detection is based on the assumption that database errors may not be properly handled by the application, i.e. if an illogical query is run, the application would disclose a raw error message. These messages, while dependant on the specific database engine, have a well defined format which can be detected using a keyword lookup - or a regular expression if more details need to be extracted from the message. Generating such input is relatively easy since the testing tool can have a list of predefined invalid queries to inject. The problem with such approach is that if server-side validation is run prior to executing a SQL query in question, the input might get rejected without ever detecting the vulnerability. This is something a tester would quickly recognize if performing a manual test but it is sightly more complex when done automatically. Nonetheless, this specific method seems to be utilized by both sqlmap and ZAP [35, 37].

Another detection method is time-based injection or inference. This technique is based on injecting a malicious payload that contains a call to the database's sleep/wait function with high enough value as argument to avoid false positives. The assumption is that if SQLi was possible for given input parameter, using payload such as `" OR sleep(10) = 1 #` would cause the request to take at least 10 seconds more than without it. This specific method is used by sqlmap as well [35]. Note that not all database systems use the same name for this function.

Other approaches that can be seen in ZAP and sqlmap are based on: (1) appending tautologies such as `AND 1=1` and comparing outputs before and after appending, (2) piggy-backed queries using semicolon, (3) union-based queries, and (4) trying to inject arithmetic operations and see if they evaluate to the same output as without the operation [35, 37, 67].

3.5.2 Cross-Site Scripting

Unlike SQLi, XSS vulnerabilities can also be partially detected using static analysis because part of the potentially responsible source code is provided to the user (i.e. JavaScript). Based on a 2015 review, static analysis was proposed in 24% of studies into XSS, while dynamic analysis was the focus of 50% of studies. The remaining approaches were either based on XSS prevention, modelling approach, or alternative approaches benchmarking [73]. For the purposes of detection, is important to remember to distinguish between the three types, i.e. reflected, stored, and DOM-based. The reason is that their payload preparation, exploitation, and verification may vary.

A common dynamic analysis approach for all of these types is that a malicious payload is prepared first, then the scanner attempts to inject it, and finally it looks for the payload's presence in the application's content in unescaped form. In fact, this the approach used by ZAP for detecting both reflected and persistent XSS vulnerabilities. The only difference is that for stored XSS, it performs a follow-up crawl to determine on what page the payload appears, whereas for reflected XSS it only needs to look at the response to its malicious request. ZAP tests several different payloads - script tag inclusion, JavaScript callback for an invalid image on error event, SVG tag on load event, and b tag on mouse over event [37, 67].

One of the static analysis approaches that is also used by ZAP or Nikto is analysis of XSS countermeasures defined in HTTP headers [32, 37]. The two main HTTP headers in this regards are `X-XSS-Protection` and `Content-Security-Policy`. The former is more of a legacy option - as stated by MDN, the associated XSS filters are retired in Edge, Chrome removed XSS Auditor and Firefox has completely omitted its implementation. Nevertheless, it is worth mentioning that this header was intended as an instruction for the receiving browsers to stop execution upon detecting reflected XSS. This specific feature is now *replaced* by content security policy which can completely block execution of inline JavaScript [31].

An alternative approach proposed by Duchene et al. in 2014, named KameleonFuzz, is a more complex solution for reflected and stored XSS that first learns the application model based on which it generates malicious inputs. In the learning phase, the scanner learns how to navigate the application, while observing reflection on certain parameters (using taint tracking). In the input generation phase, it employs a genetic algorithm for the malicious input evolution expressed in its own attack grammar [74].

Similarly to KameleonFuzz, DOM-based XSS detection is often done via taint tracking - a method in which a specific input is marked as tainted and then its path is tracked as it propagates through the application

(sinks for this value are also marked as tainted). This specific approach was first used in the Firefox-based DOMinator tool. The issue is that it can be rather time-consuming. This can be partially avoid by utilizing pretrained ML classifier that acts as a filter prior to taint tracking and, thus, reduces the search space [75].

3.5.3 Cross-Site Request Forgery

In terms of CSRF vulnerabilities, one of the static analysis approaches that can be seen being used in tools like ZAP or w3af is testing for the presence of CSRF countermeasures [37, 39]. As previously discussed, a common prevention technique is the use of CSRF tokens which should be unpredictable values that are required to be sent alongside legitimate requests to the server. As such, it is possible to look for these tokens in either all forms present on a website or just the forms that seem to be transferring sensitive-enough data. Although, determining which forms would fit this description might be problematic.

The issue with only searching for known implemented countermeasures is the potentially high false positive / false negative rate. This is because not all forms need CSRF protection - for instance search fields. On the other hand, the fact that a token field is present does not imply the server-side generation and/or verification is correctly implemented. For that reason, it might be worth considering a more advanced approach, such as Mitch - a machine learning-based detection system for CSRF. It works by comparing so called *sensitive HTTP requests* from the perspective of a legitimate authenticated user and from the perspective of a malicious user, i.e. it requires two different test accounts to achieve high precision results. HTTP requests are deemed sensitive based on a classifier trained using supervised learning [76].

It is worth mentioning that CSRF vulnerabilities can be partially mitigated if `SameSite` attribute is introduced for relevant cookies. It is also possible to employ other countermeasture for which a scanner may look, such as `Referrer/Origin` headers or `X-Requested-With` [76].

4 | Chromium Extensions

4.1 Introduction

In 2008, Google released its web browser - Google Chrome [77]. Simultaneously, they also open-sourced the underlying code named Chromium, or the Chromium project. Subsequently, Google Chrome soon became the dominant browser on the market when, in 2012, it surpassed Microsoft's Internet Explorer [78].

In the following years we have seen multiple browsers being based on the Chromium project, such as Avast Secure Browser or Microsoft Edge. This only strengthened the position of *Chromium* on the browser market, making it the leading platform for which to develop extensions since extensions are transferable between Chromium-based browsers.

While the name *Chromium*, or Chromium browser, may be seen synonymous with the specific product Google Chrome, for the purposes of this thesis it shall encompass the whole set of Chromium-based browsers. Analogously, browser extensions developed for these Chromium-based browsers shall be referred to as Chromium extensions.

4.2 Distribution

To install a Chromium extension, the user can select one of multiple ways to do so. The following list summarizes all the distinct options as of April 2022 [79]:

- from an online store,
- from source files (unpacked extension),
- through policies,
- through preferences JSON file,
- through Windows registry.

Probably the most common and well-known options is installation from the Chrome Web Store, which is host to more than 137 thousand extensions [80]. It closely resembles Android's Google Play, meaning that the installation process is a matter of a few clicks and a review of the required permissions. It should be noted that Chrome Web Store is not strictly limited to the Google Chrome browser. In fact, accessing the store from Microsoft Edge causes an informational banner to appear, notifying the user that they can install extensions directly from Chrome Web Store. Moreover, Microsoft Edge users may choose to install extensions from the Microsoft Edge Add-ons store.

For managed environments, it might be the case that the same extensions must be installed on a subset of devices. An automated solution for such scenarios may be, for instance, through policies. Both Google Chrome and Microsoft Edge allow organizations to use various types of policies, which may be used for setting home screen, default search provider, blocking specific remote sources, etc. Using these policies, it is also possible to block, allow, or enforce installation of specific extensions [81].

Administrators have one more platform-specific option. For Linux and MacOS, they may choose to include preferences JSON files. Each extension, which must be installed, requires its own preferences file placed in

the browsers *External Extensions* subfolder, where the filename must match ID of the extension in question. Contents of that file then describe how to acquire said extension (e.g. from a shared network directory). Analogously, Windows machines require the administrator to create entries in the browsers registries, one entry per extension. The value of that registry than contains information on how to procure the extension [79].

Mainly for development purposes, it is also possible to load a Chromium extension directly from source files. This approach, however, requires that the user enables developer mode within the extensions panel.

4.3 Architecture

In general, a browser extension is a certain piece of code that runs in the context of the browser and aims to enhance the user's browsing experience in some way. Such extension could, for instance, block ads on a website or change its color scheme to increase contrast between text and background. Given the nature of browsers, these extensions are usually created with common web technologies used for creating regular websites, such as HTML, CSS, and JS.

Chromium extensions are built around a single manifest file that provides metadata about the extension itself - name, required permissions, dependencies, copyright information, etc. The existence of a manifest file is the only requirement set by Chromium to constitute an extension. It then tells the browser where to find the actual code, as well as user interface definitions. Other building blocks of an extension are: background scripts, browser UI elements, content scripts, and options page.

4.3.1 Extension Manifest

The manifest refers to a JSON-formatted file that must be named *manifest.json* with an object at its root and at least the following three properties:

- `manifest_version`: an integer representing what feature set the extension requires (e.g. 3),
- `name`: a string identifier of the extension visible in the extension management UI (e.g. `"Dolos"`),
- `version`: a string of dot-separated numbers identifying the specific version of the extension (e.g. `"0.0.1"`).

Google also specifies a set of recommended properties:

- `action`: an object describing the appearance and behavior of the extension's icon in the top toolbar (next to the navigation bar),
- `default_locale`: a string representing the default subfolder containing a set of localized files,
- `description`: a string containing a short plain text description of the extension,
- `icons`: an object containing key-value pair for each available icon (key corresponds to icon size, value corresponds to the icon's file location).

The remaining properties are considered optional and their use depends on the specific use case. The exhaustive list of all possible properties for manifest version 3 can be found at https://developer.chrome.com/docs/extensions/mv3/manifest/.

4.3.2 Background Scripts

Background scripts are JavaScript files specified in the *background* manifest property. These scripts are intended for event handler definitions/registrations. For instance, a background script could register an event listener for the *chrome.ContextMenus.onClicked* event which is invoked after opening a context menu within the document. To avoid downgrading performance, Chromium keeps these scripts loaded only for a limited amount of time. More specifically, should a background script stay idle for a prolonged period of time, Chromium will unload it and only load it once a watched event is fired.

4.3.3 User Interface

To facilitate user-extension interaction, the developer can choose from various UI elements of the browser. *Browser's UI* is understood as any graphical user interface elements that are not part of the website (document) itself but rather the remaining parts of the browser. Probably the most common UI element an extension can use is the *action* button. It appears as an icon displayed (usually) to the right of the navigation bar. The developer may, for example, choose to display a settings form upon clicking the button. Before manifest version 3, Chromium distinguished between *browser action* and *page action* which both had the same functionality but differed in scope, i.e. page action was only active for specific pages while browser action remained active throughout the whole browsing session.

4.4 Permissions

Since Chromium extensions are, essentially, a web application running inside the browser, they automatically inherit all of the permissions a website would have. This can be sufficient for many extensions, however, for a security testing tool it is worth exploring what other options there are.

Before describing the individual and potentially useful permissions or APIs provided by those permissions, we have to distinguish between certain categories or types of permissions. Using the extension's manifest, the developer may declare required permissions, optional permissions, and host permissions. Each of these categories is defined as an array of strings. In the case of required permissions, the manifest keyword is `permissions` and its contents should correspond to known permission names, such as `storage`. The same logic applies to optional permissions (keyword: `optional_permissions`) with the key difference that these permissions are requested during runtime instead of requesting them in advance. Host permissions contain strings of patterns which identify to what website the extension should have access (keyword: `host_permissions`) [79].

One of the benefits of explicitly declaring what permissions are required is damage control in case of extension compromise. As of April 2022, Chromium offers more than 60 distinct permissions. For that reason, I selected only several of these that I believe to be potentially relevant [79]:

- `certificateProvider`: provides access to TLS certificates associated with a host,
  (API: `chrome.certificateProvider`)
- `contextMenus`: allows override/extension of Chromium's context menu,
  (API: `chrome.contextMenus`)
- `cookies`: allows access to, modification of, and event listening for cookies,
  (API: `chrome.cookies`)
- `scripting`: allows execution of JavaScript snippets in different contexts,
  (API: `chrome.scripting`)
- `storage`: provides access to Chromium's storage,
  (API: `chrome.storage`)
- `tabs`: gives access to open/active tab information and allows interactions with them.
  (API: `chrome.tabs`, `chrome.windows`, ...)

4.5 Networking

Normally, Chromium extensions tend to limit their functionality to the currently opened web page, i.e. once the page is loaded, the extension processes that page and optionally accesses additional APIs to provide more information. However, many of the examined existing tools rely on sending multiple network requests to determine how the server reacts and to analyze different parts of given web application. And these requests are not necessarily restricted to HTTP only - though, it seems to be quite prevalent. In section 2.6, we have seen that some of the tools also relied on lower-level protocols (with respect to the OSI model) like TCP,

UDP, or ICMP. Therefore, it is necessary to examine what are the possibilities and limitations of networking when running in the Chromium extension context.

The following sections are dedicated to the analysis of various protocols that are being used in desktop tools from the perspective of an extension. The aim of each analysis is to determine whether it is possible to communicate over given protocol, understand its limitations and alternatives.

4.5.1 Fetch API

The simplest to assess is the HyperText Transfer Protocol (or *HTTP*). As it the basis of the modern web, it is unsurprising that a browser extension can communicate over this protocol in the same manner a web page's code could. In the context of JavaScript, developers can send HTTP requests and process the responses using the Fetch API which allows specifying the HTTP method, cross-origin policy, cache mode, inclusion of credentials, additional headers, and more.

The Fetch API provides almost unlimited access to the HTTP protocol. The main issue becomes visible once the developer tries to view non-standard HTTP headers - both in requests and responses. Browsers define so called *forbidden header names* which are inaccessible from the JavaScript context and can only be read or modified by the browser. Chromium, and other browsers as well [31], splits the definition into two categories - headers starting with either prefix `proxy-` or `sec-` and headers listed in the forbidden header fields as can be seen in Source code 4.5.1. Currently, Chromium does not allow modification of the `user-agent` header even though it should [82]. This is due to a bug reported back in December 2015 but is still unresolved as of April 2022 [83].

```
bool HttpUtil::IsSafeHeader(base::StringPiece name) {
  if (base::StartsWith(name, "proxy-", base::CompareCase::INSENSITIVE_ASCII) ||
      base::StartsWith(name, "sec-", base::CompareCase::INSENSITIVE_ASCII))
    return false;

  for (const char* field : kForbiddenHeaderFields) {
    if (base::LowerCaseEqualsASCII(name, field))
      return false;
  }
  return true;
}
```

Source code 4.5.1: Chromium's safe headers: services/network/cors/cors_util.cc

4.5.2 WebSocket API

While HTTP is great for occasional retrieval of data from the server, it is not well suited for facilitating bidirectional communication between a server and a client. In 2011, a new protocol defined by RFC6455 named *The WebSocket Protocol* tried to address this problem. WebSockets serve as a thin layer on top of TCP that adds mainly web-origin based security to the packets. The WebSocket API exposes a `WebSocket` object which can be used for establishing such connection. In order to connect to an endpoint, RFC6455 defines a specific URI schema that must be used [84]:

```
ws-URI = "ws:" "//" host [ ":" port ] path [ "?" query ]
wss-URI = "wss:" "//" host [ ":" port ] path [ "?" query ]

host = <host, defined in [RFC3986], Section 3.2.2>
port = <port, defined in [RFC3986], Section 3.2.3>
path = <path-abempty, defined in [RFC3986], Section 3.3>
query = <query, defined in [RFC3986], Section 3.4>
```

The process of establishing a new connection from a web browser can be simplified to the following steps (excluding proxies and TLS):

1. The client provides a WebSocket URI as described above, at which point the connection is in state `CONNECTING`.
2. The client establishes a connection to the server if found.
3. The client sends an opening handshake in the form of a valid HTTP request with several constraints including, but not limited to, the following:
   - The `GET` method is used.
   - The HTTP version is at least `1.1`.
   - The `Host` header and the requested URI must correspond to the WebSocket URI that initiated this connection.
   - The request must contain `Connection` header including the value `Upgrade` and the `Upgrade` header with the value `websocket`.
4. The client waits for server's response.
5. The server validates the handshake and returns appropriate error code if deemed invalid.
6. If the server accepts the connection, it must respond with the code `101` and an HTTP response with similar constrain to those posed on the client's opening handshake.
7. The client validates the server's handshake.
8. If deemed valid, the connection goes to state `OPEN`.

For the full specification of an opening handshake, see [84, p. 14-25]. From the perspective of a developer, connecting, for instance, to localhost over port 8080 in JavaScript using WebSocket API may then look like this:

```
const socket = new WebSocket("ws://localhost:8080/");
```

4.5.3 WebRTC API

One specific scenario, for which WebSockets are not as suitable, is audio and video streaming between clients. WebRTC API is a technology to address this shortcoming by enabling peer-to-peer communication between server-client or browser-browser. This makes it the appropriate technology to use when, for instance, implementing a teleconferencing platform. A peer-to-peer connection is established through the `RTCPeerConnection` interface. WebRTC relies on several different technologies that make it possible, including ICE, STUN, NAT, TURN, and SDP [31].

Interactive Connectivity Establishment (or *ICE*) is a technique for establishing connection between two peers. Its purpose is (mainly) to overcome problems associated with operating offer/answer protocols going through Network Address Translators (or *NATs*). It tries help the peers find an appropriate path by which they can communicate with respect to the network topology between them. While other protocols can be used directly to achieve similar results, they tend to have topology-related pros and cons which could complicate the implementation. ICE tries to simplify that [85].

Network Address Translator (or *NAT*) is a router function proposed in 1994 that aimed to lower the impact of IPv4 address depletion. It takes advantage of the fact that not all devices need a public and globally unique IP address at all times. It works by replacing source addresses of packets passed through a primary router within a network by the router's public IP address - this way, devices behind a NAT do not need their own public IP address. However, the router must keep track of these address bindings in its translation table to ensure proper handling of incoming response(s) [86].

Session Traversal Utilities for NAT (or *STUN*) is a protocol which aims to help other protocol overcome potential issues associated with the logic of NATs. STUN provides two main functionalities - determine machine's IP address and port binding allocated by a NAT and to keep this binding alive [87].

Traversal Using Relays around NAT (or *TURN*) is a protocol intended to allow peer-to-peer communication through a dedicated relay node. Unlike other similar protocols, a single relay can be used for communication with multiple peers. The idea of TURN is to allow hosts behind a NAT to request other hosts to be TURN servers (relays). Peers can then connect to this server to communicate between each other [88].

Session Description Protocol (or *SDP*) is a protocol used for negotiation of multimedia connection details. This is usually done by other protocols that exchange SDP messages which, among other fields, contain negotiation terms for networking details (i.e. connection) and bandwidth [89].

It is important to note that, while WebRTC uses all of these protocols in order to establish and maintain a real-time connection, the developer does not have direct access to them. Browsers expose certain elements of it, e.g. the ability to add ICE candidates or creating an offer for SDP [31].

4.5.4 Lower-Level Protocols

There are several use cases, such as port scanning or tracerouting, where having access to lower-level protocols with respect to HTTP (in the OSI model) is potentially a requirement. The main motivation for the use of Transmission Control Protocol (or *TCP*) and User Datagram Protocol (or *UDP*) packets is to mimic nmap's TCP/UDP port scanning capability. However, neither regular JavaScript code embedded in a web page nor a Chromium extension has access to such low-level communication as of March 2022. During my investigation, I have found that, while it was possible to establish communication over both TCP and UDP channels with the use of `chrome.sockets.tcp` and `chrome.sockets.udp` APIs, both are now deprecated. The migration guide recommends to use WebSockets API instead [79]. To the best of my knowledge, those two deprecated APIs were the lowest-level networking protocol APIs available to a Chromium extension directly.

4.6 Storage

Scenarios such as storing results of previous test runs or keeping a local copy of a remote database are just a few examples where an extension might require using storage. For the purposes of this section, *storage* refers to a space on a machine which is running a browser extension or displaying a website where either of these pieces of software can save its data. Therefore, any form of synchronization with cloud storage is not considered.

Each origin, i.e. a combination of protocol, host, and port, has access to several different APIs to store data in the browser. It also has a certain storage quota which can be retrieved using a `StorageManager.estimate()` call, or alternatively `navigator.storage.estimate()` if not available in given browser. In the case of Chromium, this quota can be rather high since the its only goal is to leave *enough* space on the device for it function properly. Previously, this threshold was set to 2 GB but in later versions, Chromium started taking into considerations devices with small total storage (e.g. 8 GB) in which case the threshold lowers as well [82]. The logical storage units that can use this quota include: IndexedDB databases, Cache, Web Storage (localStorage / sessionStorage), and a few more.

The first, and probably the most known, option is **Web Storage** which provides the developer with a key-value pair storage. In the current implementation, the developer may choose between **localStorage** and **sessionStorage**, where the former persists after closing associated tab whereas the latter does not. They also both have size limitations. While the precise value may vary from browser to browser, the limits seem to be in the order of several megabytes, sessionStorage being generally smaller. An important aspect of both of these storage types is that their APIs are synchronous and thus blocking when data needs to be retrieved [31, 82].

An alternative to Web Storage that is more suitable for large amounts of structured data is **IndexedDB** - a transactional JavaScript-based database system. This allows for a more flexible database storage than a tranditional RDBMS would offer. Unlike Web Storage, its API is fully asynchronous, meaning it will not block the application until results are received [31]. To the best of my knowledge, there is no hard size limit to IndexedDB in Chromium other than the one imposed by previously mentioned total storage quota.

As noted above, there are also other alternatives which may be used for certain scenarios, such as Cache for custom request caching, but those are not relevant in the scope of this thesis. An important option, however, is Chromium's **chrome.storage** API. In essence, it provides the developer with an upgraded version of localStor-

age - it has the same functionality but is asynchronous, can persist even when using incognito mode, and can be easily synchronized with Chrome sync. However, it does require additional manifest permission: `storage`. Moreover, it has a 5MB limit on the total size of saved data, unless the extension has `unlimitedStorage` permission [79].

4.7 Limitations

The goal of this section is to analyze various aspects of Chromium extensions and determine the limits of its environment. For the purposes of this thesis, I analyzed the following factors that may pose functional limitations:

- port scanning / host scanning,
- tracerouting,
- storage.

4.7.1 Port Scanning

Many of the tools mentioned in section 2.6 perform various logical checks that are more based around analyzing HTTP responses which can easily be done in JavaScript, as well as many other programming languages. For instance, Wappalyzer detects certain JavaScript libraries based on commonly known file names and keywords. However, when it comes to general networking, the browser extension environment is rather limited.

One popular tool used for network analysis is the previously mentioned nmap CLI tool. Amongs its main use cases is UDP/TCP port scanning. This is something that would have been possible in the past using `chrome.sockets.tcp` and `chrome.sockets.udp` APIs which are now deprecated. The recommended alternative to that is the WebSocket API. With that in mind, I investigated if there have been any successful port scanning attempts using either WebSockets or WebRTC. During my investigation, I found analyses and proofs of concept for both technologies.

Starting with WebRTC, Jacob Baines published an article on Medium in 2019 discussing how this technology can be used for port scanning. His approach to determining if a certain port is open or closed is based on the `hostCandidate` value returned on `ICECandidateError` event [90]. However, per W3C's recommendation, ICE candidates can be prohibited to specifically prevent port scanning [91]. This property is also considered deprecated by MDN [31]. Therefore, while it may be possible to exploit this behavior in older versions of Chromium, it should no longer be possible. I tested the viability of this approach in Microsoft Edge v100.0.1185.44 and I was not able to recreate the results shown in the original article. To the best of my knowledge, there is currently no approach that would allow port scanning using WebRTC in the latest version of Chromium.

In 2016, Tom Gallagher noticed that the WebSocket specification allowed for side-channel attack to scan local network from within the browser. This was possible because of the difference in time it took for the WebSocket connection attempt to fail - a longer time period would suggest no response received, while shorter responses would imply an open port. Gallagher submitted a internet draft improving the WebSocket specification so that such timing attack would be more difficult. This draft expired in 2017 and, to the best of my knowledge, was not incorporated into any follow-up specification [92]. However, it seems that, at least Chromium, implements a certain safeguard to avoid such attack. Nikolai Tschacher noted in his article on port scanning with WebSockets that Chromium specifically tries to postpone user callback to make it harder to determine why connection failed. Despite that, Tschacher noticed there is still a certain difference in timing on average when comparing open and closed ports. His proposal was to compare 30 measurements of a possibly closed port against 30 measurements of the desired port to determine if the port is open [93]. While I did not find any change that could render this approach unusable, I was no longer able to reliably determine the state of a port with this method.

Fortunately, there is a more reliable and possibly future-proof alternative that can be used within the context of a browser extension. Tschacher mentions that trying to establish a new WebSocket connection seems to return different error messages for closed and open ports, i.e. we could infer the port state based on the

error. Specifically, closed ports seemed to return `net::ERR_CONNECTION_REFUSED` while open ports lead to various other messages. The problem is that JavaScript, in the context of a website, cannot access the error message. The user is able to view it in the developer tools panel but not from code [93]. However, a Chromium extension with an active devtools panel can access the `chrome.devtools.network` API that can read the network requests and responses [79], including WebSocket error messages.

It is important to note here that this API provides access to networking done by the currently inspected window. Therefore, if an extension should perform port scanning using this approach, it must initiate the WebSocket connection in the website's context through script injection which requires additional permission: `scripting` (API: `chrome.scripting`) [79]. A notable downside of this scanning approach is that Chromium forbids WebSocket connections to so called *unsafe ports* which fail with error message `net::ERR_UNSAFE_PORT`. These are also referred to as *restricted ports* and are listed in Chromium's `net/base/port-util.cc` file - as of April 2022, Chromium defines 80 restricted ports (several of which may be re-enabled under certain conditions). Ports 22 (SSH), 25 (SMTP), and 389 (LDAP) are just a few examples [82].

4.7.2 Traceroute

As previously discussed, the tracert utility in Windows uses ICMP packets to determine trace the packet's route through the network. Based on my investigation into networking options of Chromium extensions, the ICMP protocol does not seem reachable from code, as it is on a lower level than TCP and, in turn, WebSocket Protocol. However, the logic used with ICMP relies on incrementally increasing the TTL value of sent packet which could can be set on TCP/UDP packets as well.

Unfortunately, neither WebRTC nor WebSockets provide access to the packet TTL value, rendering it impossible to replicate this functionality directly from browser. An alternative to that might be to utilize a custom server that would run traceroute on behalf of the extension when, for instance, requested via the server's API. That, however, poses certain risks of misuse if not properly limited.

4.7.3 Storage

Based on my findings summarized in section 4.6, I would conclude that there are no actual limitations of Chromium extensions with respect to storage. Nowadays, web applications, as well as browser extensions, have enough storage (in terms of capacity) available to suit their needs. While not all related APIs may be easy to use, for instance IndexedDB's API [31], the developer has a variety of options from which they can choose.

5 | Extension Design

One of the goals of this thesis was to design and implement a proof of concept solution that would demonstrate certain parts of the previously described theoretical background. This chapter is dedicated solely to describing the proposed solution, named **Dolos**. Firstly, I shall describe the various requirements for the solution. Secondly, I will summarize the core architecture of the extension. The following section will describe individual functional components that perform the issue detection. Lastly, I shall describe the user interface of the extension.

5.1 Requirements

To following sets of requirements were compiled based on the functionalities and traits of existing tools (regardless of their platform or distribution method), the discovered limitation of Chromium extension environment, and the requirements as specified in the thesis assignment. These are:

R1. The extension can be run in a Chromium-based browser.

R2. The extension must be able to crawl selected website.

R3. The extension must implement passive scanning capabilities.

R4. The extension must implement active scanning capabilities.

R5. The extension must detect certain cases of potential sensitive information exposure, namely IP addresses, GUIDs, and email addresses.

R6. If a certain component and its version are detected (e.g. jQuery v1.8.0), the extension must report if detected component has any known vulnerabilities with assigned CVE IDs.

R7. The extension must be able to detect reflected Cross-Site Scripting vulnerabilities.

R8. Any detected weakness or vulnerability must be reported to the user.

R9. Any reported finding must include recommended remediation steps (if known), as well as additional resources on said finding where applicable.

R10. The user must have the ability to specify the website on which the test will be run.

R11. The user must be able to view previously run tests with their details.

R12. The user must be able to delete any of the previously run tests.

5.2 Architecture

This section is dedicated to the core architecture of the extension. While analyzing source code of several of the open-sourced tools listed in section 2.6, I noticed certain architectural similarities which I decided to adopt in my solution as well. One of these common traits was a addon/plugin system to provide high extensibility, as well as easier maintenance.

5.2.1 Used Technologies

Firstly, I had to decide what specific technologies (languages, frameworks, etc.) I would use to create this proof of concept. While my selection may not be the most optimal configuration for this task, it allowed for rapid development, making it possible to focus on the actual web security logic rather than various technical details. The extension was built with the following main technologies:

- TypeScript,
- React,
- antd,
- Webpack.

As described in section 4.3, a Chromium extension is built using common web technologies such as HTML, CSS, and JavaScript. It would, therefore, be completely possible to use only these 3 technologies for the whole solution. However, the development may get a bit repetitive or tedious, as well as complicated when it comes to more complex projects. Also, given that JavaScript is a loosely typed language, it is easy to introduce unexpected bug and it is hard to debug. For these reasons, I decided to use the popular superset of JavaScript called **TypeScript**. As the name suggests, it introduces additional constraints on types, however it provide much more than that. TypeScript also provides developers with concepts like type definitions, interfaces, etc. Important note for TypeScript is that it is a compiled language that gets transformed directly to pure JavaScript [94].

To simplify user interface development, since it is not the main focus of this thesis, I chose a combination of two libraries - **React** and **antd**. React provides the developer with a component-based foundation to create UI efficiently. It simplifies state management and dynamic rendering and assists to avoid repetition [95]. However, using React itself would require defining all the components from start which is why I chose to use antd, a React UI library containing many different components, such as menus, icons, buttons, or forms, that follow the Ant Design language [96].

The aforementioned technologies are usually used in unison with a certain bundler (e.g. Webpack) or task runner (e.g. Gulp). For this thesis, I selected **Webpack** with its various plugins due to my prior experience with it. Its purpose is to orchestrate the compilation and resolution of various dependencies, while combining them into a smaller production-ready output.

5.2.2 Project Structure

The implemented solution has three main components, or blocks, which communicate with each other to create the full product - **extension scaffolding**, **Dolos**, and **Dolos UI**. Both Dolos and Dolos UI shall be described in the upcoming sections and, thus, will only be briefly summarized here.

As the name suggests, **extension scaffolding** represents the part of code which facilitates the extension environment and acts as a proxy between the browser and the actual content of the extension. Given that web developers tend to work heavily with their browser's developer tools, I decided to implement the extension's UI as a dedicated panel there which is discussed later in section 5.4. To achieve this, the extension needs a manifest file such as the one in the snippet 5.2.2. Next, it has the `DevTools.html` file that registers a new developer tools tab and finally the `DevToolsPanel.html` file which actually includes and initiates the UI. This way, UI definition is clearly separated from the code necessary to constitute a browser extension which may be useful should the application be ported to a different platform.

Next, the **Dolos** module or component is what contains the functional (i.e. security testing) logic. It is centered around a single instance of the `Dolos` class. It presents the main object of the whole extension and, as such, it is responsible for test orchestration, state management, test event dispatching, and more.

Lastly, **Dolos UI** refers to a separate part of the code which is solely responsible for rendering the appropriate components while closely integrating with the Dolos instance and listening to its events.

```json
{
  "name": "Project Dolos",
  "description": "1-click solution to detect security issues in your web
  ↪  application.",
  "manifest_version": 3,
  "version": "1",
  "permissions": [
    "tabs",
    "storage"
  ],
  "host_permissions": [
    "http://*/",
    "https://*/"
  ],
  "devtools_page": "DevTools.html"
}
```

Source code 5.2.1: Manifest for the Dolos extension.

5.2.3 Website Model

Among the most important parts of the Dolos component is the `Website` class. Its purpose is to represent the currently known (i.e. explored) parts of the tested website with any additional details found by run modules - it is intended to be gradually extended and refined.

It consists of several important parts which need to be stored, each one representing an specific component or aspect of the real website that is being tested. To avoid overcomplicating the solution, the implemented solution only considers these entities:

- website,
- web page,
- server,
- application entry point,
- dependency.

While this list is not exhaustive, it should cover most of the crucial parts. Other interesting entities to consider would be, for instance, content management system, database system, hosting provider, or DNS records to name a few. From the perspective of this extension, a website object consists of multiple web pages, which may have connections between each other, and it is hosted on a server. Each web page can require certain dependencies, such as CSS or JavaScript files. Moreover, each page can also contain an application entry point which represent any component (or URL) of the underlying website that accepts user input. That could, for example, be a registration form.

5.2.4 Dolos Instance

At the core of the extension is a single class named `Dolos` which follows the singleton design pattern. It aims to provide the main point of contact needed for the user interface. As such, it is responsible for the following functionalities:

- orchestrating test runs and providing their results,
- parsing URL of the currently open and active tab in the browser,

- loading, saving, and deleting history of test runs,
- generating non-conflicting test run names,
- registering and removing event listeners for basic test-related events.

Each item on the list shall be discussed in more detail in the following several paragraphs.

Test Orchestration

Test orchestration in this scenario refers to the process of processing test run requests and initiating the appropriate tests instances. Given the asynchronous nature of the task in question, it is possible to initiate multiple tests runs at the same time. This is not an expected use case but is still a possibility, in which case tests might be fighting for resources, potentially leading to poor performance. However, for the purposes of this thesis, this scenario is not considered.

Current URL

In order to provide the user with a 1-click solution, it is reasonable to assume that, when users open the developer tools or specifically the extension's tab, they wish to run a test on the currently viewed website. While the tested URL specification is done within the user interface to allow user modifications, the interface should automatically load the current URL by default. To minimize the number of source files that are accessing Chromium's APIs, as well as to avoid repetition, I decided to delegate this functionality to the `Dolos` instance. There are multiple ways to achieve this goal, such as running a custom script in the context of the viewed page that returns the `location` instance, but I was not able to find a solution that would *not* require additional privileges. This is why, in snippet 5.2.2, it can be seen that the this solution requires the `tabs` permissions, as it provides access to the current active tab's information.

Storage

In chapter 4, multiple options for data storage were explored and discussed. In order to select an appropriate one for this solution, a few aspects needed to be considered. Firstly, the data that will be stored will most likely be rather small. The only type of data that requires storing is the test run history and potentially certain settings. With enough optimization, this type can be greatly reduced in size and could essentially be in the order of several kilobytes per test run, depending on the number of findings. Secondly, the test run data is structured and well defined. Each test run only really needs to store a few details about the run configuration and then a list of findings. Thirdly, the data needs to persist as long as the user wishes and, thus, should not get automatically purged by the browser.

Based on these three aspects and assuming only one option should be selected, I could conclude that there are 2 main candidates for storage - `localStorage` and `chrome.storage`. In this specific case, I selected the latter, as it provides a few advantages that could prove useful in any follow-up work, i.e. `chrome.storage` can persist in incognito mode and can be synchronized with Chrome sync. There are, of course, certain scenarios where the extension could benefit from the use of other options as well to yield better performance. Specifically, it might be the case that the extension needs to perform large-scale value search for which JavaScript is not well optimized but, for instance, IndexedDB could be. Moreover, even the test run history could be stored in IndexedDB, as the data is structured. However, for this specific implementation, it might put unwanted overhead on the development process given its API's relatively high complexity compared to `chrome.storage` or `localStorage`.

Non-Conflicting Names

Among the additional functionalities implemented in this solution is the option to assign names to test runs. This can increase clarity when searching through multiple runs in the extension's history. And, while in theory there is no reason why two tests could not have the exact same name, it may not be the expected behavior given that various applications, as well as file systems, tend to avoid these *potential conflicts*. For that reason, I implemented a simple procedure that determines if given run name is already taken and, if so, it adds

appropriate increment at the end. This aims to emulate the behavior commonly used in file system managers to which most users are likely accustomed.

Event Listening

One of the key considerations for the user interface was that test runs may take significant amount of time to finish. This lead me to realize there needs to be notification and/or continuous status update system to ensure the user does not feel as if the application froze while waiting for their test results. In order to achieve this goal, I implemented an event listener registration and deletion logic for both the `Dolos` instance as well as test instances.

In the former case, I defined a set of several *rough* events that might occur during the lifetime of a `Dolos` instance, specifically:

- new test was initiated,
  (`DolosEvent.NEW_TEST_INITIATED`)
- the previously initiated test was completed,
  (`DolosEvent.TEST_FINISHED`)
- specific test was successfully deleted from history,
  (`DolosEvent.TEST_DELETED`)
- the latest snapshot of the history and the settings was loaded,
  (`DolosEvent.SNAPSHOT_LOADED`)
- the latest snapshot of the history and the settings was saved.
  (`DolosEvent.SNAPSHOT_SAVED`)

In the latter case, I defined another set of event types closely resembling the individual phases through which the test goes and their rationale will become clearer in the following subsection on the modular test system:

- the test was initiated,
  (`TestEvent.TEST_INITIATED`)
- the test was canceled,
  (`TestEvent.TEST_CANCELED`)
- the test was completed,
  (`TestEvent.TEST_COMPLETED`)
- a test step was initiated,
  (`TestEvent.STEP_INITIATED`)
- the previously initiated test step finished,
  (`TestEvent.STEP_COMPLETED`)
- a test module was initiated,
  (`TestEvent.MODULE_INITIATED`)
- the previously initiated module finished,
  (`TestEvent.MODULE_COMPLETED`)
- the currently running module reported a new finding.
  (`TestEvent.NEW_FINDINGS_FOUND`)

Note that not all of these events are fully utilized in the final solution but rather serve as a placeholder for any future work. This applies to `TestEvent.TEST_CANCELED`.

```
export interface IModule {
    readonly friendlyName: string
    run(website: Website): Promise<Finding[]>
}
```

Source code 5.2.2: IModule interface used for defining new test modules.

5.2.5 Modular Tests System

One of the common features seen among a certain number of the open-sourced tools, such as ZAP or Nikto, was that they tend to have one script, plugin, or module per single issue/vulnerability and the user can choose which ones should be run in any given test. While the user interface implemented in my solution does not yet allow for such level of control over what should be run, I decided to replicate this approach from the code's perspective so that it simplifies development of new detectors as well as creation of new predefined test types. This should also simplify implementation of such feature in the user interface in the future.

For this thesis, I devised a set of three key components with simple hierarchy that form a test. At the lowest level is a module which performs the security testing logic. Modules are then grouped into steps which are simply named collections. Lastly, there is the test itself which consists of steps and performs the orchestration, as well as invokes test events when needed.

A **module** is a class that implements a very simple interface named `IModule` which can be seen in snippet 5.2.5. It should serve only one purpose, such as detecting reflected XSS vulnerabilities or port scanning, and should perform its logic only when called via the `run` method. During that call, it can both modify the passed `Website` instance and return any actual findings - here, a finding represents a potential issue that shall be reported to the user. The idea is that a module does not necessarily need to detect issues but it can also refine the details known about a website.

A **step** is an object type (`TestStep`) that, currently, only associates a name (string) with an array of module instances. These instances are then run in sequence as they are specified in the step. The purpose of this unit is to provide a more logical structure to the user. In the future, it might be worth considering parallelism for modules in the same step, in which case this structure could prove useful. As with modules, a step should be dedicated to a single type of functionality, such as dependency analysis.

A **test** is a class (`Test`) that holds all the details about a specific test run - when it started and when it finished, what is the current status, what steps should be run, etc. - and it orchestrates the test run. The orchestration logic is a simple step by step, module by module approach in the order of their definition for given test. After each module and step, it fires appropriate events primarily for UI purposes which will become more apparent later in this chapter.

5.3 Implemented Modules

The key requirement for this proof of concept is to have an actual security testing logic in place. As such, I was able to implement various types of modules based on my analysis described in chapter 3. These range from the simple regular expression-matching modules to the more complex ones that actively test malicious payloads to find vulnerabilities.

5.3.1 AdminInterfaceEnumerator

Following WSTG's recommendations from step 2 - *Configuration and Deployment Management Testing*, administrator interfaces provide a gateway to privileged access and, as such, might (and most likely will) be the adversaries' target [19]. For this purpose, I implemted the `AdminInterfaceEnumerator` module.

As hinted in Security Testing, the selected approach for this extension is black box testing, as it would be impractical to provide the server-side source code to the extension. One of the possible solution to this problem

with said approach is a regular dictionary attack using a set of well-known default administrator interface URLs. This is can be especially effective on users of popular content management systems because they tend to have a default URL for the administration login. This not only provides the tester with another application entry point, broadening the attack surface, but it also hints as to what specific CMS might be in use.

The issue with this approach is that it is generally a time-consuming process to collect enough data for the dictionary to be effective. For this proof of concept, I only prepared a small sample based on WSTG. From the code's perspective, each known administrator interface is a an object of type `AdminInterface` where:

```
export type AdminInterface = {
    name: string,
    paths: string[],
    keywords: string[]
}
```

The attribute `name` serves as a human-readable identifier of the specific interface which shall be displayed in the extension's reporting section. The `paths` array then specifies what specific URL appendices should be tested. For the individual responses it receives, the module searches for specified `keywords` to determine if its guess was successful. This, at least partially, eliminates unwanted false positives, since only receiving an HTTP 200 response does not constitute a successful match. It might be the case that a website displays a custom 404 page but does not provide the correct HTTP response code.

5.3.2 CDNDetector

The purpose of this module is to exploit the well-documented and structured format of URLs used by content delivery networks which are commonly used to procure (mainly) JavaScript dependencies - generally available libraries and frameworks. These could then be compared with the National Vulnerability Database to see if, for instance, reflected XSS attack might be possible.

Following the ideas described in section 3.2, these URLs can be easily analyzed using regular expressions in order to extract the library/dependency name and version. However, a single CDN may use multiple patterns at the same time. For that reason, the module works by iterating over all of detected JavaScript dependencies and testing them against a set of known CDNs and their associated patterns. Each known CDN is an object of type:

```
export type CDN = {
    name: string,
    homepage: string,
    description?: string,
    patterns: CDNAssetPattern[]
}
```

From the perspective of the current implementation, only the `patterns` field is used and actually relevant to the detection. Each pattern is then an object of type:

```
export type CDNAssetPattern = {
    type: DependencyType,
    pattern: RegExp,
    name?: number,
    version?: number
}
```

At its core, it has the regular expression to be used against the dependency's URL. `name` and `version` fields are used as references to capture groups of the pattern so that it would be possible to extract those details. The reason for those two being potentially undefined (the `?` symbol) is that, while name will most likely be available, version might not since the URL may contain certain keywords, such as `latest`, instead which do not represent a *globally unique* identifier of the specific version. Those would then require additional parsing and lookup in order to determine what version is actually being imported.

5.3.3 DefaultCrawler

A crucial module to the whole web security testing extension is a crawler. This specific implementation, i.e. `DefaultCrawler`, performs a breadth-first traversal to a predefined maximum depth. It starts at the user-submitted URL which is used as the website's origin or root within the scope of the test. In each iteration, the crawler tries to visit a previously unprocessed URL, extract all links from its content and add them to the queue.

There are a few design considerations to note here. The crawler itself is only responsible for orchestrating when and what URL is visited. Dispatching the underlying HTTP requests, parsing them, extracting features, etc. is all done by the *Website Model* which, in this case, refers to a collection of classes representing the various components of a website as previously described in subsection 5.2.3. Therefore, the crawler only really interacts with the `Website` instance, passed by the currently running `Test` instance, through which it has access to the `Webpage` instance that is being visited. The motivation for this design decision was that it allows crawler module(s) to focus on the crawling strategy rather than the request/response and HTML parsing process.

That being said, `DefaultCrawler` is as of this writing the only crawler type, or strategy, implemented. The reason for BFS approach is that it provide a thorough and consistent view of the website. This ensures that rerunning the test would yield the same results. The problem with this strategy is that it might explore many uninteresting pages and exhaust a significant amount of resources.

5.3.4 DependencyHashLookup

In cases where JavaScript dependencies are not procured from a CDN, it is worth considering a hash comparison approach as mentioned in section 3.2. The assumption for this is that developers might download production-ready libraries from official resources and leave them unchanged. If that is the case, it is possible to precalculate hashes of common libraries for all their versions and do a comparison with content hashes of files used by the website.

The goal of this module is to perform this exact procedure in order to determine what dependencies are imported. For each dependency, it simply does a hash comparison between its content and the extension's annotated knowledge base. The main advantage of this approach is that is relatively fast and trivial to implement. However, it does have a few downsides as well. Firstly, it either requires a *good-enough* data source from which to pull this data or a significant amount of (partially) manual work. In my solution, I decided to use the freely available API from cdn.js which allows querying all of its libraries, including precalculated content hashes and library metadata. Secondly, the knowledge base quickly grows in size with every new library added to it.

In this solution, I decided to use an integrated knowledge base which would be shipped with the extension automatically. This is a reasonable solution for the purposes of a proof of concept but would require regular updates for production purposes. Overcoming this issue could be possible through the same API used to generate the knowledge base and store the data locally using the browser's storage APIs instead. This would minimize the size of the extension, as well as avoid being out of sync with the latest changes. Moreover, the locally stored knowledge could be more fine-tuned for the applications tested by the users instead of pulling many different libraries in advance.

5.3.5 EmailDetector

Building on the thoughts expressed in subsection 3.4.1, this module aims to detect potentially leaked email addresses on each of the visited web pages. It achieves this goal using a regular expression which it applies to the document's inner text of each web page.

The reason for explicitly looking into inner text rather than the whole document is that it lowers the number of false positives. As discussed previously, the email address specification is quite permissive in terms of what constitutes a valid address. One of the key aspect of its permissiveness is that it is not restricted to public internet and, in turn, the general leveled domain structure only. Due to that, even an address like `jon@doe` is still valid, yet in terms of the public internet, registering such address would not be possible. Nonetheless, if we consider even these addresses, certain false positives start to appear due to snippets such as this:

```
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?family=Roboto:wght@100&display=swap"
↪  rel="stylesheet">
```

This specific set of link tags is used to import the Roboto font from Google Fonts platform.

If the regular expression was run against the whole raw body of the HTTP response, suddenly we would see a *leaked* email address of `wght@100` which is clearly not expected. The aforementioned problems can either be eliminated by only searching in the rendered text rather than the whole response body or by restricting the expression to the more common address formats, e.g. `jon.doe@example.com`. Both solutions have their respective issues but should perform *well enough* in most common use cases.

While experimenting with this module, I discovered an interesting aspect that might be worth considering for any future work. This specific validator, as well as many other, works on a parsed HTML version of the response. This has the added benefit of being able to leverage all the different built-in methods provided by browsers which are likely well optimized - such as searching for specific elements. However, JavaScript does not get executed in this case. This was to avoid potential attacks on the extension itself - though Chromium allows extensions to run custom JavaScript within a sandbox environment if needed.

However, while testing this module on a custom fresh installation of the popular e-commerce solution Prestashop, I noticed that it obfuscates email addresses using the following method:

```
<script type="text/javascript">eval(unescape('___hex_string___'))</script>
```

where `___hex_string___` gets unescaped to a piece of JavaScript code in the form of:

```
document.write('<a href="mailto:jon@doe.com">jon@doe.com</a>');
```

While the specific use of the function `unescape` is not recommended, as it is a legacy feature and its use in new code is discouraged [31], it does underline the fact that similar types of obfuscations might be used as well. That being said, the current implementation is not able to overcome this specific complication, but it does present interesting ideas to explore in any follow-up work.

5.3.6 GUIDDetector

Similarly to the `EmailAddressDetector`, even this detector is based on the idea of executing a regular expression on every single page of the website. Therefore, the implementation logic is exactly the same just with a different expression, as described in subsection 3.4.1.

Given that GUIDs are well defined and do not have nuances similar to email address detection, this approach can be viewed as rather reliable. There is, however, one important consideration. Let us assume that a website exposes a GUID. If that GUID is well formed, then it will (most likely) get detected by the regular expression following the definition in subsection 3.4.1. The issue becomes apparent once we look in the other direction - if it is an invalid GUID, will it get detected?

If we start with a valid GUID and then append any number of characters at the end of this string, then we would see it matches the regular expression, yet it does not seem to be a valid GUID. Again, there are two solutions to this shortcoming. We could either require that the hexadecimal sequence, which forms the GUID, is followed by a space or a dot for instance, or we accept the possibility that it is an error and the real GUID got concatenated with a different string of text by accident. Note that this issue and its possible resolutions also apply to other pattern-matching tasks.

5.3.7 JSStaticCodeAnalyzer

Up to this point, all of the described modules that analyzed source code performed their analysis on HTML. On the other hand, `JSStaticCodeAnalyzer` focuses solely on any previously detected JavaScript dependencies.

In summary, this module iterates through all the dependencies and for each one it performs the same type of analysis. As of this writing, the analysis only searches for function calls on the `console` object due to time constraints. The rationale for this analysis is that `console` calls can leak information about the website's internals. For instance, they can help uncover all the different states through which the application goes. Currently, all these calls are treated equally, meaning that if such call is detected, it is automatically reported. In any future work, it might be worth considering including a dictionary of trigger words or phrases that might score the individual findings in order to understand their severity. Alternatively, adding natural language processing capabilities might also yield interesting results.

In order to achieve this detection, there were several approaches I considered using. The first technique, following the style of other detectors, was using a regular expression (or multiple). While possible, this approach would be somewhat impractical, because the expression would quickly become complex - especially if more functions of interest were added. An alternative to that was to search for the position of a specific keyword (function call) and then perform parenthesis matching to extract the function's arguments. This approach could be easily extended and should yield reliable results. Another alternative was to use any of already existing tokenization libraries for JavaScript and look for the appropriate tokens. Having tokenized the whole file would allow for other types of analysis as well, such as analyzing comments.

In this case, however, I chose a slightly more sophisticated approach. The tokenization output usually lacks any additional structure - it is a flat array of tokens. Doing analysis on that is possible but it might be easier if those tokens were organized into an abstract syntax tree (or *AST*). That way, extracting a function call with its arguments is as simple as locating a tree node of certain type and then looking at its children - which may be more complex than a simple string or number literal. My current implementation achieves this goal using an open-sourced JavaScript library called *esprima*, developed by Ariya Hidayat.

The reason for using ASTs instead of the other approaches is that it can be further leveraged to do other types of analyses. It would, for instance, be easier to execute small portions of the code to see their output or to guess their output without execution. While adding these additional analyses would not fit the original idea of a single functionality per module, that could be easily fixed by transferring the AST ownership from the module to the dependency objects themselves. This way, other modules could also use the same AST instance.

5.3.8 NetworkAddressDetector

The aim of this validator is to determine if the web application might be leaking network addresses, specifically IPv4 and IPv6 addresses. These might have been collected by the application as part of suspicious sign in activity detection or for only allowing access from certain addresses. In which case, leak of such information could be significant - especially if also connected with other personal identifiers.

This specific implementation is split into two functional blocks - one for IPv4 dection and one for IPv6 detection. In the former case, a very simple regular expression is used to detect any quadruplet of one to three digit long dot-separated numbers. This acts as a very basic filter that still contains many possible values which are not valid IPv4 addresses. This is why, if a compliant string is found, it does additional check to determine if it is within the `0–255` range and has no leading zeros. Conversely, the IPv6 block only uses a single regular expression. As of this writing, it does not consider shortened notations for certain forms of IPv6 addresses as described in RFC5952 [97].

There are similar false positive considerations as with GUID detection. These addresses could have any random string appended at the end and would still get detected, yet they might not be valid IP addresses (or possibly prepended). For instance, a string `"127.0.0.1"` would pass the test but so would `"a127.0.0.1.5"` - note that only the valid substring would get tagged. And whether that should be detected or not is slightly more complicated topic, as it would usually depend on the context.

5.3.9 NVDLookup

One of the key actions to perform after successfully detecting various dependencies, such as libraries, framework, engines, etc. is to verify if there are any known vulnerabilities for the specific versions. As mentioned in the section subsection 2.4.1, NIST maintains a publicly accessible database of vulnerabilities that can be queried through an HTTP-based API. The `NVDLookup` module does exactly that. It transforms the detected

dependencies and their version into CPE strings so that NVD can be easily queried for those dependencies and related vulnerabilities can be retrieved. In case the module receives any results for any of the dependencies, it immediately reports it as a vulnerable component.

5.3.10 PortScanner

One of the ways to determine what applications are running on the server is to perform a port scan. Due to Internet Assigned Number Authority's (or *IANA*'s) efforts, one can easily determine what service is running on given server based on the open port numbers - assuming they did not change the default configuration. Administrators can, of course, temper with these settings but it might provide a relatively good baseline.

The implementation of this module based on the ideas shared by Nikolai Tschacher in his article [93]. The module has a sample of known ports that are mapped to certain applications and, for each port, it tries to establish a WebSocket connection to the tested site on that port. This connection is initiated by injecting a small snippet of JavaScript into the currently viewed web page and executed. The module then listens for the network response. In order to determine the state of the port, it analyzes the error message - if the error message is `net::ERR_CONNECTION_REFUSED`, then the port is closed. The reason for this approach is detailed in subsection 4.7.1.

This approach seems to provide reliable results based on local tests compared to nmap (quick scan run on local machine and a remote server hosting a PHP website). The main downside is that WebSocket connections cannot initiate connections over *unsafe* ports which limits the detection capabilities.

5.3.11 SecureConnectionAnalyzer

Communication over unsecure HTTP instead of HTTPS is viewed negatively and for good reason - pure HTTP traffic can be sniffed and requires no decryption, meaning that if the attacker can intercept the traffic, they can also gather any sensitive data that might be transferred. This idea is further underlined by Google's decision to flag HTTP pages as *not secure* starting in version 68 [98].

The goal of this module is determine if any of the already explored pages can be accessed via HTTP. If the response is not an HTTP redirect to HTTPS and, in fact, content is retrieved for the HTTP request, then the page gets flagged as insecure. The current implementation adopts a *fail fast* strategy in that the first page that fails the test is also the last and the module finishes its work at that point.

5.3.12 SQLiDetector

As discussed previously, (SQL) Injection is among the most common and also severe vulnerabilities. Moreover, there are many different types of SQLi attacks, as well as SQLi detection methods.

The `SQLiDetector` module is currently based on illegal queries. For each previously detected application entry point, it lets the associated `EntryPoint` instance to generate a set of valid input values. It then tampers with each one of those inputs, one at a time, and tries to make the query fail by including special characters that could either initiate or terminate a string or mark the beginning of a comment. If the response fails, it is marked a successful attempt at SQLi on given field. Note that *response failure* in this implementation represents a 500 response code, which is very simplistic check but with default configurations of certain server engines can yield valid results.

This approach could be further extended by looking at the content of the response. For instance, it could look for well known error messages thrown by different database systems. This, however, will require additional data collection work.

5.3.13 XSSDetector

The goal of the `XSSDetector` module, as of this writing, is to determine the possibility of reflected XSS in any of the detected application entry points.

The logic of this validator is somewhat similar to the `SQLiDetector` in that it goes through all entry points, generates valid input, tries to send its malicious variations, and then analyzes the response. Since the idea of reflected XSS is that the payload sent to the server is directly embedded in the subsequent response, the analysis only needs to performed on the immediate response. During that time, it looks for the malicious payload in the raw content to see if the unescaped version is present or not.

This approach can lead to certain false positives that would be worth considering for any future work. The value may not be embedded in a *immediately rendered* part of the document. The malicious payload may propagate into a JavaScript snippet, in which case it could both a valid and invalid finding. The exact scenario would depend on how that value is handled and if it gets later rendered somewhere. That could be determine by executing the code within a sandbox environment or by performing static code analysis of the snippet and trying to understand how it propagates through the document/code.

5.4 User Interface

A key part of the extension is its user interface which is essential to delivering a *1-click* solution that can be used by all developers regardless of their level of knowledge. The whole UI is structured into several screens, each one consisting of only a few components that together provide a frontend to the aforementioned modules and tests.

5.4.1 Dashboard

The first screen with which the user is greeted is the Dashboard (Figure 5.1). Its main purpose is to facilitate initiating a new test run. This is done through a form with 3 parameters - test run name, test type, and target URL.

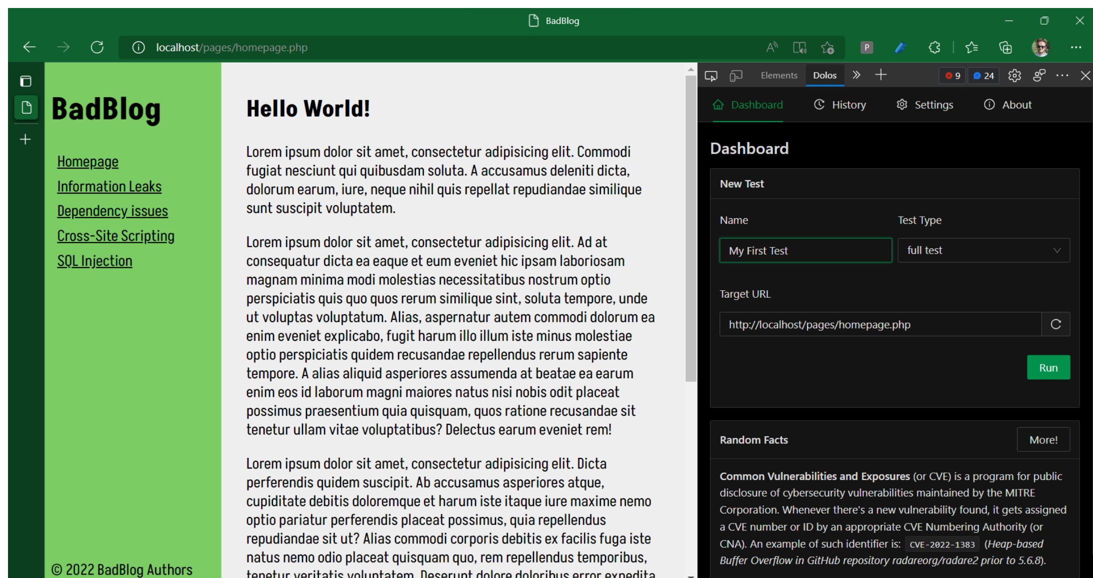

Figure 5.1: Dashboard screen of Dolos extension visible in Developer Tools.

Note that run name is an optional parameter and, if left empty, it is given the value with the following format: `"Test on **Origin**"`, where `"**Origin**"` refers to the target URL's origin. Moreover, test type is by default set to `full test`, which runs every single module that has been implemented. The URL parameter is automatically loaded from the currently inspected tab upon opening the Dolos developer tools tabs. It can be manually refreshed using the reload button to the right of the input field for URL. This is to ensure a test can be run as easily as possible.

Once the user clicks the `Run` button, the component initiates a call to the public `Dolos.runTest` method with the submitted paremeters. At the same time, it initiate a URL change from the dashboard to the history

screen with the new new test immediately opened. That way, the user can track the test's progress while it is being executed. This is as far as the logic of the dashboard goes.

An additional experimental feature inserted into the dashboard is the Random Facts component. It allows the user to view learn new information in the field of cyber security, while providing additional relevant resources as well. Using Next button, users can go through the extension's knowledge base of various facts. As of this writing, this database is rather small and mostly serves as a proof of concept.

5.4.2 History

To view the details of any test or to list all the previously run tests, users can navigate to the History screen. When manually entering this screen, the latter is displayed first (Figure 5.2). To view the details, the user needs to click on a specific item in the table in order to open it. Since it is expected that multiple tests would be run on the same page repeatedly, the UI also offers an option to delete the whole history, utilizing Dolos' public method `Dolos.purgeHistory`.

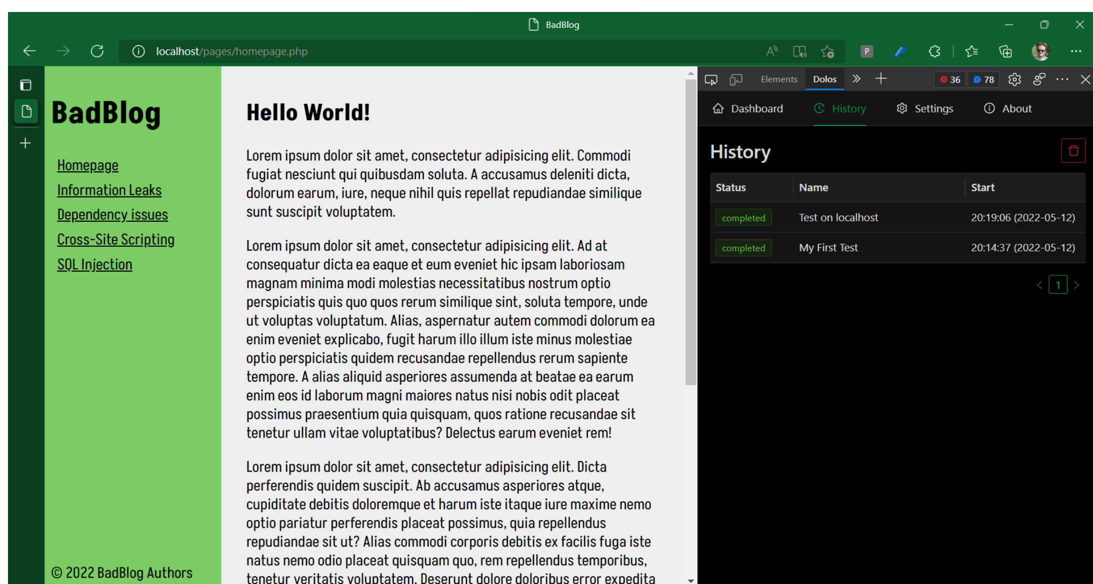

Figure 5.2: History screen of Dolos extension visible in Developer Tools.

Upon opening a specific test run, the user is presented with three main parts or cards of the run. These are: overview, progress or steps, and findings (Figure 5.3). The overview serves as a container for the most basic details, such as start time, tested URL, etc. The steps card is intended to visualize the current status of the test in more detail, i.e. show in which phase the test currently is, what module is being executed, and how much time the completed phases took. At the very end of the screen, all of the currently known findings are listed in the form of a table. Each finding has its title, which serves as a quick categorization of the issue, and a severity level. By default, the table sorts the findings by severity in a descending order, meaning that the most severe issues are at the top.

Each finding can be expanded to see more details (Figure 5.4). These consists of a brief description which answers what, where, and how a module found it, a more detailed text explaining what the issue is and how to remediate it, and lastly there is a set of links pointing the user to additional resources. Moreover, the links have three different categories defined: vulnerabilities (or CVEs), weaknesses (or CWEs), and Other.

5.4.3 Settings

The Settings screen is currently only intended as a placeholder for any future work that might require additional settings of the extension. That being said, the implemented user interface is meant to provide a few ideas as to what might be configurable but has no real underlying functionality.

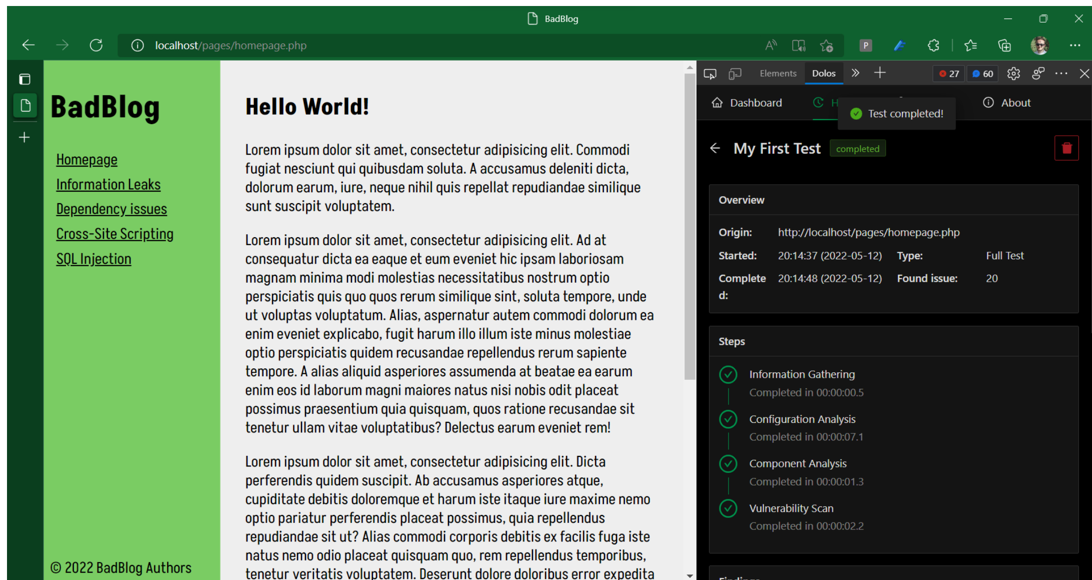

Figure 5.3: Detailed view of a test run within Dolos extension.

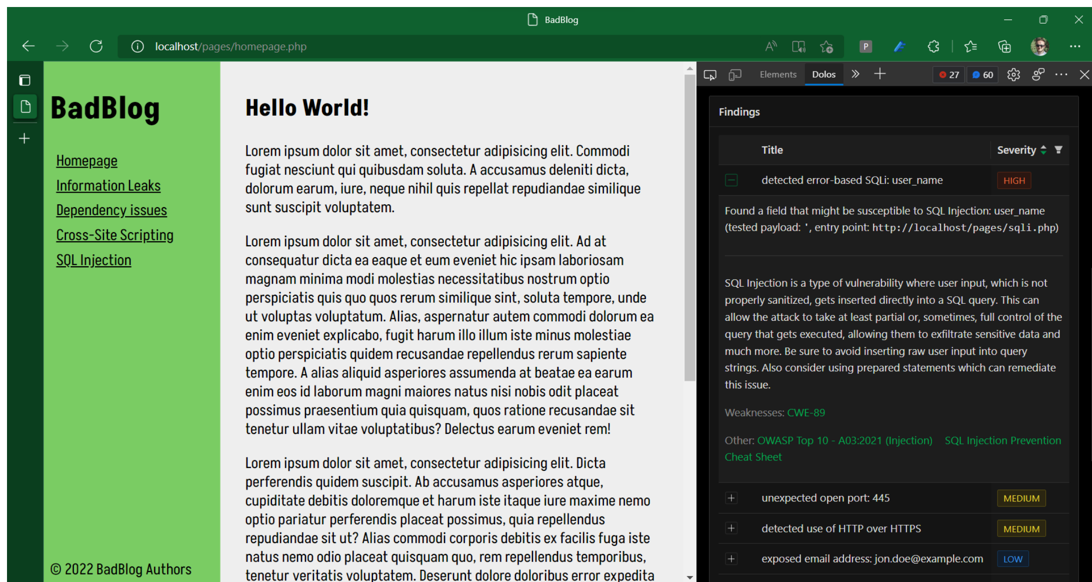

Figure 5.4: Details of a specific finding reported by Dolos.

5.4.4 About

Lastly, the About screen presents the user with a few basic, as well as a few more detailed pieces of information about this project. As such, it is not intended to have any functionality.

6 | Conclusion

The aim of this thesis was to determine the viability of browser extensions for web security testing purposes. The first chapter focused on exploring the existing frameworks, projects, and tools in the field of (web) security testing. A few important organizations were introduced, such as the MITRE Corporation or the OWASP Foundation, for their contribution to the field. Moreover, their projects and products relevant to this topic were discussed in more detail, including their connection to real-world examples. Three common vulnerabilities - SQL Injection, Cross-Site Scripting, and Cross-Site Request Forgery, were explained in detail. Moreover, several existing browser extensions, cloud-based tools, as well as desktop tools for web security testing were described.

The second chapter analyzed various aspects of web security testing process. As part of information gathering, foundations to website crawling problem were described. Moreover, various freely available APIs, which might be useful data sources for security testing, were reviewed. Next, software detection in general was discussed - various already used approaches for detecting both frontend and backend components, as well as a few proposed alternatives which exploit the nature of specific technologies. Furthermore, various types of sensitive information were discussed in terms of what are their associated standards and how those can be leveraged to simplify such data type's detection. These included email addresses, network addresses, and phone numbers among other. This information was then contrasted with the approaches to sensitive information detection implemented by the tools reviewed in the first chapter and certain research projects. Lastly, the three vulnerabilities - SQLi, XSS, and CSRF, were described from the perspective of their detection, reviewing how this problem is being approached by existing tools and what are some of the methods proposed by researchers.

The third chapter focused on Chromium extensions. Their basic structure and internal logic was explored, while explaining how to build such extensions. Given the mostly HTTP-based nature of existing tools, networking aspects of the browser extension environment were discussed in detail. This highlighted the inherent limitations of this platform since extensions, similarly to regular web applications, operate mostly over a limited number application-layer protocols with respect to the OSI model. A few approaches were discussed to overcome these shortcomings with respect to port scanning, as well as their relevancy in the current versions of Chromium. An alternatively solution to port scanning from within the browser extension environment was proposed based on these approaches using WebSockets. Apart from networking, no other limitations were determined when compared to the existing tools.

The fifth, and the last, chapter was dedicated to proposing and describing a concrete solution - a proof of concept that can perform automated web security testing from within the browser extension context. Each implemented functionality was described in more detail and potential future improvements were discussed, demonstrating the limitations and strengths of the solution.

References

[1] CERN. *A short history of the Web*. [online]. URL: https://home.cern/science/computing/birth-web/short-history-web (last accessed on 2022-03-23).

[2] United Nations. *As Internet user numbers swell due to pandemic, UN Forum discusses measures to improve safety of cyberspace*. [online]. 2021-12-07. URL: https://www.un.org/sustainabledevelopment/blog/2021/12/as-internet-user-numbers-swell-due-to-pandemic-un-forum-discusses-measures-to-improve-safety-of-cyberspace/ (last accessed on 2022-03-23).

[3] *2021 Data Breach Investigations Report*. 2021. URL: https://www.verizon.com/business/resources/reports/2021/2021-data-breach-investigations-report.pdf (last accessed on 2022-04-01).

[4] ISACA. *State of Cybersecurity 2022: Global Update on Workforce Efforts, Resources and Cyberoperations*. isaca.org. 2022. URL: https://www.isaca.org/go/state-of-cybersecurity-2022 (last accessed on 2022-04-25).

[5] Elaine Barker, Miles Smid, and Dennis Branstad. *A Profile for U. S. Federal Cryptographic Key Management Systems*. 2015-10-28. DOI: 10.6028/NIST.SP.800-152. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-152.pdf (last accessed on 2022-03-15).

[6] The OWASP Foundation Inc. *About the OWASP Foundation*. [online]. URL: https://owasp.org/about/ (last accessed on 2022-03-23).

[7] Andrew van der Stock et al. *OWASP Top 10:2021*. [online]. 2021-09. URL: https://owasp.org/Top10/ (last accessed on 2022-01-16).

[8] Brandon Forbes. *Hacking DEF CON 29*. [online]. 2021-07-19. URL: https://reznok.com/hacking-def-con-29/ (last accessed on 2022-03-23).

[9] Chris Williams. *Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug*. [online]. 2014-04-09. URL: https://www.theregister.com/2014/04/09/heartbleed_explained/ (last accessed on 2022-03-23).

[10] Peter Yaworski. *Real-world bug hunting: a field guide to web hacking*. William Pollock, 2019-07. ISBN: 978-1-59327-861-8.

[11] Pedro Canahuati. *Keeping Passwords Secure*. [online]. 2019-03-21. URL: https://about.fb.com/news/2019/03/keeping-passwords-secure/ (last accessed on 2022-03-23).

[12] Mehedi Hasan Remon. *F5 BIG-IP TMUI RCE - CVE-2020-5902 (**.packet8.net)*. hackerone.com. 2022-03-23. URL: https://hackerone.com/reports/1519841 (last accessed on 2022-03-25).

[13] The MITRE Corporation. *Common Vulnerabilities and Exposures*. [online]. URL: https://cve.mitre.org/index.html (last accessed on 2022-03-12).

[14] The MITRE Corporation. *Common Attack Pattern Enumerations and Classifications*. [online]. URL: https://capec.mitre.org/ (last accessed on 2022-03-26).

[15] Lindsey O'Donnell. *Hackers Breach Dunkin' Donuts Accounts in Credential Stuffing Attack*. [online]. 2018-11-29. URL: https://threatpost.com/hackers-breach-dunkin-donuts-accounts-in-credential-stuffing-attack/139472/ (last accessed on 2022-03-26).

[16] Tara Seals. *The SolarWinds Perfect Storm: Default Password, Access Sales and More*. [online]. 2020-12-16. URL: https://threatpost.com/solarwinds-default-password-access-sales/162327/ (last accessed on 2022-03-26).

[17] Information Commissioner's Office. *British Airways Penalty Notice*. [online]. 2020-10-16. URL: https://ico.org.uk/media/action-weve-taken/mpns/2618421/ba-penalty-20201016.pdf (last accessed on 2022-03-26).

[18] The MITRE Corporation. *Common Weakness Enumeration*. [online]. URL: https://cwe.mitre.org/index.html (last accessed on 2022-03-12).

[19] Open Web Application Security Project. *WSTG - latest*. [online]. URL: https://owasp.org/www-project-web-security-testing-guide/latest/ (last accessed on 2022-01-15).

[20] The MITRE Corporation. *Common Platform Enumeration*. [online]. 2014-11-28. URL: https://cpe.mitre.org/ (last accessed on 2022-03-23).

[21] The MITRE Corporation. *CVE-2014-0160*. [online]. 2013-12-03. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 (last accessed on 2022-03-24).

[22] Blake E Strom et al. Tech. rep. [online]. 2018-07. URL: https://attack.mitre.org/docs/ATTACK_Design_and_Philosophy_March_2020.pdf (last accessed on 2022-03-16).

[23] The MITRE Corporation. *Enterprise Techniques*. MITRE. 2021 [Online]. URL: https://attack.mitre.org/versions/v10/techniques/enterprise/ (last accessed on 2022-03-16).

[24] The MITRE Corporation. *Active Scanning: Vulnerability Scanning*. [online]. 2020-10-02. URL: https://attack.mitre.org/versions/v10/techniques/T1595/002/ (last accessed on 2022-03-24).

[25] The MITRE Corporation. *Enterprise Tactics*. MITRE. 2021 [Online]. URL: https://attack.mitre.org/versions/v10/tactics/enterprise/ (last accessed on 2022-03-16).

[26] National Institute of Standards and Technology. *National Vulnerability Database*. [online]. URL: https://nvd.nist.gov/ (last accessed on 2022-03-19).

[27] National Institute of Standards and Technology. *API Vulnerabilities*. [online]. URL: https://nvd.nist.gov/developers/vulnerabilities (last accessed on 2022-03-24).

[28] Brant A. Cheikes, David Waltermire, and Karen Scarfone. *Common Platform Enumeration: Naming Specification Version 2.3*. Tech. rep. National Institute of Standards and Technology Interagency Report 7695 (Aug. 2011). Washington, D.C.: U.S. Department of Commerce, 2011. DOI: 10.6028/nist.ir.7695. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7695.pdf (last accessed on 2022-03-19).

[29] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. "A classification of SQL-injection attacks and countermeasures". In: *Proceedings of the IEEE international symposium on secure software engineering*. Vol. 1. IEEE. 2006, pp. 13–15.

[30] Samy Kamkar. *The MySpace Worm*. 2010-05-04. URL: https://samy.pl/myspace/ (last accessed on 2022-04-02).

[31] Mozilla Corporation and contributors. *MDN Web Docs*. [online]. URL: https://developer.mozilla.org/en-US/ (last accessed on 2022-03-13).

[32] Chris Sullo. *Nikto source code*. [online]. 2012. URL: https://github.com/sullo/nikto (last accessed on 2022-03-26).

[33] Gordon Lyon et al. *Chapter 15. Nmap Reference Guide*. [online]. URL: https://nmap.org/book/man.html (last accessed on 2022-03-26).

[34] Microsoft Corporation. *tracert*. [online]. 2021-07-29. URL: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/tracert (last accessed on 2022-03-26).

[35] Bernardo Damele A. G. and Miroslav Stampar. *sqlmap documentation*. [online]. URL: https://github.com/sqlmapproject/sqlmap/wiki (last accessed on 2022-03-26).

[36] WPScan Team. *WPScan source code*. [online]. URL: https://github.com/wpscanteam/wpscan (last accessed on 2022-03-26).

[37] the ZAP Dev Team. *OWASP ZAP - Documentation*. [online]. 2022. URL: https://www.zaproxy.org/docs/ (last accessed on 2022-03-09).

[38] PortSwigger Ltd. *BURP suite documentation: Desktop editions*. [online]. 2022. URL: https://portswigger.net/burp/documentation/desktop (last accessed on 2022-03-09).

[39] Andres Riancho. *w3af - Web application attack and audit framework 1.6.54 documentation*. [online]. 2014. URL: http://docs.w3af.org/en/stable/ (last accessed on 2022-05-02).

[40] Inc. Tenable. *Nessus Professional*. [online]. URL: https://www.tenable.com/products/nessus/nessus-professional (last accessed on 2022-03-31).

[41] Snyk Limited. *Snyk | Developer security*. [online]. URL: https://snyk.io/ (last accessed on 2022-03-31).

[42] Detectify. *Attack Surface Management Tool Powered by Ethical Hackers*. [online]. URL: https://detectify.com/ (last accessed on 2022-03-31).

[43] Wappalyzer. *wappalyzer/wappalyzer: Indetify technology on websites*. [online]. URL: https://github.com/AliasIO/wappalyzer (last accessed on 2022-03-31).

[44] RapidSec. *CSP Scanner: Test, Analyze Evaluate CSP*. [online]. URL: https://chrome.google.com/webstore/detail/csp-scanner-test-analyze/eoiiiomeoogcpnkdedcodoeaacpdfmdj (last accessed on 2022-03-31).

[45] Checkbot. *Checkbot: SEO, web speed security checker for Chrome*. [online]. URL: https://www.checkbot.io/ (last accessed on 2022-03-31).

[46] webdevmedia. *Online investigation tool - Reverse IP, NS, MX, WHOIS and Search Tools*. [online]. URL: https://dnslytics.com/ (last accessed on 2022-03-31).

[47] Gojko Adzic. *gojko/bugmagnet: Bug Magnet Chrome/Firefox Extension*. [online]. URL: https://github.com/gojko/bugmagnet (last accessed on 2022-03-31).

[48] Hacker Target Pty Ltd. *IP, DNS Security Tools | HackerTarget.com*. [online]. URL: https://chrome.google.com/webstore/detail/ip-dns-security-tools-hac/phjkepckmcnjohilmbjlcoblenhgpjmo?hl=cs (last accessed on 2022-03-31).

[49] Ludovic Coulon and Riadh Bouchahoua. *LasCC/Hack-Tools: The all-in-one Red Team extension for Web Pentester*. [online]. URL: https://github.com/LasCC/Hack-Tools (last accessed on 2022-03-31).

[50] Denis Podgurskii. *DenisPodgurskii/pentestkit*. [online]. URL: https://github.com/DenisPodgurskii/pentestkit (last accessed on 2022-03-31).

[51] Google LLC. *DNS-over-HTTPS (DoH)*. developers.google.com. 2020-07-22. URL: https://developers.google.com/speed/public-dns/docs/doh (last accessed on 2022-04-27).

[52] Google LLC. *Safe Browsing Lookup API (v4)*. developers.google.com. 2016-07-28. URL: https://developers.google.com/safe-browsing/v4/lookup-api (last accessed on 2022-04-27).

[53] Inc Sublime Security. *Simple Email Reputation*. URL: https://emailrep.io/ (last accessed on 2022-04-27).

[54] April King. *Mozilla HTTP Observatory*. URL: https://github.com/mozilla/http-observatory (last accessed on 2022-04-27).

[55] Julien Vehent, Dimitris Bachtis, and Adrian Utrilla. *Mozilla HTTP Observatory*. URL: https://github.com/mozilla/tls-observatory (last accessed on 2022-04-27).

[56] Fabian Marquardt and Lennart Buhl. "Déjà Vu? Client-Side Fingerprinting and Version Detection of Web Application Software". In: *2021 IEEE 46th Conference on Local Computer Networks (LCN)*. IEEE. 2021, pp. 81–89.

[57] Jingling Zhao et al. "An AST-based Code Plagiarism Detection Algorithm". In: 2015-11, pp. 178–182. DOI: 10.1109/BWCCA.2015.52.

[58] *REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL*. European Commission. 2016-04-27. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679&from=EN (last accessed on 2022-04-01).

[59] Pete Resnick. *Internet Message Format*. RFC 5322. 2008-10. DOI: 10.17487/RFC5322. URL: https://www.rfc-editor.org/info/rfc5322 (last accessed on 2022-03-31).

[60] Jon Postel. *Internet Protocol*. RFC 791. 1981-09. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/info/rfc791.

[61] Bob Hinden and Dr. Steve E. Deering. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. 1998-12. DOI: 10.17487/RFC2460. URL: https://www.rfc-editor.org/info/rfc2460 (last accessed on 2022-04-01).

[62] Dr. Steve E. Deering and Bob Hinden. *IP Version 6 Addressing Architecture*. RFC 4291. 2006-02. DOI: 10.17487/RFC4291. URL: https://www.rfc-editor.org/info/rfc4291.

[63] Social Security Administration. *Social Security Number Randomization*. ssa.gov. [online]. URL: https://www.ssa.gov/employer/randomization.html (last accessed on 2022-04-30).

[64] International Telecommunication Union. *Notation for national and international telephone numbers, e-mail addresses and Web addresses*. [online]. 2001-02-02. URL: https://www.itu.int/rec/T-REC-E.123-200102-I/en (last accessed on 2022-04-30).

[65] Brendan Harkness. *Anatomy of a Credit Card*. [online]. 2022-03-24. URL: https://moneytips.com/anatomy-of-a-credit-card/ (last accessed on 2022-04-30).

[66] Paul J. Leach, Rich Salz, and Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. 2005-07. DOI: 10.17487/RFC4122. URL: https://www.rfc-editor.org/info/rfc4122.

[67] the ZAP Dev Team. *zaproxy/zap-extensions: OWASP ZAP Add-ons*. 2022. URL: https://github.com/zaproxy/zap-extensions (last accessed on 2022-05-01).

[68] Mariam Sulakian. *Proactively prevent secret leaks with GitHub Advanced Security secret scanning*. GitHub Blog. 2022-04-04. URL: https://github.blog/2022-04-04-push-protection-github-advanced-security/ (last accessed on 2022-04-12).

[69] David Sánchez, Montserrat Batet, and Alexandre Viejo. "Detecting sensitive information from textual documents: an information-theoretic approach". In: *International Conference on Modeling Decisions for Artificial Intelligence*. Springer. 2012, pp. 173–184.

[70] Mu Qiao et al. *Context aware sensitive information detection*. US Patent 10,984,316. 2021-4 20.

[71] V. Vijayalakshmi et al. "Survey on detecting leakage of sensitive data". In: *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*. 2016, pp. 1–3. DOI: 10.1109/STARTUP.2016.7583916.

[72] Michael Ogata et al. *Vetting the Security of Mobile Applications*. [online]. 2019-04. DOI: 10.6028/NIST.SP.800-163r1. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-163r1.pdf (last accessed on 2022-05-01).

[73] Isatou Hydara et al. "Current state of research on cross-site scripting (XSS) – A systematic literature review". In: *Information and Software Technology* 58 (2015), pp. 170–186. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2014.07.010. URL: https://www.sciencedirect.com/science/article/pii/S0950584914001700.

[74] Fabien Duchene et al. "KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection". In: *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. CODASPY '14. San Antonio, Texas, USA: Association for Computing Machinery, 2014, pp. 37–48. ISBN: 9781450322782. DOI: 10.1145/2557547.2557550. URL: https://doi.org/10.1145/2557547.2557550.

[75] William Melicher et al. "Towards a Lightweight, Hybrid Approach for Detecting DOM XSS Vulnerabilities with Machine Learning". In: *Proceedings of the Web Conference 2021*. WWW '21. Ljubljana, Slovenia: Association for Computing Machinery, 2021, pp. 2684–2695. ISBN: 9781450383127. DOI: 10.1145/3442381.3450062. URL: https://doi.org/10.1145/3442381.3450062.

[76] Stefano Calzavara et al. "Machine Learning for Web Vulnerability Detection: The Case of Cross-Site Request Forgery". In: *IEEE Security Privacy* 18.3 (2020), pp. 8–16. DOI: 10.1109/MSEC.2019.2961649.

[77] Ben Goodger. *Welcome to Chromium*. Chromium Blog. 2008-09-02. URL: https://blog.chromium.org/2008/09/welcome-to-chromium_02.html (last accessed on 2022-04-28).

[78] StatCounter. *Browser Market Share Worldwide (Q1 2009 - Q2 2022)*. 2022-04-28. URL: https://gs.statcounter.com/browser-market-share#quarterly-200901-202202 (last accessed on 2022-04-28).

[79] Google LLC. *Extensions - Chrome Developers*. Last accessed 2022-01-30. 2022. URL: https://developer.chrome.com/docs/extensions/.

[80] DebugBear. *Counting Chrome Extensions – Chrome Web Store Statistics*. 2020-06-29. URL: https://www.debugbear.com/blog/counting-chrome-extensions (last accessed on 2022-04-28).

[81] Microsoft and community contributors. *Microsoft Edge - Policies*. 2022-04-28. URL: https://docs.microsoft.com/en-us/deployedge/microsoft-edge-policies (last accessed on 2022-04-28).

[82] The Chromium Authors. *Chromium v102.0.4960.1 source code*. [online]. 2022-03. URL: https://source.chromium.org/chromium/chromium/src/+/refs/tags/102.0.4960.1: (last accessed on 2022-03-23).

[83] Bennett Blodinger. *Issue 571722: Unable to set User-Agent header*. Chromium Bugs. 2015-12-23. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=571722 (last accessed on 2022-04-28).

[84] Alexey Melnikov and Ian Fette. *The WebSocket Protocol*. RFC 6455. 2011-12. DOI: 10.17487/RFC6455. URL: https://www.rfc-editor.org/info/rfc6455.

[85] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. RFC 8445. 2018-07. DOI: 10.17487/RFC8445. URL: https://www.rfc-editor.org/info/rfc8445.

[86] Kjeld Borch Egevang and Paul Francis. *The IP Network Address Translator (NAT)*. RFC 1631. 1994-05. DOI: 10.17487/RFC1631. URL: https://www.rfc-editor.org/info/rfc1631.

[87] Marc Petit-Huguenin et al. *Session Traversal Utilities for NAT (STUN)*. RFC 8489. 2020-02. DOI: 10.17487/RFC8489. URL: https://www.rfc-editor.org/info/rfc8489.

[88] Tirumaleswar Reddy.K et al. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 8656. 2020-02. DOI: 10.17487/RFC8656. URL: https://www.rfc-editor.org/info/rfc8656.

[89] Christer Holmberg, Harald T. Alvestrand, and Cullen Jennings. *Negotiating Media Multiplexing Using the Session Description Protocol (SDP)*. RFC 9143. 2022-02. DOI: 10.17487/RFC9143. URL: https://www.rfc-editor.org/info/rfc9143.

[90] Jacob Baines. *Using WebRTC ICE Servers for Port Scanning in Chrome*. Medium. 2019-12-20. URL: https://medium.com/tenable-techblog/using-webrtc-ice-servers-for-port-scanning-in-chrome-ce17b19dd474 (last accessed on 2022-04-20).

[91] W3C. *WebRTC 1.0: Real-Time Communication Between Browsers*. 2021-01-26. URL: https://www.w3.org/TR/webrtc/ (last accessed on 2022-04-20).

[92] Tom Gallagher. *Security Enhancement for WebSockets to Prevent Private Network Mapping*. Internet-Draft draft-gallagher-hybiwebsocketenhancement-00. Work in Progress. Internet Engineering Task Force, 2016-07. 4 pp. URL: https://datatracker.ietf.org/doc/html/draft-gallagher-hybiwebsocketenhancement-00.

[93] Nikolai Tschacher. *Browser based Port Scanning with JavaScript*. incolumitas.com. 2021-01-10. URL: https://incolumitas.com/2021/01/10/browser-based-port-scanning/ (last accessed on 2022-04-20).

[94] Microsoft and community contributors. *The TypeScript Handbook*. [online]. 2022. URL: https://www.typescriptlang.org/docs/handbook/intro.html (last accessed on 2022-05-02).

[95] Inc. Meta Platforms. *React Documentation*. [online]. 2022. URL: https://reactjs.org/docs/getting-started.html (last accessed on 2022-05-02).

[96] Ant UED and community contributors. *Ant Design*. [online]. 2022. URL: https://ant.design/ (last accessed on 2022-05-02).

[97] Seiichi Kawamura and Masanobu Kawashima. *A Recommendation for IPv6 Address Text Representation*. RFC 5952. 2010-08. DOI: 10.17487/RFC5952. URL: https://www.rfc-editor.org/info/rfc5952 (last accessed on 2022-04-01).

[98] Angela Moscaritolo and Michael Kan. *Google Chrome Begins Flagging All HTTP Pages as 'Not Secure'*. PCMag. [online]. 2018-07-24. URL: https://www.pcmag.com/news/google-chrome-begins-flagging-all-http-pages-as-not-secure (last accessed on 2022-05-12).

[99] M. DiPierro. "The Rise of JavaScript". In: *Computing in Science Engineering* 20.01 (2018-01), pp. 9–10. ISSN: 1558-366X. DOI: 10.1109/MCSE.2018.011111120.

[100] Anil K Jain, Jianjiang Feng, and Karthik Nandakumar. "Fingerprint matching". In: *Computer* 43.2 (2010), pp. 36–44.

[101] Bin Wang et al. "Research on Web Application Security Vulnerability Scanning Technology". In: *2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. Vol. 1. 2019, pp. 1524–1528. DOI: 10.1109/IAEAC47372.2019.8997964.

[102] Internet Engineering Task Force (IETF). *RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1)*. Last accessed 2022-01-30. 2014. URL: https://httpwg.org/specs/rfc7231.html.

[103] Google LLC. *Chromium - Design Documents*. [online]. 2022. URL: https://www.chromium.org/developers/design-documents/ (last accessed on 2022-01-30).

[104] Jianglang Feng, Baojiang Cui, and Kunfeng Xia. "A Code Comparison Algorithm Based on AST for Plagiarism Detection". In: *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*. 2013, pp. 393–397. DOI: 10.1109/EIDWT.2013.73.

[105] The MITRE Corporation. *Enterprise Matrices*. MITRE. 2021 [Online]. URL: https://attack.mitre.org/versions/v10/matrices/enterprise/ (last accessed on 2022-03-16).

[106] JS Foundation and community contributors. *Webpack Documentation*. [online]. 2022. URL: https://webpack.js.org/concepts/ (last accessed on 2022-05-02).