SecurityAuditor: An XDriver Security Oriented Module for the Evaluation of Security Header Policies

Alexandros Savvopoulos

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor:
Assist. Prof. *Polyvios Pratikakis*
Assoc. Prof. *Sotiris Ioannidis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

This work has received funding from the European Union Horizon's 2020 research and innovation programme H2020-DS-SC7-2017, under grant agreement No. 786890 (THREAT-ARREST)

University of Crete
Computer Science Department

SecurityAuditor: An XDriver Security Oriented Module for the Evaluation of Security Header Policies

Thesis submitted by
Alexandros Savvopoulos
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science
Professor THESIS APPROVAL

Author: 

Alexandros Savvopoulos

Committee approvals:

Polyvios Pratikakis
Assistant Professor, Thesis Supervisor, Committee Member

Sotiris Ioannidis
Associate Professor, Thesis Advisor, Committee Member

Yannis Tzitzikas
Associate Professor, Committee Member

Departmental approval:

Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, March 2021

SecurityAuditor: An XDriver Security Oriented Module for the Evaluation of Security Header Policies

Abstract

Website security is an important factor for a properly functional site. The developers can set Security Header policies in order to prevent various attacks that can be fatal to the functionality of the websites. However, there are many policies' misconfigurations which can be exploited by the attackers. These attacks can even lead to the users' private data leaking.

The Selenium is a browser automation framework. It emulates a user website task as it can control the web browsers through WebDrivers. The user's functionalities can be executed by this framework in order to gather information about the websites' functionalities. However, there are many problems which can be created by this framework during the execution of users' tasks. These problems may lead to a task's failure. For this reason there is another browser automation toolset named XDriver framework. It contains mechanisms, which offer solutions for task's failures in order to finish it successfully. It also offers Selenium functionalities to reduce the code complexity as it was built on the top of the Selenium framework.

In this master thesis the SecurityAuditor module was developed. This is an XDriver module that used XDriver functionalities in order to evaluate the Security Header Policies. These policies could be implemented by the websites' developers. It also detected policies' misconfigurations which could reduce the security of the website. Using this module, a large-scale study was conducted in order to evaluate it as well as to investigate if these policies were implemented correctly by the websites' developers. It was observed that most of the domains did not implement these policies and some of the policies were detected with syntax errors or known vulnerabilities (misconfigurations). Therefore, the websites' safety could be reduced.

The comparison of the XDriver with the Selenium framework was another study which was conducted in this thesis. The XDriver error handling mechanisms were evaluated, executing browser users' tasks in a number of domains for both of these frameworks. It was concluded that the XDriver solved many Selenium exceptions.

SecurityAuditor: Ένα XDriver Εργαλείο Προσανατολισμένο στην Ασφάλεια για την Αξιολόγηση των Security Header Policies

Περίληψη

Η ασφάλεια των ιστοσελίδων είναι ένας σημαντικός παράγοντας για μια σωστή και λειτουργική ιστοσελίδα. Οι προγραμματιστές των ιστοσελίδων μπορούν να ορίσουν πολιτικές ασφάλειας σε αυτές με σκοπό να αποτρέψουν διάφορες επιθέσεις που μπορούν να αποβούν μοιραίες για τη λειτουργικότητά τους. Ωστόσο, υπάρχουν ευάλωτα σημεία στο συντακτικό αυτών των πολιτικών, τα οποία μπορούν να εκμεταλλευτούν οι επιτιθέμενοι με σκοπό ακόμη και τη διαρροή ιδιωτικών δεδομένων των χρηστών.

Το Selenium είναι ένα πρόγραμμα αυτόματης περιήγησης ιστοσελίδων. Μπορεί να μιμηθεί ενέργειες χρηστών καθώς μπορεί να διαχειριστεί ένα πρόγραμμα περιήγησης μέσω των WebDrivers. Οι λειτουργίες των χρηστών μπορούν να εκτελεστούν από αυτά τα συστήματα, προκειμένου να συγκεντρωθούν πληροφορίες σχετικά με τις λειτουργίες των ιστοσελίδων. Ωστόσο, υπάρχουν πολλά προβλήματα που μπορούν να δημιουργηθούν κατά την εκτέλεση αυτών των ενεργειών και να οδηγήσουν σε αποτυχία της εργασίας. Για αυτόν τον λόγο, υπάρχει το XDriver που είναι ένα άλλο σύστημα αυτοματισμού προγράμματος περιήγησης. Περιέχει μηχανισμούς που προσφέρουν λύσεις, όταν μια εργασία αποτυγχάνει με σκοπό την επιτυχή ολοκλήρωσής τους. Προσφέρει, επίσης, λειτουργίες του συστήματος Selenium που μειώνουν την πολυπλοκότητα του κώδικα, καθώς αυτές εκτελούν τις ενέργειες του Selenium συστήματος.

Σε αυτή την εργασία υλοποιήθηκε το εργαλείο SecurityAuditor. Αυτό το εργαλείο είναι μέρος του XDriver συστήματος και χρησιμοποίησε λειτουργίες του, με σκοπό την αξιολόγηση των πολιτικών ασφαλείας. Αυτές οι πολιτικές μπορούν να εφαρμοστούν από τους προγραμματιστές των ιστοσελίδων. Εντόπιζε, επίσης, ευάλωτα σημεία στο συντακτικό των πολιτικών που μπορούσαν να μειώσουν την ασφάλεια του ιστότοπου. Χρησιμοποιώντας αυτό το εργαλείο, πραγματοποιήθηκε μια μελέτη μεγάλης κλίμακας για την αξιολόγησή του. Επίσης, μέσα από αυτή την μελέτη ερευνήθηκε, εάν αυτές οι πολιτικές εφαρμόζονταν σωστά από τους προγραμματιστές των ιστοσελίδων. Παρατηρήθηκε ότι οι περισσότεροι ιστότοποι δεν είχαν υλοποιήσει τις πολιτικές και κάποιοι εντοπίστηκαν με συντακτικά λάθη ή ευάλωτα σημεία στο συντακτικό τους με αποτέλεσμα την μείωση της ασφάλειάς τους.

Η σύγκριση των συστημάτων XDriver και Selenium ήταν μια επιπλέον μελέτη που έγινε σε αυτή την εργασία. Διεξήχθη για την αξιολόγηση των μηχανισμών χειρισμού σφαλμάτων που περιέχει το σύστημα XDriver . Αυτή η αξιολόγηση πραγματοποιήθηκε με την ταυτόχρονη εκτέλεση εργασιών σε έναν σύνολο από ιστοσελίδες και στα δύο προηγούμενα συστήματα. Συμπερασματικά, το σύστημα XDriver έδωσε λύσεις σε πολλά προβλήματα του Selenium συστήματος.

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον επόπτη μου Πολύβιο Πρατικάκη για την συνεργασία που είχαμε κατά την διάρκεια του μεταπτυχιακού μου. Επίσης θέλω να εκφράσω την βαθύτατη ευγνωμοσύνη μου στο συνεπιβλέποντα μου Σωτήρη Ιωαννίδη για την ευκαιρία που μου έδωσε να προχωρήσω στις μεταπτυχιακές σπουδές και να αποκτήσω περισσότερες γνώσεις και εφόδια για την μετέπειτα πορεία μου στον χώρο εργασίας. Η μεταπτυχιακή μου εργασία έλαβε χρηματοδότηση από το ερευνητικό και καινοτόμο πρόγραμμα της Ευρωπαϊκής Ένωσης Horizon's 2020 (H2020-DS-SC7-2017), στο πλαίσιο συμφωνίας υποτροφίας με αριθμό 786890 (THREAT-ARREST). Επίσης, θα ήθελα να ευχαριστήσω και τον Ιωάννη Τζίτζικα για την άριστη συνεργασία που είχαμε κατά την διάρκεια του επικουρικού μου έργου.

Ευχαριστώ πολύ όλους στο εργαστήριο που με βοήθησαν αυτά τα χρόνια μέσα από διάφορες συζητήσεις και παρουσιάσεις εργασιών, που έγιναν. Επίσης θα ήθελα να ευχαριστήσω τον Σεραφείμ που με βοήθησε και με στήριξε σε όλα τα ακαδημαϊκά χρόνια καθώς και τον Χριστόφορο για όλη αυτή την βοήθεια που μου έδωσε σε οποιαδήποτε δυσκολία και αν υπήρξε. Δεν θα γινόταν να μην ευχαριστήσω τον Γιαννάκο, Κρούς, Ιέρο, Μήτσο, Αλέξανδρο, Κόντο, Μιχάλη. Είμαι τυχερός και ευγνώμων που απόκτησα αυτούς τους φίλους και περάσαμε μαζί αξέχαστες στιγμές και βραδιές όλα αυτά τα χρόνια.

Δεν θα μπορούσε να επιτευχθεί η επιτυχής ολοκλήρωση των σπουδών μου, χωρίς την υποστήριξη της οικογένειας μου. Ευχαριστώ πάρα πολύ τους γονείς μου, Λουκία και Δημήτρη, που με στήριξαν και με στηρίζουν πάντα. Φυσικά ευχαριστώ και την αδερφή μου Ιωάννα, που ήταν πάντα εκεί σε οποιαδήποτε απορία και δυσκολία μου, με πάρα πολύ υπομονή με σκοπό να με βοηθήσει.

στην οικογένειά μου

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Browser automation is the act of running various tasks in order to gather insights into the functionality and performance of site pages or even emulate users' tasks. If a user wants to do a significant number of tasks throughout the work day, these tasks can be executed by the browser automation tools. These tools help to reduce the time spent on these tasks.

Nowadays, there are some browser automation tools which help the users to do specific tasks automatically such as transferring data between servers. These tools can help the developers by testing the functionality of their applications and websites. The researchers use these tools conducting large-scale analysis of important and various kinds of studies such as advertisements.

The Selenium Project [30] is a powerful tool for browser automation that uses techniques to remotely control browser instances. It is also useful for automated testing in order to emulate users' interactions with the browser. At the core of Selenium is WebDriver, which is an interface to write instruction sets that can be run interchangeably in many browsers like Chrome/Chromium, Firefox, Opera, Safari. Furthermore, the Selenium supports many programming languages such as Python, Java, C#, Ruby. It was designed primarily for automated testing and scenarios that used web applications with known structure and behaviour. However, the researchers use this tool conducting large-scale analysis. In this analysis, the browser automation tool can execute actions and tasks to many web applications which do not have the same structure and behaviour. It can lead to the appearance of unexpected alerts and the interaction with elements which can be invisible or hidden for the users. These cases may cause failures to the analysis. Whenever a problem occurs and causes a task's failure, the researchers have to spend more time finding solutions for them in order to continue the analysis.

The most common problem of Selenium is the stale elements as they can appear in various cases during the execution of an analysis. As an example, when the user visits a website and processes certain web elements (e.g clicks certain links, fills out forms etc) checks for redirections or collects information about the landing page of the website. After each action there is probably a redirection or async page loads.

In that case the website will be reloaded as well as the browser will render again all the site page. As a result, all collected objects will become stale even if they are still present at the page from a user's perspective. This can happen because the objects may have other attribute values such as xpath, css, id values after the page reloading. Therefore, the user can not interact with them.

In addition, there are situations in which the browser and the whole process have to be restarted in order to obtain a clear browser profile. This happens in the case of many webdriver or even timeout exceptions in the sequence of failures, which lead the browser to an unexpected state. Therefore, all the collected elements of the specific website will become stale due to the browser reboot process. As a result, the problem of stale elements is also showing up.

These examples highlight the need of inserting custom code in all scripts in order to catch any error anywhere it may occur, which is not necessarily feasible. Uncaught errors might lead to fatal crashes and delay of the analysis' completion. These solutions will increase the code complexity which is not efficient. Ideally, our analysis/tasks should be performed in a website agnostic way so as to handle all these limitations.

There are also existing tools such as OpenWPM [50] and Puppeteer [22]. The first one is a well-designed framework based on the Selenium that focuses on the browser setup, management, synchronised parts of automation and dynamic interactions. It contains all the Selenium limitations as it does not have any error handling mechanisms. The second one offers specific interaction functionalities (e.g generate screenshots of pages, testing chrome extensions). Therefore, it can not be used for various kinds of studies. Additionally, the XDriver is another browser automation tool which offers many functionalities and gives solutions for the Selenium limitations. In the following Section the XDriver will be extensively described.

1.1 Overview of XDriver

XDriver [49] is an open-source custom browser automation tool, built on the top of Selenium, the official Chrome and Firefox WebDrivers. It was designed for robustness and fault-tolerance during prolonged interactions with web apps and was implemented in Python programming language. It addresses the most Selenium limitations giving solutions through error handling mechanisms. Moreover, it is suitable for website-agnostic analysis, interaction with the browsers, large-scale studies and prolonged interactions with the web applications and provides a plethora of auxiliary functionalities such as automated form filling, configurable built-in crawler, cross-browser, easy user proxy integration and traffic routing through TOR.

The error handling mechanisms are the basic goal of this framework. When a complex, large-scale analysis is conducted, there is no knowledge about the structure and the behaviour of the web applications. At any moment, any unexpected

popup or software crash may lead to a framework's module failure. This causes the block of other functionalities and of analysis in general. These problems can result in ambiguous states such as TimeoutException and Selenium WebDriver crash. The tool has to handle and solve them for going ahead with the analysis. XDriver contains a basic method called "invoke" that catches any raised Selenium exception and offers a solution for each one through calling specific exception handlers or returning default values. It saves the temporary state in the case of the browser restarting with the purpose of continuing the analysis from this state.

An example is the Stale Element Reference exception caused by some actions in web elements such as clicking certain web elements and filling out forms. At the level of Selenium, "find_element_by*" methods will be called by these actions searching the web element in the HTML page. Whenever a "find_element_by*" method is called, XDiver stores the returned object's reference as the key in a hash table and methods or given arguments as a value. If such an error occurs, the framework will refetch the requested object that raises the exception using its hash table entry. Afterwards, it replaces the old object with the new one. This is the solution of the Stale Element Reference exception that offers this framework. Otherwise, if the Selenium Project is used, the analysis will fail.

Other basic Selenium problems are the browsers, the WebDrivers and intermediates components' crashes. In these cases the appropriate XDriver error handling mechanism launches a fresh browser instance, loads a previously used profile to maintain the state and keeps previous settings (e.g. proxy configuration, virtual display, headless mode). Each handling mechanism replaces the old XDriver object with the new one. This object can be a Selenium browser driver or a web element in the case of a Timeout or Stale Element exception respectively.

1.2 Security Header Policies

Nowadays, there is a huge range of web applications which are becoming more and more complicated. There is a variety of different possible attacks in which both the websites and their users can be exposed to. This has led the browser vendors to support a wide range of additional opt-in security mechanisms (e.g HSTS, CSP). These mechanisms are contained in the HTTP response header and the web applications can use them in order to increase their security. They can be protected by known attacks (e.g SSL stripping, cross site scripting) as well as vulnerabilities using these mechanisms. The execution of malicious code inside the web application can be avoided using mechanisms such as CSP, X-XSS-Protection. There are policies that can block the render of the page in HTML tags (e.g frame, iframe tags) and browser features after using them (X-Frame-Options, Feature-Policy). The control of the information, which are included in the browser requests, and the protection of mis-issued certificates are other reasons for using important policies such as Referrer-Policy and Expect-CT.

1.3 Motivation of the master thesis

The developers can check their websites' functionalities performing browsers' tasks while they are using a browser automation framework. They can also use different tools (e.g CSP-Evaluator, CORScanner) in order to check if all the Security Header policies are correctly implemented, as there is no specific tool that evaluates all the policies. On the other hand, many researchers study different kind of websites' topics such as advertisements using browser automation tools like Selenium. They investigate the deployment of the Security Header policies in the real websites, too, writing the code for the policies' evaluation from scratch.

The above reasons led our team to extend the XDriver framework creating a security oriented module named as SecurityAuditor. The XDriver [49], is a custom browser-automation tool built on the top of Selenium, designed for robustness and fault-tolerance during prolonged interactions with web applications. Our module evaluated the Security Header policies detecting the policies' fields, syntax errors or even vulnerabilities that had been observed in published papers. These policies can protect the website from attacks and on the other side from the vulnerabilities that can downgrade the policy's mechanism. Therefore, this module can be used by the users calling only a specific XDriver functionality and giving the policies' names. Thus the users can check if their websites are functionable and if they implement the Security Header policies correctly. Furthemore, this module can be useful for research projects, as they are investigating the policies' deployment in the real websites without spending time to implement tools for the policies' evaluation or to use different tools. Therefore, this module is useful for large scale studies, as it handles the auditing process in a completely automated manner offering information about the website security according to the Security Header policies.

Using this module, a large-scale analysis was conducted so as to evaluate the Security Header policies and to conclude if the web developers followed faithfully the policies' documentation, which was leading to a correct and safe website. Furthermore, other experiments were carried out in order to compare the XDriver framework with the Selenium Project. In particular a web crawler, which emulated the users' interaction with the web apps, was developed. For each web application some actions were done (e.g click web elements, fill out forms) that caused Selenium exceptions as a result of tasks' failures. Then, it was investigated how many of them were solved by the XDriver framework and how many by the Selenium Project. The goal of this comparison was to show that the users, who use the Selenium framework, are facing up many problems during their tasks and analysis as a result of spending more and more time in order to find solutions for them. On the other hand, the XDriver framework, handles the Selenium limitations as much as possible. Therefore, the users can use this tool instead of Selenium focusing on the execution of browsers' tasks and not on finding solutions for the problems.

1.4 Contributions

To summarize, the main contributions of this master thesis are:

- Development of a security module that extends XDriver framework by evaluating the Security Header policies that exist in a range of websites. SecurityAuditor module detects known misconfigurations and faults that may exist in the policies' syntax. As a result they can downgrade the websites' security guarantees and allow the presence of known attacks.
- Execution of a large-scale analysis using the SecurityAuditor module for measuring the misconfigurations and syntax-errors and for investigating if the websites are safe or not due to the security policies implementation.
- Comparison of the XDriver framework to the Selenium Project in order to evaluate the XDriver error handling mechanisms and investigate if the solutions that these mechanisms offer, avoid the Selenium exceptions which are not solved by its framework.

1.5 Thesis organization

This master thesis was organised as follows. Chapter 2 describes the works that have been published about the Security Header Policies. It also contains information about the existing browser automation frameworks and their functionalities so as to understand the importance of them.

Chapter 3 presents the module which was developed during this thesis. It contains information for each Security Header policy such as policies' syntaxes in order to understand their importance. It also describes how the module evaluated each of them as well as the results which were produced.

Chapter 4 contains the evaluation executed by our module. Through this evaluation, results and observations about the Security Header policies are presented. It also contains some conclusions about the websites' safety according to the implementation of these policies in the websites.

The comparison between the XDriver and the Selenium framework is displayed in the Chapter 5. It provides information about the solution which was offered by the XDriver error handling mechanisms. It explains how these mechanisms worked and how the Selenium framework was behaving in the case of a failure.

Chapter 6 outlines the work that could be done in the future so as to improve the XDriver framework. Chapter 7 summarizes the basic points of this thesis and highlights its importance.

Chapter 2

Related work

2.1 Browser Automation Frameworks

Nowadays, the Browser automation frameworks are very important. They can help the users to perform daily browsers' tasks or even test their websites' functionalities automatically. These tools are also useful for different kinds of analysis such as advertisements' studies.

Selenium [30] is the most widely used tool. It is a powerful open-source [31] tool offering functionalities to users in order to interact with browsers automatically. There are some Github projects supporting Selenium. The project [31] supports the Selenium tool in Ruby and Javascript. The TestAutomation framework [18] is another Github project that builds a lot of applications with independent reusable keyword components that can be used by other web applications without spending extra effort. Furthemore, there are other works which support the Selenium tool in Java language [29, 32] in order to help users to run daily browsers' tasks.

The OpenWPM [50, 51] is an open-source tool [20, 21] based on the Selenium. It keeps commands at a high level, performing browsers' tasks and can create many browser instances executing tasks in parallel. Galen [11] is an open-source [12] tool using also the Selenium in order to collect elements from pages and test layout and responsive design of web applications. OpenTest is another tool [19] supporting browser automation processes through a Selenium server. In addition, Robot [25] is a generic open-source[26] automation framework that offers capabilities which can be extended by libraries implemented in Python or Java.

Cucumber is a tool that produces reports whether the software behaves according to the specification or not. It can work well with specific browser automation frameworks such as Selenium. There are some Github projects which support this tool in combination with the Selenium in order to offer extra functionalities [8, 33]. Cucumber implementations in Java also exist in Github [6, 5].

Serenity [34] is an open-source framework [35] that can automate tests and uses the results describing what the user's application does and how it works. It provides reporting capabilities on top of tools like Cucumber and Junit, integration

with Selenium WebDriver and patterns that make it easier to write cleaner and reusable code. There is also a Java implementation of Serenity tool in the Github project [36].

Puppeteer [22] is another browser automation framework with the Puppeteer code in the Github [23]. It provides a high-level API to control Chrome or Chromium over the DevTools Protocol. It offers many functionalities, such as generation of screenshots, automation form submission and UI testing, emulating users' tasks. It also creates an automated testing environment in order to run tests directly in the latest version of Chrome. It can test Chrome extensions, too. This framework was developed in the Javascript programming language. There is a Github project [7] which supports the Cucumber tool using Puppeteer functionalities and creating a behavioural test framework. There is also a Robot framework implementation with Puppeteer [27, 28] for the users who prefer the Puppeteer tool. WebDriverIO [40, 41] is a test automation framework running tests based on the WebDriver protocol and Appium technology. It is based on the Puppeteer in order to automate browsers' tasks.

However, there is not a framework to handle scripts' failures that can be caused during the execution of browsers' tasks. These failures refer to the Selenium limitations that all the above tools contain as they did not offer a solution. For this reason, the XDriver framework was developed, presented in the work [49]. It is an open-source tool [45] built over the Selenium and offers many useful functionalities which makes some important processes before using the basic Selenium functionalities. It also offers mechanisms which avoid Selenium limitations giving useful solutions for them. Therefore, the users can reduce their code complexity and execute browsers' tasks.

2.2 Security Header Policies

There are two online tools [13, 16] in which the user can type a website's URL and get information about the Security Header policies. They are similar to the SecurityAuditor module with the difference that the first tool does not evaluate the CORS policy and the second one both the CORS and Expect-CT. Another difference is that our module could be used without user interaction. Specifically, our module was based on a browser automation framework. It means that a user could perform browser's tasks and evaluate Security Header policies in a number of websites in a completely automated manner.

There are prior works referred to the Security Header Policies. They were focused on the Strict-Transport-Security (HSTS) and the Content-Security-Policy (CSP). These are very important policies if the attacks, which they avoid, are considered. These works referred to vulnerabilities that were observed by executed attacks or even tools which implemented the policies' mechanisms. Some works also existed for the Cross-Origin Resource Sharing (CORS) policy. CORS is also a significant policy as it refers to the websites' resources which can be loaded if the

browser permits them. These works are analyzed in the following Sections [2.2.1, 2.2.2, 2.2.3].

Table 2.1: Comparison of prior works/tools with SecurityAuditor module in relation to the Security Header policies evaluation (Y=Yes, N=No).

Works/Tools	HSTS evaluation	CSP evaluation	CORS evaluation	Expect-CT evaluation	Other policies evaluation *	User automation
Work [67]	Y	partial	N	N	N	Y
Work [61]	Y	N	N	N	N	Y
Work [53]	Y	N	N	N	N	Y
Work [59]	Y	N	N	N	N	Y
Work [65]	Y	N	N	N	N	Y
Work [64]	N	Y	N	N	N	Y
Work [70]	N	Y	N	N	N	Y
Work [46]	N	Y	N	N	N	Y
Work [54]	N	Y	N	N	N	Y
Work [69]	N	Y	N	N	N	Y
CSP-Evaluator [4]	N	Y	N	N	N	partial
Work [48]	N	N	Y	N	N	Y
Work [2]	N	N	Y	N	N	Y
HTTP Security Headers check [16]	Y	Y	N	Y	Y	partial
Header Analyser [13]	Y	Y	N	N	Y	partial
SecurityAuditor module	Y	Y	Y	Y	Y	Y

***** X-Content-Type-Options, X-Frame-options, X-XSS-Protection, Feature-Policy, Referrer-Policy

In the Table 2.1 a comparison of prior works/tools with SecurityAuditor module in relation to the Security Header policies evaluation is presented. Particularly, there are works and tools which evaluate specific and all the Security Header policies respectively. Using these tools the user should type a website's URL. On the other hand, our module evaluated all the Security Header policies in a completely automated way.

2.2.1 Prior works in the HTTP Strict Transport Security (HSTS)

In the work [68] was developed a browser extension (HTTPS Everywhere) which implemented the HSTS mechanism. It used rulesets in order to check each HTTP request and modify each insecure connection (over HTTP) to secure (over HTTPS). It also contained an option to block all HTTP requests because there were cases where a website's functionality failed under the HTTPS and led the user to risk. They also conducted experiments in order to reveal a series of implementation flaws and deployment issues in all the widely used mechanisms referred to the encryption in web services such as HSTS and some of Content Security Policy

mechanisms.

An extensive in-depth study on the privacy threats was conducted in the work [67]. It explored the exposed functionalities and information when the attackers steal the users' HTTP cookies. It was detected a pattern across websites which partially deployed the HTTPS as a result of private information leaking. In this work, it was also used the HTTPS Everywhere extension in order to investigate how a user can be protected by using it because the HTTP cookies could be regularly exposed although such extensions are used.

A measurement of the HSTS deployment was conducted in the work [61]. Transmission errors, redirection errors were detected by this measurement. An enhanced HTTPS stripping attack was also designed in order to demonstrate vulnerabilities and give some suggestions for eliminating them. The work [53] investigated the correct HSTS implementation. Common mistakes and difficulties from HSTS configuration as well as some approaches about the correct HSTS policy implementation were described in this work.

An in-depth empirical study for the HSTS and public-key pinning policy was conducted in the work [59]. Important observations were mentioned about the incorrect implementation of these policies. Some examples referred to this work, where the developers had not understood the HSTS and public-key pinning policy as a result of having vulnerabilities in their websites. The work [65] investigated these two policies, too. The quantity and the quality of these policies' implementations were observed in servers and popular browsers. Some attack scenarios were also mentioned. These works had contained some observations about the correct HSTS implementation that our study also confirmed with a few differences.

2.2.2 Prior works in the Content-Security-Policy (CSP)

The work [64] contained an analysis of how CSP deployment had evolved over the years and some observations were mentioned about the CSP deployment in websites. It was also investigated the usage of CSP in some cases such as TLS enforcement, framing control. A campaign and subsequent survey was conducted about the CSP complexity which led the websites not to enforce this policy. The work [70] was focused initially on the practical benefits of CSP and contained an analysis focusing on XSS protections. So, some classes of CSP bypasses were identified as well as the way that these cases downgraded the CSP mechanism, Another analysis was also conducted in order to observe the benefits of CSP deployment and the "strict-dynamic" keyword was proposed as an additional policy directive to the CSP documentation. Another work [46] used an analysis as well, in order to observe how the browsers behaved in the CSP deployment. This means that a number of weaknesses and misconfigurations' errors were observed, which downgraded the CSP mechanism. Therefore, the importance of a better CSP which could reduce the vulnerabilities in the websites due to its deployment was mentioned, too. These works gave to our team useful information about the weaknesses

and misconfigurations which could exist in a website due to the CSP implementation. Thus, SecurityAuditor module was detecting these information in the CSP syntax as misconfigurations or even practices which could lead a website to be vulnerable.

The paper [47] proposed an extension of CSP named as CCSP in order to protect more websites through the CSP. Therefore, the extension from different perspectives was analyzed and it was tested on major websites. It was observed that this extension could be deployed with limited efforts by using the popular content providers in order to reduce the CSP limitations.

The work [60] referred to the XSS attacks which were found out using script gadgets in websites. Many observations about this kind of attack were contained, too. Furthemore, it explained the ways that the attackers could downgrade the CSP mechanism and how the XSS attacks could be handled.

Another work based on the CSP was the [69]. A case study in data exfiltration via DNS prefetching was conducted in order to refer assumptions which could not hold the CSP mechanism in practice. A crawler also performed reporting on the CSP implementation in practice. The work [71] also contained a study about the challenges in CSP deployment. A semi-automated policy generation was used by the web application crawling on a set of popular websites, as well. Some suggestions were proposed for the CSP improvement.

The CSP-Evaluator [4] is an online tool, that the user can type a specific CSP syntax. Then useful information about the CSP evaluation will be returned to the user. This tool is similar to the SecurityAuditor module with the difference that our module can evaluate the policy syntax in a number of domains.

The article [63] concluded a study about the CSP deployment in websites. It referred to the policies syntaxes' errors identified by the analysis. It was also mentioned how much effort the websites' developers should make in order to adapt this policy. A tool named UserCSP was developed, using dynamic analysis to automatically infer CSP policies and giving the opportunity to the users to enforce this policy correctly in websites.

The work [54] was focused on the CSP deployment when it was interacted with browsers' extensions. An empirical study about the extensions which downgraded the CSP mechanism was reported as well. It also proposed an extension-aware CSP endorsement mechanism in order to facilitate a wider CSP adoption.

In the work [57] some scenarios were identified to downgrade the CSP mechanism due to successful XSS attacks. An extension, named as PreparedJS, was presented, too. This extension addressed the identified weaknesses and offered safe script templates which provided full protection against XSS attacks combining with the CSP.

There were also works, in which tools for generating or even improving the implemented CSP, were developed . The websites could be fully protected using these new or updated CSP syntaxes. One of this work was the [52] which proposed the AUTOCSP technique. It was an automated technique for retrofitting CSP to web applications. This technique leveraged dynamic taint analysis identifying

which content should be allowed. Then, it modified the server-side code in order to generate pages with right permissions. The paper [62] proposed the CSPAutoGen tool in order to enable the CSP in real-time compatible with real-world websites and without server modifications. The work [58] generated CSP in an automatic way which could protect the websites more. These works were not similar to the SecurityAuditor module and they are mentioned in order to be understandable that there were many kinds of works for this policy due to its complexity and importance.

2.2.3 Prior works in the Cross-Origin Resource Sharing (CORS)

The work [48] conducted an empirical-study about the real-world uses of CORS. The policies' complexities and the detecting new security issues of this policy were mentioned. It analyzed the reasons that these issues existed. It was also proposed some improvements and clarifications in order to address these problems. An open-source tool named CORScanner was developed which evaluated the CORS policy. The SecurityAuditor module contained a modified version of this tool for the CORS evaluation as it detected enough CORS vulnerabilities.

The work [55] developed the BrowserAudit tool. It was a tool for testing that a deployed browser could enforce the guarantees implied by the main standardised and experimental security mechanisms. This work referred not only to the CSP but also to the HSTS and CORS mechanisms. It also validated the tool by discovering both fresh and known security-related bugs in major browsers. Our SecurityAuditor module differed from this kind of tool as it evaluated the policies, checking their syntaxes from the websites' header flows.

The work [66] focused on the Same-Origin-Policy (SOP) controlling the interaction between the host document and embedded document from a SOP rules' subset. An empirical study was conducted on major web browsers in order to show the dependence of this policy from specific attributes such as CORS attributes, embedding elements' types.

Furthemore, there was a tool named CORSTest [2], which detected CORS misconfigurations. This tool was developed in order to observe how many websites were actually vulnerable.

Chapter 3

SecurityAuditor Module

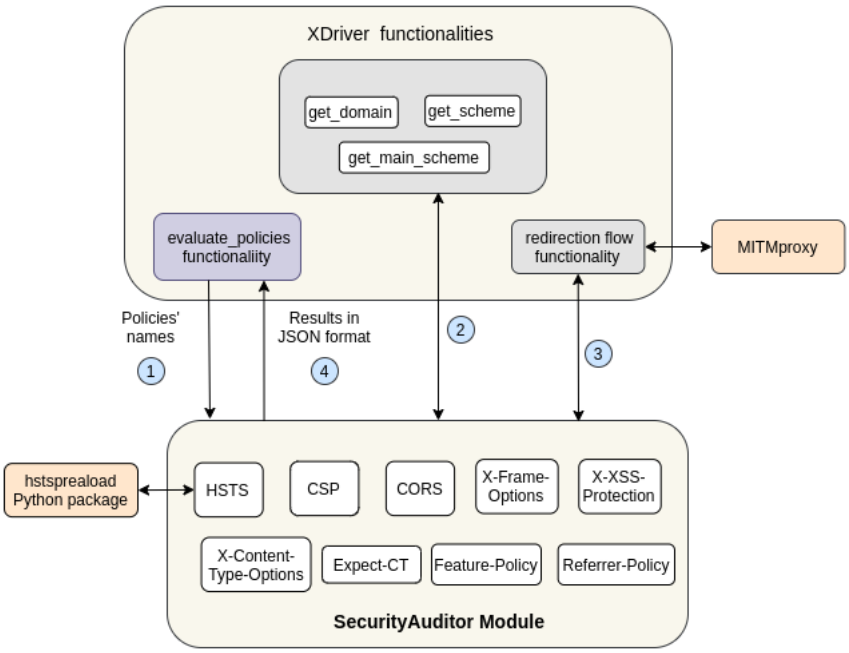

Figure 3.1: SecurityAuditor overview

In this Chapter both the existing Security Header policies and the results, obtained by the SecurityAuditor module, are described. This module was developed during this thesis and offered information about the website's security according to the Security Header policies. An overview of our module is shown in the Figure 3.1 and presents in detail how this module worked and which results were produced.

The SecurityAuditor module, that used some XDrivers' functionalities, investigated existing security mechanisms. This module could be used through calling the XDriver functionality (evaluate_policies). This functionality contained arguments, referred to the policies' names which the user wanted to evaluate. The module

collected information about the website's URL (e.g main domain, URL scheme etc) which could be needed for the policies' evaluation. These information were offered by XDriver functionalities. The module also used XDriver functionalities in order to gather the header's flow (get_redirection_flow) in the case of the real time policies' evaluation. This functionality used a MITMproxy so as to collect the request and response headers of the websites. Then, it could evaluate the policies visiting a number of websites at real time or reading the header's flow of the websites from a JSON file added as functions' arguments.

Therefore, the module searched each of the security policies in the website's headers. It detected syntax errors and faults that might exist in each policy syntax. Furthermore, it concluded vulnerabilities that could be contained in a policy syntax due to some bad practices or even faults which had been implemented by the websites' developers. If there were neither errors nor misconfigurations, the module would refer that the policy was safe for that website. Additionally, this module offered some additional notes for specific policies such as policies' fields in experimental state.

3.1 HTTP Strict Transport Security (HSTS)

HTTP Strict Transport Security (HSTS) [17] is one of the most important security policies as it protects a domain from Man-In-The-Middle (MITM) attacks. It instructs a user's browser to connect to the domain only over HTTPS, even if the user types in the address bar an explicit HTTP URL or an HTTP URL followed by the redirection flow. The browser remembers this policy for a specific period of time, which the developer can set the time with the specific HSTS field (max-age field). The directives' values are set in seconds. As this period passes, the browser has to enforce the policy again. This policy can be specified dynamically through the HSTS HTTP header by websites' developers or preloaded by the browsers for popular domains. HSTS Preload service has been implemented by Google that maintains a list of domains with their HSTS policy. This service maintains two independent lists (Chrome and Firefox) [14]. The domains have to meet some requirements in their HSTS policies in order to be acceptable from these lists. They must serve a valid certificate and HTTPS traffic to all subdomains, too. There are also some requirements for the policies' fields such as the "max-age" directive which must contain at least 31536000 seconds (1 year). The "includeSubDomains" and the "preload" directive must be specified.

Therefore, important and famous browsers (e.g Chrome, Firefox, Edge, Safari) follow one of these lists in order to enforce the HSTS policy. Each browser checks the preload list first. If the domain exists, then the website will communicate only over HTTPS flow with the user's browser. Otherwise the website sends a HTTP request to domain and gets the HSTS header from the headers flow of the domain's response. Then, it enforces the HSTS HTTP header policy and the communication between the user's browser and the domain are done over HTTPS

flow. This policy contains the "includeSubdomains" field, too. If this parameter is specified, it will be applied to all the websites' subdomains as well. The "preload" is an extra field that can be set when the domain exists or recently requests to be submitted in the preload list. The "max-age" field is an obligatory parameter of policy in comparison to the other two which are optional.

Our module first got the redirection flow for that domain using the appropriate XDriver functionality. For the main domain and the subdomains, it detected the HSTS Preload service using the hstspreload Python package [15]. This package checked the preload list regarding whether that host existed in the list or not. The module concluded this information to the HSTS results. It also checked the HSTS HTTP header following the HSTS documentation [17] and detected faults that the policy might contain. These faults perhaps downgraded the HSTS mechanisms and could be syntax errors such as the absence of "max-age" field and fields which did not exist in HSTS documentation. Additionally, the SecurityAuditor detected any known misconfigurations. When the policy was served over HTTP, the browser ignored this policy, which was one of these misconfigurations [59]. Another one was when subdomains or the base domain were not protected due to syntax errors (malformed) in the HSTS policy or opt out the policy (max-age = 0 seconds). Small max age values were also dangerous as they were able to downgrade the policy mechanism. That is why our module set "max-age" values less than 86400 seconds (1 day) as a misconfiguration observed by the experiments of the published paper [59]. Our module concluded all this information in the HSTS results. These results also contained the "enabled" boolean value as a conclusion for the website's safety, which would be True if the main domain and the subdomains had implemented the HSTS policy correctly or False in the other cases.

3.2 Content Security Policy (CSP)

Content Security Policy (CSP) [1] is another important policy that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks have the purpose of stealing website data or even of distributing malware. CSP was designed initially to restrict the execution of script content inside the website as the developer was able to create an origin based whitelist or allow the execution of inline JavaScript known as CSP 1.0 version [37]. Thus, the execution of inline JavaScript could be exposed by the attackers. That is why the CSP Level 2 [38] supported cryptographic nonces to the CSP standard. The developer was able to embed the "nonce" directive field adding an additional attribute to script HTML elements as a result to define a whitelist of specific inline scripts. However, it is widely known that there were cases in which some scripts were called by other scripts such as advertisements. This means that the inline scripts could not be executed unless they were present in the origin based whitelist. This problem might cause failure of the website's functionalities. For that reason the CSP3 version [39] added the "strict dynamic" expression. When

this expression was contained in CSP policy, all programmatically edit scripts that were whitelisted and all the programmatically additional scripts of them could be trusted. All these problems led to the CSP evolution during the years in order to avoid the existing vulnerabilities. CSP policy also offers two modes, such as the "enforce" mode, in which the browser can apply it. On the other side, the "report-only" mode allows web developers to test their policies by monitoring the effects of them and reporting all the violations to a specific URI.

The SecurityAuditor module detected the policy mode in order to inform the user, if the policy was enforced by the browser or only reports the violations. The directives (e.g child-src, frame-src, script-src etc) that the policy contained, defined the valid sources of specific objects or even HTML elements. They contained fields, which could be a source list (whitelist), scheme sources or specific parameters like self, unsafe-hashes. Our module detected the directives as well as their fields. It also checked for syntax errors (malformed) and misconfigurations.

Misconfigurations. There are some vulnerabilities that can be used by the attackers. The usage of "unsafe-inline" fields with the absence of "strict-dynamic" and nonces or hashes fields in a directive can downgrade the CSP mechanism and the attackers can cause problems to the websites [64].

When the developer sets the same directive multiple times in the same header policy with different fields, there is no clear explanation about which directives' fields will be applied by the browser. This does not happen when there are multiple Content Security Policies in the websites' response headers. In that case the browser will enforce each policy individually [64].

Furthemore, it is dangerous to contain the "data" scheme-source, in "script-src" directive without the presence of "nonce" attribute, since it allows the execution of inline scripts [70]. The syntaxes' faults (malformed syntax) and the unknown directives also referred to as misconfigurations by our module. Specifically for these directives, there was a list of them as an extra note in the results.

Bad practices. There are some dangerous practices that can be caused by a bad implementation of some CSP directives. That is why our module detected bad practices of CSP policy by notifying the researchers who used the module's results in a large study. It is necessary for the developers to be updated with the CSP changes, so as to always use the current version of the policy. Due to the evolution of the CSP, there are some obselete directives like block-all-mixed-content used by the developers.

In addition, the web developers have to be careful when they are implementing CSP. The usage of a general wildcard and not specific scheme values (http or https or data) in whitelists allows the arbitrary hosts to include content using these fields. As a result the policy is insecure since an attacker can take advantage of these arbitrary hosts [70]. When the whitelist contains a wildcard, whose scheme is not blob, data or filesystem, then any URL can be matched. An attacker, who

may control the content of the URL, can cause many problems in the website that the user visits [46]. The absence of the "require-trusted-types-for" directive is also very important, as it can lock down the DOM XSS injection sinks. This information refers to the CSP Evaluator tool [4], which gives some information about the safety of the CSP policy.

It is also a fact that the websites have to set the "default-src" directive. It works as a fallback for other fetch directives, which controls the locations where certain resource types may be loaded. Therefore, the user agent searches it in the case of absent fetch directives in order to set default values. When the CSP does not contain this directive as well as other important directives like the "script-src", it gives opportunities to the attackers to cause problems in the websites. "script-src" specifies valid sources for JavaScript including URLs loaded directly into script elements and other things which can trigger script execution (e.g OnClick event handler). As a result, when there are not these two directives, there are no restrictions to the execution of script elements. The "object-src" is another directive as it specifies valid sources for HTML elements (e.g object, embed, applet elements), which are not receiving new standardized features (e.g security attributes). These cases were detected as bad practices by our module and have referred to the [70].

To conclude, our module contained all these misconfigurations, bad practices, the detecting directives and fields. This also contained basic information that were referring to faults that existed in specific directive field syntax or in the general policy. Furthemore, the mode that the CSP had been applied on and the "enabled" information, which would give the result if the current policy was safe, were also useful results for a large scale analysis. Due to the evolution of the CSP there were some directives in experimental state, which added an extra note in the results of them for further analysis.

3.3 Cross-Origin Resource Sharing (CORS)

The Cross-Origin Resource Sharing (CORS) [3] is a complex mechanism for transferring requested resources safely among the user's browser and the requested domain. The browser makes a "preflight" request to the server hosting the cross-origin resource and sends the origin of the requesting domain. This process is done by the Origin header, added by the browser. The server generates the "Access-Control-Allow-Origin" in HTTP response headers, which contains the allowed origins. Then, the browser will check if the allowed origins from the "Access-Control-Allow-Origin" are matched with the requesting domain. If the whole process is successful, the browser will be sure that the requesting domain is the same as the domain which the browser wants to communicate with.

This policy is used by many modern websites in order to allow access from trusted third parties and subdomains. However, the web developers make mistakes due to the mechanism's complexity when they implement the CORS in their web applications. That is why there is an open-source tool named as CORScanner

[2], which follows the CORS mechanism steps. It will evaluate if a website is safe using the CORS policy or refers to the vulnerabilities, which lead the website to be vulnerable. These vulnerabilities are referred to the published paper [48] detected by the CORS tool. Our module contained the CORS policy using a modified version of the CORScanner tool. It means that some codes were replaced by the CORScanner tool with XDriver functionalities called by the XDriver mechanisms in the case of an unexpected error, such as getting the redirection flow. For this reason, our module contained the same misconfigurations as the CORScanner tool, referred to the published paper [48].

3.4 X-Content-Type-Options

The X-Content-Type-Options response HTTP header [42] deals with the MIME types. It is used to protect websites against MIME sniffing vulnerabilities, which can be caused as a user disguises a particular file type in the time that the content is downloaded from a website. This can be avoided by blocking a request for a specific MIME type using the only directive "nosniff", which this policy contains. So, they can send MIME types through their origin servers.

Our module would check if the "nosniff" directive existed in the policy and if the policy existed many times in response headers. Any syntax errors could be detected by the module, too. These informations were taken into account in order to make a conclusion for the safety of the policy, adding a boolean "enabled" field in the results.

3.5 X-XSS-Protection

The X-XSS-Protection response header [44] prevents websites from Cross-Site-Scripting (XSS) attacks by blocking the websites' loading and contains only a value (0 or 1). The value "1" can set the "block" mode. It enables XSS filtering or the "report" field combined with a URI, in which the browser will report the violation that may be caused in the case of an XSS attack.

The module detected these values, fields and the syntax errors, which might exist in the policy header. It would also check if the policy existed multiple times in the response header, which was not allowed. The above information led to a general boolean "enabled" field in which the module made a conclusion about the website's safety. This field could be True in the case of a correct policy implementation. It would be False if there were syntax errors or in the case of disabling the XSS filtering, setting the value "0" in the policy.

It is also important to note that this policy can be applied using the Content Security Policy. It can be achieved by not allowing the execution of inline JavaScript ("unsafe-inline" field). However, there are some legacy browsers such as Internet Explorer and Safari, which still apply this policy. There are enough domains that implement this policy too, which will be analysed in the Section 4.4.

3.6 X-Frame-Options

The X-Frame-Options HTTP response header [43] is used in order to instruct a browser how to behave in the site's content handling. This occurs regarding whether the browser may display the transmitted content in frame HTML tags that are part of other web pages. Therefore, the browsers can allow or not to render a page in a frame, iframe, embed or object HTML element through this policy. As a result this policy can avoid the click-jacking attacks by ensuring that their content is not embedded into other sites. Specifically, this policy contains three fields (DENY, SAMEORIGIN, ALLOW-FROM). The first one refers to the page which cannot be displayed in a frame HTML tag regardless of the site that attempts to set the "DENY" field in policy. Another field describes the page, that can only be displayed in a frame HTML tag on the same origin as the page itself, using the "SAEMORIGIN" field in policy. The last one is the "ALLOW-FROM" combined with a URI and can be set in order to be displayed on a page in a frame HTML tag only on the specific origin URI.

Our module detected these three policy directives. If any syntax errors (malformed) in policy was detected or even if the policy existed more times in response headers, then it would refer to the results. Any syntax error and the presence of multiple headers led the policy not to be safe. For this reason, an "enabled" field was contained in policy's results.

Furthemore, it is important to note that the X-Frame-Options can be implemented in the Content Security Policy using the "frame-ancestors" directive. That is why this policy is ignored by the browsers when the website implements it through the CSP. However, this is not clear for the web developers and for that reason the module evaluated it with the same importance as the other policies.

3.7 Expect-CT Policy

Expect-CT Policy [9] prevents the use of mis-issued certificates for a website. The Expect-CT header policy lets the websites to select Certificate Transparency requirements as well as to enforce them. The Expect-CT documentation [9] refers to the mechanisms that can be required by the Certificate Transparency (CT). This policy contains three directives. The "max-age" directive is obligatory and refers to seconds after reception of the Expect-CT header field while the user agent should regard the host of the received message as a known Expect-CT host. The user agent can report Expect-CT failures when the policy contains the "report-uri" directive combined with a URI. The browser can enforce the policy when the "enforce" directive exists in it in order to refuse future connections that violate its Certificate Transparency policy.

Our module detected the absence of "max-age" as a syntax error as it was an obligatory directive. The module also checked the values of this field since a developer could set zero as a value. That means the policy was opting out. This

was a misconfiguration which was contained in the module's results. Other two directives were optional and detected by the module. When the policy was enforced over HTTP, the browser ignored this header. This was another misconfiguration that existed in the module's results. It was also important to refer that the HTTP headers could not contain multiple Expect-CT headers, as it was the last misconfiguration that the module's results had. All these results were concluded to the boolean field "enabled" contained in the policy results which indicated whether the website was safe or not.

3.8 Feature Policy

The browsers can allow or block features through the mechanism of Feature-Policy header [10]. These features (e.g camera, fullscreen, geolocation) refers to the top-level of a webpage or even to embedded HTML frames. The web developers can enable, disable or change the behaviour of certain browser features through this policy. These features are known as directives in the policy documentation [10]. The website's developer has to set a directive combined with a list of origins that takes one or more values (allowlist). These values such as self, none can allow or not the specific browser feature. The directives refer to the browser features such as payment and the allowlist's values to the behaviour of them. The feature can permit all browsing contexts (wildcard field) or only the browsing contexts in the same origin ("self" field). It can also be allowed only to the iframe HTML tag, in which the feature works ("src" field) or even be disabled in top-level and nested browsing contexts ("none" field).

The SecurityAuditor module checked the policy for any syntax errors. Multiple same fields in directives could exist and the module detected them as misconfigurations. In particular, the wildcard can be contained in the origin list of a specific directive. This is a dangerous value as it allows the presence of the certain feature and all nested browsing contexts (iframes) regardless of their origin. This value is also set in a directive as default value in the case of its absence in the policy syntax. That is why they were detected as misconfiguration by our module. At this point, it is necessary to note that the careful policy implementation by the web developers is crucial, because they can cause misleadings setting this policy. The combination of the "none" parameter, which disables the feature such as microphone in top-level and nested browsing contexts, with an origin list leads to misleadings. If this exists in a policy, our module will contain it in the misconfiguration results. Multiple Feature-Policy headers were also detected by our module, as they were not allowed according to the documentation [10]. The module was checking all these results and concluding them to an "enabled" field, which indicated whether the website was safe or not. It is also important to note that the feature policy is in an experimental state and this can be changed from time to time. Currently its name was changed to Permissions-Policy. This change was investigated by the security policies evaluation in order to check how many domains contain the old

name and how many the new one.

3.9 Referrer Policy

Referrer Policy [56, 24] controls how much information can be sent through referrer HTTP header. This information can prevent the data, which may be sent by web applications to other URLs. It sends information such as the user's session identifier, user's profile URI without leaking them in the URL. This policy contains some specific directives, which each of them limitates the information that will be sent from one web application to another URLs. This policy contains some fields. When the "no-referrer" field is set in the policy, there will be no restriction about the information that is sent along with requests. The "no-referrer-when-downgrade" field, which is the default field when the policy is missing from the website, is referring to the information which will be sent in the case of protocol security level status in the same or improved communication (HTTPS → HTTPS or HTTP → HTTPS). Only the "origin" information can be sent along the requests when the "origin" field is set in the policy. When the policy contains the "origin-when-cross-origin" field means that the "origin", path and query string can be sent when they perform same-origin request. For the other cases only the "origin" information can be transferred. Something similar happens for the "strict-origin-when-cross-origin" field, with the difference that the "origin" information will be sent alone in case that the protocol security stays in the same level. The "strict-origin" field refers to the same protocol security level and sends only the "origin" information when this condition is applied. The "same-origin" field enforces the browser to send the referrer information only in the same-origin requests. The "strict-origin" field only transfers the "unsafe-url" field which will send the "origin", path and query string without any restrictions.

SecurityAuditor module detected all the directives that a referrer policy could contain. It is also important to refer that the HTTP headers could not contain multiple referrer header, as it was the only misconfiguration, included in the module's results. Any syntax errors in the policy also referred to the results. Another important field was the "enabled" field, which was a boolean value, reporting to the safety of the website.

Chapter 4

Evaluation of the Security Header Policies

A large scale analysis of the Security Header policies for 100K Alexa domains was conducted using the SecurityAuditor module. This analysis concluded all the policies except from the Cross-Origin Resource Sharing (CORS). As it was mentioned before, our module contained the implementation of the CORScanner tool [48, 2] with a few changes in order to use the XDriver functionalities which called and were called by the XDriver error handling mechanisms. Therefore, it was not essential to conclude this policy in the below evaluation.

4.1 Security Policies' overview

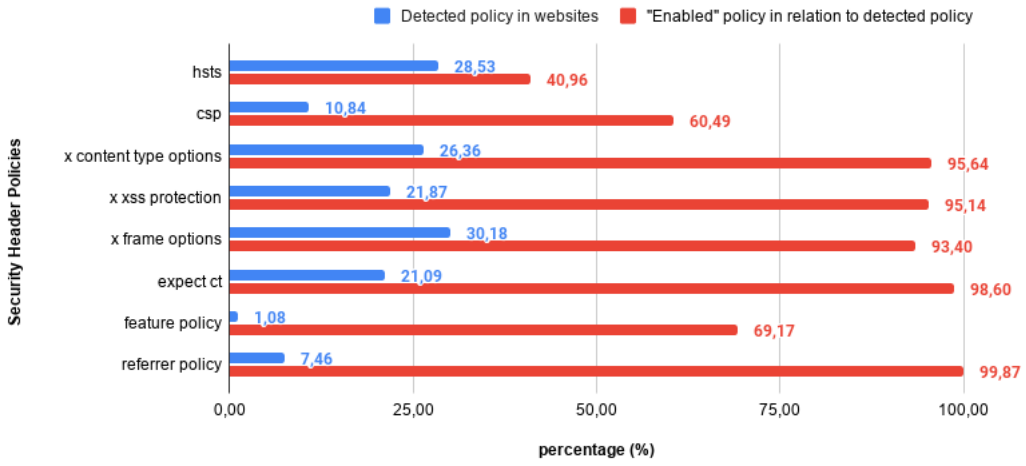

Figure 4.1: The percentage of both each policy detected in the websites and the correct implemented policies in relation to the total number of detected ones.

Figure 4.1 gathers the results of all the Security Header policies. Our module detected the Content Security Policy up to 11% of the total domains. This was a quite small percentage, as the websites have to contain this policy, because it protects them from important attacks such as XSS. This policy contains many directives and fields which makes complex the policy syntax. There are also changes being made over the years by adding or removing directives due to the presence of a larger number of vulnerabilities in websites. Therefore, the web developers are confused and do not prefer to use this policy. It was also noticed that the 61% of the CSP were correctly implemented. This means that the module detected some vulnerabilities or even mistakes in the policy syntaxes.

Another crucial policy is the HSTS, which was detected in the 29% of the websites. It was a small percentage if it is considered that this policy ensures the secure data transfer between website and user using HTTPS communication.The 41% of the HSTS enforcement policies were properly implemented. Therefore, our module diagnosed many syntaxes' mistakes for this policy.

26% of the websites had implemented the X-Content-Type-Options policy, as most of the websites contained document files to their functionalities. As a result the websites needed this policy implementation in order to prevent them from MIME type sniffing attacks. It was also observed that the developers implemented the policy correctly to 96% of it. This happened due to the simple syntax that the policy contained.

The X-XSS-Protection policy was detected by the 22% of the domains. However, many browsers stopped to support this policy, because it could be enforced through the CSP. This means that either the developers did not have the knowledge of the policy's deprecated or their websites used a legacy browser, which was the only way to support this policy. This was also a policy with few implementation's mistakes in the policy syntax, as 95% of the policies were correctly implemented.

The X-Frame-Options policy was detected by 30% of the websites. This is also a policy, which can be enforced through the CSP, using the "frame-ancestors" directive. However, the browsers still support it. The only case, in which the browser ignores it, is the presence of CSP and X-Frame-Options policy in a website. In the Section 4.5 it will be analysed more. Furthermore it did not detect many policy syntax mistakes, as 93% of this policy implementation were correct.

The Security Auditor module detected the Expect-CT policy in 21% of the total domains. 99% of them were safe by implementing this policy correctly. So 0,2% of the detecting Expect-CT policies had either syntax mistakes or misconfigurations. In Figure 4.2 is presented the percentage of each misconfiguration of the total non-correct policies (non-"enabled" policies). Each of these percentages means that the policies contained at least one of these misconfigurations. Therefore, 44% had led the browser to ignore it by setting the zero as the "max-age" value. The "max-age" was also an obligatory field and it was not included in 9%. It was also observed that 26% enforced this policy using HTTP communication. This was a case for the browser to ignore the Expect-CT policy. As a result, these domains were acting in the same way as when they did not implement it. Last but not least,

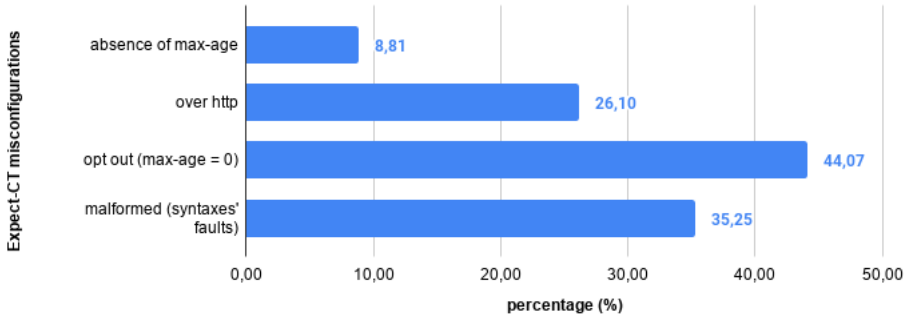

Figure 4.2: The percentage of each misconfiguration in the incorrect (non-"enabled") Expect-CT policy

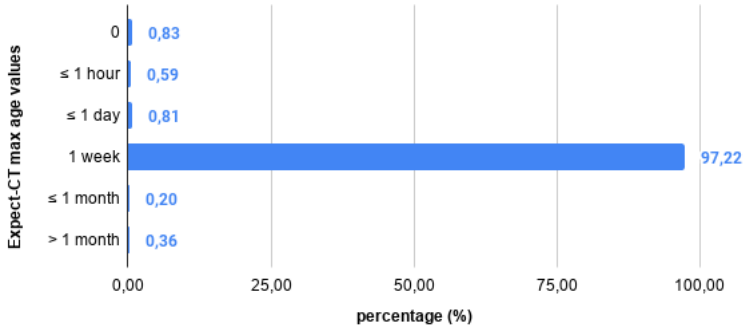

Figure 4.3: Histogram of max-age values for Expect-CT

our module detected 35% syntax errors (malformed) in this policy. A histogram of the "max-age" values is presented in Figure 4.3 and was designed in order to show the most preferred values for this field. It was noticed that 97% of the total "max-age" values had been set exactly in 604800 seconds (1 week).

Additionally, the Feature-Policy was used by very few domains. 1% of them implemented it. This means that they did not allow or disable features to their websites. There were also 165 domains, which used the new name of this policy (Permissions-Policy). However, there were 912 domains with the old one (Feature-Policy). This means that the browser could not recognise the policy in the domains which contained the old policy's name. It was also observed that 86 domains, such as passportindia.gov.in and www.metrodeal.com, were trying to implement this policy inside the HTML code. However, the web browser could not recognise this policy with this way. Additionally, some domains such as hashflare.io had set the policy in a comment HTML line code. The developers might have understood that this policy could not be set in the HTML code. So, they removed it by setting this line code in comments.

The Referrer-Policy was detected only by 7%. This happens because the developer could only restrict the information which could be transferred inside the web requests. So, this policy did not protect the websites from attacks in comparison with the other policies. That's why the web developers did not prefer it .

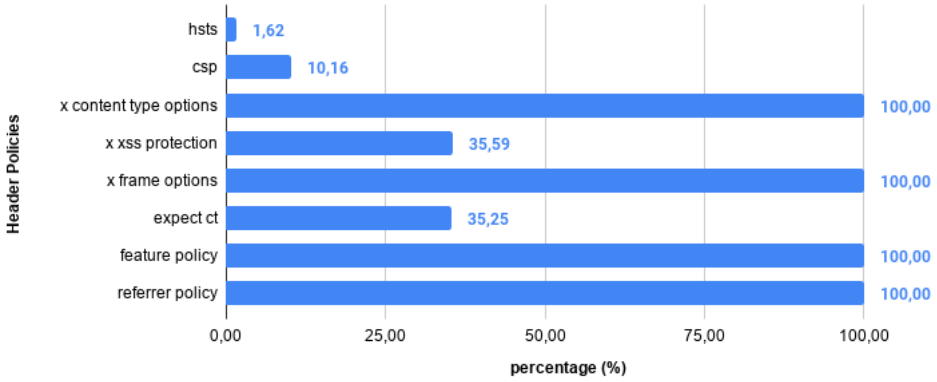

Figure 4.4: The percentage of each Header Policy with syntax errors

In Figure 4.4 the percentages of the policies syntax errors (malformed) of the total non-correct policies are presented and were up to 100% for the X-Content-Type-Options, X-Frame-Options and Referrer-Policy. This happened because there were no known misconfigurations for these policies. As a result our module detected only syntax errors in the case of an incorrect policies' implementation. There was not detected any misconfigurations in the Feature-Policy and the Expect-CT had syntax errors (malformed) up to 35%, because there were many opt out misconfigurations (max-age = 0) which led the browser to ignore this policy. The same result came out for the X-XSS-Protection, as there were many websites which disabled the XSS filtering by setting the appropriate field in policy (x-xss-protection: 0). As a result the websites were not protected by Cross-Site Scripting attacks. The HSTS and CSP were detected with syntax errors in approximately up to 9% and 1% respectively. The rest were other misconfigurations, which will be analysed in the Sections 4.2 and 4.3.

4.2 Analysis of HTTP Strict Transport Security (HSTS)

The HSTS policy is one of the most important policies. In Figure 4.5 the percentage of each HSTS field of the total correct HSTS policies (safe policy) is presented. It was observed that 100% of the websites, which implemented this policy, used HTTPS communication. That happened, because a recommendation of a safe HSTS policy was the HTTPS communication and not the HTTP, which was detected as HSTS misconfiguration by our module. It was noted that 49% of the safe HSTS policies were enforced by the policy to their subdomains. 32% of the safe HSTS domains contained the "preload" directive in the policy. However, 13% of

the domains, which were protected by the attacks using the HSTS policy, were in the HSTS preload list. This means that many of them did not exist in the HSTS preload list although they were used in the "preload" directive in the policy. This happened, because the HSTS preload list is not updating every day, as a result recently submitted domains are not included in this list and therefore, they are not detected as "preloaded" domains by our module.

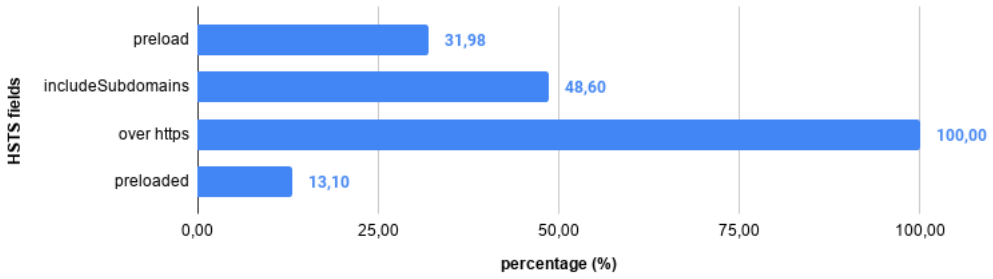

Figure 4.5: The percentage of each HSTS field in the detected HSTS policies

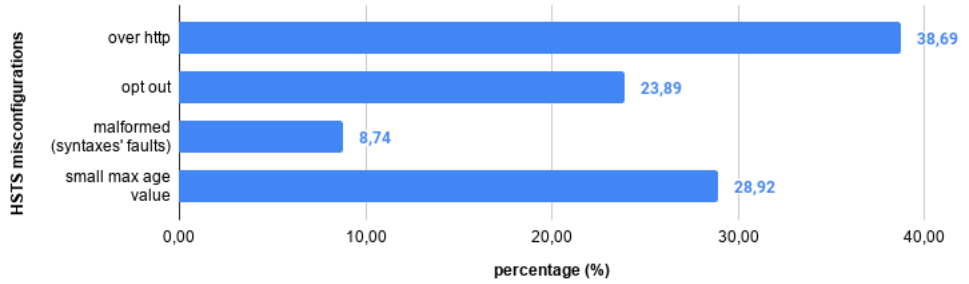

Figure 4.6: The percentage of each misconfiguration in the incorrect (non-"enabled") HSTS policy

The percentage of each HSTS misconfiguration to the total incorrect HSTS policies are presented in Figure 4.6. Each of these percentages refer to the unsafe policies, which were detected with one or more of these misconfigurations. 39% of the total unsafe policies were enforced using HTTP communication, which led the browser to ignore this policy. 24% of them had set the "max-age" value with zero, which is another way for the browser to ignore the HSTS policy. 9% of the unsafe policies had been detected with incorrect syntax. 29% had set the "max-age" fields with values less than 86400 seconds (1 day) detected as small max-age values by our module. It was also observed that two domains (venezuelaaldia.com, panampost.com) had set the HSTS policy using the CSP syntax which was an unexpected case. There was also another strange observation. These two domains used not only the CSP syntax but also the same syntax. It was obvious that

the developers, who implemented these policies, had copied this policy syntax from an online forum or these websites were served by the same company. In addition, the sonycreativesoftware.com domain used the HSTS policy by setting the "max-age" field using quotes characters, which is an incorrect syntax (max-age="31536000"). So the developer was perhaps confused with other policies, which uses these characters in their fields' syntax. Another explanation might be, that the developer had not totally understood the policies' syntax of the HSTS documentation. The renderotica.com domain used the HSTS policy by setting the value "31536000" without referring to a specific field. In this case the developer might want to set the "max-age" field.

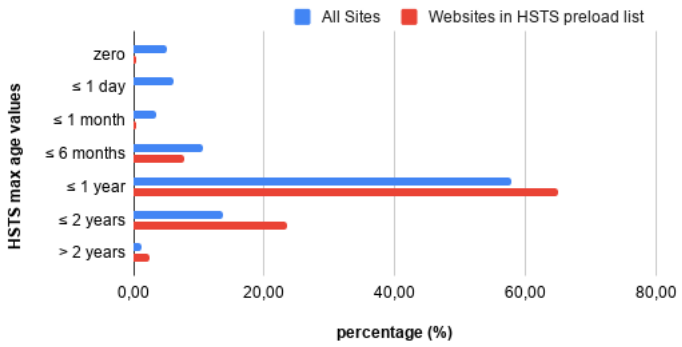

Figure 4.7: Histogram of max-age values for HSTS

The "max-age" values were also analysed. In Figure 4.7 the "max-age" values from all the websites, which implemented the HSTS policy as well as the preloaded domains, are presented. It was observed that most websites had used values from 6 months up to one year. The second preference was the period from one to two years. That happened for all domains which implemented this policy and for the HSTS preloaded domains as well, because the "max-age" value of one year was acceptable for the domains, which wanted to be included in browsers' HSTS preload list. The value of two years was a recommendation according to the HSTS preload lists in order to protect these websites more. On the one hand, these reasons were acceptable for the HSTS preload domains. On the other hand, the domains, which were not in the HSTS preload lists, might have followed the preload lists directions for more safety. For these domains, there was also the explanation that some of them were recently submitted to the preload lists. These cases could not be detected by our module as the list did not update each change.

4.3 Analysis of Content-Security-Policy (CSP)

A website can select in which mode the CSP will be set. It can use this policy only for reporting the violation for the testing cases (report-only mode) and enforce the CSP (enforcement mode). It can also use both of them in order to check the

violations and at the same time to enforce it. In Figure 4.8 it was observed that 90% of the detecting Content Security Policy was enforced. 7% of them reported only the violations and 3% used both of the modes. It is obvious that most domains used this policy in order to protect the websites. It is also important to note that the CSP has been changing its name over the years. First the CSP was called X-WebKit-CSP and then it was renamed to X-Content-Security-Policies. Our module did not detect domains, which used the old names. This means that the developers are using the recent name (Content-Security-Policy).

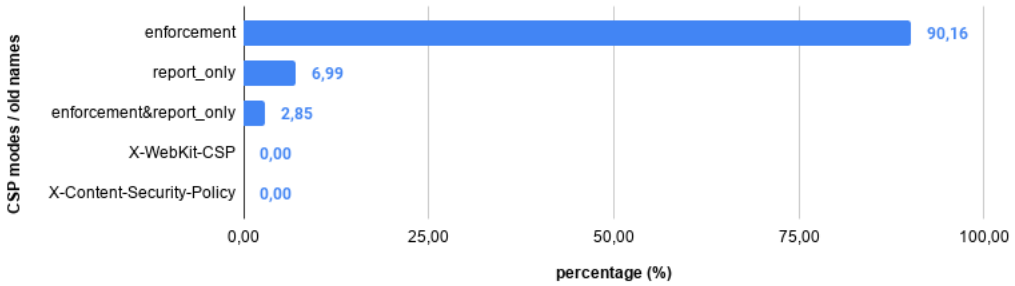

Figure 4.8: The percentage of CSP modes/old names in the detected CSP

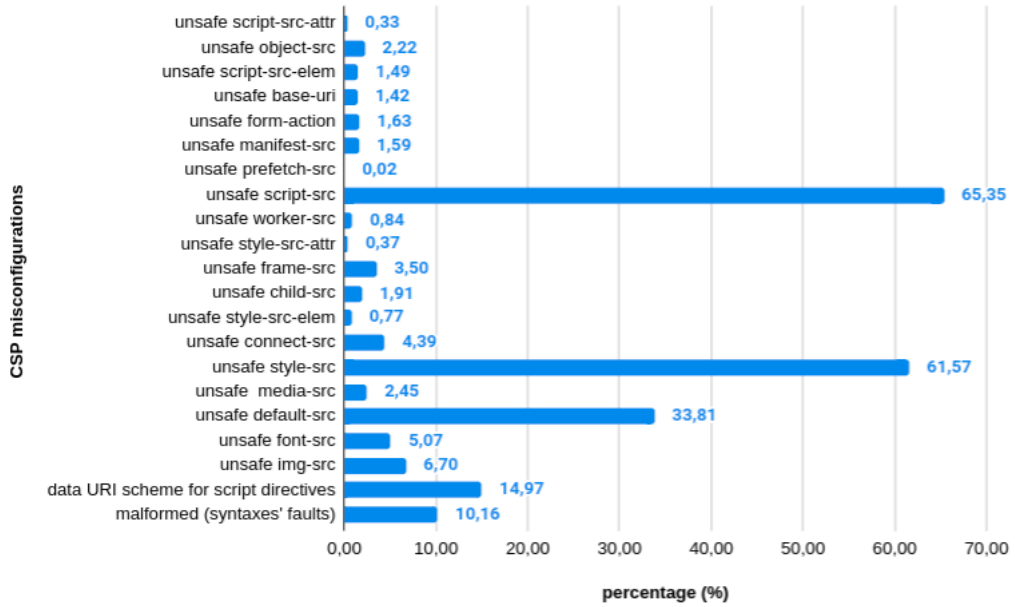

Figure 4.9: The percentage of each misconfiguration in the incorrect (non-"enabled") CSP policy

The CSP misconfigurations (see Figure 4.9) were also investigated. 15% of all the incorrect Content Security Policy (non-"enabled" policies) used the "data" source scheme with the combination of the "script" directive or "script-src-attr"

directive or even the "script-src-element" directive. 10% of the CSP misconfigurations used incorrect policy syntax (malformed). 20 domains used CSP directives from older CSP versions. Two of these domains (teamliquid.net and liquiddota.com) used the "disown-opener" old CSP directive and the others (e.g bnpparibasfortis.be, gls-group.eu, siemens.com.cn), the "reflected-xss" old CSP directive. The rest of misconfigurations referred to the directives, which did not set the "strict-dynamic" field and cryptographic nonce values, but they contained the fields "unsafe-eval" or "unsafe-inline". These cases did not completely protect the website, as they could not restrict the behaviour of the inline scripts. The "script-src" directive was one of the most unsafe directives which used this way observed up to 65% of the incorrect CSP. This happened because it is widely known that the websites have to allow the execution of inline JavaScript in order not to cause problems to their functionalities. The "style-src" directive was the second preference detected to 62%. This directive specifies valid sources for stylesheets in order to improve the appearance of the websites. So, this is another website functionality, which is commonly used by the developers without giving attention to the protection through the CSP. The "default-src" directive is an important directive that serves as a fallback for the absent CSP directives. 34% of the misconfigurations were used in this unsafe way. It was a small percentage as the user agents searched it in order to set default values for the absent directives which control the locations where certain resource types may be loaded (fetch directives).

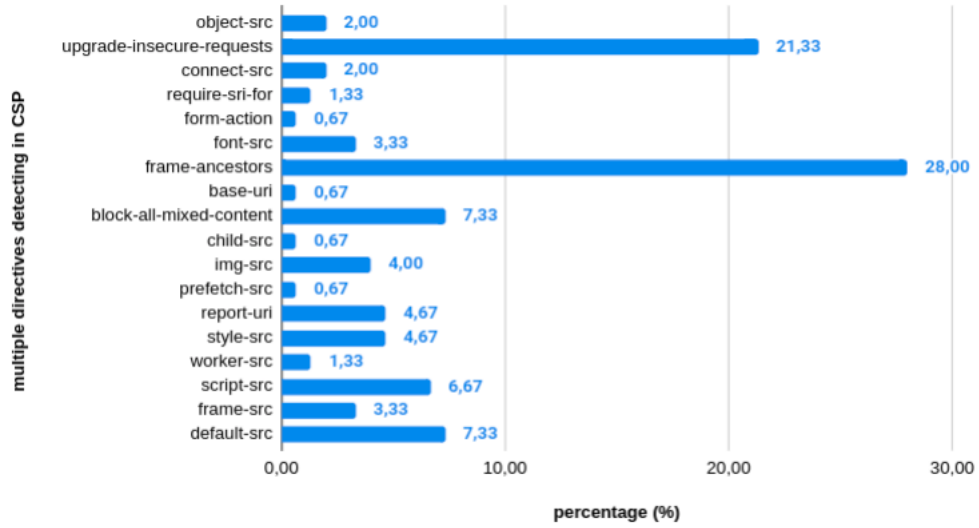

Figure 4.10: The percentage of each CSP directive appeared more than once in the policy's syntax

In Figure 4.10 the percentage of all the CSP directives, which were detected more than one time (multiple directives) in the policy syntax, are presented. Most of these directives were the "frame-ancestor" and the "upgrade-insecure-request".

The first one specifies valid parents that may embed a page using specific HTML tags such as frame, iframe. It was detected to 28% of the total multiple directives. The second one was detected to 21% and was used in order to rewrite URLs, which were insecure (served over HTTP). Then it was replaced by secure URLs (served over HTTPs). There was no reason to set multiple times this directive as it was a simple one without any additional fields.

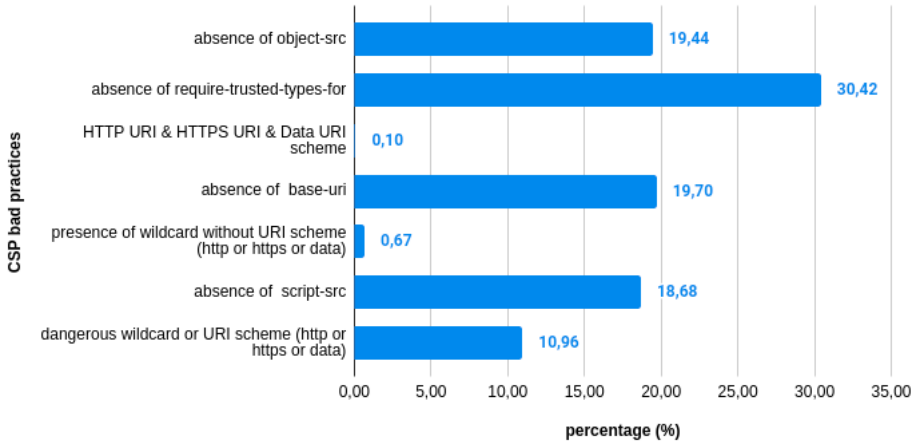

Figure 4.11: The percentage of each CSP bad practice in relation to the total number of them

Figure 4.11 describes the **bad practices** which were detected by our module. First of all, the absence of the "object-src", "base-uri" and "script-src" directives were approximately up to 19% of the total detecting bad practices. Especially for the "script-src" directive, it was important to be set in the CSP in order to restrict the script executions inside the website. The most common bad practice was the absence of the "require-trusted-types-for" directive which was detected in 31%. This directive could reduce the DOM XSS attacks by instructing the user agents to control the data passed to DOM XSS sink functions such as eval function. It was also noticed that only a few policies had set the wildcard to the whitelist without any URI schemes (http or https or data). This was a positive result for the websites' safety as it did not restrict the URIs or the URLs schemes when it had been set. Last but not least, it was the presence of the wildcard or URIs or URIs schemes which detected to 11%. This was another security risk for the websites. It means that the website's developer did not restrict these directives for specific either URIs or URIs schemes as a result to know the risk.

Figure 4.12 shows the CSP directives, which contained the "unsafe-inline" field allowing the use of inline resources. It was observed that the majority of these directives were the "style-src" and the "script-src" to 32% and 34% respectively. As a result the websites' safety was reduced. The "default-src" directive contained this dangerous field up to 17% of the total directives, which was another risk for the website security.

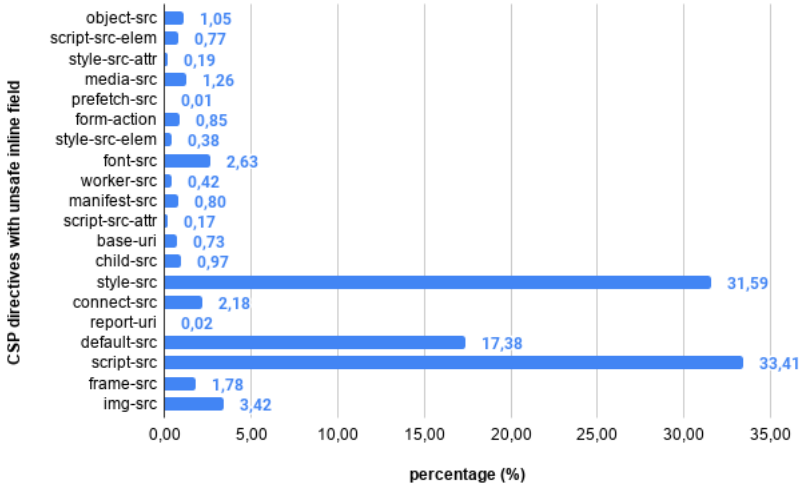

Figure 4.12: The percentage of each CSP directive, which contained the "unsafe-inline" field, in relation to the total number of them

4.4 CSP and X-XSS-Protection

The browsers can implement the X-XSS-Protection policy through the CSP. For this reason, they stopped to support the X-XSS-Protection. The only way for the developers to implement this policy is the support of a legacy browser in their websites.

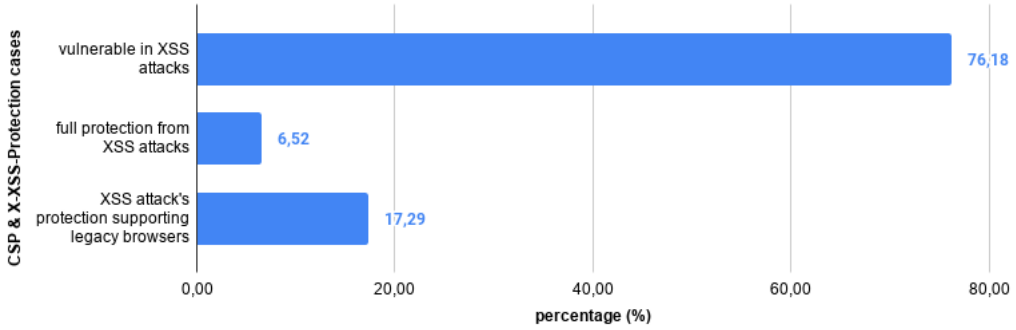

Figure 4.13: The percentage of each case, in which the websites were protected or not, in relation to the total number of domains using the CSP and X-XSS-Protection

In Figure 4.13 the percentages of the cases, where the websites were vulnerable or not in XSS attacks, is presented. These percentages refer to the total domains (100K) in which the analysis was conducted. 76% websites were detected to be vulnerable to XSS attacks. These cases referred to the websites which did not set

both of these policies or even did not implement them with a correct syntax. 7% of the websites was protected by the XSS attacks as they had implemented the CSP without allowing the execution of inline javascript. The rest (17%) referred to the cases in which they did not set the CSP or did not implement it correctly, but they applied the X-XSS-Protection policy correctly. Therefore, only the websites, which supported a legacy browser, could be safe in this case.

4.5 CSP and X-Frame-Options

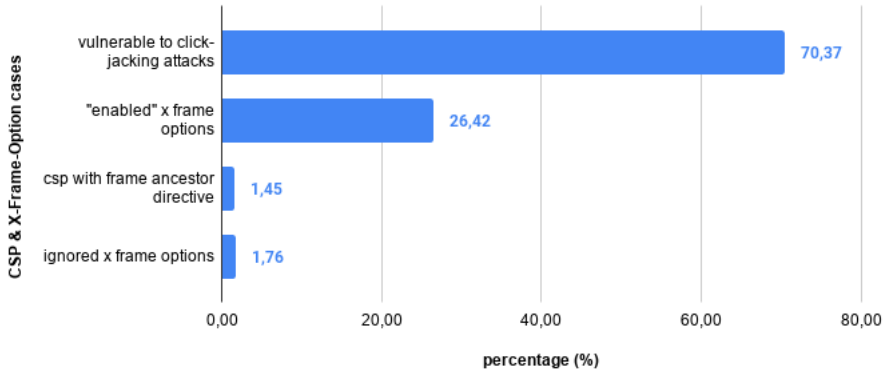

Figure 4.14: The percentage of each case, in which the websites were protected or not, in relation to the total number of domains using the CSP and X-Frame-Options

The browser can implement the X-Frame-Options policy though the CSP. It can be achieved only in the case where the website sets the "frame-ancestor" directive in the CSP. If a website implements both of the policies, then the browser will ignore the X-Frame-Options.

In Figure 4.14 it is observed that 70% of the total domains were vulnerable to click-jacking attacks as they did not implement at all or correctly any of these policies in order to avoid them. 26% of the domains had set the X-Frame-Options policy correctly and 1,5% had implemented the X-Frame-Options through the CSP, using the "frame-ancestors" directive. 1,8% of the domains had implemented both of these policies as a result the browser ignored the X-Frame-Options policy.

Chapter 5

Comparison XDriver with Selenium

This Chapter presents the results of the comparison between the XDriver and the Selenium framework. Through this analysis, it was investigated how many failures' cases were solved by the XDriver error handling mechanisms and how the Selenium behaved in these cases.

5.1 Experimental Orchestration

A web crawler was developed for the comparison of XDriver framework with the Selenium Project. Firstly, a manager ran the XDriver and the Selenium in parallel execution (see Figure 5.1). The Selenium instance created a Chrome browser instance with the same settings as the XDriver such as maximized browser window. Both of the frameworks had set the same time (timeout = 120seconds) in which the browser waited for the website to respond.

Each of the frameworks initialized a Chrome browser using appropriate functionalities. The "boot" functionality was used for the XDriver, which created the browser with preferences given by the manager. For the Selenium, some functionalities were used (ChromeOptions, add_arguments, driver.Chrome(args)) for the browser's initialization.

Afterwards, each of the frameworks visited a number of domains and performed the same specific tasks in order to investigate their behaviour on these tasks. Firstly, they searched web elements with path expression (xpath syntax) using the appropriate functionalities for each framework. It was achieved by using the "find_elements" functionality in the Selenium and the XDriver. The XDriver functionality had the difference that it made some processes in order to have functionable elements. It was accomplished by collecting the element's DOM path and simulating the "find_element_by_xpath" functionality in order to ensure that the elements existed in the website. During these processes, there were crucial points in which a failure could occur and the error handling mechanisms were used in

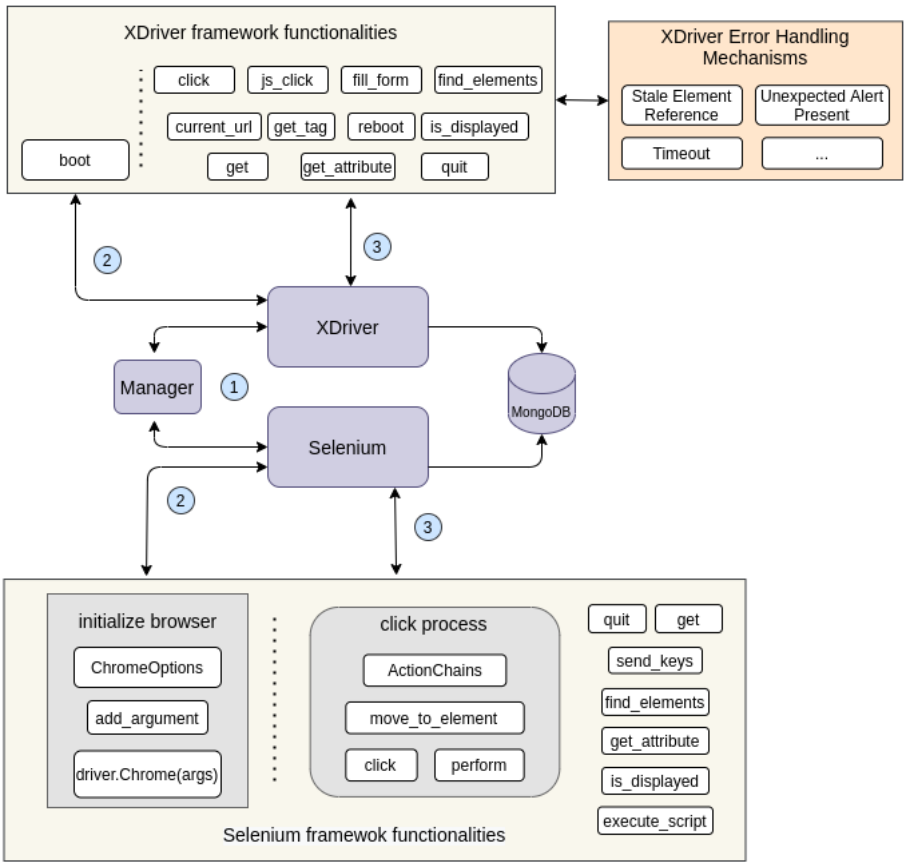

Figure 5.1: Design of the experiment

order to solve it.

Then, they collected the maximum number of available web elements according to the managers' parameters. In the case of invisible elements, the Selenium executed JavaScript using a specific functionality (execute_script) and the XDriver used the "js_click" functionality. "js_click" tried to trigger the element's "OnClick" or its childrens' "OnClick". This happened because there were cases in which the element that the user wanted to click, was not the actual clickable element but a parent one. Then, it had to click some of its children in order to perform the process. In the case of visible elements the process was simpler. The Selenium used specific functionalities which moved the cursor over the element in order to click it. The XDriver functionality (click) not only performed the same process as Selenium but also used the error handling mechanisms in the case of an exception.

After that, they checked if any website redirection was caused by the click action as there were web elements, which were redirecting to another internal page of the domain. When it happened, the crawler returned to the website, in which the collected web elements referred. This "click" task was repeated for the rest of

the collected web elements.

Subsequently, they continued to the second task. This task collected the form elements of the website in order to fill them out as a real user. The frameworks searched for form elements using the "find_elements" process which was analysed before. A form element (e.g signin, login) could contain many elements (e.g email, password, age) named as input elements. So, the frameworks tried to type each of these input elements. This action was carried out by using both the Selenium (send_keys) and the XDriver functionality (fill_form). These two functionalities have been analysed in detail in the Section 5.2.1.

The next task was to collect the script source filenames that the website contained. The frameworks performed this task using the "find_elements" functionality in order to search the available scripts as some of them did not have their codes in the website. In particular, the programmers might have their code in a private server from which the website executed them. Another reason could be that the third party scripts were executed by external libraries such as advertisement libraries. After that, the frameworks used the "get_attribute" functionality in order to collect the filenames which were contained in the script's source attribute.

Finally, the frameworks checked if the collected forms were enough for the domain in which all the above tasks were performed. If not, they had to collect five Hyperlinks of the landing page, searching for those that contained one of the specific words (sign-in, signup, login, password) in the text attribute of the link. In this way, the collected Hyperlinks might contain more form elements as many websites contained them with input elements referred to these words. Then, for each of these Hyperlinks the above tasks were repeated. Thus, more cases were investigated for the filling out form task.

While the web crawler was performing the above tasks, the results of them were stored in a Mongo database. These results contained useful information for the collected element such as web element ids, form element ids, input element ids, script filenames. They also involved the exceptions which were caused during these tasks as well as those which passed from the error handling mechanism to the XDriver framework.

5.2 Experimental Evaluation

The analysis was conducted for 20K alexa domains and approximately 80K Hyperlinks. Therefore, the tasks were executed to 100K URLs. The analysis was focused on specific Selenium exceptions (stale element reference, timeout, webdriver, unexpected alert present exceptions), in order to evaluate the XDriver error handling mechanisms. These exceptions are the most common Selenium limitations, which the researchers have to solve during a study.

For the purpose of a fair comparison between the two frameworks (XDriver, Selenium), the below results and observations referred not only to the same domains, but also to the same number of them. There were domains which did not respond

to the browser requests, because they were not available (HTTP 404 Error). But there were domains that only one of the frameworks did not visit due to client or server errors. These errors existed at the time that the browser of the frameworks' tasks requested these domains. Therefore, they were not included in the analysis' results.

5.2.1 Results of crawler tasks and exceptions

Table 5.1: The number of elements performed by each XDriver task, when there were either no exceptions or exceptions solved by XDriver mechanisms.

XDriver Tasks	without exception	exceptions solved by XDriver mechanisms	number of elements
clicked web elements	124K (22,80%)	246K (45,05%)	545K
got script filenames	141K (66,70%)	687 (0,03%)	210K
searched form elements	173K (99,90%)	112 (0,06%)	170K
typed input elements	322K (99,77%)	739 (0,23%)	323K

Table 5.2: The number of elements performed by each Selenium task, when there were no exceptions.

Selenium Tasks	without exception	number of elements
clicked web elements	157K (32,33%)	487K
got script filenames	1256K (66,96%)	1876K
searched form elements	155K (99,69%)	156K
typed input elements	88K (30,37%)	289K

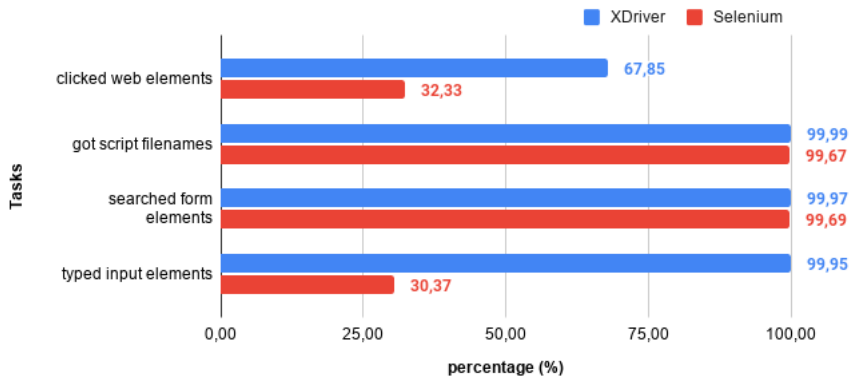

Figure 5.2: The percentage of each framework's task correctly performed

Figure 5.2 presents the tasks' results for each framework. For each task, the percentage of the properly executed cases according to the total amount of the elements is shown. The failed cases referred to the stale element reference exceptions. This happened, because the action of these tasks were selected in order to evaluate this kind of Selenium exceptions. The rest of exceptions will be analysed in Section 5.4. Table 5.1 presents the analytically XDriver tasks' results and Table 5.2 the Selenium ones.

For the first task (clicked web elements), 68% click actions were performed by the XDriver. 23% of them were executed without failure and 45% of the occurred exceptions were solved by the XDriver error handling mechanisms (see Table 5.1). However, the Selenium performed 32% of the click actions properly. On the one side, there was a difference between the task's percentages (∼35%) due to the useful solutions of the XDriver stale element reference mechanism. This mechanism searched the problematic elements according to their ID attributes in the pages' HTML DOM. The mechanism used this attribute, as it was difficult to change it when a page was reloaded, comparing it with other attributes (e.g xpath, css). If the element was not contained in the page, the mechanism could not find a solution. However, if it existed, the mechanism would re-fetch the element and would update the old webelement's reference in order to have in hand a functionable web element. On the other side, the XDriver functionalities, which were used in order to execute this task, helped the click actions as it was mentioned in Section 5.1. Therefore, the XDriver executed two times more click web element actions properly, due to the useful solutions of the stale element reference mechanism and the further processes in the XDriver functionalities, although the XDriver collected 7K more web elements than the Selenium (see Tables 5.1,5.2)

For the second task (got script filenames), it was observed that both of the frameworks collected approximately 100% of the script filenames. The frameworks collected properly 66% of them and the rest (33%) was detected without the source attribute. These scripts did not have the filenames available on the website because they might have existed in a private server or might have been executed by an external library. Therefore, there was no difference between the two frameworks because this task was simple. It also searched for the available scripts in the website using specific functionality as it was analysed before (Section 5.1). As a result, the task did not have many functionalities to perform.

For the third task (searched form elements and typed input elements), both of the frameworks searched forms' elements and collected inputs' elements in approximately 100% without any exception. The first part of the task searched only elements using the functionality (find_elements) as it was analysed in Section 5.1. That is why there was no difference between the two frameworks. The second part referred to filling out the forms, by typing each input's element of them. The XDriver typed properly 99,95% of the input elements and the Selenium 30,4%. This huge difference could not be explained by the exceptions, which were solved by the XDriver mechanisms, as they were only 0,2%. This happened due to the XDriver functionality (fill_form) which typed the elements according to its type.

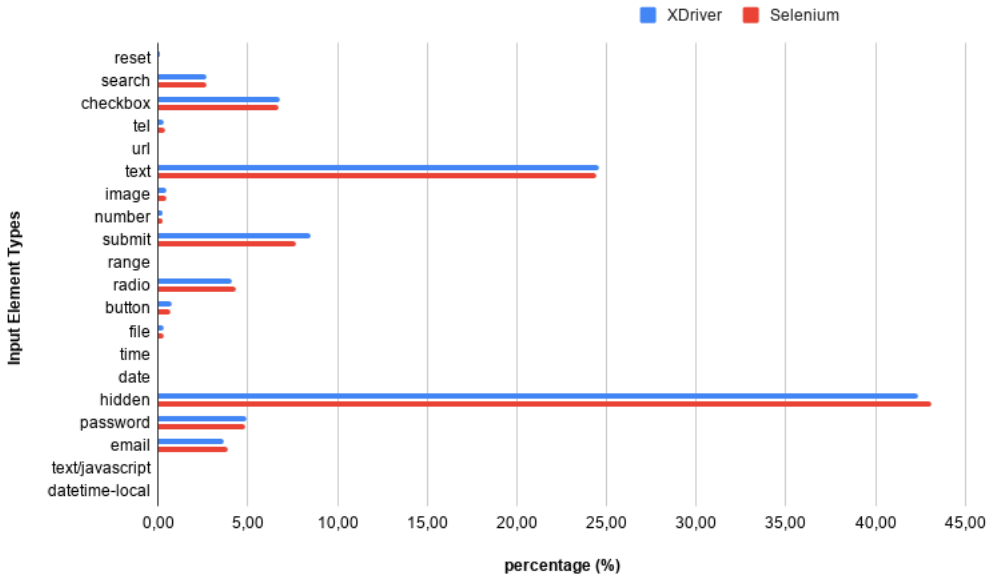

Figure 5.3: The percentage of input elements' types collected by each framework

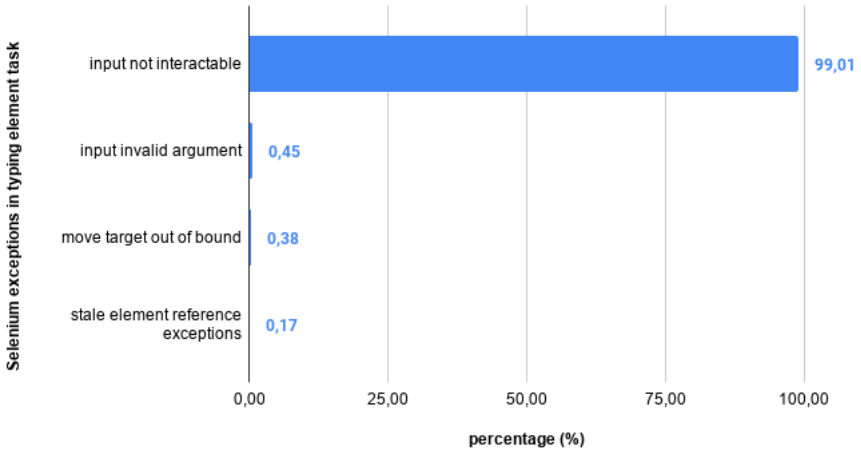

Figure 5.4: The percentage of exceptions caused during the input element task in the Selenium.

Figure 5.3 presents the elements' types, collected by each framework during this task in order the variety of them to be observed. All the elements could not be typed only by using the basic Selenium functionality (send_keys) such as file elements. The exceptions, which occurred in these cases for the Selenium, are presented in Figure 5.4. The non-interactable elements referred to non-clickable or invisible elements. This means that these elements either had to be clicked in

order to be displayed or could not be clicked at all. Another problem was the move target out of bound exceptions. Specifically, the webview had to be moved over the elements, which would be typed, using the scroll functionality. The invalid argument exceptions were caused when more processes had to be performed before the elements were typed. This means that it could not be used in the same way to type all the input elements, but it had to be done according to their types. For example, if the XDriver had to type a text input element, then the functionality would check the input's length and would use the "send_keys" Selenium functionality setting an appropriate text value. If the element needed a time value, the functionality would fill it out with a static fake object in order to complete the process. Therefore, the users had no need to insert more code in order to perform the process of filling out input elements using the XDriver functionalities. These tasks explained why the XDriver was more useful than the Selenium, as it contained the further processes which were needed for specific functionalities.

5.2.2 Results of other exceptions

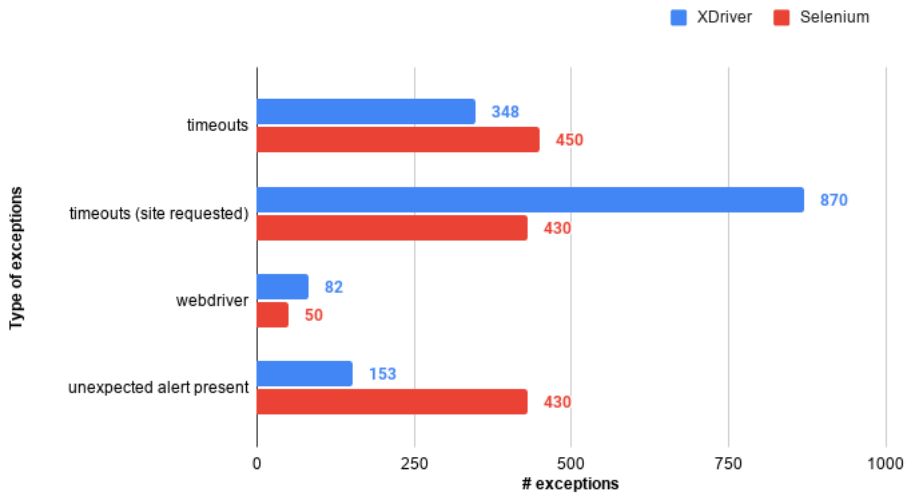

Figure 5.5: The number of other exceptions during the analysis of each framework.

Figure 5.5 presents other exceptions, which were caused during the execution of all the tasks. Firstly, they were referring to timeouts which could be caused if a domain did not respond (timeouts in site requested). For these cases it had been set by 120 seconds as the max time that the browser waited for the domain to respond. It was observed that there were 870 timeouts (timeouts in site requested) in the XDriver and 430 in the Selenium. In this case the XDriver timeout handling mechanism rebooted the browser and returned to the last called functionality with a fresh browser instance. In the case of other timeouts, in which the browser's driver waits for some processes to be executed (e.g searching a web element which

is not immediately present), there were 348 exceptions in the XDriver and 450 in the Selenium. The XDriver timeout handling mechanism rebooted the browser and requested again the problematic domain. If a timeout exception was caused again, it would repeat this process ten times. When the problem was not solved, then the mechanism returned to the last functionality with a boolean value instead of raising the exception.

The webdriver exceptions were another common Selenium problem. It was observed that 82 webdriver exceptions were caused in the XDriver and 50 in the Selenium. In this case the XDriver timeout handling mechanism detected that the exception was a webdriver and rebooted the browser in order to avoid it. It is important to note that 45 webdriver exceptions returned a boolean value in the crawler side (XDriver) as the XDriver mechanisms did not solve them. It means that this XDriver error handling mechanism did not give a useful solution for them and the rest of exceptions (37) were solved by the XDriver mechanism.

Another exception was the unexpected alert present, which was induced when popups or alerts appeared after actions in elements such as fill out forms and click web elements. There were 153 of these exceptions in the XDriver and 430 in the Selenium. For these exceptions the XDriver handling mechanism tried to solve the problem by clicking the cancelling button of the alerts and preventing the page from popping more alerts using the "alert dismiss" Selenium functionality. No alert present exceptions could be caused either by synchronizations' issues, when the pages were loaded, or by the alerts which were not "checked", although the checked operation had been performed. For these kinds of exceptions, the XDriver mechanism would reboot the browser and continue the functionality, from which the exception was caused.

The XDriver error handling mechanisms gave useful solutions to the above exceptions except from 3% of them (45 webdriver exceptions). Therefore, there were only 45 tasks' failures in the XDriver in contrast to the Selenium with 1360.

Chapter 6

Future Work

The SecurityAuditor module had the need of better proxy functionality. This means that many problems existed in the proxy side. It would be a good improvement for the XDriver to find a way to catch the proxies' problems and try to solve them. But it is not simple because a proxy runs out of the XDriver scope. Therefore, if the proxy was executed inside the XDriver scope, a basic solution would be the proxy's reboot in a way to keep the state in which the task would fail. This could also help the analysis which has the need of the redirection flow of the websites.

A future work for the SecurityAuditor module would be to evaluate the CSP when there are multiple times in the same response header. The module evaluates only the first policy header in the case of multiple policy headers but the browser enforces them independently each one. The combination of the multiple CSP headers in one, could be a solution in order to be evaluated and give all the information to the user in this case.

It would also be a good idea to conduct an analysis which would perform actions using other XDriver functionalities and evaluate all the XDriver error handling mechanisms. It would help to improve these mechanisms, as a result there would be even an improved framework which could be extensively used by the community.

Chapter 7

Conclusion

In this master thesis, our initial effort was focused on the implementation of the SecurityAuditor module evaluating the Security Header policies. These policies protected the websites from different kinds of attacks and were implemented by the websites' developers. Therefore, a large-scale analysis was conducted in order to evaluate our module. This analysis investigated how many of the Security Header policies were applied by the websites as well as the mistakes and the misconfigurations which were detected in the policies' syntax.

In conclusion most websites did not apply the policies. The X-Frame-Options was detected to 30% of 100K domains as the most preferred policy and the Feature-Policy to 1% as the least preferred one. The HSTS was detected to 29% and this percentage was small because this policy is very important. Another essential policy is the CSP detected to 11%. This happened due to its complexity which led the users not to include it in the websites' headers. There were also syntax errors and vulnerabilities in the policies' syntaxes both in the HSTS and CSP, as there were 60% and 40% incorrect policies' syntaxes, respectively which led to unsafe websites.

The comparison of the XDriver framework with the Selenium was another analysis which was conducted in this thesis. In this way, the effectiveness of the XDriver error handling mechanisms' solutions was investigated. The XDriver clicked 30% more web elements than the Selenium because of the solutions given by the XDriver error handling mechanisms. Moreover, the XDriver filled out all the forms successfully due to its functionality, which filled out forms according to their elements' types. On the other hand the Selenium filled out forms only to 30%. Therefore, the XDriver performed more tasks because of the XDriver error handling mechanisms as well as functionalities which made some necessary processes before the execution of basic Selenium functionalities.

Bibliography

[1] Content Security Policy (CSP) Documentation. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy.

[2] CORSTest tool. https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html.

[3] Cross-Origin Resource Sharing (CORS) Documentation. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[4] CSP Evaluator Tool. https://csp-evaluator.withgoogle.com.

[5] Cucumber framework using selenium and java. https://github.com/vinodkrane/cucumber-testing-framework-using-selenium-java-maven.

[6] Cucumber with java. https://github.com/selenium-cucumber/selenium-cucumber-java.

[7] "Cucumber&Puppeteer".https://github.com/patheard/cucumber-puppeteer.

[8] "Cucumber&Selenium".https://github.com/waheedahmed55/Hybrid-Selenium-Cucumber-BDD.

[9] Expect-CT Policy Documentation. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Expect-CT.

[10] Feature Policy Documentation. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Feature-Policy.

[11] Galen framework. http://galenframework.com/docs/about/.

[12] Galen framework in Github. https://github.com/galenframework/galen.

[13] Header Analyser tool. https://securityheaders.com/.

[14] HSTS Preload Lists. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/strict-Transport-Security#preloading_strict_transport_security.

[15] HstsPreload Python Package. https://pypi.org/project/hstspreload.

[16] HTTP Security Headers check tool. https://www.serpworx.com/check-security-headers/.

[17] HTTP Strict Transport Security (HSTS) documentation. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security.

[18] "Java Selenium WebDriver". https://github.com/shirishk/java-selenium-testng-automation-framework.

[19] OpenTest framework. https://getopentest.org/docs/architecture.html.

[20] OpenWPM in Github. https://github.com/mozilla/OpenWPM.

[21] OpenWPM in Gitlab. https://gitlab.com/bosi/ba-openwpm.

[22] Puppeteer. https://pptr.dev/.

[23] Puppeteer in Github. https://github.com/puppeteer/puppeteer.

[24] Referrer Policy Documentation. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy.

[25] Robot framework. https://robotframework.org/#libraries.

[26] Robot framework in Github. https://github.com/robotframework/SeleniumLibrary/.

[27] Robot Framework with Puppeteer. https://qahive.github.io/robotframework-puppeteer.github.io/.

[28] Robot Framework with Puppeteer in Github. https://github.com/qahive/robotframework-puppeteer.

[29] Selenium in Java. https://github.com/edinc/java-selenium-framework.

[30] Selenium Project. https://www.selenium.dev.

[31] Selenium Project in Github. https://github.com/SeleniumHQ/.

[32] "Selenium Wrapper in Java". https://github.com/wasiqb/coteafs-selenium.

[33] "SeleniumCucumber". https://github.com/rahulrathore44/SeleniumCucumber.

[34] Serenity framework. http://www.thucydides.info/#/.

[35] Serenity framework in Github. https://github.com/serenity-bdd/serenity-core.

[36] Serenity framework in Java. `https://github.com/serenity-bdd/serenity-junit-starter`.

[37] W3C Content Security Policy 1.0. `https://www.w3.org/TR/2012/CR-CSP-20121115`.

[38] W3C Content Security Policy 2. `https://www.w3.org/TR/CSP2`.

[39] W3C Content Security Policy 3. `https://w3c.github.io/webappsec-csp`.

[40] WebDriverIO framework. `https://webdriver.io/`.

[41] WebDriverIO framework in Github. `https://github.com/webdriverio/webdriverio`.

[42] X-Content-Type-Options Documentation. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Options`.

[43] X-Frame-Options Documentation. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options`.

[44] X-XSS-Protection Documentation. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection`.

[45] XDriver framework in Gitlab. https://gitlab.com/kostasdrk/xdriver-open.

[46] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content security problems? evaluating the effectiveness of content security policy in the wild. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1365–1375, New York, NY, USA, 2016. Association for Computing Machinery.

[47] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. CCSP: Controlled relaxation of content security policies by runtime policy composition. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 695–712, Vancouver, BC, August 2017. USENIX Association.

[48] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We still don't have secure cross-domain requests: an empirical study of CORS. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1079–1093, Baltimore, MD, August 2018. USENIX Association.

[49] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 195–1970, New York, NY, USA, 2020. Association for Computing Machinery.

[50] S. Englehardt, C. Eubank, P. Zimmerman, Dillon Reisman, and A. Narayanan. Openwpm : An automated platform for web privacy measurement. 2016.

[51] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1388–1401, New York, NY, USA, 2016. Association for Computing Machinery.

[52] M. Fazzini, P. Saxena, and A. Orso. Autocsp: Automatically retrofitting csp to web applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 336–346, May 2015.

[53] Lucas Garron, A. Bortz, and D. Boneh. The state of hsts deployment: A survey and common pitfalls. 2014.

[54] Daniel Hausknecht, Jonas Magazinius, and Andrei Sabelfeld. May i? - content security policy endorsement for browser extensions. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, DIMVA 2015, pages 261–281, Berlin, Heidelberg, 2015. Springer-Verlag.

[55] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. Browseraudit: automated testing of browser security features. pages 37–47, 07 2015.

[56] I.Dolnak. Implementation of referrer policy in order to control http referer header privacy. In *2017 15th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 1–4, Oct 2017.

[57] Martin Johns. Script-templates for the content security policy. *Journal of Information Security and Applications*, 19, 07 2014.

[58] Christoph Kerschbaumer, Sid Stamm, and Stefan Brunthaler. Injecting csp for fun and security. pages 15–25, 01 2016.

[59] Michael Kranch and J. Bonneau. Upgrading https in mid-air: An empirical study of strict transport security and key pinning. In *NDSS*, 2015.

[60] Sebastian Lekies, Krzysztof Kotowicz, Samuel Grob, Eduardo Vela Nava, and Martin Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. 2017.

[61] Xurong Li, Chunming Wu, Shouling Ji, Qinchen Gu, and Raheem Beyah. Hsts measurement and an enhanced stripping attack against https. pages 489–509, 01 2018.

[62] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. Cspautogen: Black-box enforcement of content security policy upon

real-world websites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 653–665, New York, NY, USA, 2016. Association for Computing Machinery.

[63] Kailas Patil and Braun Frederik. A measurement study of the content security policy on real-world applications. *Int. J. Netw. Secur.*, 18:383–392, 2016.

[64] Sebastian Roth, Timothy Barron, S. Calzavara, N. Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. In *NDSS*, 2020.

[65] Sergio Santos, Carmen Torrano, Yaiza Rubio, and Felix Brezo. Implementation state of hsts and hpkp in both browsers and servers. pages 192–207, 11 2016.

[66] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-origin policy: Evaluation in modern browsers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 713–727, Vancouver, BC, August 2017. USENIX Association.

[67] S. Sivakorn, I. Polakis, and A. D. Keromytis. The cracked cookie jar: Http cookie hijacking and the exposure of private information. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 724–742, May 2016.

[68] Suphannee Sivakorn, Angelos D. Keromytis, and Jason Polakis. That's the way the cookie crumbles: Evaluating https enforcing mechanisms. In *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, WPES '16, pages 71–81, New York, NY, USA, 2016. Association for Computing Machinery.

[69] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Data exfiltration in the face of csp. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 853–864, New York, NY, USA, 2016. Association for Computing Machinery.

[70] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, 2016.

[71] Michael Weissbacher, Tobias Lauinger, and William Robertson. Why is csp failing? trends and challenges in csp adoption. volume 8688, pages 212–233, 09 2014.