USING GUI TESTING TO AUTOMATE WEBSITE SECURITY ANALYSIS

by

Jeroen Hoebert

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering
to be defended publicly on Monday 05 September 2022, at 15:00.

Student number:	852248694	
Course code:	IM9906	
Thesis committee:	dr. ir. Hugo Jonker (chairman),	Open University
	dr. Pekka Aho (supervisor),	Open University

Contents

ABSTRACT

This thesis examines the added value of GUI testing to the field of security analysis. Because in an increasingly digital world, software security is more important than ever. It is important to find those vulnerabilities before they are exploited. Unfortunately, using dedicated tooling to search for security vulnerabilities is not always within reach of the software developers. That is why implementing security analysis in GUI testing would enable developers to find vulnerabilities that otherwise would have gone unnoticed.

To enable security analysis, the GUI testing tool TESTAR was extended with the ability to detect SQL injection and XSS vulnerabilities, HTTPheader misconfigurations and session token invalidation. The performance of TESTAR's security analysis was measured by running it against the OWASP benchmark. Code coverage was measured in a real world application using OpenCover. These results were compared to those of the dedicated security analysis tool OWASP ZAP. Token validation analysis was tested in a synthetic scenario created for this research, no comparison was made to dedicated tooling because none of them supported this scenario. Showing that TESTAR is able to do types of security analysis that are not feasible with dedicated security analysis tooling.

TESTAR was not able to beat the dedicated tools, but was able to match their performance for misconfigurations and performed sufficiently in others. The biggest limitation of this approach was the lack of attack services, because TESTAR is limited to the GUI. This research showed, that GUI testing is able to deliver added value to security analysis. In particular for applications that are analyzed with dedicated tooling at a late stage in their development or not analyzed at all.

1

INTRODUCTION

In an increasingly digital world, security vulnerabilities are more important than ever. Finding and patching these vulnerabilities before someone takes advantage of them is a critical part of keeping the web safe. Finding those vulnerabilities however, requires time and effort that could be spent elsewhere. This means finding vulnerabilities has a resource budget. Automated tooling reduces the resources that need to be spend on finding these vulnerabilities. That is why automated tooling for finding those vulnerabilities can increase the security of the web drastically.

However, even with automated tooling, there are a lot of websites out there with security vulnerabilities. This is at least partially because security tools are not accessible enough for software developers [SDM20]. Another reason could be the lack of priority for security analysis in organizations.

GUI testing is a way of testing that uses the same graphical user interface (GUI) a human user would use to test an application. When an automated GUI testing tool is used, the tool will click through the application looking for bugs or expected behavior. In this document, we propose a research that explores the option of integrating security analysis in automated GUI testing.

What makes this combination interesting is that a GUI tester interacts with a system in a way a human would. Meaning that a GUI tester interacts with the system in the way the system was intended to be interacted with. Making it a lot easier to test for certain vulnerabilities, ones requiring scenario's navigating though the application for example. This enables a GUI tester to do kinds of security analysis that are not feasible with dedicated security analysis tooling. Like analysing whether a session token is invalidated after logout.

If this type of analysis was sufficient it would also be possible to do the same analysis during normal browsing. A possible future for this type of analysis could be a security analysis extension in a browser that warns the user when a website has vulnerabilities. Another possibility is the use in the GUI testing tools themselves, making security analysis more accessible for software testers.

This research is aimed at answering the question: "What can GUI testing contribute to the field of security analysis?". To answer this question, two hypothesis are tested. The first, "A GUI tester is able to do security analysis", and the second: "Security analysis with a GUI tester contributes to the safety of the web". The first hypothesis can be proven by enabling a GUI testing tool to do security analysis. This is something that should be possible because a

GUI tester interacts with the system in much the same way a pen-tester would. The second hypothesis can be answered in two ways. Either a GUI tester is easier to use and enabling security analysis using a GUI tester would bring security analysis to a larger audience. Or a GUI tester is better in certain types of security analysis. For example security vulnerabilities that require multiple UI (User Interface) steps to run into.

In regular GUI testing, a script is used to guide the testing tool through the system. A resent trend is scriptless GUI testing, where the testing tool finds its own way through the system. TESTAR is an open source tool that does scriptless GUI testing, TESTAR is one of the most comprehensive tools within this category according to Vos et al. [VAR+21]. That is why TESTAR is the tool that will be used for this research.

In this research some background knowledge is given about the GUI testing tool TESTAR, the web interface WebDriver, and the most common web vulnerabilities according to the OWASP top 10. After that, some related work will be discussed. This includes research into automated testing, the vulnerabilities themselves and similar projects from the past. Then follows the chapter domain analysis, where the different vulnerabilities are analysed to select a set that could be implemented in TESTAR. After that, the research itself will be discussed including the algorithms for vulnerability detection that will be implemented in TESTAR. As well as a way of validating those abilities using a benchmark. The chapter is followed by Design and Development, where some of the development chooses will be elaborated. This chapter goes deeper into the development of the framework used to test vulnerabilities with TESTAR. After that the results of the validation will be analysed and the research question will be answered. A summary of the results of this research is given in the conclusion. Finally the thesis ends with a future work chapter.

2

BACKGROUND

2.1. LIST OF ABBREVIATIONS

In this thesis, several domain specific abbreviations were used. That is why a list of those abbreviations is included below 2.1.

Abbreviation	Meaning
CDP	Chrome DevTools Protocol
CIL	Common Intermediate Language
CSP	Content-Security-Policy (header)
DAST	Dynamic Application Security Testing
GUI	Graphical User Interface
IAST	Interactive Application Security Testing
SAST	Static Application Security Testing
SQL	Structured Query Language
SSRF	Server Side Request Forgery
SUT	System Under Testing
UI	User Interface
XSS	Cross Site Scripting

Table 2.1: List of abbreviations

2.2. PROCESSES

2.2.1. GUI TESTING

GUI (Graphical User Interface) testing is a way of testing certain software by inspecting the GUI of the Software. GUI testing is aimed at making sure the functionalities of software application behaves as intended. A user won´t see the source code when visiting a website, instead it sees a graphical representation of what the code implies, a GUI tester will see the same [1]. Because a GUI tester works with the same end-result as the user, it can

[1] https://www.guru99.com/gui-testing.html

detect a niche of errors no other tool can. An example of this would be unexpected render behaviour, because this is only detectable when looking at the end result of the render.

There are two kinds of GUI testing, scripted GUI testing and monkey testing. With scripted GUI testing, the user defines a path through the application that the GUI tester will follow. A disadvantage of this technique is that once the application changes, the path needs to be re-established. With monkey testing, this path is not specified in advance. The GUI testing tool will instead chose its own navigational path trough the application. Advantages of this approach are that the tool only needs limited configuration and there is no testing script to update if the application changes. Another advantage is that a monkey tester will test everywhere on an application, not only the parts someone deemed necessary to test.

2.2.2. SECURITY ANALYSIS

Software security analysis is a process with the purpose of determining the security state of software. This can be done by hand, for example with pen-testing or with automated tooling. In this thesis, the focus is on finding vulnerabilities in web applications with automated tooling.

Two major types of tools for finding security vulnerabilities are Static Application Security Testing (SAST) tools and Dynamic Application Security Testing (DAST) tools. SAST tools are white-box tools and use the source code of a system for security analysis. DAST tools on the other hand, implement a black-box approach. They interact with a system to find security vulnerabilities and have no need for knowledge of the source code. Both SAST and DAST tools have their pros and cons, and it depends on the situation and what you are trying to find which is better.

For our research the focus is on DAST tools. GUI testers and DAST tools both interact with the system from the outside, without knowledge of the internals. Because of this, it makes sense to compare the security analysis potential of GUI testing tools to that of DAST tools and leave SAST tools out of scope.

2.3. TOOLS

2.3.1. TESTAR

TESTAR is a tool which has been designed to shift the understanding of GUI testing. The goal is to "go from developing scripts to developing intelligent AI-enabled agents". TESTAR allows the scriptless automated GUI testing of desktop, web and mobile applications [2].

From 2010 till 2013 the project FITTEST ran. FITTEST stands for Future Internet Testing and was a European project. The project´s goal was to "address these testing challenges, by developing an integrated environment for automated testing, which can monitor the Future Internet." [3]. After the project ended in 2013, TESTAR was continued by the Universidad Politecnica de Valencia, Utrecht University and the Open University The Netherlands for further development. The program was funded through various national and European enterprises.

TESTAR works by running in a continuous loop, called the 'inner loop'. This inner loop is started when TESTAR is ready to start a testing sequence. The inner loop starts by getting

[2] https://testar.org/about/
[3] http://crest.cs.ucl.ac.uk/fittest/project.html

the state, this detects all the available widgets on a website. After that TESTAR will derive the actions that can be executed based on the widgets found. Then TESTAR will select an action to execute and execute that action, the way TESTAR selects the action depends on the protocol used. The protocol describes how TESTAR should behave. TESTAR has protocols that are completely random, protocols that are machine learning based, and protocols that are somewhere in between. The last step in the inner loop is getting the verdict. This is where the oracles reside in TESTAR. All the information gathered within the iteration of the loop can be processed and a verdict can be given. After getting the verdict, a new iteration will start [VAR+21]. This process continues until the predefined end of the sequence is reached. The inner loop is included in Fugure 2.1.

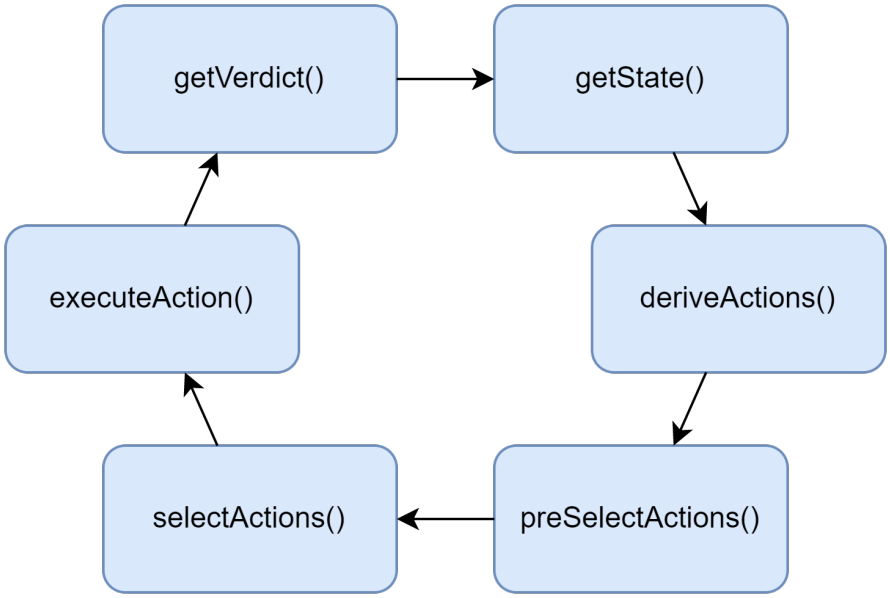

Figure 2.1: Inner loop of TESTAR

2.3.2. OWASP ZAP

OWASP ZAP is a DAST tool developed by the OWASP foundation, it is free and open source. ZAP was started in 2010 and is still actively developed. The motivation for ZAP was to deliver a simple pen-testing tool to developers to detect vulnerabilities in an early stage [OWA].

The testing process for OWASP ZAP is devided into two stage, the 'Spider' and the 'Active Scan'. The Spider is the crawler that maps the application. After running the spider, the Active Scan can be executed. This Active Scan interacts with the system and can analyse SQL injection and XSS vulnerabilities for example. To do the active scan, a section of the crawled application needs to be selected.

2.3.3. OWASP BENCHMARK

The OWASP Benchmark project is a benchmark made by the OWASP foundation for validation of security analysis tools. The benchmark exists of a web application written in Java, that has deliberate vulnerabilities build in, and a result generator to generate a score based on the output of the benchmark.

The benchmark contains 2.740 test cases, divided over eleven vulnerability categories. These test cases are pages in the web application. The pages are grouped by category, this makes it possible to only run the benchmark for certain vulnerabilities. These test cases are either vulnerable or not. By comparing the results of the security analysis tool, with the expected results of the benchmark, a score can be tied to the benchmark [Owa16]. This comparison can be done by the result generator for tools supported by the benchmark. If the benchmark is used with unsupported tools like TESTAR, the results need to be generated with custom code.

2.3.4. SELENIUM WEBDRIVER

Selenium WebDriver is a tool for browser automation. This enables an application to interact with a browser natively [4]. That is important for this research because TESTAR uses WebDriver to interact with the SUT if the SUT is a website. Because WebDriver acts like an interface between TESTAR and the browser, TESTAR is dependent on WebDriver for information all about the SUT. The browsers supported by WebDriver are: 'Chromium/Chrome', 'Firefox', 'Edge', 'Internet Explorer' and 'Safari'. This wide range of support makes it possible to automate GUI testing on multiple platforms. The interaction between a GUI tester and a website using WebDriver is depicted in Figure 2.2.

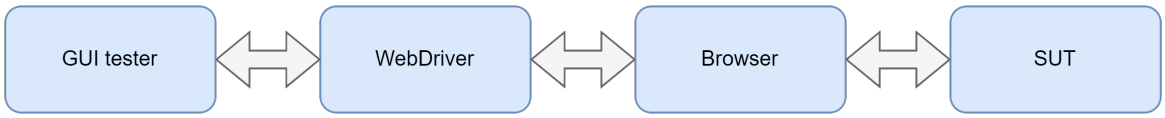

Figure 2.2: WebDriver chain of command

WebDriver also enables the application to read and manipulate things which are normally only available to the browser. This access is provided by the Chrome DevTools Protocol (CDP) [5]. Unlike the name suggests, CDP is available in Chrome, Edge and Firefox. CDP provides access to a set of tools developers can normally use in the browser. Like insights in the network traffic, cookies and local storage. This access makes it possible to use WebDriver for security analysis.

2.3.5. OPENCOVER

OpenCover is a tool for measuring code coverage in .NET applications [6]. While code coverage is often related to unit testing, it can be interesting to measure code coverage outside of this scope. OpenCover offers the option to measure code coverage while the application is in use. This enables OpenCover to measure the code coverage while manually testing, acceptation testing or in our case, security analysis. OpenCover was the only tool found that has this ability for .NET applications and is free.

OpenCover works by modifying the Common Intermediate Language (CIL) of the SUT [7]. This is the language .NET applications get compiled to. OpenCover adds its own CIL instruction to the compiled SUT code. These instructions are able to detect execution. This

[4] https://www.selenium.dev/documentation/webdriver
[5] https://www.selenium.dev/documentation/webdriver/bidirectional/chrome_devtools/
[6] https://github.com/OpenCover/opencover.
[7] https://blog.many-monkeys.com/open_cover_first_beta_release/.

way OpenCover is able to detect what part of the SUT is executed. One of the biggest advantages of this technique is that is does not require setup in the source code of the SUT. This is because OpenCover is able to run on an already compiled version of the SUT.

2.4. HTTP HEADERS

HTTP headers are a way of communicating additional data in an HTTP request or response. These headers can influence the behaviour of the server and client. Some headers can even have security implications if not set correctly. For this research, those are the headers of relevance. Some of these headers are listed in the table 2.2 below. Followed by an explanation for each of the headers in the table. This list of security related headers is not complete, but it contains a large enough variety of headers to prove the concept of detecting HTTP headers reliably.

Header	Prevents
Strict-Transport-Security	Header is present
X-Content-Type-Options	Contains 'nosniff' flag
X-Frame-Options	Header is present
X-XSS-Protection	Conaints '1; mode=block' flag
Set-Cookie	Contains 'secure' flag if header is present

Table 2.2: Security headers

2.4.1. STRICT-TRANSPORT-SECURITY

Visiting a website without an encrypted connection is undesirable because it enables third parties to intercept the connection using a man in the middle attack. This risk exists every time a website is visited over HTTP, even is the website immediately redirects to HTTPS. That is were the Strict-Transport-Security header comes in to play. This header tells the browser to only connect using HTTPS. The browser stores this information, and the next time the user navigates to the HTTP version of the website, the browser will connect to the HTTPS version instead [8].

2.4.2. X-CONTENT-TYPE-OPTIONS

The X-Content-Type-Options header tells the browser what to do if the Content-Type header is missing. The Content-Type header tells the browser how to handle the content, an image for example should be displayed, not executed [9].

If the Content-Type header is missing, the browser will use MIME Type Sniffing, evaluating what needs to be done to properly display the content. This becomes a problem when the image was injected with code in its metadata, the browser could think that the best thing to do is execute the metadata instead of displaying the image.

To avoid this behaviour and make sure the browser will not use MIME Type Sniffing to evaluate the content, the X-Content-Type-Options header can disable sniffing. The header should have the value 'nosniff' to prevent this kind of injection attacks.

[8] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security
[9] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Options

2.4.3. X-FRAME-OPTIONS

The X-Frame-Options header defines whether the browser is allowed to render frames. The HTML tags that can be disabled by this headers are: '<frame>', '<iframe>', '<embed>' and '<object>'. These tags allow a different site to be loaded as part of the initial site. Disabling these tags prohibits click-jacking attacks[10]. There are two possible settings for the X-Frame-Options header, 'DENY' and 'SAMEORIGIN'. The first disables the rendering of all frames, and the second only disables frames if they serve a different website than the initial one.

The X-Frame-Options header is made obsolete by the Content-Security-Policy header. However, the header should still be included for older browsers.

2.4.4. X-XSS-PROTECTION

The X-XSS-Protection header stops a page from loading when a reflected XSS attack is detected. This sounds great however the header never achieved full adoption. Firefox never supported the header and is not planning on doing so either. This header is replaced by the Content-Security-Policy header in modern browsers. Older Browsers still rely on this header to protect against XSS however [11].

2.4.5. SET-COOKIE

One example of a header with an impact on the systems security is the Set-Cookie header. The Set-Cookie header can have a 'secure' flag. This flag will make sure the cookie is only sent over an encrypted connection (HTTPS). If this flag is not set, the cookie could be sent over an HTTP connection, enabling third parties to steel the cookie with eavesdropping. This is especially a problem if it concerns session cookies. If the secure flag is enabled however, the cookies will only be send over HTTPS, preventing this from happening [12].

2.4.6. CONTENT-SECURITY-POLICY

Several of the mentioned HTTP headers are replaced by the Content-Security-Policy header. This header specifies what resources are allowed to load on the page. This header replaces among others the earlier mentioned X-XSS-Proteciton and the X-Frame-Options header [13]. This header is used by newer websites and browsers. However because the header was added as recently as 2013, adoption is not self-evident yet. This means that the old headers should still be supported by the websites.

[10] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options
[11] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection
[12] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie
[13] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy

3

RELATED WORK

3.1. VULNERABILITY DETECTION

3.1.1. SQL INJECTION

Injection vulnerabilities come in many shapes and forms, but they all have the same basis. Somewhere in the application, there is an ability to insert code that will be executed by some part of the system. According to Haldar et al. some of the most prevalent attacks are: 'Command injection', 'Cross Site Scripting', 'Hidden field tampering' and 'Cookie poisoning' [HCF05]. Command injection is user input that contains a executable command that is injected into the program. The most common case is SQL injection, where the command is part of an SQL query.

Detecting SQL injection with an automated tool has been done before by Kals et al. in their paper SecuBat [KKKJ06]. Kals et al. explain that SQL injection vulnerabilities can be analysed by injecting a single quote in an input field. Based on the exception that is returned, the likelihood of an injection vulnerability can be estimated. Kals et al. analysed the responses of the server to indicate the likelihood of the presence of SQL injection vulnerabilities. This simple implementation of SQL injection has potential for its use in a proof of concept.

Schwartz et al. detect SQL injection a different way, they use 'Taint Analysis' [SAB10]. Schwartz et al. describe Dynamic Taint Analysis as tracking information between source and sink. They describe that in taint analysis, data is tainted at the source and tracked trough the system. Depending on the taint policy, operations that are triggered by tainted data taint more data. So a specific subset of related data to the input data becomes tainted. Newsome et al. introduce the concept of dynamic taint analysis [NS05]. It is based on the promise that input data should never be used as jump address of format string in the code execution. If this does happen, it is a sign of vulnerability exploitation.

There are a couple of disadvantages of this system however. First, this approach requires a tool to actively monitor the execution of a program. Valgrind [NS03] was the tool used in their research. Another disadvantage is that, when this technique is used in runtime for the detection of attacks, the attack already happened when it is detected. Advantages of this technique is that there is a low false positive rate, and the tool does not have to have any knowledge of the source code.

3.1.2. Cross Site Scripting

The detection of Cross Site Scripting however, is more complex than just validating input sanitation. For detecting XSS, the use of machine learning model might be needed. Garn et al. have done so [GRG+19]. This article describes a model for detecting XSS with an accuracy of almost 80%. To validate this model, the researchers have done a real world evaluation on sex web applications. This paper could be interesting for this research because we could use this model to detect XSS or compare our results to their method.

A simpler implementation of XSS vulnerability detection is offered by Kals et al. [KKKJ06]. In their paper, they detect XSS by triggering an alert pop up.

```
1 <script>alert('XSS')</script>
```

They were not only able to detect XSS vulnerabilities, they were also able to evade XSS sanitation. By encoding part of the XSS input, filters had trouble detecting the malicious input as such. The example of the encoded XSS Kals et al. used is shown below.

```
1 &#60;ScRiPt&#62; alert&#40;'XSS'&#41;&#60;/ScRiPt&#62;
```

Image tags can also be misused for XSS. Kals et al. showed that they were able to inject a script by using an image tag [KKKJ06]. With this method, the script is not directly given as input. Instead a path to a page that contains the malicious script is supplied. The example Kals et al. gave is shown below.

```
1 <IMG SRC=JaVaScRiPt:document.forms[2].action=&quot;http://evil.
    org/evil.cgi&quot;>
```

Vogt et al. introduce a client side version of Dynamic Taint Analysis [VNJ+07]. The tainted data here, is the sensitive client side data like session cookies. The idea is that this analysis is done by the browser which checks the flow of tainted information to third parties. So the taint analysis will be triggered when someone tries to steal the cookies with an XSS attack. Whenever the cookies are send to a third party, the user is prompted with the ability to stop the action.

3.1.3. Headers

The session hijacking vulnerabilities that Drakonakis et al. have researched would make a great candidate for testing our framework [DIP20]. What makes this research so suitable is the large number of functionalities that it requires. The paper starts by eavesdropping on network traffic, making sure no cookies are send over a non encrypted connection. So our framework should be able to eavesdrop, analyse the presence of encryption and read the cookies. The paper continues by automatically signing in to websites. While automated login is out of scope for our research, the ability to script an oracle is not. In The Cookie Hunter, the network data from the login process is used to observe if the application is vulnerable for session hijacking. Our application should be able to use the data collection within its oracles to enable similar functionality. After that the researchers log out and use the cookies in different compositions to try and regain access to their lost session. So to replicate this research, our framework should be able to manipulate the cookies.

3.1.4. Session security detection

Calzavara et al. have been looking at web session attacks in their paper "Surviving the Web" [CFST17] The study shows how common attacks on web sessions work, and how they could be avoided. The emphasis of their research is on the possible ways to increase session security. Solutions they propose contain HttpOnly and Content-Security-Policy cookies. The takeaway for this research however, is that session cookies are at risk of being stolen and the way sessions are handled is an important part of application security.

3.1.5. Outdated component detection

The detection of outdated javascript libraries is done by Lauinger et al. in their paper "Thou Shalt Not Depend on Me" [LCA+18]. Lauinger et al. did not only analyse the web for outdated JavaScript libraries, they also define a step by step guide on how to do so. This is important for this research because by detecting outdated components, we can detect a large array of vulnerabilities we cannot detect natively. Being able to do so for JavaScript components would be great. Unfortunately this seems to be quite complex, Lauinger et al. Have used GitHub to obtain reference files from a list of packages to detect the versions a specific web application uses.

3.2. Research that automates certain tasks

3.2.1. Logging in

A lot of websites have login functionality, meaning a large part of the website is not accessible without logging in. For automated testing tools, this can be a challenge, because what is not accessible, cannot be tested. TESTAR supports pre-specified actions, the user can define a login sequence with this feature. When the login page is reached, this sequence will be triggered, and TESTAR will be logged in to the website [VAR+21]. This approach will work fine while testing one or a hand full of websites, but does not scale well at all. A login sequence has to be defined for every website individually.

Jonker et al. created Shepherd [JKKS20], a tool for automated login for websites. Based on the 'BugMeNot' data set, the researchers have found a way for automatic logins that has a lot higher success rate than previous attempts. A tool like this makes it possible to do post-login scanning on a large scale.

Calzavara et al. in "Measuring Web Session Security at Scale" [CJKR21], build on top of the Shepherd tool to do large scale web session analyses, analysing the password strength and session invalidation.

Drakonakis et al. have created a login framework in their paper The Cookie Hunter [DIP20]. Their method is scanning the website for login forms, and filling each of them out with the credentials. In addition to login, account creation is done.

3.2.2. Frameworks

The list of possible vulnerabilities with HTTP configuration is endless. That is why Calzavara et al. wrote "Postcards from the Post-HTTP World" [CFN+19]. In their paper they define 8 definitions for an HTTP vulnerability. These definitions make it possible to detect a well defined and well scoped vulnerability.

Drakonakis et al. have developed an automated, black box framework for exploring cookie-hijacking susceptibility [DIP20]. Drakonakis et al. use this framework in a study on

25.000 domains and discovered that almost half of them are vulnerable. Drakonakis et al. do not only demonstrate the need for analysing cookie related vulnerabilities, Drakonakis et al. also explain how they were able to analyse this vulnerability. This paper is especially interesting because a large part of The Cookie Hunter research, overlaps with the proposed research in this paper.

3.3. Security analysis tool benchmark

3.3.1. Benchmarking security analysis tools

The OWASP foundation has designed a benchmark for testing the accuracy, coverage and speed of automated software security scanners. This benchmark makes it possible to compare different tools and understand their strengths and weaknesses. The OWASP Benchmark is a runnable web application that contains thousands of test cases [Owa16]. The benchmark can translate the results into a graph that visualizes the tools performance. Indicating the True and False positive rate of the vulnerability scanner.

Amankwah et al. have used this benchmark to do an analysis of different vulnerability scanners [ACKT20]. The tools they compared were Acunetix, WebInspect, AppScan, OWASP ZAP, Skipfish, Arachni, IronWASP and Vega. One of the results of this paper is a graph indicating the True and False positive rates of these vulnerability scanners 3.1.

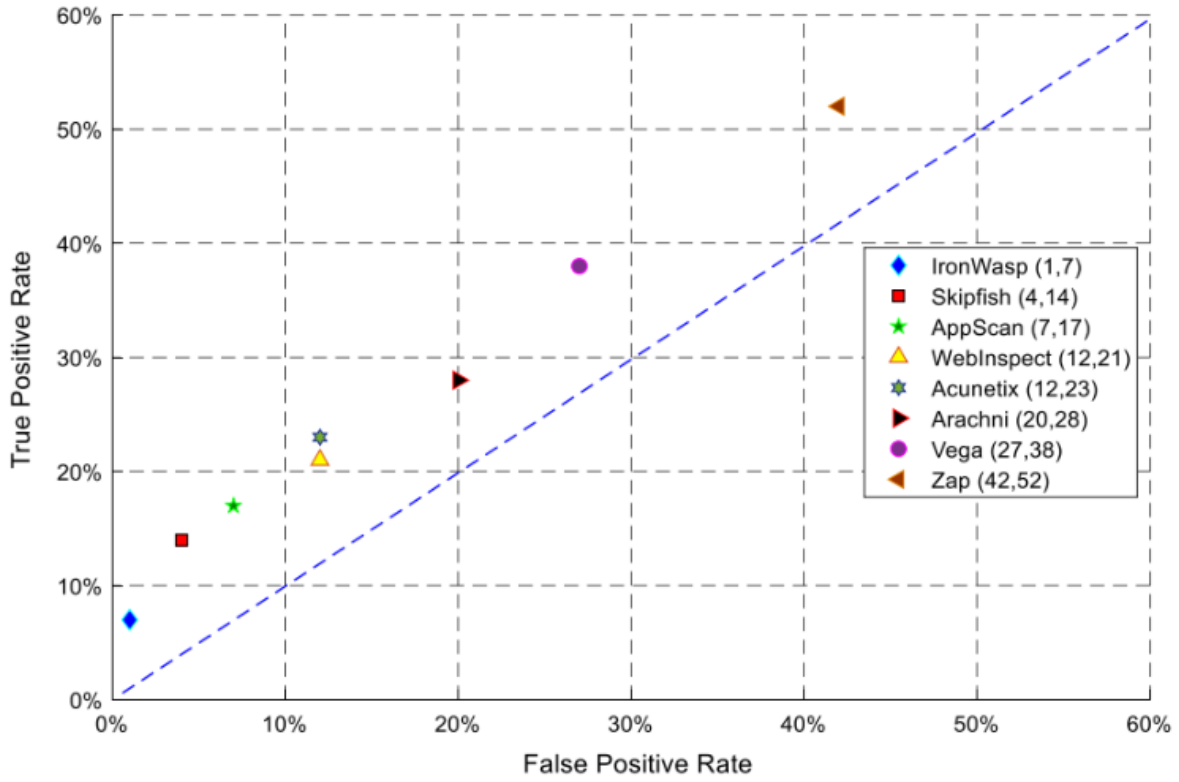

Figure 3.1: Tool performance in OWASP WBE graph (taken from Amankwah et al. [ACKT20])

A vulnerability scanner is good if it has a high true positive rate, and a low false positive rate. The blue line trough the middle is where the result the tool would guess everything. The distance from that line indicates how good the tool is. In this graph ZAP is shown top right and IronWasp is shown bottom left. However, both tools are equally far from the line.

The results of Amankwah et al. show that, from the tools that have been tested, no tool performed significantly better or worse than any other tool.

Lim Kah Seng et al. have researched the qualities of different security vulnerability scanners more toroughtly [SIS18]. They used the OWASP top 10 to categorize their findings, but did not use the OWASP benchmark. Instead they compared 46 different security vulnerability scanners based on hundreds of applications and benchmarks. They did not run all these benchmarks themselves, they combined the results of existing papers. Making their research an excellent reference for comparing specific security analysis tools to each other. Because of their clear focus of the methodology, its possible to compare the results to a new tool without having to rerun all the benchmarks for the existing tools.

3.3.2. Code coverage

One of the metrics that will be used to quantify the performance of different security analysis tools is code coverage. This means the amount of source code that is executed in a certain scenario. In the context of this research, it is the amount of source code of the SUT while running security analysis tools. Therefore, the code coverage of the SUT needs to be measured in runtime. This is something that requires additional tooling. As Horvath et al. describe, there are two kinds of code coverage, bytecode coverage, and source code coverage [HGB+19]. The big difference is that bytecode analysis can be done on compiled code, while source code analysis requires recompiling the SUT. According to Chen et al, this requirement makes the SUT no longer resemble the field behavior and is therefore less ideal [CSX+18]. Chen et al. made use of the code coverage tool JaCoCo because it is widely used in both research and practice. They did however found a few inconveniences with this tool. The results for instance were incomplete. Only the modules that were directly invoked were measured in the code coverage. The modules that were invoked indirectly, meaning by other modules, were not measured. A similar problem was found by Horvath et al. [HGB+19]. However, JaCoCo still seems to be the best Java code coverage tool at the moment.

3.4. Cost of security analysis

3.4.1. Use of security analysis tools

The previous section shows that there are many security analysis tools out there. The problem with these tools however is that they are not used enough. Smith et al. [SDM20] have researched the impact of usability of static security analysis tooling on the use by software developers. Because this research was focused on SAST tools and this research is focused on DAST tools, not all their findings apply here. Some of their findings are however transferable. They state that the integration of tools in the development workflow is one of the reasons security analysis tools are not used more effective. This is a problem that could be solved by integrating Security Analysis in another process, like GUI testing. Smith et al. conclude their paper with a list of recommendations about how security tools should interface with the user to be effective. Security tools need to show how to fix the faults they find, integrate the alerts into editable code, integrate with existing workflows and contextualize the results. The last recommendation involves adding variables to the message to place the message in context. DAST tools have no access to the code and therefore no knowledge of the root of the problem, Because of that, showing how to fix the vulnerabilities and inte-

grating them with editable code does not apply to DAST tools. Because this research is a proof of concept for security analysis, usability itself is not a priority. However, the fact that integration into the workflow is a hurdle for the use of security analysis tools shows that there is a demand for security analysis integration in existing processes, like GUI testing.

3.4.2. Cost of security analysis

To add to the case for good integration of security analysis tools in the development workflow, Curphey et al. have analysed the cost of fixing vulnerabilities [CA06]. In their paper they emphasize the importance of finding vulnerabilities in an early stage. They explain it using a graph like the one in Figure 3.2. In this graph they show that the earlier vulnerabilities are found in the development process, the cheaper they are to repair. Fixing vulnerabilities found early in the testing stage, a lot cheaper that fixing vulnerabilities found in a later stage. Because of this, they emphasize the value of finding the low hanging fruit in an early stage.

The takeaway for this research is that part of the value of security analysis in a GUI tester is that it is often already integrated in an early stage of the development workflow. Every vulnerability found in this early stage yields a cost benefit, even if a GUI tester is only able to find the low hanging fruit. This still applies if the GUI tester is accompanied by a dedicated security analysis tool in a later stage. It also shows that the performance of the GUI testing tool does not have to be on par with existing security analysis tools to add value.

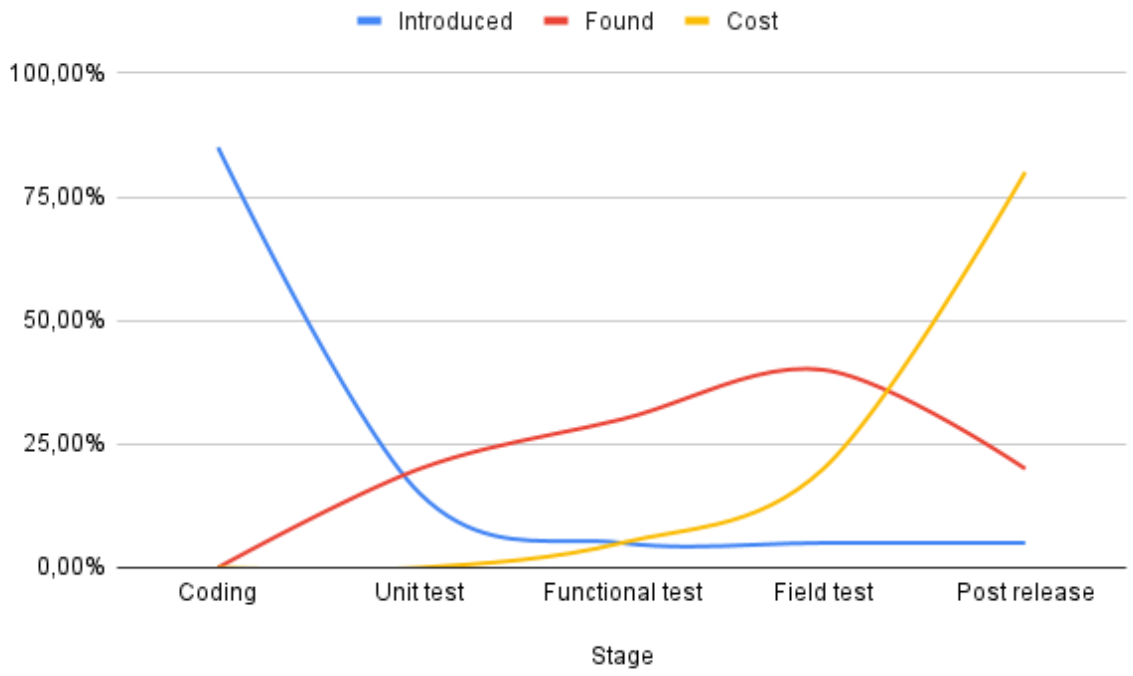

Figure 3.2: Abstract representation of the cost of repairing vulnerabilities (from [CA06])

4

DOMAIN ANALYSIS

This chapter is aimed at defining the vulnerability scope for this research. Trying to answer what vulnerabilities from the OWASP top are suitable for analysis with a GUI testing tool. Followed by a reasoning of which of those vulnerabilities should be implemented in the proof of concept for this research.

4.1. VULNERABILITIES

The OWASP foundation was created in 2001 with the objective to make the wolds software more secure. One of the means of doing this is releasing the OWASP top 10 every few years. This top 10 contains a list of 10 of the most critical security risks to web applications according to the OWASP foundation. This list is determined based on data from different sources, including security firms, bug bounty programs and a sector wide survey [owa21]. The ranking within the list is based on the impact of the vulnerabilities, and the frequency of occurring. This list is relevant for this research because it gives us an idea on what would be interesting vulnerabilities to analyse. The latest iteration of this list is the OWASP top 10 for 2021 [owa21]. The vulnerability categories are listed in Table 4.1.

place	category
1	Broken access control
2	Cryptographic Failures
3	Injection
4	Insecure Design
5	Security Misconfiguration
6	Vulnerable and Outdated Components
7	Identification and Authentication Failures
8	Software and Data Integrity Failures
9	Security Logging and Monitoring Failures
10	Server-Side Request Forgery

Table 4.1: OWASP top 10 - 2021 [owa21]

To know what vulnerabilities make scenes to look for with a GUI tester, the suitability of the vulnerabilities needs to be determined. This is done by giving some explanation

for each of the vulnerabilities of the OWASP top 10, and examine why they are or are not suitable for analysis using a GUI tester.

4.1.1. Broken access control

Broken access control is a flaw that enables unauthorised access to resources from authenticated users [1]. The category of broken access control is a broad category. It includes the tempering with JWT tokens and brute forcing of authentication for example. A large part of these vulnerabilities are not suitable for detection with a GUI testing approach. There are however vulnerabilities in this category that would be possible to analyse, some examples of them are elaborated further below.

Missing access controls is one of the examples of broken access control that can be analysed by a GUI tester. It would be possible to try and call every POST, PATCH, PUT or DELETE endpoint encountered in the application without a security token. The GUI testing approach would contribute by discovering and identifying all the endpoint. It would also enable the oracle to put the endpoint in context, making an educated guess about whether an endpoint should be secured based on where in the application it is used. The method of categorizing the endpoints based on their use in the frontend is unique to the GUI testing approach and could yield interesting results.

A second example is the ability to view or edit someone else's account by manipulating the session identifier. This could be done by identifying user or resource identifiers and changing them to other known user or resource identifiers. This is something that could be tested using a GUI tester. The advantage of a GUI tester is that they generate legit calls during their GUI testing process. These calls can be modified with different resource and user identifiers, making it unnecessary to build up from the ground up.

A last example is the elevation of privilege, by acting as a user while not being logged in as one. The GUI tester could record parts of a test run while logged in as user or admin. After that, the GUI tester could try and replicate parts of those runs without the necessary privileges. Using a scriptless GUI tester in this way would make scene because the GUI tester explores the application by itself. Meaning it can generate logical sequences within the portal without users having to interfere. Because dedicated security analysis tools do not follow the application flow like a GUI tester does, this approach is limited to GUI testers.

This shows that broken access control is something that is partially suitable for analysis using a GUI testing approach.

4.1.2. Cryptographic Failures

Cryptographic Failures are caused by the use of insufficient or old cryptographic algorithms [2]. This could include the use of HTTP connections and old versions of TLS. Some of the options for analysing cryptographic failures are the checking of the handshake information and whether the protocols that are specified really are the protocols that are accepted.

The advantage of a GUI tester in such a scenario is that the GUI tester is navigating the application anyway. This means that all analyses can be done on a extensive part of the SUT. If the analysis can be done without interfering with the GUI testing process, it would require no extra runtime. An example of a cryptographic failures that can be analysed in

[1] https://owasp.org/Top10/A01_2021-Broken_Access_Control/
[2] https://owasp.org/Top10/A02_2021-Cryptographic_Failures/

the background by a GUI testing tool is the presence Strict-Transport-Security HTTP header. This header makes sure that an encrypted connection is always used.

These are also some limitations to the GUI testing approach. A GUI tester often uses an interface to interact with the SUT, in TESTAR's case this is WebDriver. This limits the access the GUI tester has to certain information. WebDriver does not expose handshake information for instance. There are workarounds, but evaluating them is outside the scope of this thesis.

In summary, there are cryptographic failures that are suitable for analysis using a GUI tester.

4.1.3. INJECTION

An injection vulnerability is the ability to insert code or values that will change the behaviour of the system in an unintended way. For example by executing the code or manipulating privileges [3]. According to Haldar et al. some of the most prevalent attacks are: Command injection, Cross site scripting, Hidden field tampering and Cookie poisoning [HCF05].

Command injection is a form of injection where user input contains a command in a way that the program will execute that command. The most common case of this is SQL injection. In this case, part of an SQL statement is injected into the code. As an example we take a login screen where the user fills in the username. To get this data from the database, the following query is constructed. "SELECT * FROM user WHERE user_name = 'user_input';". If the user would insert "name' or 1 = 1'"; the resulting query would be "SELECT * FROM user WHERE user_name = 'name' or 1 = 1';". This expression will always be true, so all the users will be returned from the database.

XSS is another prevalent vulnerability. In this case, the server computer is not the target, but the client machine, in JavaScript. There are two kinds of XSS, stored XSS and reflected XSS. The former kind works with JavaScript code that comes from the server, because the attacker was somehow able to get this code in the database. An example where such a situation could occur would be a forum, where anybody can write anything. The latter works a bit different. It is called reflected XSS and with reflected XSS the user has to click on a link containing the script. What both kinds have in common however is that they get JavaScript code as input, and try to execute is, an example is shown below.

```
1 <script>alert('XSS')</script>
```

This example is harmless, because it only trows an alert pop-up. However, the JavaScript code that is injected could also have malicious intent, stealing cookies for example.

The problem with injection vulnerabilities however, is that detecting them is a field on its own. While there is no reason this could not be done while GUI testing, it would be very hard to detect these vulnerabilities with high accuracy within the scope of this thesis. That is why this research should not be aimed at doing SQL injection at the level of dedicated security analysis tools. This thesis should be aimed at proving that TESTAR is able to do security analysis and that it could be able to do it at the level of dedicated security analysis tools.

A possible way of detecting SQL injection would be to escape the SQL statement using a ' and look for an exception indicating that the input is not sanitized. This method was used

[3] https://owasp.org/Top10/A03_2021-Injection/

by Kals et al. [KKKJ06].

For Cross Site Scripting, there are two possible approaches. Executing XSS and detecting the presence or lack of presence of XSS mitigation measures. Kals et al. used the first method by trying to inject and detect an alert pop up [KKKJ06]. The second way of detecting possible Cross Site Scripting vulnerabilities could be by detecting the lack of HTTP headers that are supposed to prevent XSS.

In conclusion, GUI testing tools are able to do SQL injection and XSS vulnerability analysis. However, the scope of implementing injection vulnerabilities within this research should be aimed at the proving the concept, not the best way of detection itself. Because detecting injection vulnerabilities is a field by itself.

4.1.4. Insecure Design

Insecure Design is a broad category about missing or ineffective control design [4]. This category is specifically about design and not about implementation. Because it is only possible to test the implementation with web scraping, this category is not suitable for analysis using a GUI tester.

4.1.5. Security Misconfiguration

Security Misconfiguration is a problem with the configuration of security aspects of a system [5]. While most of this broad category will not be within the scope of this research, there are security misconfigurations that are detectable using GUI testing.

An example of this is the presence of HTTP headers and HTTP header flags. The GUI tester triggers the endpoints of the SUT like they would be triggered in normal operation. Because of this, all the HTTP headers that would be present in normal operation can be analysed on every call. This also applies to every security misconfiguration that can be detected from the client side. The GUI testing approach has the most value for configurations that apply to every call, because they are able to analyse every call during the GUI testing process.

Another example of a misconfiguration that would be suitable for analysis using a GUI tester would be the reveals of stack traces to users. If an exception is triggered in the backend, does the exception contain a stack trace? This is a problem because it exposes information about the flaws and the inner structure of as system. That is information that could be used by an attacker to gain knowledge needed to exploit vulnerabilities. If an exception can occur during the usage of the system, a GUI testing tool would be a good candidate for triggering and analysing it.

In summary, GUI testing is a suitable way of testing certain security misconfigurations. Because of its ability to constantly analyse the results of these configurations during normal use.

4.1.6. Vulnerable and Outdated Components

Vulnerable and Outdated Components can impact the security of a system because issues that are known to exist and have been resolved in newer versions of the components, are

[4] https://owasp.org/Top10/A04_2021-Insecure_Design/
[5] https://owasp.org/Top10/A05_2021-Security_Misconfiguration/

still present in the applications [6]. Checking this would require a data source with the latest version of all components and access to the component versions used by the website.

Lauinger et al. [LCA+18] have shown that it is possible to detect Java Script libraries without access to the source code. While it is possible, a black box approach for detecting libraries is very complex. A white box approach on the other hand, is not. So while it would be really interesting to detect component versions using a GUI tester, it would also be very complex. But most important of all, it would not add any value to the field of security analysis.

4.1.7. Identification and Authentication Failures

Identification and Authentication Failures are flaws that enable someone to be misidentified for someone else [7]. The system is unable to verify an identity securely.

There are many identification and authentication failures that a GUI tester would be able to analyze. For instance, invalidating a session on logout. The GUI tester could remember the token before logout and try to use the same token after logout to access the site again. If the token still works, the session never really ended.

Another example is, does the system allow brute force attacks or does it stop the attacker at some point? To continue on that subject, are there other automated attacks allowed? For instance, can we run a list of default usernames and passwords without the system protesting? Those are vulnerabilities that a GUI tester could analyse quite well.

The last example of an identification and authentication failure a GUI tester could analyse is weak password rules. Trying to set a series of passwords to figure out the password policy for example. Or as a last example, look for exposed session identifiers in the URL.

This shows that there are a lot of identification and authentication failures that could be analysed using GUI testing. However there are disadvantages to using a GUI tester for these vulnerabilities. The GUI tester would require SUT dependent setup for example. Pointing the GUI tester to the right pages and fields to complete the scenarios.

4.1.8. Software and Data Integrity Failures

Software and Data Integrity Failures originate from infrastructure and code that have a lack of protection against integrity violations [8]. This can easily arise when trusting code or data from untrusted sources, like user input. This is not something that can be checked by scraping a web page, so this type of vulnerability is not suitable for detection using a GUI testing tool.

4.1.9. Secure logging and monitoring

Secure logging and monitoring helps to detect active security breaches [9]. Logging and monitoring are very hard to test with web analysis, because it requires access to the logs. That is why this category is also not suitable for analysis using a GUI tester.

[6] https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/
[7] https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/
[8] https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/
[9] https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/

4.1.10. Server Side Request Forgery

Server Side Request Forgery or SSRF is an attack where an attacker can make the server side of an application make a call to a third party system [10]. Detecting SSRF will be outside of the scope for this research.

4.1.11. Recap

The vulnerabilities that could be identified with TESTAR can be classified into two categories. First, there are vulnerabilities that are passively detectable, like detecting the presence of Security headers. Secondly, there are vulnerabilities that are actively detectable, like the invalidation of a session after logout, as we can see in Table 4.3. The list of vulnerabilities is not complete, this is the overlap of vulnerabilities that are featured by the OWASP foundation [owa21] and vulnerabilities that would seem suitable for analysis using a GUI tester. Because of the grouping by OWASP categories, some vulnerabilities overlap, like the HTTP header related vulnerabilities 5 (Strict-Transport-Security header), 8 (X-XSS-Protection header) and 11 (HTTP security headers).

place	category	analysable?
1	Broken access control	✓
2	Cryptographic Failures	✓
3	Injection	✓
4	Insecure Design	
5	Security Misconfiguration	✓
6	Vulnerable and Outdated Components	
7	Identification and Authentication Failures	✓
8	Software and Data Integrity Failures	
9	Security Logging and Monitoring Failures	
10	Server-Side Request Forgery	

Table 4.2: OWASP top 10 [owa21], with analysability

4.2. What vulnerabilities should we analyse?

Now that there is a clear picture of what vulnerabilities could be analysed using a GUI tester, the question arises, what should be implemented in TESTAR during this research? The numbers used to refer to the vulnerabilities in this sections will be the numbers from Table 4.3. The passively analysable vulnerabilities could be analysed during a normal GUI-testing session. The actively analysable vulnerabilities would have to interrupt a GUI testing session or start a separate TESTAR instance.

4.2.1. Passively analysable vulnerabilities

The passively analysable vulnerabilities are interesting because they would not have to influence the GUI-testing session. TESTAR could sniff and analyse all the network traffic on the background and report any vulnerabilities that appear during a normal GUI testing session. This would add value to TESTAR, even if it is not explicitly used to do security analysis.

[10] https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/

Nr	vulnerabilities	category
Actively analysable vulnerabilities		
1	Missing access controls	Broken access control
2	View or edit someone else's account	Broken access control
3	URL modification	Broken access control
4	Elevation of privilege	Broken access control
6	SQL injection	Injection
7	XSS	Injection
12	Session invalidation	Identification and Authentication
13	Brute force protection	Identification and Authentication
14	Automated attack protection	Identification and Authentication
15	Weak password rules	Identification and Authentication
9	Enabled default accounts	Security Misconfiguration
Passively analysable vulnerabilities		
5	Strict-Transport-Security header	Cryptographic Failures / Security Misconfiguration
8	X-XSS-Protection header	Injection / Security Misconfiguration
10	Stack-trace reveals	Security Misconfiguration
11	HTTP security headers	Security Misconfiguration
16	Exposed session identifiers	Identification and Authentication

Table 4.3: Vulnerabilities that could be analysed using a GUI tester

There are five vulnerabilities within the category 'passively analysable vulnerabilities'. Numbers 5 (Strict-Transport-Security header), 8 (X-XSS-Protection header) and 11 (HTTP security headers) are all HTTP header related, so analysing HTTP headers would be a quick win in this category. Another benefit of analysing HTTP headers is that TESTAR would be able to do as good a job as any other HTTP headers analysis tool and it is not necessary to configure anything application specific. This makes analysing the HTTP headers a clear example of something TESTAR should analyse.

Next in this category is number 10, 'Stack trace reveals'. This is a vulnerability to get exited about because TESTAR is made to trigger exceptions. Because of this, TESTAR has a clear advantage over dedicated security analysis tools. TESTAR would not only be able to analyse this vulnerability, TESTAR might actually be better at this than dedicated security analysis tools. Detecting stack trace reveals can be split into two challenges. Finding the stack trace if it occurs, and identifying the stack-trace as such. With the potential of analysing this vulnerability with TESTAR, it is clear that this vulnerability has to be analysed.

The last vulnerability in this category is number 16, 'Exposed session identifiers'. This is a harder choice within this category. There are many forms of session identifiers, and identifying a session identifier in an URL is easier said than done. TESTAR would need to classify any value in the URL as either a session identifier or not a session identifier. If this classification process is not absolutely perfect, there will be either false positives or false negatives. It would be possible that there is a scenario in which TESTAR is able to correctly identify exposed session identifiers, but identifying most exposed session identifiers correctly is a whole other problem. The problem here is detecting the session identifier without knowl-

edge on what the session identifier might look like. That is why this vulnerability will not be within the scope of this research.

4.2.2. ACTIVELY ANALYSABLE VULNERABILITIES

Actively analysing vulnerabilities can be done with security oracles. This is an override within TESTAR that executes a specific action based on a specific trigger. There are a couple of disadvantages of testing these vulnerabilities. First, this will interrupt the normal GUI-testing process, that is why this type of analysis should probably be used in its own session. This means that analysing these security vulnerabilities alters the flow and adds time to the testing process. Additionally, these triggers are application specific, not every application has its password reset screen on the same URL or look the same. Likewise, the elements that will be interacted with require additional mapping. Not every password input field has 'password' as element identifier for example. This makes the process of doing active security analysis with TESTAR take some effort, which takes away from the added value of this security analysis.

That being said, there are many vulnerabilities that could be tested this way. Besides, TESTAR can already execute preprogrammed sequences, like logging in and filling out forms. This would make TESTAR a good candidate for executing security related sequences. Because of the limited added value in comparison to the passive security analysis, and the limited resources available for this research, not all vulnerabilities will be implemented. Instead, some of the vulnerability detection oracles will be implemented, proving that it is possible and leaving a framework for implementing other vulnerability detection oracles in the future. The following section will iterate over the actively analysable vulnerabilities that are going to be implemented within this research and elaborate on why they are good candidates. Due to time constraints, the other actively analysable vulnerabilities were not implemented.

SQL INJECTION

SQL injection is a field on its own, and it is not reasonable to expect cutting edge SQL injection performance in this research. However, the concept of SQL injection is not that complicated. By sending a malicious string as input to the server, actions on the database can be performed. For this concept it does not matter if the string is added directly or via an input field. Inputting strings in input fields is something that GUI testers are able to do quite well, so proving that TESTAR has the ability to do a form of SQL injection detection should not be far-fetched.

Injection attacks have the second greatest criticality of all web vulnerabilities according to the OWASP top 10 [owa21]. This is why it would be a good idea to take SQL injection as one of the vulnerabilities used to demonstrate the potential of GUI testing in the field of security analysis. That is why SQL injection vulnerability detection will be implemented within this thesis.

XSS

XSS has a lot in common with SQL injection. Like SQL injection, XSS falls in the third most critical category of web vulnerabilities according to the OWASP top 10 [owa21]. XSS also makes use of input values to inject malicious code. Unlike SQL injection however, this code is executed in the GUI itself. Because XSS is a security vulnerability in the GUI, it is a natural point where security analysis and GUI testing overlap.

Not all XSS would be equally difficult to implement though. Reflected XSS does not store any values and executes code at the moment of injection. This takes away the variable of time and order of execution. Stored XSS requires a two stage process, starting with one step of injecting the code, and a second step of the code being executed. Because of this two step process, Stored XSS has added complexity that will be out of scope for this thesis. Reflected XSS however is a good candidate for implementing in TESTAR and falls in the scope of this research.

SESSION INVALIDATION

The last active vulnerability that will be implemented within TESTAR is Session Invalidation. Session tokens are used to identify the client session. This way the server knows who is interacting with the system and what resources they can access. If the session is ended by a user logging out, the server should not accept the token anymore, because the session has ended. If the token is later retrieved from the client computer in some way, the token cannot be used to continue the session. That is a scenario that is not desirable and can be classified as a security vulnerability.

Because this is a vulnerability that is not easily checked with existing security vulnerability tools it can increase the contribution GUI testing can make to the field of security analysis. A second reason is the complexity of this vulnerability. It requires a scenario where TESTAR logs in to an application, gets the token from the cookies and logs out of the application. After that, TESTAR has to set the cookie manually and redirect to a page of the application where a session is required. The complexity of this scenario can show TESTAR's potential to automatically test a variety of security oracles. Because this increases TESTAR's contribution to the field of security analysis and shows the potential TESTAR has, this vulnerability will be implemented within this research.

RECAP

For this section, the vulnerabilities are summarized one more time, this time with one of the following categories in Table 4.4. 'Yes' has been used for vulnerability detections that will be implemented within this research, and 'No' has been used for vulnerabilities that will not be implemented within this research. All the vulnerabilities mentioned seem possible to implement in TESTAR, so they might be implemented later. The vulnerabilities that will be implemented are: '5. Strict-Transport-Security header', '6. SQL injection', '7. XSS', '8. X-XSS-Protection header', '11. HTTP security headers' and '12. Session invalidation'.

Nr	description	implement
Actively analysable vulnerabilities		
1	Missing access controls	
2	View or edit someone else's account	
3	URL modification	
4	Elevation of privilege	
6	SQL injection	✓
7	XSS	✓
9	Enabled default accounts	
12	Session invalidation	✓
13	Brute force protection	
14	Automated attack protection	
15	Weak password rules	
Passively analysable vulnerabilities		
5	Strict-Transport-Security header	✓
8	X-XSS-Protection header	✓
10	Stack-trace reveals	
11	HTTP security headers	✓
16	Exposed session identifiers	

Table 4.4: Vulnerabilities and whether they should be analysed with TESTAR

5

METHODOLOGY

This research is aimed at showing how TESTAR can contribute to the field of security analysis. To accomplish this, TESTAR will first be extended with the vulnerability detection abilities proposed in the previous chapter. These vulnerabilities are: 'SQL injection', 'XSS', 'HTTP header misconfiguration' and 'session token invalidation'. To validate these abilities, TESTAR is run against a benchmark.

To measure the real contribution TESTAR is making to the field of security analysis, it is important to compare TESTAR to existing security analysis tools. That is why existing DAST tools are also run against the benchmark to compare the results to those of TESTAR. Both TESTAR and these tools are also used on a real web application, to measure and compare the code coverage. Part of the value of using TESTAR is the lack of cost and knowledge required to set up a second tool, so the setup processes of the tools are also compared.

5.1. FINDING VULNERABILITIES USING TESTAR

5.1.1. HTTP HEADERS

HTTP headers are returned by the server for every request. To find vulnerabilities regarding misconfigured HTTP headers, the network traffic has to be analyzed and the present headers need to be validated. During this analysis, as many endpoints have to be called in as many ways as possible to get the best coverage of the SUT. Because TESTAR already generates this network traffic during GUI testing, we only needed to listen to network traffic and validate the headers.

An alternative to using the GUI testing algorithm would have been to modify or make our own algorithm. This could have created a more optimal form of network traffic for HTTP header analysis. We still decided to use the original GUI testing algorithm because one of the strong points of TESTAR is the ability to explore a website. Designing an algorithm that outperformed the TESTAR algorithm is out of scope for this thesis.

5.1.2. XSS INJECTION

To analyze Cross Site Scripting vulnerabilities using TESTAR, we need a way of finding XSS vulnerabilities in general. The first way is to test the ability to detect Cross Site Scripting vulnerabilities in an application by trying to execute XSS. The second way is by checking for the absence of Cross Site Scripting protection, like the X-XSS-Protection header. Because

this header is controversial, we decided to go with the first approach of trying to trigger XSS. The approach we have used is derived from the work of Kals et al., they tried to trigger an alert box to see if the application was vulnerable to XSS [KKKJ06]. Our approach has a slight modification however, we used different alert texts for every alert, to identify the source of the injection.

To enable TESTAR to detect XSS vulnerabilities, we needed to replace the random text generator with a dedicated security text generator. This generator will generate script tags with alert boxes instead of random text. A second modification was the detection and identification of the alert boxes if they appeared, to determine the presence of XSS.

5.1.3. SQL INJECTION

Detecting SQL injection is a field by itself, and creating a GUI testing tool that can reliably detect SQL injection is out of scope for this research. That does not make it useless to try and detect SQL injection with TESTAR however. The most dangerous SQL injection vulnerabilities are those that are easiest to find.

To detect SQL injection, we chose to break queries instead of manipulating them. Adding two quotes enables code to be injected into an existing query, this is dangerous because this gives people the ability to manipulate your database. Adding one quote however will make executing the query impossible, like Kals et al. demonstrated [KKKJ06]. An error response from the server would indicate that something has gone wrong and that the application is possibly vulnerable to SQL injection.

Extending TESTAR with this ability builds on the security text generator used for the XSS detection. This security text generator was extended with the ability to inject single quotes. A second adjustment needed to be made to detect the error response created by the SQL injection. To do that, we extended the network listener to detect when an SQL injection was actually executed, and if it resulted in a HTTP 5xx response.

5.1.4. SESSION INVALIDATION

The last ability TESTAR is extended with is the ability to detect session invalidation vulnerabilities. These tokens are sent by the server to the client when a user logs on to a portal. It is best practise to store those tokens as cookies, however session storage or even local storage are also known to be used in some implementations. For this research a scenario where the tokens are stored in cookies is assumed. This is to simplify the implementation which sole purpose is to prove the concept.

For TESTAR to be able to detect this vulnerability, it first needs to login on the application. Once it gets a session token, TESTAR will need to store that session token somewhere safe. At the same time, a web page will be loaded from inside the portal, the URL of this web page will also need to be stored by TESTAR. The next step is for TESTAR to logout, this will redirect TESTAR to a page outside the portal and remove the session token from the cookies. Now TESTAR still has the token and will attempt to see if it still works. It does this by adding the token to the clients cookies and navigating to the URL from inside the portal that has been saved. If TESTAR navigates successfully to inside the portal, the token is not invalidated on logout. On the other hand, if TESTAR is not able to successfully navigate to that page, the session token is invalidated properly.

5.2. VALIDATION AND EXPERIMENT DESIGN

To validate TESTAR's new abilities and compare TESTAR to existing security analysis tools, a series of experiments was executed. First, TESTAR and the existing security analysis tools were run against a benchmark, this validated TESTAR's abilities to detect XSS, SQL injection and header misconfiguration. This also made it possible to compare TESTAR's abilities to those of existing security analysis tools. The second experiment ran both TESTAR and dedicated security analysis tools against an existing application to measure the code coverage. This would give an indication on the relative code coverage of TESTAR in comparison to dedicated security analysis tools. The last experiment was intended to measure TESTAR's ability to detect session token invalidation. During the experiments, the set-up complexity of both TESTAR and the dedicated tools was recorded to enable a comparison of set-up complexity.

5.2.1. SECURITY ANALYSIS TOOL

The dedicated security analysis tool used to compare TESTAR to was OWASP ZAP [OWA]. This tool was the only free tool that would work without any problem on a Windows system. Other tools in the running were Arachni and Wapiti, but it was not possible to get them working within the constraints of this research. Wapiti did not function for unknown reasons, it was likely due to problems with existing dependencies on the computer used for this research. Arachni did function, but was not able to test localhost URL s, making it complex to run against SUTs that ran on the same machine. Because of the limited resources available for this research, the decision was made to continue with a single dedicated security analysis tool.

5.2.2. BENCHMARK

For the benchmark the OWASP benchmark project was used. This is a benchmark that is created by the same organization that maintains the OWASP top 10, and enabled us to test XSS, SQL injection and header misconfiguration. The 1.2 version of this benchmark was used for this experiment. This version contains 504 tests for SQL injection, 455 tests for XSS and 67 for secure cookie flags. The secure cookie flag was used to test TESTAR's ability to detect HTTP header configurations. Because the secure cookie flag is communicated within the set-cookie header, it is probable that TESTAR would perform similar in analysing other HTTP headers. Each of the tools was run against the benchmark for every vulnerability twice to get a result that was representative of the abilities of the tools, resulting in six runs per tool.

The benchmark contains many more test cases for different vulnerabilities, but only the vulnerabilities that were implemented in TESTAR were analysed. The benchmark provides a list of the test cases and expected results, that way it is possible to compare the output of TESTAR and ZAP to calculate a score.

The way TESTAR navigates the SUT was altered to make it run the benchmark more efficient. TESTAR was able to interact with every widget (element) on a page twice. It was also not able to interact with a widget for the second time before every widget was interacted with the first time. This made sure TESTAR has interacted with every widget on the page at least once before clicking a submit button. This also resulted in TESTAR running every test case in the benchmark at least once before repeating. The alteration was made to min-

imize TESTAR's runtime on the benchmark. It does however mean that the results of the benchmark say something about TESTAR's ability to detect vulnerabilities once the location of the vulnerability was navigated to, not about TESTAR's ability to find all vulnerabilities in an application.

5.2.3. CODE COVERAGE

To be able to say something about how good TESTAR is at exploring an application the code coverage was measured. To measure the code coverage of running applications OpenCover was used. OpenCover is a tool for calculating test coverage for .NET applications. Another tool considered for measuring the code coverage was JaCoCo. The main difference between the tools is that OpenCover is compatible with .NET applications while JaCoCo is compatible with Java applications. Because of the available experience, working with .NET applications was preferred.

The SUT used for this experiment is the open source web CMS Blogifier [1]. This is an open source .NET 6.0 application that contains a couple of advantages that makes it very suitable for measuring the code coverage. The first advantage is that Blogifier supports SQLite databases, eliminating the need to run a separate database. Another advantage is that Blogifier uses Blazor WebAssembly instead of JavaScript for its front end. This means that the front-end does not need to be hosted separately and that the front-end code is measured in the code coverage measurements. The last advantage of this software is that it requires minimal setup, one Blogpost was created before each test to make sure the tools were able to reach a reasonable part of the application.

Only one SUT was used because it has proven difficult to find open source .NET applications that were suited for this research. Because of the time limitations of the thesis, the decision was made to only use one application for measuring code coverage. However, another application that was considered as SUT was NopCommerce [2]. An open source e-commerce application written in .NET. This application did however pose a fatal flaw. Because it is an e-commerce platform, it requires a lot of setup to enable the tools a chance to explore the whole application. This setup needs to be reset between every test run to get consistent results. Because this application was so big, it was already hard to get consistent results. That is why this application was eventually not used as SUT.

Both TESTAR and OWASP ZAP were run against the SUT multiple times, this ensured consistent results for both tools. Before each run the database was reset and one blog post was created. While OWASP ZAP was run twice, TESTAR was run eight times This ensured consistent results over different configurations of TESTAR. The configurations were combinations of the protocols 'webdriver_generic' and 'webdriver_statemodel', and sequence limits 100 and 200. Each combination was executed twice.

5.2.4. TOKEN INVALIDATION

To validate TESTAR's ability to analyse token invalidation, both a SUT with and without token invalidation were required. Because tempering with the cookies in production environments did not feel right, an in house application was altered to fit both of these use cases. The application is built in .NET and makes use of JWT tokens. For this experiment, three versions of the application were created. The first invalidated the session token on

[1] https://github.com/blogifierdotnet/Blogifier
[2] https://github.com/nopSolutions/nopCommerce

logout, when the token was used again, it was not accepted by the application. The second version of the application did not invalidate the token, allowing TESTAR to continue exploring the application after logout. The third implementation did invalidate the token, but waited 10 seconds to do so. This is to simulate applications with asynchronous token invalidation. TESTAR was run twice on each of the applications to make sure the results were consistent.

5.2.5. EXPERIMENT OVERVIEW

So we ended up executing six experiments to answer our research question. These experiments are listed in the table below 5.1.

Nr	experiment	SUT
1	HTTP headers	OWASP Benchmark
2	XSS	OWASP Benchmark
3	SQL injection	OWASP Benchmark
4	Code coverage	Blogifier
5	Token invalidation	Custom application
6	Setup time	Blogifier

Table 5.1: Experiments

5.3. ALGORITHM DESIGN

5.3.1. HTTP HEADERS

To detect HTTP headers with TESTAR, we will use WebDriver to listen to all the network traffic between the client and the server of the SUT. This is done by extracting the DevTools from WebDriver instance used. These DevTools allow the setting of a listener for specific events. The event that is interesting for HTTP header analysis is the 'Network.responseReceivedExtraInfo' event. This event is triggered every time the server responds to a network request, this response includes the HTTP headers. A code example for retrieving the HTTP headers is shown below.

```
public void addListener(DevTools devtools)
{
    devTools.addListener(Network.responseReceivedExtraInfo(),
        responseReceived -> {
            Headers headers = responseReceived.getHeaders();
            // process headers
        });
}
```

When a request comes in, the headers are processed. In the case of TESTAR, these headers need to be stored until TESTAR is ready to give a verdict. This is because it is possible for multiple network requests to be executed in one TESTAR inner loop.

For this research, six HTTP headers were analysed. These headers are listed below in Table 5.2. These headers and their required rules are derived from Mozilla [Moz]. The set

of headers implemented in this research does not contain all the headers used for web security. It does however contain some fundamental headers and is a large enough set to demonstrate TESTAR's abilities.

Nr	description	requirement
1	Strict-Transport-Security	Header is present
2	X-Content-Type-Options	Contains 'nosniff' flag
3	X-Frame-Options	Header is present
4	X-XSS-Protection	Conaints '1; mode=block' flag
5	Set-Cookie	Contains 'secure' flag if header is present

Table 5.2: HTTP headers that will be analysed

When TESTAR is forming a verdict, the headers from the past request need to be analysed. Each header should comply to the rules above for each request. An example for an oracle that validates the Set-Cookie header can be found below. This method would be called by TESTAR's getVerdict() method.

```
public boolean validateSetCookieHeaders(Map<String, String>
   headers) {
    for (Map.Entry<String, String> header : headers)
        if (header.getKey().equals("Set-Cookie")) {
            if (!header.getValue().contains("Secure;")) {
                // vulnerability found
                return true;
            }
        }
    }
    // no vulnerability found
    return false;
}
```

5.3.2. XSS

To analyse XSS vulnerabilities without access to the source code, it is necessary to try and trigger XSS vulnerabilities on the SUT. To do so, TESTAR had to be extended to inject something that could be detected on execution. Kals et al. ran in to the same problem with and provided alert boxes as the solution [KKKJ06]. By injecting the code below, an alert box was triggered when the application was susceptible to XSS.

```
<script>alert('XSS')</script>
```

Using this technique with TESTAR introduces added complexity. First, TESTAR needs to be able to detect the alert boxes when they are thrown. Then TESTAR needs to be able to determine if the alert box is thrown because of an XSS vulnerability. Lastly, TESTAR needs to interact with the alert box to make it disappear again.

Because this approach was overly complex, a different approach was used. Instead of the alert boxes used by Kals et al. [KKKJ06], logging was used.

```
<script>console.log('XSS')</script>
```

By making the XSS write records to the console log, it did not interfere with the exploration of the SUT. TESTAR was able to continue testing while XSS exceptions appeared automatically in the log in the background. A code sample to enable TESTAR to read the logs and determine if XSS vulnerabilities were present is shown below.

```
public boolean validateXss(WebDriver webDriver)
{
    logs = webDriver.manage().logs().get(LogType.BROWSER);
    for (LogEntry entry : logs) {
        if (entry.getMessage().contains("XSS"))
        {
            // XSS vulnerabilities found
            return true;
        }
    }
    // no XSS vulnerabilities found
    return false;
}
```

For an attacker it is much more interesting to execute code on someone else's computer than on their own. That is why reflected XSS is often hidden in links. That is why it makes sense for TESTAR to add the ability to analyse XSS vulnerabilities in the URL. To do so, TESTAR looks at if there are parameters parsed in the URL. In the example below we see a URL with two parameters.

```
https://mywebpage.com/page?parameter1=value1&parameter2=value2
```

TESTAR needs to detect the presence of URL parameters, it does this by looking for and & signs in the URL. If the URL does contain a string, TESTAR will replace all the characters between an = and a & or the end of the URL with a predetermined injection string. A code example that shows how this works is shown below.

```
public String getXssUrl(String url)
{
    String injection = "<script>console.log(%27XSS%20detected
       !%27);</script>";
    if (url.contains("?"))
    {
        url = url.replaceAll("=.*" + "&", injection + "&");
        url = url.replaceFirst("[^=]*$", injection);
    }
    return url;
}
```

After manipulating the URL, TESTAR navigates to this new URL. The XSS is often not executed while loading the new page, the XSS from the URL might only be executed when

the value from the URL is used. That is why TESTAR explores the new page again to see if the URL is vulnerable to XSS.

Something that stands out in the code example above, is the fact that the special character in the injection string are replaced with %27 for ' and %20 for a space. This is because some special characters need to be encoded to be interpreted correctly in an URL.

This method of finding XSS vulnerabilities does not cover all XSS vulnerabilities. One of the ways TESTAR will not find an XSS vulnerability is when it uses filters to filter out the script tags. This makes it impossible to inject XSS using script tags, however there are many more ways to evade XSS filters. One of the ways to evade XSS filters is by decoding the XSS, this technique was used by Kals et al. [KKKJ06]. There are however a lot more methods of filter evasion, a large part of them is listed by the OWASP foundation [3].

The current implementation of TESTAR does not make use of filter evasion, this means that it cannot detect XSS injection vulnerabilities when a filter is used. Even though the XSS vulnerabilities can still persist. A second limitation of the current implementation is that it only tests from fields and URL s. There are more ways of injecting reflected XSS, but TESTAR will not be able to find them at this point. This decision was made to reduce the complexity of the implementation, enabling it to fit into the scope of the thesis.

5.3.3. SQL INJECTION

Finding SQL injection with an automated tool can be done in one of two ways. The first is by analysing the source code, this does not require the actual execution of SQL injection, but requires access to the source code. The second method is by executing SQL injection attempts and analysing the response to determine SQL injection vulnerabilities. Because TESTAR is a GUI testing tool, which does not have access to the source code, it makes use of the second method for detecting SQL injection. The second constraint of this research is that the intention is not to break the SUT. So the SQL injection method used should be non destructive.

Kals et al. offered a solution to this problem in their paper Secubat [KKKJ06]. They proposed that breaking an SQL injection string could determine the presence of SQL injection vulnerabilities. This eliminated the need of actually running SQL queries on the database. Instead they introduced a quote in the injection string. If the input value was a normal string, the query would look like this:

```
1 SELECT * FROM user where username = 'user123' limit 1;
2 SELECT * FROM user where value = '12'3' limit 1;
```

If the input value contained classic SQL injection like "' OR 1 = 1'", the query would look like this:

```
1 SELECT * FROM user where username = '' OR 1 = 1'' limit 1;
```

But if the SQL injection just contains a single quote, the query would look like this:

```
1 SELECT * FROM user where value = ''' limit 1;
```

The main difference between the first two queries and the last query is that the first two queries can actually be executed. For the last query, that is not possible. The SQL server will trow an exception, and the web server will not be able to execute the request. Because this

[3] https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html

exception is unexpected, the request will likely return an HTTP 500 exception, indicating that something went wrong on the server side.

For detecting SQL injection vulnerabilities with TESTAR using this method, an HTTP listener needs to be added to WebDriver, this listener listens to the responses after an SQL injection attempt was done. A code example getting the status code with WebDriver can be found below.

```
public void addListener(DevTools devtools)
{
    devTools.addListener(Network.responseReceivedExtraInfo(),
        responseReceived -> {
            int statusCode = responseReceived.getStatusCode();
            // process process status code
        });
}
```

Because the injection of the value and sending the value to the server can occur in different inner loops, it is not enough to analyse the exceptions only when SQL injection is performed. If the injection is done on a form field that is submitted many inner loops later, the relevant response is the response of the submit request. To mitigate this issue, TESTAR looks for 500 responses for 10 inner loops after the injection attempt.

A second limitation of this approach is that the escape character used in this case might be different on different databases. This means that the approach has to be extended to include more escape characters, or be altered depending on the type of database the SUT uses. This limitation is accepted within the scope of this thesis and no further effort was taken to mitigate it.

One of the options looked into for detecting SQL injection vulnerabilities was the analysis of stack traces. The stack traces contain detailed information about what exactly went wrong in the system. By analysing the stack trace it is possible to determine if the error derived from SQL injection. However, it is best practice in software engineering to not expose the stack traces. This is due to the fact that they give attackers a lot of information about bugs in the system, increasing the chance of exploitation. In cases were the stack traces were exposed however, they could have decreased the false positive rate. Stack trace analysis was still not implemented because different frameworks and languages generate different stack traces. This makes determining if a given stack trace is originating from an SQL injection difficult. Because the stack traces should not be exposed in the first place and the complexity of analysing them, analysing the stack traces was deemed out of scope for this research.

TESTAR does not determine what the cause of the 500 exception was, it only detects that a 500 exception has occurred after an injection attempt. This means that unrelated 500 exceptions could be flagged as possible injection vulnerabilities. To prevent this in the future, scenario tests can be used, where TESTAR finds out what field is responsible for the 500 exception. After that, TESTAR can inject different values to determine if the 500 is SQL injection related. An example of such an injection is "' and 1 = 1'", this performs SQL injection but is not expected to trigger a 500 exception while doing so. If a 500 exception is still returned, the exception is not likely to indicate an SQL injection vulnerability, but originates from something else. A code example of that query can be seen below.

```
SELECT * FROM user where value = '' and 1 = 1'' limit 1;
```

5.3.4. SESSION TOKEN INVALIDATION

Session token invalidation is a vulnerability where a lot of TESTAR's abilities come together. TESTAR needs to navigate, read cookies, set cookies, login, log out and do all of this automatically based on a predefined sequence. A flow diagram of this predefined sequence is shown in Figure 5.1. This predefined sequence will be elaborated further in the following section.

This sequence starts by logging in to the website. At this point, session tokens will be set as cookies. Because it is unknown to TESTAR at this point, which of the cookies is the session token, TESTAR will copy all of the cookies and store them. TESTAR will also store the current URL to redirect to later. After this, TESTAR will logout of the web application. At this point, the session tokens should be invalid. The cookies snapshot will be restored to the logged in state. A code sample for taking and restoring cookies snapshots using WebDriver is shown below.

```
private Set<Cookie> cookies;

public void takeCookieSnapshot(WebDriver webDriver)
{
    cookies = webDriver.manage().getCookies();
}

public void restoreCookieSnapshot(WebDriver webDriver)
{
    webDriver.manage().deleteAllCookies();
    for (Cookie cookie : cookies) {
        webDriver.manage().addCookie(cookie);
    }
}
```

After restoring the cookies to their logged in state, TESTAR will try to navigate to the stored URL. This stored URL is within a session that has been invalidated, so TESTAR should not be able to redirect there. If TESTAR is indeed unable to, TESTAR has detected session token invalidation. If TESTAR is able to navigate to the URL, this does not mean the session token is not invalidated. In the case of distributed systems, where the backend is run on multiple servers at the same time, it could take some time for the services to sync up. To avoid false positives, TESTAR will wait a minute for the servers to spread the token invalidation. After this minute, TESTAR will try to reload the page. If TESTAR is able to stay within the portal, it is likely that the session token is not invalidated.

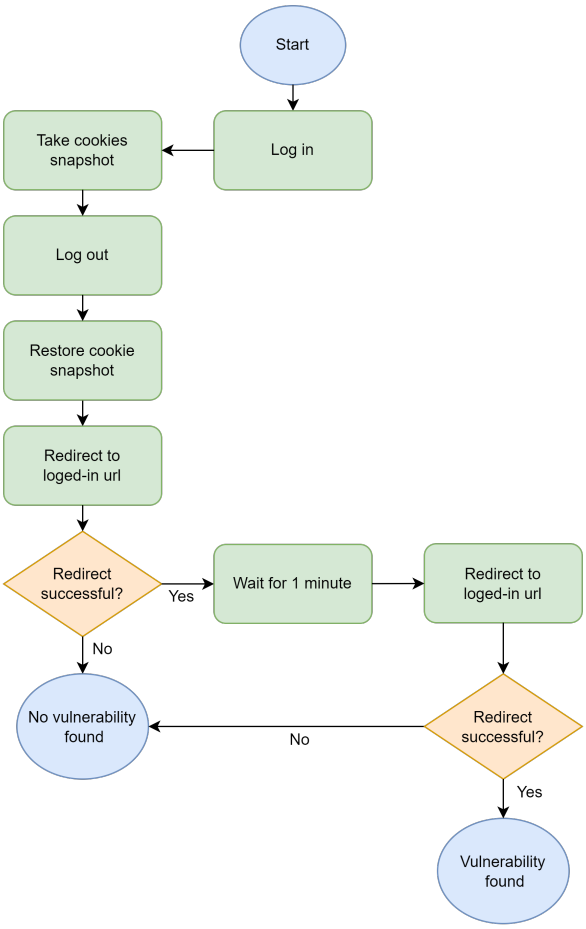

Figure 5.1: An abstract representation of a token invalidation detection algorithm.

6

DESIGN & DEVELOPMENT

In this chapter the implementation aspects of security analysis with TESTAR will be discussed.

6.1. FRAMEWORK

6.1.1. SECURITY ORACLES

To make the current security analysis possible and allow for future extensions, a framework was created. This framework introduces security oracles, which are small self contained units of code that allow for scanning for security vulnerabilities. These security oracles come in two flavors: passive security oracles, and active security oracles.

Passive security oracles have the ability to add listeners to the WebDriver instance. And to come to a verdict based on the information obtained. One of the implementations of the passive oracle is the HTTP header analysis oracles. The oracle adds a listener to the network traffic of the WebDriver and reads out all the headers. This process is a constant process happening in the background and not interfering with any other part of the testing. When it is time to come to a verdict, the HTTP headers are analysed and added to the existing verdict. Because this process does not interfere with the rest of the system, multiple passive oracles can run at the same time.

Active security oracles can do every thing passive oracles can. They are also able to introduce their own actions, pre-select the actions and have the knowledge which action is selected for execution. This enables the oracle to actively influence the testing process. An example of such an oracle is the XSS analysis oracle. This oracle adds two types of actions: form field inputs and URL inputs. These actions are combined in the 'deriveActions' method with the actions proposed by the protocol. The second interaction the oracle has is in the pre-selection process. The XSS security oracle prefers form inputs over URL inputs, so if both are present in at this stage, the oracle will remove the URL inputs. To avoid interfering, only one active oracle can be used at a given time. To help with the visualization, a simple UML diagram of the oracle structure as been added 6.1.

The security oracles are not limited to one inner loop, this makes it possible to answer security questions that require a multi stage scenario. An example of such a question is "Do session tokens get invalidated after logout?". Answering this question required the oracles to log in, read cookies, log out again, set the cookies and redirect. This sequence of

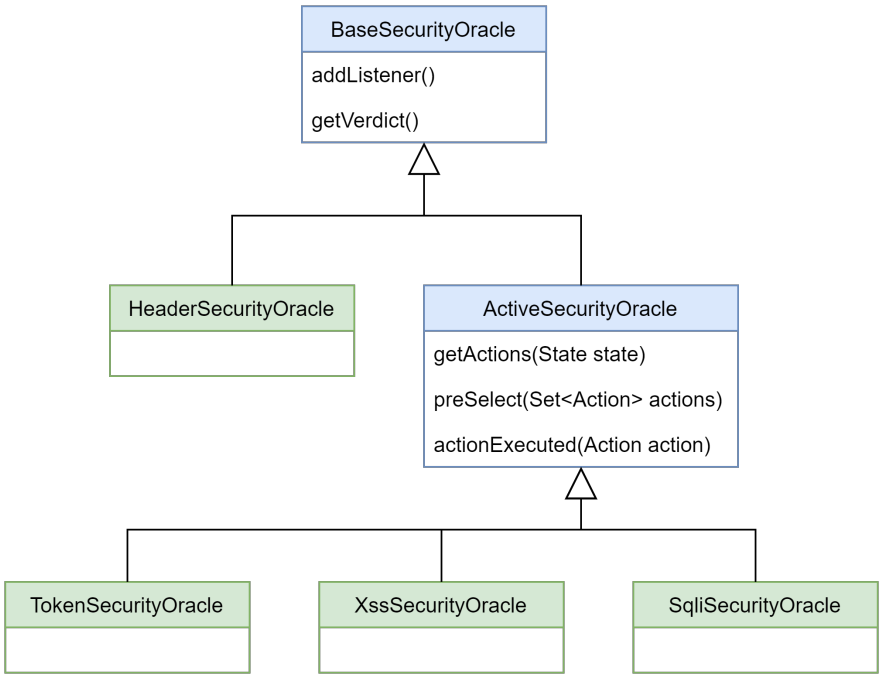

Figure 6.1: An abstract representation of the oracle structure.

event can all be programmed into the same oracle. By defining the action that needs to be executed in each state and keeping count on what actions have been executed, the oracle is able to make TESTAR run the required scenario's. The oracle can store all the data and give a verdict at the end of the sequence.

A key feature of the security oracles is the extensibility they offer. Because the security oracles are self-contained, no knowledge of the inner workings of TESTAR is required. Even the protocol or navigational algorithm used is irrelevant for the oracle. The oracle knows as little as possible, and the surroundings know as little as possible about the oracle. This results in simple oracles that can be written in as little as ten minutes, depending on the complexity of the question.

In summary, the security oracles created for this research enable TESTAR to execute and answer predefined security oracles. The passive security oracles can answer questions without interacting with the SUT, that enables them to run during any GUI testing session. Active security oracles can interact with the SUT, enabling them to answer more complex security questions. Because of the decoupled and simple nature of the oracles, creating new oracles requires limited time and complexity.

6.1.2. ORACLE ORCHESTRATOR

To connect the oracles to the protocol, the oracle orchestrator was introduced. The oracle orchestrator acts as a broker between multiple oracles and the protocol. For every stage of the TESTAR inner loop, the orchestrator is called. It is the job of the orchestrator to pass these calls on to the oracles. The orchestrator is in charge of the creation of the oracles, and the interaction of TESTAR with the oracles. A sequence diagram of this interaction is included in Figure 6.2.

The decision to decouple the protocol and the oracles was made to weaken the cou-

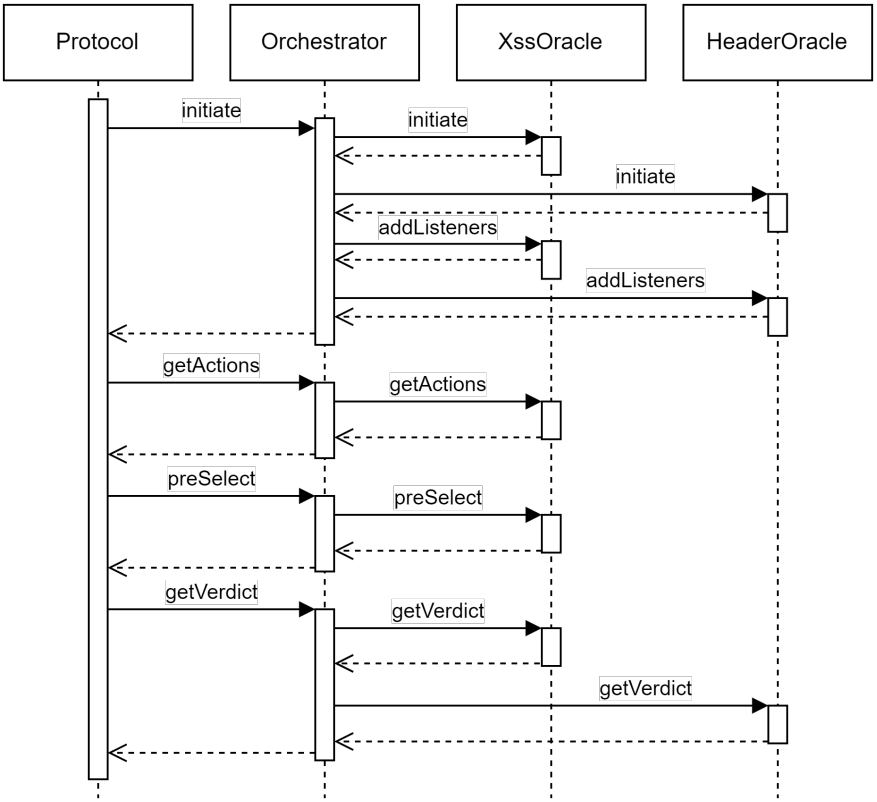

Figure 6.2: A sequence diagram of the interaction between the protocol and oracles.

pling between the exploration algorithm and the security oracles. This is not only desirable for future flexibility, it was required for this research to. Because of the unnatural structure of the benchmark, TESTAR was extended with a separate algorithm for exploring the it. This way the validation of the oracles could be separated from the validation of TESTAR's algorithms.

The implementation of the orchestrator can even be done at the 'WebDriverProtocol' level, the base class all WebDriver protocols extend from. This enables any protocol to run any oracle, just by adding them to the configuration file. This is especially useful for the passive oracles, because it would be desirable to be able to use them during normal GUI testing runs. The active oracles are less suitable for combining with normal GUI testing, because of the interference in the action selection process.

7

RESULTS

This chapter contains the results of the experiments proposed in the methodology chapter. To give an overview of the experiments that have been executed, the experiments are listed once again below in Table 7.1.

Nr	experiment	SUT
1	HTTP headers	OWASP Benchmark
2	XSS	OWASP Benchmark
3	SQL injection	OWASP Benchmark
4	Code coverage	Blogifier
5	Token invalidation	Custom application
6	Setup time	Blogifier

Table 7.1: Experiments

7.1. EXPERIMENT RESULTS

7.1.1. BENCHMARK RESULTS

Experiment	test cases	vulnerable test cases	true positive	false positive
TESTAR				
Insecure Cookie	67	36	36	0
XSS	455	246	156	0
SQL injection	504	272	222	52
OWASP ZAP				
Insecure Cookie	67	36	36	0
XSS	455	246	179	0
SQL injection	504	272	167	26

Table 7.2: Benchmark results absolute

The first three experiments all use the OWASP benchmark to validate TESTAR's ability to find vulnerabilities. The first Table 7.2 contains the number of test cases and the number of vulnerable test cases in the benchmark, as well as the performance of TESTAR in those test cases. The second Table 7.3 contains the scores from all the tools tested against the benchmark.

Category	true positive	false positive	runtime (h:mm)	Sequences
TESTAR				
Insecure Cookie	100%	0%	0:15	1.000
XSS	63%	0%	4:12	15.000
SQL injection	82%	22%	3:03	10.000
OWASP ZAP				
Insecure Cookie	100%	0%	0:01	N/A
XSS	73%	0%	0:28	N/A
SQL injection	61%	11%	0:01	N/A

Table 7.3: Benchmark results relative

7.1.2. CODE COVERAGE

The fourth experiment measures the code coverage of TESTAR and OWASP ZAP, the results of this experiment are shown below in Table 7.4.

Tool	section	Classes	Methods	Branches	Time (h:mm) Sequences
TESTAR, webdriver_generic					
1	(71 of 295)	(217 of 780)	(426 of 1.613)	0:02	100
2	(71 of 295)	(217 of 780)	(426 of 1.613)	0:03	100
3	(71 of 295)	(217 of 780)	(426 of 1.613)	0:05	200
4	(71 of 295)	(217 of 780)	(426 of 1.613)	0:05	200
TESTAR, webdriver_statemodel					
1	(71 of 295)	(217 of 780)	(426 of 1.613)	0:08	100
2	(71 of 295)	(217 of 780)	(426 of 1.613)	0:07	100
3	(71 of 295)	(217 of 780)	(426 of 1.613)	0:16	200
4	(71 of 295)	(217 of 780)	(426 of 1.613)	0:15	200
OWASP ZAP					
1	(84 of 295)	(237 of 780)	(465 of 1.613)	0:06	N/A
2	(84 of 295)	(237 of 780)	(465 of 1.613)	0:06	N/A

Table 7.4: Code coverage

7.1.3. SETUP TIME

QUICK SETUP

For the first part of this experiment the quick setup of both TESTAR and OWASP ZAP is measured in actions required. The steps required by both tools are listed below in Table 7.5.

Step	TESTAR	ZAP
1	Start TESTAR	Start ZAP
2	Select 'security analysis'	Select 'Automated Scan'
3	Enter test URL	Enter test URL
4	Click the 'Start generate'	Click 'Attack'

Table 7.5: Quick setup

LOGIN

For login scenarios, both tools required additional setup. This setup is described in Appendix 9.9.

7.1.4. TOKEN INVALIDATION

To validate the token invalidation detection abilities of TESTAR, three test cases were created. These test cases were run three times on three different versions of the same application. The results of this experiment are listed in Table 7.6.

Run	successful tests
1	(3 of 3)
2	(3 of 3)
3	(3 of 3)

Table 7.6: Token invalidation results

7.2. ANALYSIS OF RESULTS

7.2.1. HTTP HEADERS

TESTAR was able to detect 100% of the Insecure Cookie flags correctly, with no false positives. Because TESTAR detects all HTTP header data the same way, this result should be representative for the ability of TESTAR to detect and validate HTTP headers. The performance difference between the detection of the SET-Cookie secure flag and any other HTTP header should come down to the oracle used.

The algorithm TESTAR is using to detect HTTP headers is only listening and not interacting with the system. Because of this, the header detection algorithm could be used during regular GUI testing without influencing the results. Because of the high accuracy of the algorithm, the interpretation of the results should not be labor intensive.

The performance of TESTAR is comparable to that of OWASP ZAP. Once the HTTP call is made, it is very easy to validate the headers. The challenge is in the ability to navigate

the website completely, making sure the call is actually executed. Because of this, it is not surprising that TESTAR performs good in this category.

What TESTAR does not do great in this category in comparison to OWASP ZAP is the runtime. TESTAR takes a lot longer to run the HEADER analysis than OWASP ZAP does. However, this is not a problem for TESTAR because it can analyse the HEADERS during normal GUI testing. This introduces no additional runtime in comparison to not analysing the HTTP headers.

In conclusion, TESTAR is able to analyse HTTP headers as well as a dedicated security analysis tool. With the added efficiency of combining this analysis with regular GUI testing, making it take no additional time. Finally, because of the accuracy of the results, interpretation should not require more resources than with dedicated tooling.

7.2.2. Cross Site Scripting

TESTAR is able to find 63% of the XSS vulnerabilities in the OWASP benchmark without any false positives. While this percentage is not impressive in itself, is does mean that TESTAR is able to detect a large part of the easy to detect XSS. In comparison with the 73% detection rate of a dedicated DAST tool, the 63% of TESTAR is very impressive.

There are a few known problems with the way TESTAR detects XSS, and some of these problems can explain this difference. First, TESTAR does not use encoded XSS or filter evasion. This means that if the application filters out the script tags from the input, TESTAR will not be able to detect XSS vulnerabilities. This is something OWASP ZAP does a better job at. TESTAR will however be able to overcome this obstacle in the future by extending the list of possible injections.

The second problem is the difference in runtime. While OWASP ZAP took very long to analyse XSS in comparison to the other vulnerabilities, TESTAR took almost ten times as long. This is due to the fact that TESTAR interacts with an application in a way a GUI tester does, making it hard to overcome this issue. One of the ways this could be mitigated in the future is by running multiple TESTAR instances in parallel. This would enable TESTAR to run one main instance for exploring the application, and background instances to execute security oracles on the widgets found. While this could reduce TESTAR's runtime by a lot, it is unlikely that is will come close the that of OWASP ZAP.

The final problem with the current implementation in TESTAR is the fact that it only tests for reflected XSS. This means that stored XSS vulnerabilities are not detected. To test the stored XSS in a way it would be used by attackers using TESTAR is very hard because an undefined sequence of actions needs to be executed to use it. It would however be possible to inject data coming from the server with XSS injection, determining if the front end is susceptible at all.

In conclusion, while TESTAR has the potential to replace dedicated tooling for detecting XSS vulnerabilities in the future, it still has a long way to go. TESTAR should make use of encrypted XSS and detect reflected XSS to improve the results. TESTAR should also be optimized to reduce the runtime. However, the fact that TESTAR's results are less complete than the ones of OWASP ZAP, they are still beneficial to the security of the SUT.

7.2.3. SQL injection

TESTAR was able to detect 82% of the SQL injection vulnerabilities correctly. That result is very high. However TESTAR also has a false positive rate of 22%. In the benchmark, where

the ratio between vulnerable and non vulnerable test cases is roughly 50/50, this means TESTAR is mostly right. However, in a real world scenario, the amount of positive cases is much lower in comparison. This would mean that the results will consist of mostly false negatives.

In comparison to the 61% true positive and 11% false positive rates of the dedicated DAST tool, TESTAR´s performance is not inferior to the dedicated DAST tool. While the true positive rate of TESTAR is higher, this does not mean that TESTAR is performing better. The false positive rate seems to be the problem while finding this vulnerability. With a better implementation of the SQL injection oracle, TESTAR could be able to perform on the same level as OWASP ZAP.

One of the problems that leads to a high false positive rating in TESTAR is the inability to detect exactly where the SQL injection is coming form. This is a problem in the implementation and not a fundamental problem with TESTAR. If the implementation is altered to make TESTAR aware where the SQL injection is coming from, it would be able to test the same field with multiple values. Testing the same field with multiple value before deciding a verdict would enable TESTAR to achieve a much lower false positive rate.

In summary, TESTAR is able to detect SQL injection, but not on a level that would produce easy to interpret results. TESTAR is able to express suspicions about SQL injection vulnerabilities, but they will most likely be wrong a large portion of the time in a real world scenario. However, TESTAR does have the potential to improve by a lot, but this would require a more refined implementation of the SQL injection oracle.

7.2.4. CODE COVERAGE

The code coverage experiment was added to measure how much of an application TESTAR discovered in comparison to OWASP ZAP. It is important to note that, when analysing these results, the absolute values are irrelevant. It is unknown what the theoretical maximum values are that are discoverable, and how much of the code is relevant for finding security vulnerabilities. What it does show however, is how much of the application TESTAR is able to discover in comparison to OWASP ZAP. For this tests, TESTAR used the 'webdriver_generic' and 'webdriver_statemodel' protocols with a sequence limits of 100 and 200.

The first thing that stands out is the lack of deviation in TESTAR's runs using different configurations. This indicates that the sequence limit and protocol are not limiting TESTAR's ability to discover the application in this experiment.

Secondly, TESTAR covers significantly less of the application than OWASP ZAP. This could be due to the fact that OWASP ZAP does not use the GUI in the same way and could therefore touch parts of the system TESTAR could not. This also shows the fundamental limitation of security analysis using a GUI testing tool. The attack service analysed will always be limited to the GUI. Endpoints that still exist, but are not used in the GUI anymore, are potential weaknesses TESTAR is unable to find.

Based on these results it can be argued that a security analysis using TESTAR touches a large portion of the system that an analysis using a dedicated tool would. By combining this information with the benchmark results, it is possible to draw a conclusion about TESTAR's abilities as a security analysis tool. It can be concluded that, while TESTAR is worse than dedicated tools, it should be able to find a significant part of the vulnerabilities in a system.

However, the experiment itself does not tell the whole story. Stability issues while running with OpenCover limited the number of runs that was executed. On top of that there

was an inability to find SUTS that were compatible with OpenCover and fit for this research. Because of this, there was only one application used for this experiment. To get a really good picture of TESTAR's ability to explore the application, more SUTs need to be tested and more runs need to be done. However this did not fit into the time limitations of this thesis.

In conclusion, the results of the code coverage experiment show that TESTAR is able to discover a large part of the application that OWASP ZAP did. The code coverage results in combination with the benchmark results show that TESTAR is able to find a large portion of the vulnerabilities a dedicated tool would find. However, the amount of runs and SUTs in this experiment is low and more SUTs and runs would yield more meaningful results.

7.2.5. SETUP TIME

To show that it is easier to use TESTAR than to use a dedicated security analysis tool, we have compared the setup actions required for both TESTAR and OWASP ZAP in this experiment. The experiment measures the action required to setup the tools for a new SUT, this does not include any part of the installation process. Both tools need 4 basic steps to run on a new SUT. Both tools need very limited setup and are very similar in effort. A side note is that the OWASP ZAP application requires the user to select the parts of the application to analyse after crawling the application. This is not a SUT specific setup step, but should be mentioned.

The second part of the experiment looked at the setup of automated login for a new SUT. This is where the tools diverge in both approach and effort. TESTAR requires the user to define a pre-specified action and place that action in the startup process of a protocol. The user might need to alter the id of the field, depending if the SUT fields contain the username and password. The user also needs to enter the name of the form and their username and password. This means there are actually three steps: 1. inspect the application in the browser, 2. find out how the username, password and form elements are called, 3. copy and paste the code from Vos et al. [VAR+21] to the startup of the TESTAR's protocol and 4. add the element names and the username and password to the code.

OWASP ZAP requires 12 steps to define a login sequence as seen in Table 2. The advantage of the setup with ZAP is that it does not require the user to touch any code, all the steps can be achieved using the user interface. A second advantage is that OWASP ZAP does not require the user to search for field names by inspecting the elements of the SUT.

In conclusion, setting up a new SUT is equally easy for both tools. There were big differences in setting up the login process. It required a little more skill and knowledge to setup a login sequence in TESTAR. However, setting up the login process in OWASP ZAP required a lot more effort.

7.2.6. TOKEN INVALIDATION

The token invalidation experiment shows whether TESTAR is able to detect whether a token is invalidated on logout of not. In all three test cases in all three test runs TESTAR was able to detect the token invalidation correctly.

The application that was used for this experiment was altered to create the three different test cases that perfectly fit TESTAR's abilities. A benchmark or a series of real world application would have given more meaningful results about TESTAR's ability. A benchmark that tested this ability could however not be found and creating one fell outside of the time

constraints of this thesis. Manipulating the cookies in real world applications without permission is an action that could have unforeseen consequences. That is why the decision was made to create three test cases.

This means that this experiment does not show in what percentage of applications TESTAR would be able to detect session token invalidation. But it does show that TESTAR is able to detect session token invalidation in some cases.

A limitation of the current implementation is that it will only work if the session token is stored as a cookie. If local session storage is used, TESTAR is unable to do so. The second limitation of this approach is that TESTAR needs to be pointed at the login and logout buttons in the SUT. This means that analysing session token invalidation cannot be done entirely automated yet.

None of the DAST tools researched for this research had the ability to validate session token invalidation by default. This means TESTAR would be able to do something that the most popular dedicated security analysis tools cannot. However, it is possible to add custom validation rules to existing tools, making it in theory possible to analyse session token invalidation with those tools as well. This would be functionality that has to be implemented by the user, so that TESTAR would be able to do this out of the box would be a first.

In conclusion, session token invalidation is possible using TESTAR. To know on what percentage of websites this will work more research is required. A disadvantage of this feature is that some setup is required. This experiment does however show that TESTAR is fit for automated scenario testing, more scenario tests to be added in the future. But the most meaningful result of this development is that if this ability makes it in a production version of TESTAR, TESTAR would be the first tool that can validate session token invalidation out of the box.

8

CONCLUSION

This research aimed to demonstrate the potential role TESTAR could play in security analysis. Based on the security analysis executed with TESTAR and the comparison to existing security analysis tools, it can be concluded that TESTAR can contribute to the field of security analysis. In particular that TESTAR is able to do security analysis and that security analysis by TESTAR is beneficial to the security of web applications.

The reason security analysis with TESTAR, or GUI testing in general, is so appealing is that there is already interaction with the system. Being able to use this existing interaction to reduce the number of vulnerabilities in a web application, instead of relying on the presence of dedicated security analysis, could have serious impact on the security of web applications. In this research, TESTAR was extended to actively analyse SQL injection and XSS vulnerabilities, as well as passive analysis of HTTP headers.

To enable TESTAR to find these security vulnerabilities, a framework was created. This framework introduced security oracles into TESTAR. These self contained units of code enable TESTAR to answer specific security question and enable TESTAR to be extended to find new vulnerabilities in the future. The security oracles come in two forms: passive oracles and active oracles. The passive oracles run in the background, collecting data without influencing the interaction with the system. The active oracles are able to influence the actions TESTAR is executing. This enables the active oracles to do more complex forms of security analysis, like finding injection vulnerabilities.

Exceeding or matching dedicated security analysis tools in performance was never the objective. The aim was to reliably finding a large enough portion of security vulnerabilities to give security analysis with TESTAR real world value. The OWASP benchmark was used to validate TESTAR's ability to find a vulnerability when encountered. This resulted in a 100% score for header analysis, 63% score for XSS vulnerability detection and a 82% score for SQL injection vulnerability detection. With 0% false positives for both header analysis and XSS analysis, and a 22% false positive rate for SQL injection analysis. Where the dedicated security analysis tools scored, on average, slightly higher. With the same 100% score for header analysis, 73% of XSS vulnerabilities were found and 61% of SLQ injection vulnerabilities with a false positive rate of only 11% for SQL injection.

These results show that TESTAR is not only able to find SQL injection vulnerabilities, it also has the potential to find even more vulnerabilities with fewer false positives. While the ability of dedicated tools exceeds TESTAR's abilities, the difference is smaller than expected and there is more potential in TESTAR's abilities. Further more, GUI testing is expected to be an excellent way of detecting XSS vulnerabilities, because TESTAR has both access to the GUI, its configurations and the network traffic. This gives TESTAR the potential to not only test for XSS vulnerabilities, but be one of the best tools for detecting XSS vulnerabilities. The current implementation however, is still lacking behind the implementation of the dedicated security tool.

TESTAR has proven itself to be an excellent tool for passive security analysis. Because it is able to listen to interaction that is already provided by TESTAR, it is able to passively analyse certain vulnerabilities at negligible additional costs to the runtime. TESTAR's algorithm is specialized in exploring web application. This enables passive analysis to have the potential to find a large part of the, from the GUI detectable, misconfigurations in an application. In this research TESTAR is only able to analyse the headers, however there is a lot more that can be analysed passively that would increase TESTAR's impact on application security.

The second validation for TESTAR's added value to application security, is the code coverage. This indicates how much of the application TESTAR is able to touch while testing. By combining the code coverage of TESTAR with the results from the benchmark, it is possible to say something about performance in the real world. For this research, the code coverage of a C# application was measured using OpenCover for both TESTAR and dedicated security analysis tools. This is where the real limitations of TESTAR were shown. TESTAR is fundamentally limited to the GUI. The security analysis tools were able to analyse wherever they were pointed, TESTAR was limited by what it could interact with trough the GUI. This shows that, while TESTAR is very useful for testing the security vulnerabilities in the GUI, it is less useful for testing the security vulnerabilities in the entire system.

The third validation came in the form of the setup. For TESTAR to have added value for the security of a web application, it is important that the use of TESTAR has benefits over the use of a dedicated security analysis tool. This benefit came in the form of setup. While all the security analysis tools were relatively easy to setup, they all needed additional configuration to deliver the best results. This configuration takes both time and knowledge that is not by definition available. Being able to run the security analysis with TESTAR during the GUI testing process, takes these limitations away. These findings have shown that, while security wise the results would be better using a dedicated security analysis tool, using TESTAR for security analysis can offer a solution better tooling or knowledge is not available.

The final validation was to validate TESTAR's ability to find token invalidation vulnerabilities. This does not only validate TESTAR's ability to find these vulnerabilities, it also shows the frameworks ability to answer more complex security questions. Because the experiment existed of three generic test cases, it did not show the real world performance of TESTAR's token invalidation oracle. However, it did show that TESTAR is able to analyse this vulnerability in a synthetic environment. None of the other tools used for this research

were able to analyse this vulnerability out of the box. This means that TESTAR is able to do security analysis that is not feasible with dedicated security tooling.

So using TESTAR for security analysis has a positive impact on the security of the systems that are tested with TESTAR, especially if a dedicated security analysis tool is not used. This paper shows that TESTAR can be great at certain parts of security analysis (like analysing headers), and sufficient in others (like SQL injection). The comparison in configuration to an existing security analysis tool shows that TESTAR has a role to play in this field. Because of the middle ground TESTAR offers between no security analysis and security analysis with a dedicated security analysis tool.

9

FUTURE WORK

9.1. FINDING MORE VULNERABILITIES WITH TESTAR

One of the features considered in this research was the detection of exposed session identifiers in the URL. Because of the complexity of accurately identifying session identifiers, this feature did not make it into this research. However, this would be a nice extension to TESTAR, because it would be able to run passively during a normal GUI-testing session.

More vulnerabilities detection oracles were proposed but not implemented during this research. The complete list is included in the table below 9.1.

Nr	description
Actively analysable vulnerabilities	
1	Missing access controls
2	View or edit someone else's account
3	URL modification
4	Elevation of privilege
9	Enabled default accounts
13	Brute force protection
14	Automated attack protection
15	Weak password rules
Passively analysable vulnerabilities	
10	Stack-trace reveals
16	Exposed session identifiers

Table 9.1: Vulnerabilities that could have been analysed

9.2. TAINT ANALYSIS

Taint analysis is a great way of detecting injection vulnerabilities. Aside from the complexity, the need for an extra program to monitor runtime performance of the SUT is an obstacle. A different, yet still complex approach, would be a way of doing taint analysis using log files. The problem with the current injection detection of TESTAR is that it has a

lot of false positives. Mainly because TESTAR is unable to determine whether there really is a vulnerability or not. Using taint analysis in the right way would enable TESTAR to confirm the existence of injection vulnerabilities with a very high level of precision. This would take away the need to use a second tool to confirm the suspicions that TESTAR has.

9.3. Going beyond WebDriver

The abilities of TESTAR are limited by the functionalities that WebDriver exposes to the user. One of the limitations of WebDriver we ran into during this research was that WebDriver does not give detailed information about the encryption used. Because of this, TESTAR is not able to assess the quality of the encryption used. Another limitation is the lack of ability to manipulate the network traffic. Being able to manipulate the network traffic would enable TESTAR to test stored XSS in a more reliable way. Extending TESTAR with a proxy that enables TESTAR to read and manipulate all aspects of the network traffic would be a step up for TESTAR's security analysis abilities.

9.4. Improving SQL injection

TESTAR is able to detect SQL injection in a very rudimentary way. By triggering an injection, TESTAR is able to estimate the chance of an SQL injection vulnerability being present. While this method shows that TESTAR is fundamentally able to detect SQL injection vulnerabilities at some level, it does not deliver the maximum potential of TESTAR testing for SQL injections. TESTAR could be extended with more escape characters, like " and '. There could also be more checks to deliver a lower false positive rate. For example by comparing the results of different injection statements. This means that a potential avenue for future research would be a deep dive into SQL injection methods with TESTAR.

9.5. Splitting explorer and active analysis

In the current state, TESTAR will explore the GUI and during this exploration, it will try security tests. This means that every check will be executed in series, slowing the testing process with every oracle added. By splitting the scanning and pen-testing processes, this time penalty could be mitigated, executing these tasks in parallel. When the TESTAR explorer finds an opportunity for pen-testing, one or more new instances of TESTAR could be started to pen-test the specific opportunity in the background. There could be a background pool of TESTAR instances to pick up these tasks.

9.6. Attack service

While TESTAR is able to analyse security vulnerabilities and has the potential of analysing security vulnerabilities a lot better than it can now, there will always remain a limitation. Because TESTAR is a GUI testing tool, it makes sense to be limited to a single attack service, the GUI. What is unknown however, is the impact vulnerabilities by attack service. Finding out the impact of a vulnerability that is accessible through the GUI, compared to a vulnerability that is not, would help understand the role that TESTAR can play in the overall security of web applications.

9.7. MITIGATION DETECTION

Because TESTAR is limited by the GUI for detecting security, it could be fooled by front-end mitigation of vulnerabilities. For example, if a front-end sanitizes every request before it is sent to the back-end, the detection of SQL injection through the front-end becomes impossible. This however, would not mean that there is no vulnerability. The existence of such a system could be detected by TESTAR, by analysing network traffic, TESTAR could compare the network request to the back-end with the input. If the request to the back-end lacks the special characters, this would indicate that SQL injection mitigation is handled in the front-end, requiring additional attention from dedicated tooling to estimate the real risk of SQL injection vulnerabilities.

9.8. MORE COMPLETE VALIDATION

The current research compares TESTAR with a single dedicated security analysis tool on a single benchmark, and measures code coverage on a single application. The reason for this were the time constraints for this project. However, to really test TESTAR's abilities, a broader comparison is necessary. Because of the stability issues experienced using OpenCover during this research, using JaCoCo in a new attempt is recommended.

For this research, the validation was split into a benchmark part and a code coverage part. The combination of these results gave an estimation of how good TESTAR would be in the real world. It would however be an interesting addition to validate TESTAR on real world applications.

9.9. BROWSER EXTENSION FOR PASSIVE ANALYSIS

TESTAR shows in this research that it is possible to find certain security vulnerabilities from the background during normal testing sequences. This means that these vulnerabilities could be detected in the same way during normal usage of an application. A really cool next step for this research would be an implementation as a browser extension, that validates passive security aspect during normal browsing.

The extension could give each website a score based on how many of the vulnerabilities were found, making the user aware of the security state of the website used. This extension could also be used to gather statistics about vulnerable websites of notify the owners of a website if vulnerabilities are found. A problem with this research is that processing this data for analysis or notification would require processing part of the users web activity, which is undesirable from a security and privacy perspective.

BIBLIOGRAPHY

[ACKT20] Richard Amankwah, Jinfu Chen, Patrick Kwaku Kudjo, and Dave Towey. An empirical comparison of commercial and open-source web vulnerability scanners. *Softw. Pract. Exp.*, 50(9):1842–1857, 2020. 13

[CA06] Mark Curphey and Rudolph Arawo. Web application security assessment tools. *IEEE Secur. Priv.*, 4(4):32–41, 2006. 15

[CFN+19] Stefano Calzavara, Riccardo Focardi, Matús Nemec, Alvise Rabitti, and Marco Squarcina. Postcards from the post-http world: Amplification of HTTPS vulnerabilities in the web ecosystem. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 281–298. IEEE, 2019. 12

[CFST17] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. Surviving the web: A journey into web session security. *ACM Comput. Surv.*, 50(1):13:1–13:34, 2017. 12

[CJKR21] Stefano Calzavara, Hugo Jonker, Benjamin Krumnow, and Alvise Rabitti. Measuring web session security at scale. *Comput. Secur.*, 111:102472, 2021. 12

[CSX+18] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. An automated approach to estimating code coverage measures via execution logs. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 305–316. ACM, 2018. 14

[DIP20] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1953–1970. ACM, 2020. 11, 12

[GRG+19] Bernhard Garn, Marco Radavelli, Angelo Gargantini, Manuel Leithner, and Dimitris E. Simos. A fault-driven combinatorial process for model evolution in XSS vulnerability detection. In Franz Wotawa, Gerhard Friedrich, Ingo Pill, Roxane Koitz-Hristov, and Moonis Ali, editors, *Advances and Trends in Artificial Intelligence. From Theory to Practice - 32nd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2019, Graz, Austria, July 9-11, 2019, Proceedings*, volume 11606 of *Lecture Notes in Computer Science*, pages 207–215. Springer, 2019. 11

[HCF05] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *21st Annual Computer Security Applications Conference (ACSAC*

2005), 5-9 December 2005, Tucson, AZ, USA, pages 303–311. IEEE Computer Society, 2005. 10, 18

[HGB+19] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergõ Balogh, and Tibor Gyimóthy. Code coverage differences of java bytecode and source code instrumentation tools. *Softw. Qual. J.*, 27(1):79–123, 2019. 14

[JKKS20] Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and Marc Sleegers. Shepherd: a generic approach to automating website login. In *Proc. 2nd Workshop on Measurements, Attacks and Defenses for the Web (MADWEB'20)*, pages 1–10. IEEE, 2020. 12

[KKKJ06] Stefan Kals, Engin Kirda, Christopher Krügel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006*, pages 247–256. ACM, 2006. 10, 11, 19, 27, 31, 32, 33

[LCA+18] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William K. Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *CoRR*, abs/1811.00918, 2018. 12, 20

[Moz] Mozilla HTTP headers. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers. Accessed: 11-08-2022. 30

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electron. Notes Theor. Comput. Sci.*, 89(2):44–66, 2003. 10

[NS05] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. The Internet Society, 2005. 10

[OWA] Owasp zap. https://www.zaproxy.org/. Accessed: 10-06-2022. 6, 28

[Owa16] Owasp benchmark project. https://owasp.org/www-project-benchmark/, 2016. Accessed: 11-08-22. 7, 13

[owa21] Owasp top 10 - 2021. https://owasp.org/Top10/, 2021. Accessed: 11-08-22. 16, 21, 23, iv, v

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 317–331. IEEE Computer Society, 2010. 10

[SDM20] Justin Smith, Lisa Nguyen Quang Do, and Emerson R. Murphy-Hill. Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In Heather Richter Lipford and Sonia Chiasson, editors, *Sixteenth Symposium on Usable Privacy and Security, SOUPS 2020, August 7-11, 2020*, pages 221–238. USENIX Association, 2020. 2, 14

[SIS18] Lim Kah Seng, Norafida Ithnin, and Syed Zainudeen Mohd Said. The approaches to quantify web application security scanners quality: a review. *International Journal of Advanced Computer Research*, 8(38):285–312, 2018. 14

[VAR+21] Tanja E. J. Vos, Pekka Aho, Fernando Pastor Ricós, Olivia Rodriguez Valdes, and Ad Mulders. testar - scriptless testing through graphical user interface. *Softw. Test. Verification Reliab.*, 31(3), 2021. 3, 6, 12, 45, iv

[VNJ+07] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*. The Internet Society, 2007. 11

LOGIN WITH TESTAR AND OWASP ZAP

LOGIN

For login scenarios, both tools required additional setup. The steps required for TESTARwere derived from the TESTARpaper [VAR+21]. For TESTARto log in, a pre-specified action had to be added to the startup sequence. A variant of the code snipped from [VAR+21] is shown below.

```
CompoundAction.Builder builder = new CompoundAction.Builder();
for (Widget widget : state) {
    if (widget.get(WebTags.Id).contains("username")){
        builder.add(new WdAttributeAction("username", "key", "
            value"));
    }
    else if (widget.get(WebTags.Id).contains("password")){
        builder.add(new WdAttributeAction("password", "key", "
            value"));
    }
}
builder.add(new WdSubmitAction("Form_Name")).build();
```

The steps required for OWASP ZAP are quoted directly from their website [owa21] and are listed below in Table 2.

Nr.	step
1	"Explore your app while proxying through ZAP"
2	"Login using a valid username and password"
3	"Define a Context, e.g. by right clicking the top node of your app in the Sites tab and selecting 'Include in Context'"
4	"Find the 'Login request' in the Sites or History tab"
5	"Right click it and select 'Flag as Context' / 'Form-based Auth Login request'"
6	"Check that the Username and Password parameters are set correctly - they almost certainly wont be!"
7	"Find a string in a response which can be used to determine if the user is logged in or not"
8	"Highlight this string, right click and select 'Flag as Context' / 'Logged in/out Indicator' as relevant - you only need to set one of these, not both"
9	"Double click on the relevant Context node and navigate to the "Users" page - check the user details are correct, add any other users you want to use and enable them all"
10	"Navigate to the Context "Forced User" page and make sure the user you want to test is selected"
11	"The 'Forced User Mode disabled - click to enable' button should now be enabled"
12	"Pressing this button in will cause ZAP to resend the authentication request whenever it detects that the user is no longer logged in, i.e. by using the 'logged in' or 'logged out' indicator."

Table 2: OWASP ZAP login setup (taken from [owa21])