Master Thesis
Computing Science

Improving the Adoption of Dynamic Web Security Vulnerability Scanners

Author:
Y.R. Smeets, BSc
yannic.smeets@student.ru.nl
Student No. 4244249

Internal supervisor:
Dr. G. Gousios
`g.gousios@cs.ru.nl`

Second reader:
Dr. ir. G.J. Tretmans
`tretmans@cs.ru.nl`

External supervisor:
Prof. dr. ir. J. Visser
`j.visser@cs.ru.nl`

November 18, 2015

Abstract

Security vulnerabilities remain present in many web applications despite the improving knowledge base on vulnerabilities. Attackers can exploit such security vulnerabilities to extract critical data from web applications and their users. Many dynamic security vulnerability scanners exist that try to automatically find such security vulnerabilities. We studied the adoption of these tools and found out they are rarely used by web developers during the development process of a web application. Through interviews, we investigated the main cause of the lack of adoption is the difficulty to use such tools. In order to improve the adoption of dynamic security vulnerability scanners, we introduce the Universal Penetration Testing Robot (UPeTeR). UPeTeR is a class library that allows web developers to easily set relevant data for many dynamic vulnerability scanners by providing them with an abstraction of required configuration data. Plugins, ideally created by experts of the scanners, transform this abstraction into an optimal setup of such scanners. A prototype has been created which was used to validate UPeTeR's acceptance by web developers at the Software Improvement Group, a software consultancy company in the Netherlands. The acceptance experiment demonstrated that web developers are willing to try out and work with UPeTeR.

Acknowledgments

Even though it was my task to do this research and write this thesis, I am not the only one who worked on this. Many people contributed in various ways and I would like to take this opportunity to thank them.

Firstly, I would like to express my gratitude to Joost Visser, Barbara Vieira, and Theodoor Scholte for their guidance, support, and positive attitude. In the many situation of doubt or uncertainty, they helped me pull through and put me on the right track.

Secondly, I want to thank Georgios Gousios and Jan Tretmans for their academic supervision and critical thinking in order to improve this thesis. Although the few sessions we had were tough and critical, it challenged me to rethink my study and improve its design.

Thirdly, I am grateful for all SIG colleagues and interns who made me feel welcome and made my internship a pleasant experience. I was pleasantly surprised by the professional yet informal working environment in which everyone was helpful, kind, and down to earth. It was a honor to work with these highly skilled professionals and to experience their dedication and perfectionism.

Furthermore, the support of friends and family helped to a great extend with the motivation and enjoyment for this thesis.

Finally and most of all, I want to dedicate this thesis to my father. From the day I was born, he supported me, helped me, and guided me through challenging moments of life. Even though he did not have knowledge of the topic of this research, he advised me in any way he could. Unfortunately, he could not witness the results of this study as he passed away unexpectedly just before the end. Thank you with all my heart for everything.

Yannic Smeets
Nijmegen
October, 2015

Contents

Chapter 1

Introduction

1.1 Problem Statement

Due to the openness and accessibility of the Internet, many services have moved to the web in the form of web applications. Sending emails, performing bank transactions, and administering our taxes are only a few examples of the activities we can do with applications on the web. With the increasing reliance on web applications to perform our daily routines, the need for, and importance of, safe and trustworthy applications is rising [37]. However, guaranteeing a secure web environment is hindered by the increasing complexity and interconnectivity of the applications.

OWASP, the Open Web Application Security Project, is an organization focused on improving the security of software [11]. One of their projects is the OWASP Top 10, of which the latest edition came out in 2013 [14]. It raises awareness on application security by listing the ten most common vulnerabilities found in applications, as well as describing possible risks, means of detection, and ways of prevention. The top 10 listing has been around for many years, indicating the difficulty of application security.

An example of the difficulty of creating and maintaining a secure web application is given in a study by the Dutch Consumers Association on 100 Dutch webshops [20]. Webshops were tested on the top 10 security issues from 2013 according to OWASP [14]. A staggering two third of examined webshops proved to be unsafe. Some serious XSS vulnerabilities were found in 41% of the webshops that would allow hackers to steal information from customers. Even more worrying, one webshop was vulnerable to an SQL Injection vulnerability which allowed attackers to take over the entire webshop.

Fixing these vulnerabilities costs more the longer they remain in the software [23]. However in comparison, if found early enough in the development process, fixing a flaw might only take three minutes [25], given that the location of the flaw is known to the developer. A way to enable developers to determine the location of flaws is using dynamic web vulnerability scanners. However, even though many tools exist for dynamic web vulnerability scanning, many vulnerabilities remain present in web applications.

1.2 Research Questions

Finding and fixing bugs in software is better and cheaper the earlier this is done in the development process. Unfortunately, many security vulnerabilities are continued to be found in released software. Therefore, the main goal of this research is to improve the detection of security vulnerabilities in web applications early on in the development process, preferably before the application is released. Automated dynamic vulnerability scanners are tools that can aid developers in finding security vulnerabilities. Even though many automated vulnerability scanners are available for developers to use, vulnerabilities continue to be found in released web applications. This makes us believe developers are not adopting these scanners into their way of development. To verify this assumption and to understand why this is the case, we address the following research questions:

1. How widespread is the use of dynamic vulnerability scanners by web developers during development processes?
2. What are the impediments for web developers to make use of dynamic vulnerability scanners?
3. Are web developers more willing to use vulnerability scanners if they are offered ways to hide the impediments?

1.3 Research Method

In order to answer the first and second research question, interviews were conducted with security experts and web developers using a semi-structured approach. Interviewees were asked questions on their experience with dynamic vulnerability scanners, the current state of security testing, and what they feel are impediments in using these vulnerability scanners.

The next step of this research was gaining hands-on experience with available vulnerability scanners. This approach was taken for two reasons. First, we could compare our findings with the results of the interviews. Second, benchmarks could be performed on the tools to gain a better insight in their performance and capabilities. The scanners were put to the test by scanning several applications within the OWASP Broken Web Applications [12]. This project from OWASP consists of a virtual image holding a collection of deliberately vulnerable web applications.

From identified impediments, we created a design for a tool that could tackle some of the impediments. The design was implemented in a working prototype for one vulnerability scanner, and partially for another scanner.

To validate the acceptance of our tool, we organized two focus groups with web developers. We gave a presentation on the functionalities during the workshops. Also, to leave a concrete impression of the usage of the tool, a demonstration was given in which the tool ran security scans on a demo web application. After the presentation and the demo, the attendants were given a questionnaire with questions related to the earlier found impediments.

1.4 Research Context

This study was conducted at the Software Improvement Group (SIG), a software consultancy firm based in Amsterdam, the Netherlands. They aim to *Getting Software Right* by evaluating the quality of software systems and advising clients in ways to improve their code and development process. Source code is analyzed using the SIG Maintainability Model [35] and the Security Risk Assessment [60] among other things. Their goal is to reduce the cost and increase the quality of software [39].

1.5 Limitations

This research is subjected to several limitations. First of all, the experiments we conducted with several dynamic vulnerability scanners were done with no prior knowledge or experience with those scanners. We cannot guarantee that our use of the scanners is the optimal way, and therefore all findings are merely ours and do not necessarily reflect findings of other developers. However, as other web developers might also lack knowledge of adequate usage of the scanners, it is plausible they also come across similar issues.

Secondly, the prototype of UPeTeR we created only fully supported the scanners OWASP ZAP. We made attempts to incorporate the Arachni tool into the prototype, but due to limited time we only achieved a partial implementation. This could have consequences for the validation of UPeTeR. However, as a more complete implementation would only improve the prototype, we feel current results could only be enhanced. Therefore, the impact of this limitation is limited.

1.6 Thesis Outline

In Chapter 2, we present some background on web applications, security testing, several security vulnerabilities and their lifecycle, and some vulnerability scanners. Chapter 3 gives an in depth description of the interview methods we conducted and their results. The process of gaining hands-on experience is explained in Chapter 4. This is followed by the design of our solution in Chapter 5. The validation of this solution is explained in Chapter 6. Finally, we conclude this thesis in Chapter 7.

Chapter 2

Background

The aim of this chapter is to provide background information on the concepts of this study. Section 2.1 will give a definition of security testing. This is followed by an explanation of common security vulnerabilities (Sec. 2.3). Some information on the vulnerability lifecycle in software development projects is given in the next section. The final section will give an overview of information found on security web vulnerability scanners through literature study.

2.1 Security Testing

Security testing is large subject, which is part of the even broader area of testing. Understanding the meaning of security testing requires a definition of security and of testing.

Several definitions of testing can be found in literature. According to the Standard Glossary of Terms Used in Software Testing, testing is: *"the process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects"* [55]. A more compact definition is given by Myers: *"Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended"* [43]. Additionally, the OWASP Testing Guide states that: *"Testing is a process of comparing the state of a system or application against a set of criteria"* [40].

Even though security is not regarded as a functionality, it is becoming an increasingly important aspect in the development of software [37]. McGraw's definition of security states: *"Software security is the idea of engineering software so that it continues to function correctly under malicious attack"* [41]. Therefore, security testing aims at verifying that software satisfies the security requirements.

Three major areas exist in the world of security testing. Static Analysis Security Testing, or SAST, is a white-box testing technique in which an application is analyzed without actually executing it. One of several methods to do this is dataflow analysis of backend code to verify whether unvalidated input reaches sensitive areas. Static analysis typically has access to the source code of the application and can therefore reach all parts of the application and analyze all paths the execution of the code can follow. The code coverage is thus high. However, many false positives arise from static analysis as it is difficult to precisely determine whether a piece of code is actually vulnerable.

Dynamic Analysis Security Testing, or DAST, is a black-box testing technique that analyzes an application by running it. Communication is done via the exposed interface. Inputs are generated and submitted to the application. Observed output is then matched with expected output to determine if the application has functioned correctly. This is where the main advantages of dynamic analysis lie. Detection of vulnerabilities is done by actually abusing them, resulting in fewer false positives. However, as dynamic analysis is black-box, it cannot ensure all code is covered and could therefore miss vulnerabilities in code that is difficult to reach.

Gray-box testing, or often called white-box fuzzing, is a new technique that tries to combine the advantages of dynamic and static analysis [33]. Dynamic analysis depends on solid information gathering algorithms to discover points in the application where attacks can be introduced and that can reach all parts of the backend code. This is difficult for ordinary dynamic analysis as the source code is not analyzed. Static analysis is able to do this, and can thus be used initially to find attack vectors and even determine different inputs that can achieve a high code coverage. The tool SAGE, developed by Microsoft, is an example of this [31].

For this study, only DAST tools will be taken into account.

2.2 Web Applications

Web applications are client-server applications. One part of the application is executed on a remote computer, or server, that is connected to the internet. Server side code is usually responsible for retrieving and storing data. Programming languages that are commonly used for server side code are PHP, Java, and C#.

Users can access the web app via a web browser on their own computer. Browsers render and visualize the user interface of the application. This UI is written in the HTML language. HTML is a markup language that consists of tags containing attributes. The *<p id='id'>text</p>* tag is an example of a paragraph tag holding the attribute *id*. The attributes are used to

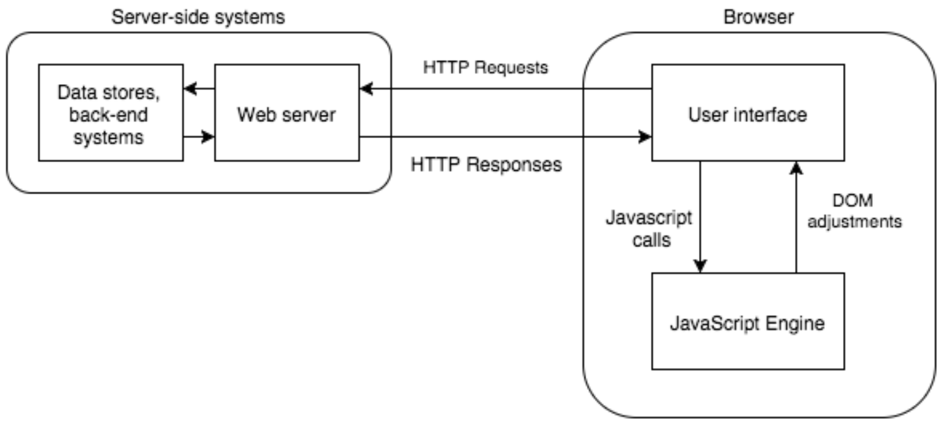


Figure 2.1: Overview of web application communication [46]

apply styling. This can be achieved by creating cascading stylesheet (CSS) files. Also, all browsers try to parse the HTML tags as a Document Object Model (DOM). The DOM is an browser independent representation of the data structure in HTML documents. Browsers can interact with the DOM by adding, removing, or changing elements within it, thereby dynamically interacting with a web page. The scripting language JavaScript is mostly used for this, and browsers therefore have a JavaScript engine built in.

Communication between both sides is done with the use of requests and responses. Initially, the client side requests the information of a web page. The server receives the request, generates and collects the results and sends that back over to the client. A URL is enough to simply request the data of a web page. However, many situations require more information from the client than just a URL. If a user wants to log into the application, it needs to send over its login credentials. This is achieved by filling in forms. The HTML that is displayed on the client side can contain a form tag which holds various types of input fields that allow users to enter information that is to be sent to the server. Figure 2.2 is an example of a login form. A user enters its username and password, and clicks on the 'Sign in' button. Whatever the user filled in is sent within the request to the server.

2.3 Security Vulnerabilities

Many different types of security vulnerabilities have been discovered and exploited in the history of the web. These vulnerabilities can be divided into several subcategories. Authentication vulnerabilities affect the integrity a web application. The use of default credentials, credentials being sent over unencrypted channels, a weak lockout system after multiple failed log in attempts, and a weak password policy all belong to this category. Privilege

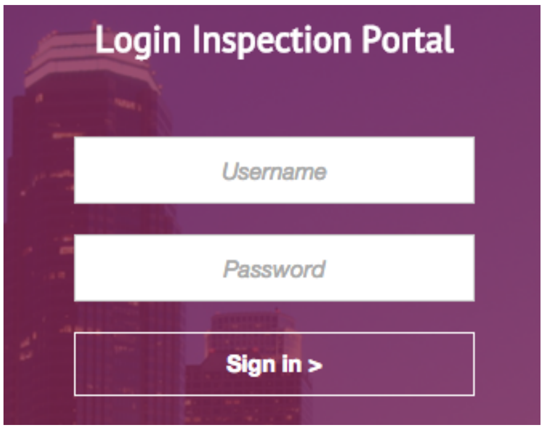


Figure 2.2: Login form for SIG Inspection Portal

escalation is a example of an authorization vulnerability, which allows users to gain access to parts of the application for which they should not have access to. Finally, one of the largest categories are input validation vulnerabilities, which includes vulnerabilities caused by a lack of validation on input generated by a user of the application [49]. This thesis mainly focuses on input validation vulnerabilities.

2.3.1 SQL Injection

SQL injection attacks belong to the most dangerous types of attacks for applications [14], including web applications. SQL injection vulnerabilities within a web application allow attackers to control communication messages to the database. This can result in the extraction and alteration of critical information, and possibly even grant access to the underlying operating system.

Web applications use databases to store data that needs to persist. This can vary from user information and blog posts to orders on a webshop and creditcard details. Data is retrieved from and stored to the database via SQL queries. Many web applications usually have a database containing a table with user information. A query such as the following is usually sent to the database in order to retrieve the information of all users.

Listing 2.1: SQL query that selects all users

```sql
SELECT * FROM Users;
```

This query selects all information from the Users table. Such static constructions of queries is used whenever the same type of data is requested for a web page, i.e. a page showing all users. However, due to the dynamic nature of web applications, a more dynamic way of constructing queries is needed to provide the correct data. User input is sent to the application, after which it is transformed into a query. This interaction between the user, web application, and database is depicted in figure 2.3.

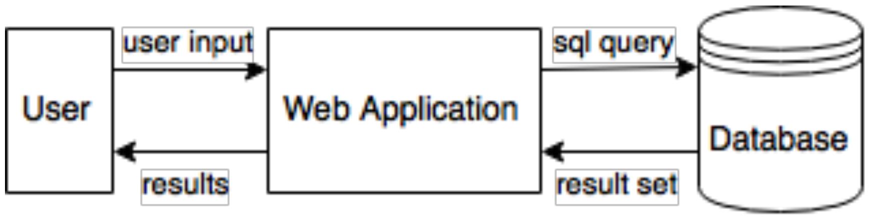


Figure 2.3: Web application interactions [53]

An authentication procedure is an example of this. Listing 2.2 illustrated Java code for authentication. Credentials like username and password are entered by the user. This is sent to the web application, where a query is generated to search for a user with the given username and password. All users that match the criteria are returned. After executing the query, the resulting set is checked to see whether it has at least one user in it. If so, the user is logged in.

Listing 2.2: Java based authentication mechanism

```
String query = "SELECT * FROM Users WHERE username = '" +
    request.getParameter("username") + "' AND password = '" +
    request.getParameter("password") + "';"

ResultSet users = database.executeQuery(query);
if (users.next()) {
  // User is logged in
}
```

Even though concatenating user input in predefined queries seems reasonable, resulting queries could potentially be completely altered. For instance, if user input is not restrained or validated and a password such as *' OR '1' = '1* is transmitted, the resulting query will be:

Listing 2.3: Malicious query

```
SELECT * FROM Users
WHERE username = 'administrator' AND password = 'whatever' OR '1'
    = '1';
```

Even though the administrator's password is not 'whatever', the final condition will always be evaluated as true. Therefore, the 'administrator' user will always match the query and be returned to the program. This would allow anyone to log in as any user without knowing their password. However, potential dangers within this vulnerability are not limited to login bypassing. A given password can even contain an entire different query. As the character *;* denotes the end of a query, a password beginning with *;* followed by another query results in two queries being executed by the database. The second query could be anything that the user's imagination can think of. For example, *'; DELETE FROM Users;* would result in a secondary query that deletes all users.

Listing 2.4: Secondary query generation

```
SELECT * FROM Users
WHERE username = 'username' AND password = '';
DELETE FROM Users;
```

SQL injection is a part of the input validation vulnerability category, thus a lack of input validation is its main cause. Unvalidated strings can contain character that are SQL keywords, like the character *'*. Stripping such keywords or rejecting inputs containing keywords might seem like a proper defensive coding technique. However, this approach prohibit users to specify legit input containing such keywords. For example, searching for people whose surname is *O'Brian* would be impossible. A better way would be to encode such keywords in a way that databases interpret them as regular input [34].

Encoding all parameters inside a query before it is executed is something even the most experienced developer can occasionally forget. A common method to avoid manual encoding is to use prepared statements [15] [54]. In ordinary SQL statements, variables are concatenated into the query string, making them vulnerable to changes. Prepared statements differ in the manner of building the query. When using prepared statements, a developer first needs to specify the static part of the query. Question mark characters are used to specify where variables will be inserted at a later stage. Upon inserting the user input into the variables, the method used for that automatically encodes the user input, thereby removing any SQL injection attacks. An authentication algorithm using prepared statements in Java will look like:

Listing 2.5: Authentication with prepared statement in Java

```
String username = reqeust.getParameter{"username");
String password = request.getParameter{"password");

String query = "SELECT * FROM Users WHERE username = ? AND
    password = ?;"
PreparedStatement ps = connection.prepareStatement(query);
ps.setString(1, username);
ps.setString(2, password);

ResultSet results = ps.executeQuery();
```

Since encoding is automatically performed in the *setString* function, developers cannot forget this.

2.3.2 Cross Site Scripting

Cross-site scripting vulnerabilities, or XSS vulnerabilities, are common in many web applications, as demonstrated by [20]. Strings returned by the web application to the user can get executed by the web browser when it is not properly encoded. This is normal behaviour, as a HTML page is basically a large string which should be loaded by the browser. However, parts of the returned HTML depending on unencoded user input enables malicious users to insert custom HTML or JavaScript [32].

There are two types of XSS vulnerabilities: non-persistent and persistent. When vulnerable to non-persistent XSS attack, a web application directly inserts user input into resulting HTML. Search engines are typical examples for this issue. To improve user experience, many search engines display the used search terms in the list of results. When leaving this unencoded, malicious users could search for a term that is actually valid HTML. The following listing shows an example in which JavaScript is directly reflected by the web application.

Listing 2.6: Reflected search term

```
<p>
  You searched for:
  <script>alert('hi');</script>
</p>
```

Obviously a user would not search with malicious code as he will be the only one affected. To take advantage of such vulnerabilities, attackers can try to steal information from valid users, as is illustrated in Figure 2.4. Some parameters, like search terms, are commonly transmitted to the server via a parameter within the URL, like `http://www.example.com/search=malicious+code`. Therefore, the first step is to send such a link to an unsuspecting user. Upon clicking on the link, the user sends the malicious code to the vulnerable web server. Since the web server reflects the parameter directly, the sent payload will be embedded within the resulting HTML. This is then executed by the browser, which has access to the user's cookies. Finally, the malicious code sends the information within the cookies to the attacker.

For persistent XSS, malicious code is not immediately reflected back to the user, but is saved into storage first. Whoever retrieves the malicious data will be exposed to it. A common example is a web forum. All posts are stored into the database and all users looking at the post will retrieve and execute the malicious code. For this reason, persistent XSS is far more

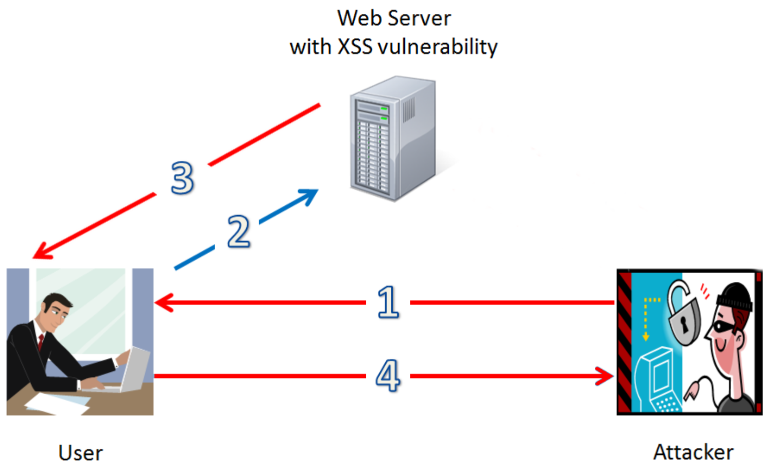


Figure 2.4: Reflected XSS [19]

dangerous than its non-persistent form as users do not need to click anything to be exposed to it.

The best practice to make a web application resistant to XSS attacks is, similar to SQL injection, proper input validation. Every user input that is sent to the server can contain characters that alter the meaning of other HTML. To prevent being vulnerable to XSS, developers need to encode all special characters that are sent to a user.

Besides implementing server-side preventive measures, developers can also help to mitigate the impact of an XSS attack by applying client-side best practices. Firstly, developers should set the HTTP-only cookie flag for session cookies. Attackers are often after information that is stored in cookies. HTTP-only cookie are not visible through the DOM (Document Object Model). Therefore, scripts are not permitted to access the data within the cookies [61], thereby making it less likely attackers can obtain secret information.

Secondly, Content Security Policies (CSPs) can be implemented. This mechanism is implemented in almost all popular web browsers [5], and allows developers to specify a whitelist of locations from which resources are allowed to be loaded [4]. A CSP consists of one or more *Content-Security-Policy* HTTP headers that specify from which sources content can be trusted. For instance, a header *Content-Security-Policy: script-src 'self'* `https://apis.google.com` notifies browsers that any script file from `https://apis.google.com` is allowed to be executed. Loading a script file from `http://apis.evil.com` would not be allowed and will thus be blocked by the browser. This prevents attackers from inserting scripts from other sources via a XSS attack.

Finally, another header called the *X-XSS-Protection* header can be used to counteract XSS attacks. By setting this HTTP response header, web applications inform browsers to enable their built-in XSS filters that block potential reflected XSS attacks.

2.3.3 Path Traversal

Almost all web applications make use of included local resources, like images, scripts, or other documents to be downloaded by users. Server-side scripts are used to manage the access to such files. Unfortunately, many web applications do not manage this properly, leading in some cases to path traversal, or directory traversal, attacks.

Upon exploiting such an attack, aggressors aim to access files and directories they should not have access to. Source code of the application could be valuable as some developers include hard coded passwords or other sensitive data within the source code. Furthermore, attacks also try to break out of the web root directory with the use of a *dot-dot-slash* (../) technique. Operating systems use the sequence ../ to traverse up the directory tree. By manipulating file paths with ../ sequences, attackers can get to critical system files like databases or password files. An example of this is a password file, containing system passwords, located at */etc/passwd* on most UNIX-like operating systems. Malicious users try to gain access by requesting the file *../../../../etc/passwd* [40]. The four dot-dot-slashes traverse to the root of the system, after which the /etc/passwd file is retrieved.

To prevent users from breaking out of the web root directory, all user inputs containing ../ and ..\(Windows directory separator) sequences should be rejected as that clearly indicates an attempt to tamper with file paths. This also includes rejecting their character encoded counterparts. Furthermore, keeping a white-list of allowed files and matching user input to white-listed files thwarts any attempts to access prohibited files.

Allowing users to specify which file they want to access is dangerous yet necessary in many cases. A solution is to use indexes rather than actual file names [13]. Developers can assign an identifier to a file, after which that file is only accessible through that identifier. Users can therefore only retrieve files which have an id assigned to them. This makes retrieving any other files impossible.

2.4 Vulnerability Lifecycle

All vulnerabilities have a lifecycle (Fig. 2.5), which starts when a coding mistake is put into the source code of an application. Whenever a vulnerability is released into the wild, there comes a moment of discovery after which attempts are made to exploit it. Upon disclosure of the vulnerability, the creators will create a patch to fix the vulnerability.

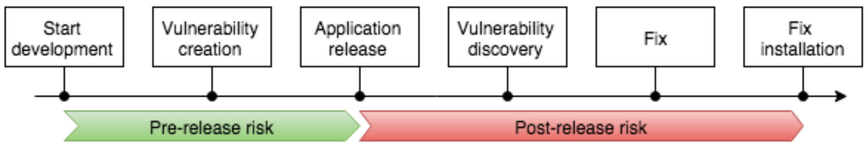


Figure 2.5: High Risk Vulnerability Lifecycle (based on [28])

Security risk exposure increases the longer the vulnerability is exploitable [30] [51]. After releasing vulnerable software, many users can be affected by malicious attacks caused by those vulnerabilities. Therefore ideally, discovery happens as soon as possible during the development phase before the application is released (Fig. 2.6). The difference between Figures 2.5 and 2.6 shows the importance of early discovery in order to keep risks to a minimum. The benefits of early discovery include quicker and cheaper solutions [25] [23], and less risk exposure [29].

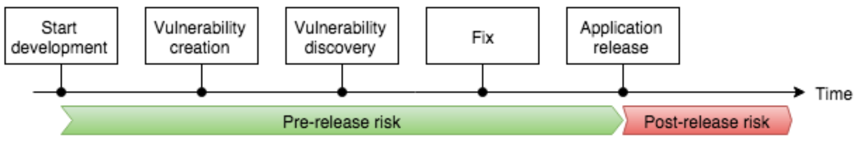


Figure 2.6: Low Risk Vulnerability Lifecycle (based on [28])

2.5 Dynamic Web Vulnerability Scanners

One way to discover security vulnerabilities of a web application early is to use dynamic web vulnerability scanners [27] throughout the development process. We define dynamic vulnerability scanners as tools that test for security vulnerabilities by simulating known, predefined attacks on a running instance of a web application. Scanner typically can spider a web application, trying to reach as many pages and states as possible. This is done to gain as much knowledge as possible of the web application in order to increase their attack surface (e.g. HTTP GET and POST parameters, cookies, and HTML form fields). After that, by probing the application with malicious input, scanners analyze the observed responses to determine whether a security vulnerability has been triggered.

This section gives an explanation of some dynamic web vulnerability scanners. An overview of many common vulnerability scanners is presented first. This is followed by a more in depth description of some open-source scanners.

2.5.1 Overview

Developers have a variety of dynamic vulnerability scanners available to them, listed in Table 2.1. Most of the scanners are commercially available, either as a standalone application or as part of a penetration testing service. Others are free and open-source, like the scanners W3af [17], Arachni [1], OWASP ZAP [18], and sqlmap [16]. These scanners have been chosen for further analysis due to the easy access of their source code. Each scanner has their own method of finding vulnerabilities, and some scanners can detect different type of vulnerabilities compared to others. Table 2.2 illustrates which scanner is able to scan for which vulnerability type.

Scanner	**Vendor**	**Commercial**	**Open-source**
IBM AppScan	IBM	x	
WebInspect	HP	x	
Acunetix WVS	Acunetix	x	
Tinfoil Security	Tinfoil Security	x	
w3af	w3af devs		x
Arachni	Arachni devs		x
Burp Suite	Portswigger	x	
Netsparker	Netsparker	x	
OWASP ZAP	OWASP		x
N-Stalker	N-Stalker	x	
sqlmap	sqlmap devs		x

Table 2.1: Overview of dynamic web vulnerability scanners

2.5.2 Sqlmap

Sqlmap is an open source automatic SQL injection and database takeover tool. It aims to automate the process of finding and detecting SQL injection vulnerabilities and exploiting them. Many different techniques for finding SQL injection vulnerabilities are used. By fingerprinting the database behind a web application, it tries to generate specific payloads that target that specific database type. A whole range of database systems are supported including MySQL, Oracle, PostgreSQL, and Microsoft SQL Server. All SQL injection vulnerabilities found are stored in a CSV file.

Sqlmap allows a user to specify which URLs it should attack in three different ways:

- A single url
- A file containing multiple URLs

- A starting url to spidering from

Sqlmap comes with a built in spider capable of searching for URLs. Logging in to a web application is not possible however. To reach pages that can only be accessed after authentication, users must log in to the application manually and provide sqlmap with the resulting session cookie. This allows sqlmap to take over the session and act as if it is an authenticated user.

There are two different ways to use sqlmap: via its command-line interface and through its REST API. The CLI starts one instance of sqlmap that can scan multiple URLs with different threads. The REST API starts a server that can initiate multiple instances of sqlmap, yet only one URL is passed to each instance.

Vulnerability Type	**Arachni**	**sqlmap**	**W3af**	**ZAP**
Code injection	x			x
CSRF	x		x	x
Local file inclusion	x		x	
LDAP injection	x		x	x
OS Command injection	x		x	x
Path traversal	x			x
Remote file inclusion	x		x	x
Session fixation	x			x
SQL injection	x	x	x	x
Unvalidated redirect	x		x	x
Xpath injection	x		x	x
XSS	x		x	x
XXE	x			x

Table 2.2: Overview of supported vulnerabilities

2.5.3 W3af

W3af describes itself as a web application attack and audit framework whose goal is to create a framework to help secure web applications by finding and exploiting all web application vulnerabilities [17]. Similar to sqlmap, it is built in python and is open source.

All functionalities within w3af are implemented as plugins. These functionalities are divided into three different types of plugins. Firstly, crawl plugins are responsible for finding new URLs, forms, and other injection points. The web_spider plugin is one of the crawl plugins. It is a classic web spider designed to navigate through the application and extract all URLs from web pages it encounters. To allow spiders to spider as an authenticated user, log in information can be given to *auth plugins*. This process can be helped further by providing URLs that are known beforehand. These URLs can be imported through a CSV file via the import_results plugin. Furthermore, the spider_man plugin is a local proxy that can be used for gaining knowledge of the web application when it contains a lot of client-side code like JavaScript. By enabling the proxy, users can manually navigate through the application in order to show w3af new injection points.

Secondly, audit plugins are designed to use injection points, found by crawl plugins, to send specially designed payloads in order to trigger vulnerabilities. Unlike sqlmap, w3af audits web applications for many vulnerabilities, as is shown in Table 2.2.

However, even though w3af supports the auditing of more vulnerabilities than sqlmap does, w3af tests SQL injection less thoroughly. Whereas sqlmap uses numerous techniques, w3af only uses two types of attacks.

Finally, users can activate attack plugins to exploit vulnerabilities. Those plugins usually try to operate a shell on the server of the web application, or manipulate the database in case of SQL injection vulnerabilities.

W3af is implemented with a GUI, a CLI, and a REST API. Users can select plugins to be used during the scan through each interface. The REST API is implemented in python.

For storing and reporting purposes, users can use *output plugins*. Supported report formats are CSV, HTML, plain text, and XML.

2.5.4 OWASP ZAP

The Zed Attack Proxy, shortly ZAP, is the flagship open source project within the OWASP community. It started as a fork from the Paros Proxy Project, which allowed ZAP to easily use the proxy functionalities within the Paros Proxy. ZAP is built in Java and is intended to be an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to support both developers and testers with a wide range of security expertise [18].

ZAP uses a concept of contexts to scan web applications. Each web application has its own context containing information specific for that web application, like the scope of URLs belonging to the application, authentication details, technologies used by the web application, and user credentials. The information within such contexts is used throughout the process of scanning the application.

Similar to w3af, functionalities within ZAP are implemented as plugins, or add-ons. These add-ons are separated by their maturity, of which three levels are present. The lowest ones are alpha add-ons. An add-on of this level can be created by anyone and has no special requirements. Alpha add-ons usually indicate that the functionalities within them are still under development and have not been fully tested. The second level is the beta level. In order for add-ons to qualify for this level, they need to be of reasonable quality (according to lead project members) and mostly fit for purpose. However, beta add-ons can still need some testing. Finally, the released level indicates that add-ons are of high quality and well tested. Core release functionalities, like a spider and standard rules for testing various vulnerabilities. Others can be downloaded from within ZAP.

Spidering is implemented in two ways. The basic, traditional spider is automatically installed and is able to discover URLs by scanning and analyzing hyperlinks on each page. By utilizing authentication information of a context, the spider is able to manage a valid log in session throughout the spidering process, enabling it to reach content for which authentication is required. Another, more novel, method of spidering is implemented in the AJAX spider add-on, which operates Crawljax [6] [42].

Like w3af, ZAP has the ability to scan for multiple different security vulnerability types including the ones stated for w3af. However, the detection of SQL injection is more advanced than w3af. ZAP is better at targeting its attacks due to the ability to specify which technologies are used. Similar to sqlmap, it can generate payload specific to a certain type of database. An add-on that mimics the payload generation of sqlmap is currently in beta level and in development.

Users can operate ZAP in three different manners. The GUI can be used for interaction with all functionalities. This allows users to do additional manual testing. Another way to use ZAP is via a CLI. A normal call to ZAP from the CLI will start the GUI. Additionally, a headless instance can be executed by running ZAP in daemon mode. Some options can be set, however little could be found in the documentation. Therefore, in order to communicate with ZAP in daemon mode, users have to make use of its REST API. However, not all functionality is available through the API, as the API tends to lag behind the available features from the GUI. The API grants users the ability to interact with ZAP programmatically. Class libraries have been created in Java and Python, and are automatically generated by ZAP to stay up to date.

2.5.5 Arachni

Arachni is a vulnerability scanning Ruby framework aimed towards helping penetration testers and administrators evaluate the security of web applications [1]. It is a highly modular system whereby functionality can be extended via custom components. A web interface has been designed for a user-friendly access to Arachni. A CLI can also be used to control Arachni.

Arachni is built around the headless PhantomJS browser. This allows Arachni, to properly handle client-side code by interacting with it through PhantomJS. Therefore, it can spider highly complex web applications which make heavy use of JavaScript. Furthermore, the trainer component enables

Arachni to train itself by monitoring the web application's behavior and looking for new auditable elements on a web page. The *autologin* component allows users to specify authentication information in order to grant Arachni access to pages behind authentication.

To further assist Arachni in its spidering, users can enable a proxy component. Manual interaction between the user and the application is recorded and analyzed by the proxy, giving more information about the application to Arachni.

Like w3af and ZAP, Arachni has detection capabilities for many different vulnerabilities. Boolean-based, timing-based, and error-based SQL injection detection is supported. Furthermore, due to its ability to fingerprint technologies used by the target application, payloads for detecting SQL injection (and all other vulnerabilities) will automatically be tailored to the used technologies.

By utilizing client-side code through its built-in browser, Arachni is able to test not only reflected and stored XSS, but also DOM-based XSS [38] wherein attack payloads are executed by arbitrary client-side code. This form of XSS can not be detected by just statically analyzing received HTML. Since Arachni also analyses changes to the DOM, it is able to detect these vulnerabilities whereas ZAP and w3af are unable to do so.

The output produced by Arachni is formatted into a custom type called the *Arachni Framework Report*. This can be transformed by the *Arachni_reported* executable into other, more common, types like JSON, HTML, XML, YAML, or even plain text.

Arachni can be executed via a web interface or a CLI.

2.6 Related Work

Other researchers have investigated various aspects of dynamic vulnerability web scanner usage. Doupé et al. [27] studied the performance of a number of scanners by creating their own realistic web application containing various known vulnerabilities. It uses features that are commonly found in modern applications and tend to make spidering difficult. Shay Chen [24] created a benchmark application that incorporates many different cases for different vulnerability types. He put numerous scanners to the test and analyzed their ability to identify actual vulnerabilities and ignore false positive cases.

Earlier research was done on the adoption of general security tools. By interviewing 42 software developers, Xiao et al. [58] studied social effects on the adoption of security tools with the use of the established framework for understanding adoption called *diffusion of innovations*. This work was followed up by Witschey et al. [57], with the purpose of quantifying the relative importance of factors that predict security tool adoption. This was done with an online survey for software developers in 14 different companies.

Their research shows that, even though developers recognize the importance of security, security tools are rarely used for various reasons related to working environments. Our results differ in the fact that we also studied other aspects besides environmental incentives.

Chapter 3

Adoption of Vulnerability Scanners

Our aim is to investigate whether dynamic web vulnerability scanners are being used by web developers during the development process. We also intend to increase our understanding of the impediments for web developers to use dynamic web vulnerability scanners during the development process. Therefore, the following research questions are addressed:

1. How widespread is the use of dynamic vulnerability scanners by web developers during development processes?
2. What are the impediments for web developers to make use of dynamic vulnerability scanners?

In this chapter, we show that scanners are rarely being used. Furthermore, we present impediments for the adoption of dynamic vulnerability scanners by web developers that were found in five interviews. Interviews were held with two security consultants, one ethical hacker, and two web developers.

3.1 Interviews

3.1.1 Methodology

Due to the exploratory nature of the research questions, we have chosen a qualitative research approach in the form of semi-structured interviews [36] [48] [50], similar to other exploratory studies [58] [59]. By having a predefined set of questions, interviews can be steered towards answers for the research questions, while at the same time leaving room for improvisation and the exploration of new insights [48]. In order to focus on the questions and the conversations, an audio recording was made for each interview in accordance with the interviewee. This allowed us to represent the made statements in a reproducible manner [36].

3.1.2 Interviewee Sample

The research questions can be answered from two perspectives. On one hand, web developers know best whether vulnerability scanners are being used in their development process, what impediments they encounter, or what their reasons are for not using them. On the other hand, security experts have better knowledge of the scanners and thus have a better impression of any impediments these scanners have. Therefore, we have interviewed five people with backgrounds ranging from security experts to experienced web developers.

The first two interviewees were web developers with many years of experience. The third and fourth interviewees were security consultants at SIG. Even though they have little time themselves to develop applications for the web, their profession as consultants allows them to visit and experience many different environments and speak to many web developers. Therefore, they have a large pool of experience to tap from with regards to the adoption of vulnerability scanners. Finally, we have interviewed an hacker who works as a security expert.

3.1.3 Interview Design

Each interview consisted of three parts. It started with an introductory part, followed by an investigation into the state of security testing in the web development industry. The interviews were concluded with exploration of ideas for improvements.

The introductory part consisted of questions on the experiences the interviewee had with dynamic vulnerability scanners. This gave us an indication of the knowledge the interviewee had on the subject. Furthermore, web developers could provide an answer to the first research question, while interviewees with scanner experience could also give some insights into the second research question.

The following part of the interviews included questions regarding the current state of security testing in companies. This also included exploring the importance of security for companies, and the amount of effort that is put into security testing. The aim of this line of questions is to find impediments from a company culture point of view. Interviewing security consultants was beneficial in this stage, due to their experience of cultures within many different companies.

Finally, we finished the interviews with a set of questions focused on the ideal situation for security testing according to the interviewees. We wanted to get a better understanding of current impediments and possible solutions by giving interviewees an opportunity to come up with ideas to improve the current state of security testing.

3.1.4 Questions

Due to the exploratory nature of the interviews, the actual questions asked during the interviews might not exactly match the questions proposed below. Depending on the course of the interviews, some questions have been left out while new questions were asked when appropriate.

Experience with dynamic vulnerability scanners

- What is your experience regarding automated black-box vulnerability scanners?
  - How difficult are the tools to use without experience?
  - How much knowledge and experience is needed to use the tools in the most effective way?
- What is your opinion on using automated security vulnerability scanners during the software development process?
  - What would currently be the added value of the scanners?
  - What could be possible downsides of using the scanners?

State of security testing in companies

- In general, how important is security for companies?
- For the companies that find security important, how do they make sure their web applications are secure?
  - How much effort do they put into security testing?
  - What tools do they use for security testing?
    - What are the benefits of the tools?
    - What are downsides of the tools?
  - What do you think is the reason that companies have chosen for their current test approach?

Requirements of a security test approach

- What would be important requirements for an approach to test the security of a web application?
- Do you think current security testing tools are suitable to be integrated in the software development processes of companies?
  - If so, why do not all companies do this in your opinion?
  - If not, what is required to make the tools suitable?

3.1.5 Data Analysis

The interviews have been transcribed according to the audio recordings. To save time, the transcriptions are not precise word-for-word written reports, but summarized versions of the interviews where deviations from the main research questions are left out. The summarized reports can be found in appendix A.

The answers given by interviewees regarding usage of the scanners was analyzed to answer the first research question. The insights and judgments of the experts were put together to illustrate general consensus.

In order to extract impediments from the interview reports, the data was coded through the method of thematic analysis [26]. This has been used successfully in the past by other studies [56]. First of all, similar to the Grounded Theory Approach [56], segments of texts are given codes or keywords that represent an answer to a research question. This is followed by a translation from codes to themes, where a theme combines several codes into a more meaningful, all-embracing unit [26].

3.2 Results

All interviewees explained that vulnerability scanners are seldom being used by companies at all, let alone integrate them into the development process. Some even claim security is rarely being tested at all. Table 3.1 shows comments made by the interviewees on this subject. Therefore, we are comfortable to state that vulnerability scanners are hardly used during the development process.

Table 3.2 represents an overview of the codes and matching themes that emerged from the interviews. These codes and themes represent impediments that web developers face when using, or trying to use, dynamic vulnerability scanners according to the interviewees. The matching of codes and summarized text of each interview can be found in appendix B

The opinion of each interviewee on each code is shown on the right side of the table. Interviews #1 and #2 were held with security consultants at SIG. Interview #3 was done with the ethical hacker. The final two interviews were conducted with web developers.

3.2.1 Usability

The theme that combines the most codes is usability. It encompasses the following codings:

- Ease of Use
- Ease of Installation & Mainenance

Interviewee	**Comment**
#1	Generally, companies do not use vulnerability scanners.
#2	In general, security is important for companies. However, this does not mean that a commensurate amount of time, attention, and money is put into it. Usually, there is not structured approach.
#3	Some developers are aware of security, but they do not have the time to implement it properly. That is why security is rarely tested.
#4	Security is important for companies, but it is not always being tested.
#5	Currently, we do not test for security at all. Our customers are not wealthy, and creating and running security tests takes time and therefore cost money.

Table 3.1: Comments on security testing and vulnerability scanner usage

- Ease of Context Configuration
- Ease of Integration
- Handling Different Output Formats

These codings were mentioned often during the interviews by both web developers and security experts/consultants. Integration, installation, and maintenance of the scanners came up in the interviews with security consultants. They explained these tasks require a lot of time and effort. Both interviewees noticed an absence of a continuous build pipeline, thereby forcing companies to manually operate the scanners.

This would not be a problem if scanners were user friendly. However, the difficulty to use and to configuring the scanners to operate on a target web application were mentioned as problem areas for many scanners. The ethical hacker, interview #3, emphasized the importance of proper contextual configuration on multiple occasions, as this is essential for effective security scanning. This is backed up by the web developer in interview #4. Having some experience with a few scanners, he explained he would rather focus on manual testing with personal knowledge of the application than use tools which lack this information. Even though web developers have the most knowledge of their application, transferring this knowledge is difficult.

Finally, the difficulty of handling different output formats was mentioned as a downside. According to a security consultant, updates for scanners

Theme	**Code**	**Interview**				
		#1	#2	#3	#4	#5
Usability	Ease of Use	+	+			+
	Ease of Installation & Maintenance	+	+			
	Ease of Context Configuration		±	+	+	+
	Ease of Integration	+	+			
	Handling Different Output Formats		+			
Lack of Effort	Priority		+	+	+	+
	Security Knowledge	±	+			+
Performance	Effectiveness	±	+	±	+	
	Test Speed	+		+	+	-
	False Positives	±	+	+		+
Others	Data Contamination				+	
	Understanding Tool Functionalities				+	

+	interviewee mentioned the impediment
±	interviewee is undecided
-	interviewee did not find this an impediment

Table 3.2: Themes and Codings

can cause them to return a different output format, requiring a significant investment to keep a scanner functioning correctly. This also matches to the coding for ease of maintenance.

3.2.2 Lack of Effort

Two codes have been connected to this theme:

- Priority
- Security Knowledge

In four out of the five interviews, priority was mentioned as a large impediment of the adoption of security scanning tools. This can be explained in two ways. First of all, four interviewees explained the battle between security and deadlines. Performing security tests takes time away from developing features, which have a greater visible value than security. According to both web developers, security is not explicitly requested. Therefore, not much time is reserved for security testing. Secondly, a false sense of security may develop when companies use up to date web development technologies, like Microsoft's .Net Framework [10] or Facebook Login [7]. Both web developers claim these technologies or services automatically mitigate common vulnerabilities like SQL Injection and XSS. Even though this might be true for some vulnerabilities, this does not always hold. The .Net Framework's countermeasures for CSRF have to be manually applied. A developer can forget to do this, or even fail to understand the need for this. Therefore, relying only on other technologies could cause a false sense of security.

Security Knowledge, or a lack thereof, was also discussed in three interviews. Two interviewees thought this is an impediment, while one had some contradicting statements. He blamed nescience for the absence of scanner usage. However, he also stated it should not be a problem, since vulnerabilities are easy to explain.

3.2.3 Performance

The Performance theme consists of the following codes:

- Effectiveness
- Test Speed
- False Positives

Interviewees had different opinions on the importance of the effectiveness of the scanners. All interviewees mentioned scanners only find low-hanging fruits, simple flaws that can easily be spotted. According to a security consultant, manual security tests can find far more interesting problems than automated vulnerability scanners can. However, interviewees #1 and #3 pointed out that even these findings are useful, as it is better than having no findings.

The majority of the interviewees said that the slow speed of scanners is a downside. Integration of the scanners into a build pipeline would result in a huge slowdown in the building process, which should be fast. On the other hand, one web developer suggested this would not be a complication, since the work is done automatically. To prevent a slowdown of the building process, the tests could be run at night or in the weekends, circumventing the delay.

Four interviewees mentioned false positives as being a problem. All of them state that the time required to process these false positives is a drawback. Nonetheless, one security consultant stated it is better than not doing security tests at all.

3.2.4 Others

The final theme is meant for codes that do not fit well within the other themes, and we feel are not relevant for this thesis. The reasons for this choice will be given in Section 3.3.1. Codes included in this theme are:

- Data Contamination

- Understanding Tool Functionalities

3.3 Discussion

During the interviews, some main problems came to our attention. All subjects mentioned some aspect of the difficulty to use scanners as a problem. In the interview with the ethical hacker, we discovered that configuring the scanners to fit the application under test has a great impact on the performance of the scanners, yet it is not trivial to do for many scanners. Although web developers know the context of their web application better than anyone, it requires great effort and knowledge to apply this to configuring the scanners.

Another interesting impediment we found was the handling of different output formats. The security consultant explained that not only the configuration and setting up of the scanners requires time and effort, but also the processing of the results. The output format of a scanner could change in every update, which would require a change in the handling of these results. However, we also figured that using multiple scanners would further increase this hassle even more.

Priority and security knowledge have both been given the theme Lack of Effort. Many interviewees pointed out the battle between security and deadlines. Features have more visible value than proper security. Companies will therefore put much more effort into the development of functionalities instead of proper security testing. This results in a reluctance to use the scanners. We figured that a lack of security knowledge corresponds with a lack of effort as well. As the first interviewee pointed out, *"security vulnerabilities are not rocket science"*. Many web developers will be able to learn and understand the risks and consequences of vulnerabilities. However, because not much attention is given to security, gaining security knowledge falls short.

Companies cannot be forced to put more effort into security if they feel the costs are too high. However, making scanners easier to use might help with lowering the expenses. The Usability and Lack of Effort themes are therefore closely related.

Effectiveness is a topic that was discussed by four interviewees, yet only two considered this important. Obviously, the more these scanners find, the better they are. However, as subjects #1 and #3 stated, every vulnerability found, even the easy ones, are useful.

Similarly, four interviewees mentioned test speed, of which one did not consider this of great importance. Scans can take quite some time. However, as interviewee #5 says, these scans are automatically performed. Developers can focus on other tasks while a scan is running. Interviewee #1 claims long security scans cause increased building times. However, this can be circumvented by running the scans nightly instead of on every commit.

3.3.1 Other Findings

Subject #4 suggested that data contamination could be an impediment for many companies to adopt vulnerability scanners. In order to observe possible flaws, test data has to be sent to the server. For example, to find out if a text field is vulnerable for a reflected XSS attack, a payload to trigger the attack must be sent. The input could be saved in a database by the application, causing the database to be contaminated with test data. We see this argument as laziness. Best practices of web application testing state using a proper testing environment is critical regarding web testing [52]. Having a separate testing environment that is independent of the production environment eliminates data contamination as a problem.

Secondly, subject #4 also mentioned he prefers to know how scanners function, and how tests for vulnerabilities are performed. This includes every payload that is being sent by scanners in order to determine flaws. While this might be interesting for some developers, it would generate huge amounts of data that provide little extra knowledge on vulnerabilities. Even though the actual payload with which a fault is detected is important, we feel the rest can be omitted since only one subject mentioned this.

Chapter 4

Automated Vulnerability Scanners

In the previous chapter we show that web developers are not integrating security vulnerability scanners into the development processes of web applications. This is due to the amount of effort required to use and incorporate the scanners into the development process. Furthermore, the success of the scanners greatly depends on correctly setting up scanners in order for them to understand the context of the application. This is difficult to do according to the developers, and therefore requires to much effort and time.

To verify the impediments that were found during the interviews, we intended to get hands-on practice with several scanners and see whether we experienced those impediments as well. Aside from the literature study that was done before hand, we did not have any practical experience with the scanners.

We focused on the impediments for usability and performance, as a lack of effort is not testable in this setting. Furthermore, as we do not feel test speed is of significant importance, we did not take testing time into consideration. Those impediments can be overcome by automating the testing process and executing it at night or during weekends. Finally, the similarities and differences between generated output of the scanners are evaluated.

4.1 Experiment Applications

In order to gain hands-on experience with scanners, we chose to run scans on three different test applications. This allows us to experience how to setup scanners to properly run a scan on an application. Two benchmarking applications were selected, as well as a demonstration application containing deliberate vulnerabilities.

4.1.1 Web Input Vector Extractor Teaser

WIVET is a benchmarking project that tests how well scanners can spider through a web page by finding all web links, thereby assessing the ability of the spiders to locate and extract additional resources and input delivery methods. This increases the attack surface of the scanners, and with that improves the chance of finding vulnerabilities. We have used the latest available version, Version 3, of WIVET for this study.

The project is a website that contains many web pages, each requiring different approaches to reach a certain success state. A total of 56 unique success states can be reached. Some examples of techniques used to test spider capabilities are the use of JavaScript, jQuery, iFrames, and even Adobe Flash. The session logs which success states have been reached so statistics of the spider can be gathered.

Furthermore, the benchmark also requires spiders to exclude the logout page from spidering. Logging out of the session will stop the benchmark and start a new one, as the collection of data is done throughout a session. Spiders should thus avoid entering that web page.

4.1.2 WAVSEP

The Web Application Vulnerability Scanner Evaluation Project, or WAVSEP in short, is an evaluation platform for dynamic security vulnerability scanners. It contains a variety of common vulnerabilities designed to assess the accuracy of detecting security vulnerability issues by scanners. Common and advanced scenarios are presented to establish the scanner's abilities. Furthermore, the project also contains false positives cases in which scenarios give the impression of being vulnerable but in reality are not.

Version 1.5 of the WAVSEP was used for this study. Table 4.1 shows the number of true positive and false positives cases for each vulnerability type that are present in WAVSEP.

Vulnerability Type	**# of true positive cases**	**# of false positive cases**
Path Traversal	816	8
Remote File Inclusion	108	6
XSS	73	7
SQL injection	132	10

Table 4.1: Cases for each vulnerability type in WAVSEP

4.1.3 BodgeIt Store

While the benchmarking applications give a good impression of possible capabilities of scanners and provide us with some experience in setting up and using the scanners, they lack a simulation for realistic scenarios. We therefore decided to also work with an application which simulates real life applications yet has known vulnerabilities to test on.

The BodgeIt Store [2] [3] is an application that satisfies the criteria. It is designed as a simulation of a webshop to help people learn about penetration testing. For this reason, it contains deliberate vulnerabilities. This includes a SQL injection vulnerability in the login page and two XSS vulnerabilities in the search and contact page.

4.2 Evaluation

The four scanners that were discussed in Section 2.5 have been put to the test and were used to scan the applications discussed above. We used a MacBook Pro that was running OS X Yosemite. All target applications are part of the OWASP BWA [12]. VirtualBox version 5.0.4 was used to run the BWA, and set it up in such a way that communication between the virtual image and the native operating system was possible. Table 4.2 shows the results of the scanners on the benchmark applications. Table 4.3 reflects the evaluation of impediments we encountered for each scanner.

4.2.1 Sqlmap

As recommended by the developers of sqlmap, a cloned version of the github project [8] was used to install the latest version, version 1.0-dev-27707be, of sqlmap. The CLI was used to execute sqlmap.

Ease of installation & maintenance Sqlmap only requires Python version 2.6.x or 2.7.x. As version 2.7.10 was pre-installed on our machine, sqlmap ran without requiring further installation steps. Also, updating sqlmap can be achieved by merely providing the argument *–update* and sqlmap will update itself automatically. Therefore in our experience, installing and maintaining sqlmap is not difficult.

Ease of configuration Setting up sqlmap turned out to be a hassle. By default, sqlmap does not spider an application, and the argument –*crawl* is required to enable spidering. Even then, it only detects pages with GET parameters. The argument *–forms* needs to be set in order to enable spidering and scanning of POST parameters. Additionally, sqlmap is not able to automatically handle sessions. Sessions are used by web applications, like the BodgeIt Store, to identify which account a user belongs to, and are therefore required in order to spider and scan pages that are only accessible to authenticated users. By using the argument *–cookie*, sqlmap can be given

Scanner	**WIVET precision**	**WAVSEP results**								**BodgeIt results**	
		SQLI		XSS		RFI		Path Traversal			
		Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
sqlmap	7%	100%	100%	-	-	-	-	-	-	0%	0%
OWASP ZAP	21%	100%	100%	89%	100%	78%	99%	68%	100%	67%	67%
Arachni	96%	100%	100%	90%	100%	100%	100%	96%	100%	67%	100%

Table 4.2: Benchmark results

Impediment	**sqlmap**	**ZAP**	**arachni**	**w3af**
Ease of installation & maintenance	-	±	±	+
Ease of configuration	+	+	+	
Ease of integration	+	±	+	
Ease of use	+	+	+	
Handling output format	±	-	±	
Effectiveness	±	±	-	
False positives	±	+	-	

+	impediment is found
±	undecided
-	impediment is not found

Table 4.3: Impediment evaluation of scanners

a session id. Unfortunately, this required the user to manually get a session id in order to give it to sqlmap. On the other hand, sqlmap does allow users to specify the DBMS that is used by the application, by using the argument *–dbms*.

Ease of integration Integrating sqlmap into the development process is not necessarily difficult, but it takes quite some effort. Much manual control is required to operate sqlmap. During the spidering and scanning process, the user is continuously presented with choices that have to be made for sqlmap to continue. Examples of this are the questions whether the user wants to check for the existence of a sitemap, or whether the user wants to test a certain URL. These have to be answered multiple times in similar situations during one scan, making it a tedious process. To overcome this, the argument *–batch* can be added. This makes sqlmap automatically fill in default answers for the choices. However, some of these default choices are not desirable. For instance, after detecting one SQL injection vulnerability, it will ask the user whether to continue. The default behaviour in batch mode will not continue, even though we want to find all vulnerabilities. Users can overwrite the default answer for a question with the *–answers* argument.

Ease of use As explained in the points above, sqlmap is not very usable for automated use. Sqlmap needs to be manually controlled by default. Alternatively, sqlmap can be activated in batch mode, which automatically fills in default answers that are not always desirable. Those default answers need to be overwritten. Yet to know which answers need to be overwritten, a developer needs to go through the logs to find the 'wrong' answers. Especially for developers that are not experienced with sqlmap, this takes much time and effort. Furthermore, testing authenticated pages requires the developer to obtain a session id before scanning, which we found not intuitive.

Handling output format The output of sqlmap consists of a simple CSV file containing the URLs and parameters of found vulnerabilities. As sqlmap only detects SQL injection, the vulnerability type that is found is obvious.

When sqlmap is merely used for spidering, the output is more difficult to collect. Unlike scanning, the output file that is created is not a CSV file but a simple text file with all found URLs. The text file is stored as a temporary file which is difficult to access. Furthermore, the argument *-m filename* can be used to provide sqlmap a text file holding all URLs it needs to use as starting point for spidering. Unfortunately, sqlmap handles each URL separately, resulting in different temporary output files for each starting URL. Aggregating all different output files would therefore be required to obtain a list of all found URLs.

Effectiveness Sqlmap only found 4 out of the 56 states of the WIVET benchmark, which is very low. It cannot handle JavaScript events and any other technique besides plain HTML.

Since sqlmap can only scan for SQL injection, only those cases containing SQL injection vulnerabilities were used from the WAVSEP benchmark. Only two URLs were passed along to sqlmap, namely the page that links to all true positive cases and the one that contains all false positives cases. Sqlmap has to spider those URLs to find all cases and found all of them. This shows that unlike the negative results from the WIVET benchmark, sqlmap can spider links from static web pages.

The results of the WAVSEP benchmark on the SQL injection vulnerabilities is shown in Table 4.2. All true positive cases were identified as vulnerable, whereas none of the false positive cases were flagged. This shows sqlmap is very competent in finding SQL injection vulnerabilities.

However, different results came out of the BodgeIt Store scanning. The only SQL injection vulnerability that is present in the BodgeIt Store is located on the login page. Unfortunately, even after directly targeting that page, sqlmap could not find this vulnerability. Furthermore, it did flag another page which turned out to be a false positive. Results from real life applications are therefore less reliable.

False positives None of the false positive cases were identified in the WAVSEP benchmark. However, one false positive result was given when scanning the BodgeIt Store. Even though this is only one case, it was continuously reported throughout multiple scanning sessions. As it is merely one, it is not a large issue but it does take time and effort to process it.

4.2.2 OWASP ZAP

For this study, we looked at the Java API version of ZAP 2.4.2.

Ease of installation & maintenance Downloading and installing ZAP was easily done from their github page. The application itself required no further installation. Updating to the latest version can also be done by calling one method.

Additional features in the form of plugins were easily downloaded from the GUI with the use of the internal marketplace system that comes with ZAP. Unfortunately, this feature is not accessible from the API. Users are therefore required to use the GUI in order to install additional features and plugins.

Ease of configuration As ZAP relies on its internal proxy to follow browser traffic, users need to alter the proxy settings within their browser to connect to the proxy. Users are able to see and alter the proxy connection settings, so we had no trouble configuring the proxy and browser.

However, configuring ZAP to spider and scan our applications was more unclear. ZAP uses a concept of contexts, where a context contains much information on the application under test. This includes URLs that are in and out of scope, and which users can login to the application. All of this data can be set via the API, yet it is not clear in what format that data should be provided. For instance, URL scoping is done with regular expressions. Unfortunately, it is not immediately clear how to insert an entire section of the application in scope. Even though the difference between *url/section* and *url/section/* seems insignificant, it is a large difference in ZAP. For ZAP to scan an entire section, it needs to start at the root of that section. However, if that root is not included in the scope, it refuses to scan the section. Therefore, that tiny difference results in an entire section not being scanned.

Furthermore, we also experienced setting up authentication details to be sensitive to errors. The API needs configuration parameters which includes the login request post data. That data usually holds the username and password of a user. As ZAP can login with different users that are within the context, the data should have a string placeholder so ZAP knows where to inject the login credentials of a user. The string that should be used is {*%username/password%*}, which is nowhere explained in the API and was difficult to find within the documentation.

Also, not all functionalities within ZAP are available through the API. An example of this is the AJAX spider. In the GUI, a user can set which elements the AJAX spider should click on. This is not possible in the API. This makes it difficult to operate ZAP through the API to its full potential.

Finally, ZAP allows users to set a strength of scan plugin. Intuitively, this means that, the higher the strength, the more thorough it will scan for that type of vulnerability. However, the difference between different strength levels is not explained or indicated.

Ease of integration ZAP has an API that utilizes a running instance of ZAP that runs in headless mode. The API is created in Java and PHP and can easily be used by developers to interact with ZAP by writing their own code. This code can then be called from an automated build pipeline, making it easy to integrate ZAP into the development process. Unfortunately not all functionalities are available through the API, which is a drawback.

Ease of use Some usability problems have been described above already. Most of those issues are caused by a lack of clear documentation. For example, some settings have only a few possible valid options which the GUI represents as a select box to the user. However, the API does not do this. It expects a string with the exact name of the option and it does not always provide a list of available options. Users would need to start the GUI to find out which options are available. Also, parameter types are very inconsistent throughout the API. For example, a context id is usually needed to retrieve information of a context or select a context. However, occasionally a context name is asked for. This can be very confusing.

Handling output format Users can get the results from both spider types from the API. The regular spider returns a list of all found URLs, which can easily be processed. However, the AJAX spider gives much information except the found URLs which we found not intuitive.

Results from scans can easily be obtained through the API. Findings contain data on the URL and parameter where the vulnerability was found. Furthermore, payload that was used to detect the vulnerability, a description of the vulnerability type, and possible solutions are included. This information helps developers solve the vulnerability, which is good.

Effectiveness ZAP has two different types of spiders, a regular one and an AJAX spider. Surprisingly, the regular spider did extremely poorly, as we could only find one state out of 56 within the WIVET benchmark. The results significantly improved when using the AJAX spider, which found 12 states. However, this is still not very good as it misses 44 other states.

ZAP can, unlike sqlmap, scan for more than just SQL injection vulnerabilities. Therefore, benchmarks for all vulnerability types were scanned by ZAP. The strength of all required scanning plugins were set to insane, its highest setting, in order to get the best results.

Two different plugins are available for scanning for SQL injection vulnerabilities. One tests for simple short cases, while the other one tests for specific DBMSs. The first plugin detected 118 vulnerabilities out of 132, while the second one detected 120. Interestingly, combining both gave the best result where all vulnerabilities were found.

The other vulnerability types could be tested for with one plugin for each type. Not all cases for these vulnerability types were found. The precision ranged from 68% for path traversal to 89% for XSS. Interestingly, some false negatives could be explained by bad spidering. Some cases required the submission of form data. Apparently ZAP was unable to do this properly for all cases which resulted in missing several vulnerabilities.

Finally, we tested the BodgeIt Store for SQL injection and XSS vulnerabilities. The SQL injection vulnerability was found, and one XSS vulnerability was detected.

False positives The number of false positives that are reported differs per plugin and vulnerability type. In the WAVSEP benchmark, only one false positive RFI vulnerability was flagged whereas others were not detected. Furthermore, only one false positive SQL injection vulnerability was given when scanning the BodgeIt Store.

4.2.3 Arachni

We installed Arachni version 1.2 from its homepage, and used the CLI for executing the scanner. All scan reports were transformed to html via the arachni reporter application which is included in the default installation.

Ease of installation & maintenance Arachni can be downloaded in a self-contained package, meaning no additional installation steps are required in order to execute Arachni.

Unlike sqlmap and ZAP, Arachni does not have a built in update system. Whenever a new update is available, users need to download that version again. Even though this is not the most easy way to upgrade, it does not require a large amount of effort due to the easy, almost non-existent, installation process. Thefefore, installation and maintenance of Arachni is an easy task.

Ease of configuration The CLI of Arachni comes with many different options to choose from. Setting up a scope is trivial with the arguments *–scope-include-pattern* and *–scope-exclude-pattern*. Furthermore, Arachni can automatically keep track of session ids, but an option to manually set such session ids is also available. Even setting used technologies can easily be achieved with the use of the argument *–platforms*.

On the other hand, some parts of configuration are less obvious. Enabling vulnerability types to scan for is done with the argument *–checks*, followed by the vulnerability scanner methods. Some types have more than one method to test for them. XSS is an example of this, and has eight checks that can be selected. Selecting them all separately would require a lot of checks to be selected, so it is made possible to select all by selecting *xss**. However, this also includes the check xst which checks for an entirely different vulnerability type.

Ease of integration Similar to sqlmap, Arachi is a command line tool. Automated scripts can easily call Arachni from the command line with several options in order to run scans.

However, interpreting the results might be more challenging. Arachni outputs a file in a custom *afr* format, which is not readable by default. The report needs to be transformed to an alternative, more readable format by the Arachni reporter. Unfortunately, those reports contain huge amounts of information which take quite some time and effort to figure out how to parse properly.

Ease of use Various arguments can be used to tweak the scan process of Arachni. The developers have created shortcuts to improve the usability. Scoping can be done with patterns. A single word can be entered to include or exclude all URLs that contain that word. This allows users to easily set up scoping without having to set difficult regular expressions or precise URLs. Also, enabling multiple checks for one vulnerability type is easy with the wildcard * character. By enabling the *sql** check, all checks for SQL injection are enabled. Entering long strings for multiple scanners of similar types is therefore unnecessary.

However, we also experienced less user-friendly parts in the CLI. Even though setting checks is easy, knowing which checks to set is less obvious. Checks have a brief description explaining their functionalities in one sentence, which is often not enough to grasp their full potential. Similarly, plugins hold many different features which are not obvious to inexperienced users. Finally, one major issue we encounters is the output. As mentioned, Arachni outputs a custom format which is unreadable unless transformed. The Arachni reporter has the ability to do so, but requires a different command compared to calling Arachni. We experienced this as very counter intuitive.

Handling output format The report format Arachni produces is not useful as it is not readable without a transformation. Therefore, extra effort is required in order to view the results of a scan.

Found URLs are all listed in a sitemap within the result. Reported issues follow up the sitemap in the result. All issues are sorted by vulnerability type, which has a name, description, and some remedy guidance. Then each location where that vulnerability type is found is presented along with a lot of request and response information. Even though all important information on vulnerabilities and URLs is present, the report quickly becomes unclear due to the amount of information that is presented.

Effectiveness Running Arachni on the benchmarks gave impressive results compared to the other scanners. Arachni only missed two states related to adobe flash in the WIVET benchmark.

Arachni was also the best among the tested scanners in the WAVSEP benchmark. Perfect detection scores were achieved for SQL injection and RFI. Only a few cases were missed for XSS and path traversal.

Finally, Arachni was the only scanner to detect both XSS vulnerabilities from the BodgeIt Store. However similar to sqlmap, it missed the SQL injection vulnerability that is present in the login page.

False positives Results from the WAVSEP benchmark show none of the false positive cases were flagged as positive. Furthermore, the false positive case in the BodgeIt Store that was detected by both sqlmap and ZAP was not found by Arachni. Therefore false positives seem less likely to be a problem.

4.2.4 W3af

W3af was downloaded from github. At the time of this experiment, the latest version was 1.7.6. Unfortunately, we were not able to properly install w3af. On the initial run of the execution script, w3af claimed it needed to download additional dependencies. It created a script that would automatically install all required dependencies. However, this process failed on certain dependencies. Manually trying to install those dependencies did not resolve the problem.

4.3 Discussion

During the experiments, we noticed that the studied scanners do not focus on usability. This is most notable by the lack of proper documentation. Documentation is often either incomplete, out of date, or hard to find at all. New users therefore need a lot of time and effort to learn how to use the scanners. Proper documentation would come a long way to improving the use of such scanners. However, that only addresses some usability issues while leaving some other issues, as described below, untouched.

It must be noted though that we studied scanners that are open-source and free, meaning they are likely maintained by a group of volunteers which spare time. Commercial tools that are developed by companies have much more resources to invest into proper documentation and support. Therefore, bad usability in such scanners is less likely. However, these tools are often very expensive which could scare developers.

The quality of the scanners were surprising. Arachni was also able to spider nearly all states of WIVET. Nearly all vulnerabilities of the WAVSEP benchmark were found by all scanners. However, we feel these scores might not be representative for real applications. Both benchmarks are known within the scanners community, so developers have likely investigated the cases within the benchmarks. We believe the scanners are optimized for those cases and therefore perform well. Running the scanners on the BodgeIt Store shows the performance of the scanners decreases on real-life applications as none of the scanners found all vulnerabilities. Similarly, Doupé et al. [27] concluded this as well in their study.

We also found the real problem behind false positives to be interesting. Initially, we figured the impediment of false positives was related to the number of findings that are not actually vulnerable. However, as shown in the WAVSEP results, not many false positives are reported. The downside to false positives is their return in every scan. No code will be changed as the location of the finding is not vulnerable. Therefore, subsequent scans will continue to report the finding. This is annoying and is distracting from other, real, findings.

Finally, we have noticed the importance of proper spidering in order to find vulnerabilities. For example, ZAP missed some XSS, RFI, and path traversal vulnerability cases from WAVSEP due to its inability to find certain states or pages. When we accessed those pages or states through the proxy, ZAP detected them and could find the vulnerabilities afterwards. Being able to detect as large of an attack surface as possible allows for the possibility of finding more vulnerabilities.

Chapter 5

UPeTeR

As argued in Chapters 3 and 4, a lack of adoption of security vulnerability scanners for web applications by web developers is, to a great extend, caused by the difficulty to use such tools. Even though security experts have much experience with operating security vulnerability scanners, they usually lack sufficient knowledge of the system under test (SUT) [22]. On the other hand, web developers do have the knowledge of their system, yet their know-how on security vulnerability scanners in insufficient. As is explained in Chapter 3, this is likely caused by the effort required in order to obtain adequate knowledge due to the scanners being difficult to use.

The solution for this problem proposed in this thesis aims to utilize the expertise of both user groups. Web developers should focus on specifying information about their web application, like authentication credentials. On the other hand, security experts should handle setting up security scanners in the most effective manner with respect to the information set by the web developers.

In this chapter we introduce the idea behind the Universal Penetration Testing Robot (UPeTeR). This chapter starts with an explanation of the philosophy behind UPeTeR and the problems it intends to solve. Next, a description of the processing steps within UPeTeR is given. This is followed up by an elaboration on the internal design of UPeTeR. Included in the elaboration are the configuration data objects, plugins that are to be created by scanner experts, the controller that is tasked with regulating the order of execution of the scanners. Finally, the design of the findings and the reporter are presented, which aim to provide developers with flexibility in the format of the results.

5.1 Process Description

The process designed to fulfill these ideas is illustrated in Figure 5.1. A developer initially sets the information required to perform a scan. This information is then used to determine the best configuration for a scanner. The scanner is run with the configuration and results are returned. These results are then transformed into any format desired by the developer.

As most web developers have insufficient knowledge to setup the scanners themselves effectively, UPeTeR relies on security experts to determine proper setups based on the information provided by developers. Additionally, security experts are also relied upon to instruct UPeTeR how to run scanners with the determined setup. Finally, UPeTeR facilitates a means for developers to structure output in a manner that is useful for them.

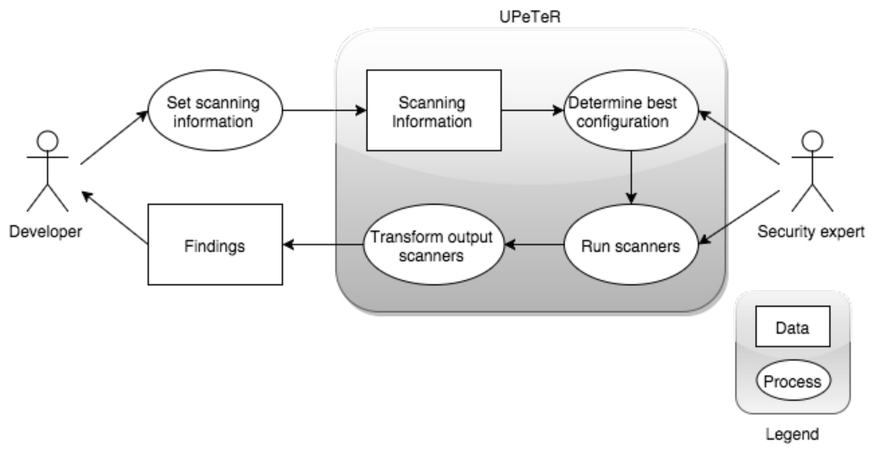


Figure 5.1: UPeTeR process

5.2 Design

Figure 5.2 depicts the information flow of the implementation of the process described in Section 5.1. The information required for scanning, which has to be set by the developer, consists of three objects:

- SUTConfig - The System Under Test configuration data containing information specific to the application to be tested (Sec. 5.2.1)
- SpiderOptions - Options that a developer can set to influence the spidering process (Sec. 5.2.2)
- AttackOptions - Options that a developer can set to influence the attacking phase (Sec. 5.2.3)

Determining the best configuration for scanners, running the scanner, and processing the results of the scanners is done by plugins (Sec. 5.2.4). These plugins are to be created by experts of scanners as they are best suited to transform the scanning information to an optimal setup configuration.

Since these plugins are created by experts, they are not considered to be a part of UPeTeR.

Finally, the reporter is responsible for transforming the output of the plugins to a form that is useful for the web developer. A collection of findings could suffice, however some cases will require an HTML or XML report.

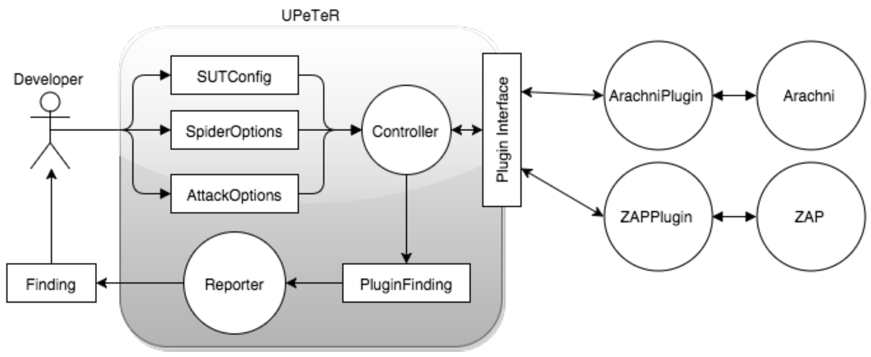


Figure 5.2: UPeTeR Flow Diagram

5.2.1 System Under Test Configuration

The SUTConfig is a data object containing information about the application that is tested. To be able to utilize as many different scanners as possible, we have created an abstraction of the information that is required by the scanners we have looked in to. Even though those scanners operate in different manners and require alternate steps to set up, the information they require is similar. A diagram of the SUTConfig information is given in Figure 5.3.

The first type of information that is required is scoping information. Scanners need to know which URLs belong to the application, as web pages could contain links to other websites. These other websites should not be scanned as they are not part of the application. On the other hand, it might also be the case that a certain section of the application should not be tested. This can be achieved by setting those URLs out of scope.

Many web applications contain content that is only available for authenticated users. To reach these sections, scanners need login information to automatically authenticate. The following data is required by scanners to achieve this:

- **Login URL** - The URL to which the login request is sent;
- **Logout URL** - The URL to logout of the application;

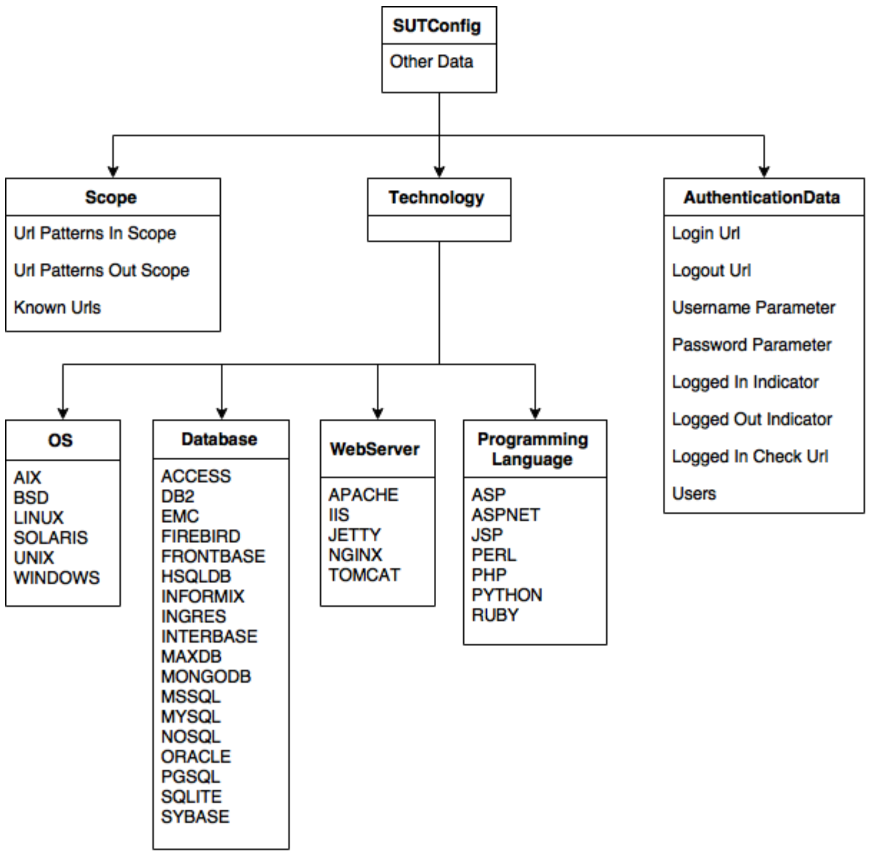


Figure 5.3: UPeTeR SUTConfig Diagram

- **Username parameter** - The parameter name as which the username is sent to the server;
- **Password parameter** - The parameter name as which the password is sent to the server;
- **Logged in indicator** - A pattern that is present whenever a user is logged in;
- **Logged out indicator** - A pattern that is present whenever a user is logged out;
- **Logged in check url** - To determine whether a login has succeeded, some scanners check this url for the logged in indicator;
- **Users** - User information (username & password) of zero or more users;

To increase the detection rate and decrease scanning time, scanners are often able to target specific technologies for their payload generation and attacks. SQL injection is an example of this, where it does not make sense to create MySQL specific attacks when the application uses a Microsoft SQL Server database. Four types of technologies are supported by scanners: databases, web servers, operating systems, and programming languages. Commonly supported technologies are represented in Figure 5.3.

5.2.2 Spider Options

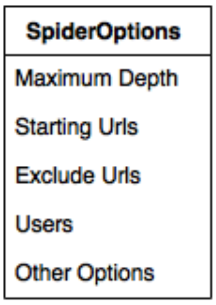


Figure 5.4: UPeTeR SpiderOptions Diagram

As is explained in Section 2.5, each reviewed scanner starts its analysis by spidering the application. Figure 5.4 depicts the SpiderOptions data object, which enables developers to influence this spidering process. Firstly, developers can set where to start the spidering from. Typically, this will be the root url of the application. Secondly, it might not be desirable to spider certain sections of the application. For instance, the url used for logging out of the application is not very useful to attack. Moreover, spidering the logout URL might result in the scanner not being able to log back in. Therefore, the SpiderOptions object contains information on URLs that should be excluded from spidering. Thirdly, users can be selected to access the authentication-protected sections of the application. Combining users to spider as together with the authentication data within the SUTConfig gives scanners enough information for logging in. Finally, developers can influence the spider quality and duration by inserting a maximum spider depth. The depth of the spider indicates the number of other pages traversed to reach a page. Altering the depth to spider can be useful in situations in which a section is infinitely deep, like a calendar in which each month is similar to other months. Spidering further into a calendar will likely not result in the discovery of additional vulnerabilities. Therefore, it is valuable to lower the spidering depth.

5.2.3 Attack Options

The spidering phase is followed by the attacking phase in which the scanners generate and send payloads to URLs discovered during the spidering

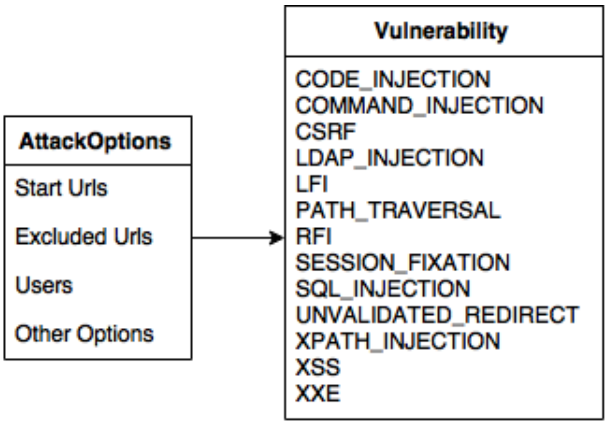


Figure 5.5: UPeTeR AttackOptions Diagram

phase. The AttackOptions data object (Fig. 5.5) provides developers with the ability to specify options for the attacking phase. Similar to SpiderOptions, AttackOptions contain information regarding URLs to exclude from attacking, and users to attack. Furthermore, it allows developers to specify which vulnerabilities should be scanned for. In the case where the application does not have any functionalities handling XML, scanning for XML External Entity (XXE) is not relevant. Not scanning for that vulnerability will therefore result in a reduced scanning time. A list of possible vulnerabilities, taken from commonly supported vulnerabilities by the scanners discussed in Section 2.5, is presented in Figure 5.5.

5.2.4 Plugins

The idea behind UPeTeR is to provide a means for developers to circumvent the intricacies of setting up and using scanners. By determining an optimal setup for each scanner from the application information in an automated fashion, developers are not required to learn the workings of each scanner. Since each scanner expects setup information differently, UPeTeR cannot do this out of the box. Therefore, people with experience in using a particular scanner have to provide this transformation. This is facilitated in the form of plugins which have to be created by people with expertise in a scanner. Each plugin is responsible for their own scanner, and have three main tasks:

- **Installation & Maintenance** - Installing scanners and keeping them up to date is time consuming and requires much effort. Therefore, plugins need to be able to assure developers that a scanner is working. Furthermore, it is desirable to keep scanners up to date with the latest version, as newer releases likely add functionalities or improve accuracy and performance.

- **Setup & Execution** - Without enough experience in using a scanner, it is difficult for developers to properly setup and execute them. By delegating these tasks to plugins allows developers to focus on providing relevant information about the application.
- **Result Handling** - No standardized output format exists for dynamic vulnerability scanners. This makes comparing and combining similar results of different scanners time consuming. Therefore, all plugins are required to transform scanner results into a single format, thereby enabling comparing and combining the results. This format is presented in Section 5.2.6.

5.2.5 Controller

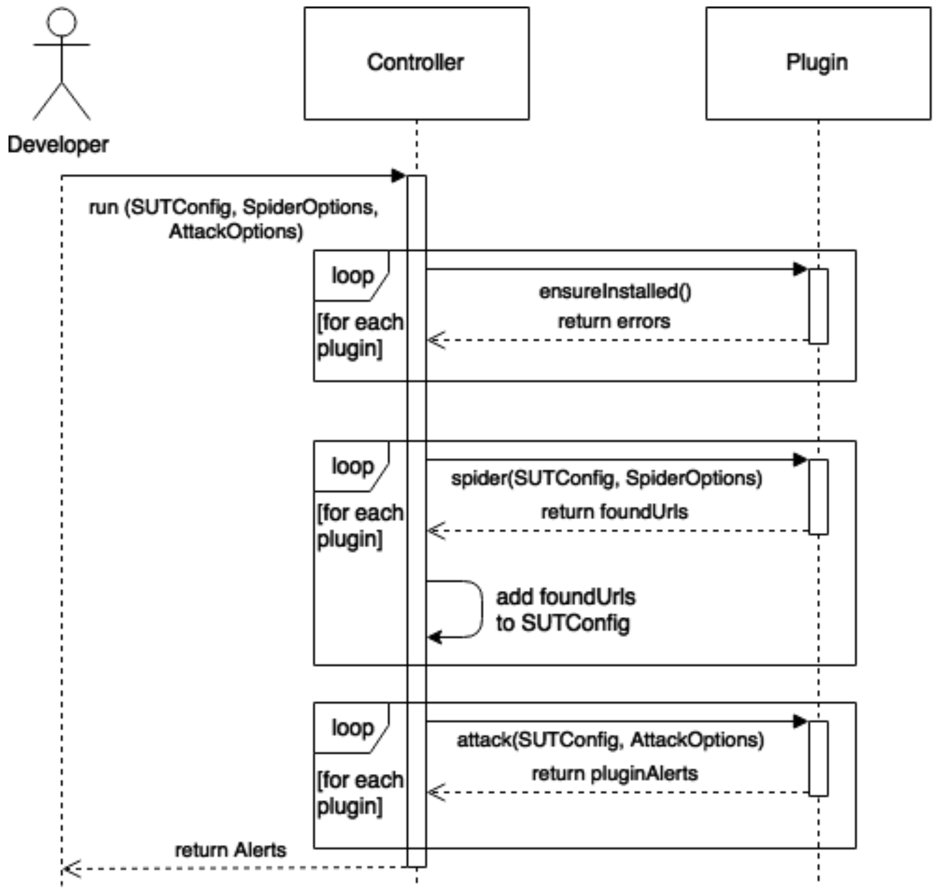


Figure 5.6: UPeTeR scanning phases UML Diagram

Calling plugins is done by the controller in three different phases, as depicted in Figure 5.6. Initially, the controller needs to determine which plugins can be executed, i.e. which scanners are installed and up to date.

Plugins should return any errors encountered during this phase so that developers can solve these problems. Furthermore, whenever at least one error is returned, the plugin cannot ensure the scanner is installed correctly. Therefore, the scanner will be excluded from the next two phases.

The next two phases represent the spidering and attacking phases that are used in each scanner, as is explained in Section 2.5. Both phases of each scanner can be executed sequentially before moving on to the next scanner. However, the ethical hacker interviewed in interview # 3 (Appendix A.3) explained the importance for scanners to have as much context information as possible. This includes the knowledge of urls that belong to the application. As Doupé et al [27] claim, spidering is arguably the most important part of finding security vulnerabilities, and many scanners have different approaches to spidering. Therefore, in order to get the best results out of the spidering process, the controller initially only runs the spider from all scanners. URLs found by spiders are inserted into the scoping information of the SUTConfig (Sec. 5.2.1). Since this data object is given to the plugins, the next spider can utilize the previously found URLs as additional information. After all scanners have spidered, the attacking phase is started in which all scanners are instructed to test for security vulnerabilities.

5.2.6 Findings

Developers should be able to easily use and interpret the results from all scanners. No standard output format exists for the results of scanners, so UPeTeR provides that in the *PluginFinding* object. Plugins transform the results of a scanner to the UPeTeR format. This format consists of the type of vulnerability found, the URL it is found on, and the parameter it is found on. Data on the manner of detection is also given to help developers understand how the vulnerability was discovered and how to reproduce it. Finally, most scanners provide information on a solution for a vulnerability to help developers solve the problem. This information is also present in the PluginFinding.

As multiple scanners can be used to find vulnerabilities, it is possible that *PluginFindings* from different scanners cover the same vulnerability. Just returning a list of *PluginFindings* could therefore result in an unclear list of findings of which some match and some might not. Combining similar results could thus be useful for developers to get a good overview. However, a desirable combination or format might differ for different developers. For instance, some developers might want the list of vulnerabilities to be sorted on vulnerability type, or on URL. The *Reporter* is designed to provide developers with the ability to specify their own final output format. This could be a list of aggregated findings, an XML file, or an HTML file. Default reporters can be implemented to provide often used output formats.

Chapter 6

Acceptance of UPeTeR

Due to the difficulties web developers face when trying to test a web application, scanners are rarely being used. UPeTeR was designed to increase the acceptance of these scanners and to overcome barriers for acceptance that were found through interviews. The following research question was formulated to investigate whether UPeTeR is accepted or not:

- Are web developers more willing to use vulnerability scanners if they are offered ways to hide the impediments?

This chapter presents the qualitative research that has been done to determine if the proposed abstraction is actually accepted by web developers. First of all, Section 6.1 explains the method used to do the research. This is followed by results of the research in Section 6.2. Finally, we conclude this chapter with a discussion of the research method and results.

6.1 Focus Group Design

To assess whether UPeTeR improves the acceptance of vulnerability scanners, web developers were asked to participate in a focus group. A total of ten developers participated and were divided into two focus groups, each consisting of five participants. Each session took roughly one hour, and was structured in three parts:

- Presentation
- Demonstration
- Acceptance questionnaire

Since web developers are the intended users of UPeTeR, the focus groups were formed by developers at SIG who have at least some experience in developing web applications. Even though the core business of SIG is consultancy, they have supporting web applications designed by developers and consultants. At the time of designing the sessions, a web application was being developed at SIG. Developers working on that applications were invited to participate in the focus group. Other participants were security consultants who have developed web applications in the past and were interested in the findings of this research.

6.1.1 Presentation

For developers to give their opinion on UPeTeR, they had to know how UPeTeR works and how it is designed. For that matter, each session started with a presentation explaining this research. Firstly, results of the initial interviews (Sec. 3.2) were shown to give participants an overview of found impediments. After that, the design of UPeTeR was given by using diagrams similar to the ones displayed in Chapter 5. The set of diagrams differed in complexity due to the limited space available on the slides. Lacking components were verbally addressed. The presentation concluded with a description on how developers can use UPeTeR.

6.1.2 Demonstration

The presentation was followed up by a demonstration of UPeTeR. The aim of the demonstration was to provide participants a visualization of UPeTeR in action, in order for them to better understand the workings of UPeTeR. Two things were required to perform the demo. First of all, a working implementation (prototype) of the design was needed. Second, an application to perform vulnerability scans on was required. Preferably, this application contains some vulnerabilities that can be detected by vulnerability scanners, so that results can be shown. The BodgeIt Store that was used in earlier experiments (Ch. 4) was used again for this.

Prototype

A prototype was created that partially implements the design of UPeTeR. Due to limited time available, we chose to focus on implementing the abstraction of setup data and the reporting of findings. Installing and maintaining features were left out. Those functionalities would take a lot of time to implement and that would be time consuming to demonstrate and is not required for developers to grasp its usefulness.

Plugins are needed for UPeTeR to execute scanners. All open-source scanners discussed in Section 2.5 were considered to be supported. However, due to time limitations not all of them were included. Sqlmap was left out as it solely scans for SQL injection. An attempt to create a plugin for Arachni was made, yet did not fully succeed. The Arachni plugin uses the command line interface to operate Arachni. The plugin can transform the setup information to parameters for the CLI, and execute Arachni with the parameters. However, Arachni outputs its results in a custom format which has to be transformed by the Arachni reporter executable. The reporter creates a file in a given format, which has to be processed by the Arachni plugin. This final process proved to be time consuming to implement and was thus left out. Fortunately, a fully functioning plugin was created for OWASP ZAP. Therefore, the prototype fully supports ZAP and partially supports Arachni.

Due to time constraints, only one default reporter was implemented for the prototype. This default reported merges equal findings of different scanners and returns a list of all merged findings. Two findings are considered equal if and only if the vulnerability type found, the url, and the parameter on which the vulnerability is found are the same. Merged findings contain payloads and proposed solutions of each scanner that found the vulnerability. By storing these in one finding, developers will not be overwhelmed by similar findings for each scanner and will get as much information as possible, such as triggers for the vulnerability and solutions to solve it.

6.1.3 Acceptance questionnaire

We loosely followed the method of Riemenschneider et al. [47] that describes a method based on a questionnaire to test the acceptance of methodologies. Other studies [21] have also used this methodology with success. Though the usage of a tool is different from usage of a methodology, there are certain points of similarity, suggesting that some tool adoption determinants may be applicable to methodology adoption decisions [47]. We have chosen to focus on the criteria ease of use, usefulness, compatibility, perceived behavioral control - internal, and behavioral intention. The first three criteria are part of the identified impediments in previous exploratory interviews (Ch. 3). Other categories were chosen due to their general applicability in acceptance. The questionnaire that was given to participants can be found in Appendix C. Questions were formulated as statements with which the participant could agree or disagree with. The different possible answers are based on a Likert scale, ranging from Strongly Disagree (1) to Strongly Agree (5). The final question was an open question that allowed participants to give their opinion on potential improvements.

6.2 Results

Figure 6.1 shows the outcome of the validation sessions for UPeTeR. Each bar represents the scores obtained for a question, where S1 is the first statement, S2 is the second statement and so on. Different sections of each bar represent the number of participants that rated the statement with the score that belongs to that section. For instance, one participant disagreed with

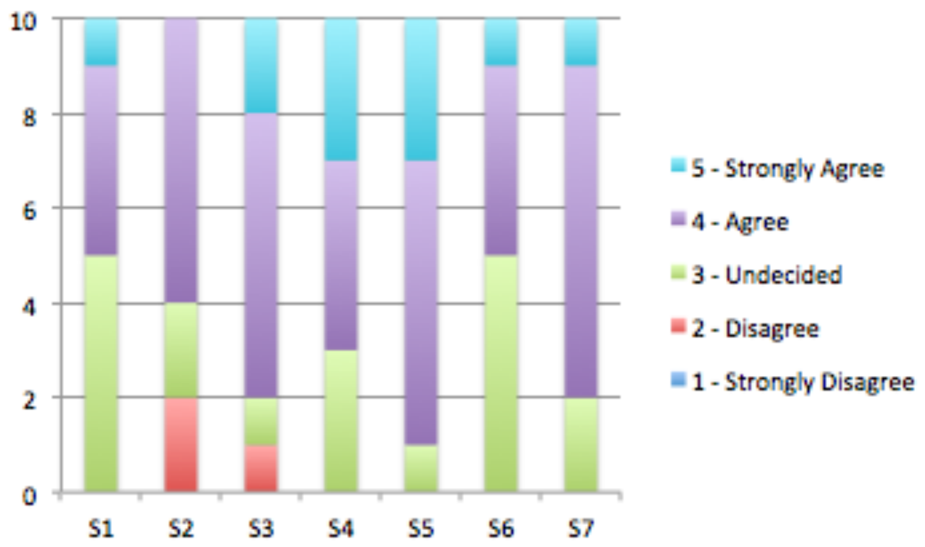


Figure 6.1: Scores given for valdation questions

the S3, one was undecided, six agreed, and two strongly agreed. As depicted in Figure 6.1, none of the participants strongly disagreed with any of the given statements.

Results shown in Figure 6.1 are supported by data given in Table 6.1. The results show that, on average, all participants agree with the given statements, indicating a positive stance towards UPeTeR.

S2 received the lowest scores. It had the lowest minimum, maximum, and average score of all statements. No one strongly agreed and two people disagreed with it. This indicates that participants were uncertain whether UPeTeR fits well in the current way of development.

S5 obtained the highest scores, with three participants strongly agreeing and six agreeing with it, while only one was undecided. These scores resulted in the highest average score of 4.2. Participants therefore think UPeTeR is useful for testing the security of a web application.

S1 and S6 have varying results, as in both cases four developers agreed with it while it left five others undecided.

6.2.1 Suggested Improvements

Participants were given the ability to write down any improvements they could think of.

Implement different interfaces to use UPeTeR. The prototype is implemented in the form of a class library, where its classes have to be imported into a user created test case in order to run the scans. Three participants indicated they would rather use a different interface. Examples

Statement	Min	Max	Median	Average
1	3	5	3.5	3.6
2	2	4	4	3.4
3	2	5	4	3.9
4	3	5	4	4
5	3	5	4	4.2
6	3	5	3.5	3.6
7	3	5	4	3.9

Table 6.1: Statistical results of validation questions

could be a user interface, a headless interface that utilizes a configuration file which the developer has to create, or even integrated interface into a continuous integration environment.

Implement more plugins. The success and usefulness of UPeTeR largely depends on the number of vulnerability scanners it can command. Therefore, implementing plugins for more scanners is vitally important. The prototype can only fully control OWASP ZAP, and can partially utilize Arachni. Implementing plugins for more scanners requires adequate knowledge of the scanners. Two participants mentioned this improvement.

Support testing of more technologies and vulnerabilities. With the rise of *Web 2.0*, the internet has become increasingly diverse in technologies being used by web applications [45]. Many applications implement lightweight approaches like SOAP- and REST-based services. Furthermore, developers often resort to scripting languages like JavaScript to interact with web pages from within the browser [44]. New methods are required to test these new technologies. Fortunately, vulnerability scanners are developing methods to detect new vulnerabilities inside the technologies. For instance, a SOAP scanner is being developed for OWASP ZAP. Also, the Crawljax spider[42] and the built in spider of Arachni can cope with JavaScript. However, as UPeTeR utilizes the capabilities of vulnerability scanners, implementing ways to test new technologies is outside of the scope of UPeTeR.

Filter out false positives. As was discovered during initial interviews (Sec. 3.2), false positives are a major issue when doing security scans. It takes time to determine whether a finding is truly a vulnerability. Having to determine the validity of the same vulnerability is therefore a tedious and time consuming task. UPeTeR does not yet introduce a means to combat this, and is therefore an issue for future researchers to solve. This will be further discussed in Section 7.3.3.

Create output in a meaningful way. The prototype only contained one default output format. Participants therefore suggested to increase the number of ways to output results.

The presentation that was given and the prototype that was created for the focus groups was mainly focused on the abstraction of setup data.

Due to time constraints, only a default reporter was implemented and shown. In hindsight, the potential use of the reporter was not displayed which resulted in this suggestion. Even though the design of UPeTeR has this suggestion covered, the feedback from the participants suggests they value useful default reporters so developers do not have to put effort into that themselves.

Chapter 7

Conclusion

The contributions of this study are:

- An identification of impediments encountered by web developers when using dynamic security vulnerability scanners for web applications through interviews;
- A presentation of impediments found in practice through a qualitative study of using scanners on several benchmarking applications;
- A proposed design for a universal vulnerability scan runner that uses an abstraction of data required for scanners;
- A partial implementation of the proposed design in the form of a prototype;
- An evaluation of the acceptance of the proposed design and an analysis of potential future improvements;

7.1 Results

The goal of this research was to improve the detection of security vulnerabilities in web applications early on in their development process. To do this, we investigated how web developers make use of dynamic vulnerability scanners and how this can be improved. In this section, we discuss the results regarding the research questions that were established in the beginning.

RQ1 - How widespread is the use of dynamic vulnerability scanners by web developers during development processes?

Initially, we asked ourselves whether web developers actually use dynamic vulnerability scanners during developing web applications. To answer this question, we performed interviews with five people. Two of them were web developers, two others were security consultants, and one was an ethical hacker. All subjects claimed dynamic vulnerability scanners are rarely being used by web developers at all, let alone during the development process.

RQ2 - What are the impediments for web developers to make use of dynamic vulnerability scanners?

The interviews continued with questions to find these impediments. We transcribed all interviews and coded parts which we deemed relevant to the research question. Thematic analysis was performed to combine several similar codes into a single theme. The results show three main problem areas for web developers to start using dynamic vulnerability scanners. First of all, scanners are not easy and intuitive to use. Setting up the tools correctly is perceived to be a difficult and time-consuming task. Scanners therefore require a substantial amount of time and effort to operate and integrate properly. This brings us to the second problem. Even though companies are aware of the importance of security, it is not given much attention. Functionality is considered to be more important and the amount of effort required to use the scanners is therefore deemed too much. This lack of effort is increased by the idea that the scanners do not perform well. The number of issues found by those tools is perceived low, and what is found can easily be spotted manually.

Next, we experimented with four free dynamic vulnerability scanners to verify these findings. All scanners were used to scan two different benchmarking applications and one webshop application which included several intentional vulnerabilities. Scanners were evaluated on the issues found during the interviews. The issues of usability were confirmed, as all scanners required decent amount of knowledge to operate properly, which took a substantial amount of time and effort to obtain. Additionally, results of all scanners were different and therefore difficult and time consuming to compare. On the other hand, performance was found to be decent yet diverse between different scanner. We concluded that each tool has its own strengths and weaknesses.

RQ3 - Are web developers more willing to use vulnerability scanners if they are offered ways to hide the impediments?

Finally, in order to improve the use of dynamic vulnerability scanners, we created the Universal Pen Test Robot (UPeTeR). UPeTeR uses an abstraction of information required by studied scanners to performed a scan. This allows developers to focus on providing that information about the web application, while security experts can use their knowledge to create plugins for scanners that transform the abstraction of information into a proper configuration. Additionally, all results are transformed into one format. UPeTeR also enables developers to further alter the results in the form of a reporter.

To answer the research question, we had to establish whether UPeTeR is accepted by web developers. This was done through a questionnaire that was given to participants of two focus groups consisting of five web developers each. Participants were given a presentation and demonstration of UPeTeR, after which they filled in a questionnaire on their acceptance of UPeTeR. Results indicate a positive stance towards UPeTeR, and some improvements were suggested to increase the usefulness of UPeTeR even further.

7.2 Threads to Validity

Several threads have been identified that could weaken the validity of the results. These threads have been split into three types as suggested by Runeson and Höst[48].

7.2.1 Construct Validity

Construct validity is an aspect of validity that reflects on whether the actual measurements represent what the experiment intends to measure [48].

Partial implementation of acceptance methodology We used the method described by Riemenschneider et al. [47] to measure the acceptance of UPeTeR, yet not all factors were included in our study. Instead, we focused our questions on the impediments found in initial interviews (Ch. 3). Those results show factors for the current lack of adoption, so we therefore focused on those factors. However, one thread to construct validity could be that factors that were not deemed troublesome in initial interviews have become worse in UPeTeR.

Lack of hands on experience During the focus groups, participants were given a presentation and a demonstration on UPeTeR to familiarize them with its functionalities and capabilities. However, participants were not given the opportunity to work with UPeTeR and test it for themselves. Therefore, the possibility remains that the participants did not fully grasp the usability of UPeTeR.

7.2.2 Internal Validity

Internal validity applies to causal relations between my actions and the subjects responses, and whether those responses are not affected by other factors [48].

Participant sample of validation Participants of the focus groups were all employees of SIG, effectively making them our temporary colleagues which could have introduced bias. We have attempted to mitigate this by explicitly asking participants to answer according to their opinion and not let that be influenced by being colleagues. Additionally, all participants have done research and we therefore trust their professionalism and understanding of the importance of unbiased feedback.

Statement formulation in validation questionnaire Another improvement would be to change the formulation of statements of the questionnaire. All statements were structured in a positive manner and could therefore have influenced the participants. Unambiguous or alternating positively and negatively structured statements could have further reduced the risk of bias.

7.2.3 External Validity

External validity is related to the ability to generalize the findings [48].

Generalization of initial interviews Initial interviews were only held with five people, which is a limited sample size. This limits the generalizability of the findings. Experts from different view points were interviewed in order to mitigate this thread. For instance, the ethical hacker and the two security consultants have seen a large variety of working environments, so their experience is more diverse than the experience of interviewed web developers who have only working in a few different companies and environments. However, more subjects need to be interviewed for better results.

Generalization of impediment verification Our verification of found impediments is purely our own experience. We did not have any experience prior to this study and can therefore be considered generic web developers. Therefore it is plausible other generic web developers experience similar issues. However, the experience of other web developers would need to be studied to improve the results.

Open-source vs commercial scanners We have chosen to only study open-source scanners to verify impediments, as these are freely accessible to everyone and code can be viewed to determine internal workings. Therefore, we do not know whether commercial scanners have similar issues.

Generalization of validation All participants of the focus groups were employed at SIG, thus the sample of participants is limited. This in turn limits the generalizability of the results. On the other hand, all participants have different levels of experience with web developers. The sample therefore does represent different levels of expertise.

7.3 Future Work

Due to time constrains, many aspects of this research have not been executed as precise as we would like. Future work will therefore need to be done to improve UPeTeR and the results of this study. This section presents possible extension points for this research.

7.3.1 Impediments Analysis

This study started with an analysis of the acceptance of the current web vulnerability scanners by web developers during their development processes, and what possible impediments could be for developers not to use the scanners. The interview sample consisted of five participants from different companies, which impacts the ability to generalize the results. Further research could pursue a larger and broader sample to better qualify the results.

7.3.2 UPeTeR Implmentation

The design of UPeTeR was only partially implemented in the prototype for the demonstration. Only ZAP is fully supported by a plugin, while arachni is partly implemented. Further research could investigate how to properly setup other scanners.

Furthermore, new web technologies become available and mainstream and techniques to test and scan those technologies will also arise. It could be that those techniques require different information to start the scan. For instance, Crawljax allows users to specify which elements on which pages have JavaScript actions attached to them, and what values are valid for specific forms [42]. Keeping the abstraction of information these scanners require is vital to the success of UPeTeR.

Finally, more research should also be done to find out what types of output is generally wanted and required by developers. As of now, only one default reporter has been created to visually show the results of the scans. However, continuous integration tools like Jenkins [9] require HTML or XML reports to interpret and present the results.

7.3.3 False Positive Handling

One of the impediments that was identified in the interviews was false positives and the amount of effort that is required to process them. Therefore, repetitive false positive results cost much time and money. This research has not looked into solutions for this problem, which thereby remains a problem that needs to be addressed in further research. Since UPeTeR is able to utilize multiple scanners, we believe research in the area of statistical analysis could be a good option. If findings could be statistically analyzed, developers could filter on the percentage of certainty of the finding being a true positive. This would allow for the removal of findings with a low prediction.

7.3.4 Acceptance Analysis

The two focus groups with which the acceptance analysis was performed solely contained developers employed at SIG. The background of the participants differed, as some had more experience with security while others practiced in web development, yet they all work in the same working environment. SIG does develop their own software including web applications, yet this is not their core business. It might be the case that focus groups with developers in different working environments would have a different outcome. Further research is required to confirm or contradict this.

Bibliography

[1] Arachni homepage. http://www.arachni-scanner.com.

[2] Bodgeit store homepage. https://code.google.com/p/bodgeit/.

[3] Bodgeit store source code. https://github.com/psiinon/bodgeit.

[4] Content security policy. https://www.owasp.org/index.php/Content_Security_Policy.

[5] Content security policy support statistics. http://caniuse.com/#feat=contentsecuritypolicy.

[6] Crawljax homepage. http://crawljax.com/.

[7] Facebook login. https://developers.facebook.com/docs/facebook-login/v2.4.

[8] Github page of sqlmap. https://github.com/sqlmapproject/sqlmap.

[9] Jenkins homepage. https://jenkins-ci.org/.

[10] Microsoft .net framework. https://www.microsoft.com/net.

[11] Owasp. https://www.owasp.org/index.php/Main_Page.

[12] Owasp bwa. https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project.

[13] Owasp path traversal. https://www.owasp.org/index.php/File_System#Path_traversal.

[14] Owasp top 10 vulnerabilities. https://www.owasp.org/index.php/Top_10_2013-Top_10.

[15] Sql injection prevention cheat sheet. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet.

[16] Sqlmap homepage. http://sqlmap.org/.

[17] W3af homepage. http://w3af.org.

[18] Zap homepage. https://www.owasp.org/index.php/ZAP.

[19] Reflected xss image. http://4.bp.blogspot.com/-s-yIqt30cXI/U-Ef3aDtYEI/AAAAAAAAAew/uxVMedhggxc/s1600/diagram.png, August 2014.

[20] Veel veiligheidslekken bij webwinkels. http://www.consumentenbond.nl/veilig-online-betalen/extra/veiligheidslekken-bij-webwinkels/, May 2015.

[21] Anwar Aldris, Ariadi Nugroho, Patricia Lago, and Joost Visser. Measuring the degree of service orientation in proprietary SOA systems. *Proceedings - 2013 IEEE 7th International Symposium on Service-Oriented System Engineering, SOSE 2013*, pages 233–244, 2013.

[22] Jim Bird, Eric Johnson, and Frank Kim. 2015 State of Application Security: Closing the Gap. Technical report, SANS Institute, 2015.

[23] Barry Boehm. Barry Boehm Software Engineering Economics. *IEEE Transactions on Software Engineering*, 10(1):4–21, 1984.

[24] Shay Chen. The web application vulnerability scanners benchmark. http://sectooladdict.blogspot.ro/2014/02/wavsep-web-application-scanner.html, 2 2014.

[25] Dan Cornell. Remediation statistics: What does fixing application vulnerabilities cost? 2012.

[26] Daniela S. Cruzes and Tore Dyba. Recommended Steps for Thematic Synthesis in Software Engineering. *2011 International Symposium on Empirical Software Engineering and Measurement*, (7491):275–284, 2011.

[27] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6201 LNCS:111–131, 2010.

[28] Stefan Frei. The Known Unknowns. *Scientific American*, 311(1), 2013.

[29] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-Scale Vulnerability Analysis. 2006.

[30] Stefan Frei, Bernhard Tellenbach, and Bernhard Plattner. 0-Day Patch Exposing Vendors (In) security Performance. *Retrieved from http://www. blackhat. com/presentations/bh-europe-08/Frei/Whitepaper/bh-eu-08-frei-WP. pdf on September*, 14:2009, 2009.

[31] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20, 2012.

[32] Jeremiah Grossman. Cross-Site Scripting Worms & Viruses. (June):1–18, 2007.

[33] William G J Halfond, Shauvik Roy Choudhary, and Alessandro Orso. Improving penetration testing through static and dynamic analysis. *ICST 2009, the Second IEEE International Conference on Software Testing, Verification and Validation*, Volume 21(Issue 3):195–214, 2011.

[34] William G J Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL Injection Attacks and Countermeasures. *Preventing Sql Code Injection By Combining Static and Runtime Analysis*, page 53, 2008.

[35] I. Heitlager, T. Kuipers, and J. Visser. A Practical Model for Measuring Maintainability. *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39, 2007.

[36] Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. *Proceedings - International Software Metrics Symposium*, 2005(Metrics):203–212, 2005.

[37] Leon Kappelman, Ephraim Mclean, and Natalie Gerhart. The 2014 SIM IT Key Issues and Trends Study. 2014(December), 2014.

[38] Amit Klein. Dom based cross site scripting or xss of the third kind. http://www.webappsec.org/projects/articles/071105.shtml, July 2005.

[39] Tobias Kuipers. Monitoring the Quality of Outsourced Software. *International Workshop on Tools for Managing Globally Distributed Software Development (TOMAG 2007). Center for Telematics and Information Technology*, 36, 2007.

[40] Project Leaders, Matteo Meucci, and Andrew Muller. Testing Guide 4.0. (Cc).

[41] G. McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.

[42] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web*, 6(1):1–30, 2012.

[43] Glenford J Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing.* John Wiley & Sons, 2011.

[44] Alex Nederlof, Ali Mesbah, and Arie Van Deursen. Software engineering for the web: the state of the practice. *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, pages 4–13, 2014.

[45] Tim O'Reilly. What is Web 2.0: Design patterns and business models for the next generation of software. *Communications & strategies*, 65(4580):17–37, 2007.

[46] Linda Dailey Paulson. Web Applications with Ajax. *IEEE Computer*, 38(10):14–17, 2005.

[47] Cynthia K. Riemenschneider, Bill C. Hardgrave, and Fred D. Davis. Explaining software developer acceptance of methodologies: A comparison of five theoretical models. *IEEE Transactions on Software Engineering*, 28(12):1135–1145, 2002.

[48] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[49] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Have things changed now? An empirical study on input validation vulnerabilities in web applications. *Computers and Security*, 31(3):344–356, 2012.

[50] C.B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[51] Muhammad Shahzad, M Zubair Shafiq, and Alex X Liu. A Large Scale Exploratory Analysis of Software Vulnerability Life Cycles. *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[52] Glenn A. Stout. Testing a Website : Best Practices. 2001.

[53] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. *ACM SIGPLAN Notices*, 41(1):372–382, 2006.

[54] Stephen Thomas and Laurie Williams. Using automated fix generation to secure SQL statements. *Proceedings - ICSE 2007 Workshops: Third International Workshop on Software Engineering for Secure Systems, SESS'07*, 2007.

[55] Erik van Veenendaal. Standard glossary of terms used in Software Testing Produced by the ' Glossary Working Party ' International Software Testing Qualification Board. 1:1–36, 2005.

[56] Kristian Wiklund, Daniel Sundmark, Sigrid Eldh, and Kristina Lundvist. Impediments for automated testing - An empirical analysis of a user support discussion board. *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*, pages 113–122, 2014.

[57] Jim Witschey, Olga Zielinska, and Allaire Welk. Quantifying Developers ' Adoption of Security Tools.

[58] Shundan Xiao, Jim Witschey, and E Murphy-Hill. Social Influences on Secure Development Tool Adoption: Why Security Tools Spread. *Computer Supported Cooperative Work (CSCW)*, pages 1095–1106, 2014.

[59] Jing Xie, Heather Richter Lipford, and Bill Chu. Why do programmers make security errors? *Proceedings - 2011 IEEE Symposium on Visual Languages and Human Centric Computing, VL/HCC 2011*, pages 161–164, 2011.

[60] Haiyun Xu, Jeroen Heijmans, and Joost Visser. A practical model for rating software security. *Proceedings - 7th International Conference on Software Security and Reliability Companion, SERE-C 2013*, pages 231–232, 2013.

[61] Yuchen Zhou and David Evans. Why aren't HTTP-only cookies more widely deployed. *Proceedings of 4th Web*, 2010.

Acronyms

AJAX - Asynchronous JavaScript And XML

API - Application Programming Interface

CLI - Command-line Interface

CSP - Content Security Policy

CSV - Comma-separated Values

CSRF - Cross-Site Request Forgery

CWE - Common Weakness Enumeration

DAST - Dynamic Analysis Security Testing

DBMS - Database Management System

DOM - Document Object Model

GUI - Graphical User Interface

HTML - Hypertext Markup Language

HTTP - Hypertext Transfer Protocol

JSON - JavaScript Object Notation

LFI - Local File Inclusion

OWASP - Open Web Application Security Project

REST - Representational State Transfer

RFI - Remote File Inclusion

SAST - Static Analysis Security Testing

SIG - Software Improvement Group

SOAP - Simple Object Access Protocol

SUT - System Under Test

SQL - Structured Query Language

UML - Unified Modeling Language

UPeTeR - Universal Penetration Testing Robot

URL - Uniform Resource Locator

XML - Extensible Markup Language

XSS - Cross-Site Scripting

XXE - XML External Entity

YAML - YAML Ain't Markup Language

Appendix A

Exploratory Interview Summaries

A.1 Interview #1

A.1.1 Experience with dynamic vulnerability scanners

At SIG, we wrote our own selenium tests for testing security. We chose to use selenium because we were already using it. This way, we didn't have to use too many different tools for testing. However, the downside of this approach is that this does not test security for all input fields. Therefore, you do not have 100% test coverage. We do not even know how much coverage we actually have. What would be better is a tool that automatically tests every input field.

I also have some knowledge of SQLMap. I found that SQLMap is very difficult to use and only tests a small part. It is nice for testing SQL injection, but that is not all you want to test for. If other tools also solely check for one vulnerability, you would need a lot of different tools. This takes too much time, since tools like SQLMap are not created for integration. For that, you want to be able to put it into the build pipeline, but that is not possible. However, SQLMap finds vulnerabilities and that makes it useful.

Furthermore, a possible downside of integrating these tools into the build pipeline could be that, if running the tests takes a lot of time, it increases the building process. This should be fast.

False positives could also be a problem, but there is no way to prevent this. It is better than nothing.

A.1.2 State of security testing in companies

Generally, companies do not use vulnerability scanners. This is due to nescience. For many companies, the testing process is still in its infancy. Agile sprints are being done, but after that the software is given to a test team who will test for two weeks by clicking through the application. There is no build pipeline that automates these steps. A lack of knowledge on security vulnerabilities could be a cause, but I do not think this has to be an issue. Basic stuff like SQL Injection is easy to explain and everyone will encounter these at least once in his or her career.

Companies do not make a conscious choice for only using penetration testers to perform security tests. This is the current concept of security. We, at SIG, also did this for our own website. No security requirements were made, so people became agitated at the end of the development. Only then did we start to look at security. This is too late, because we would have to change everything in hindsight if something would have been vulnerable. Nescience is not the enemy here, since people know security is important. However, the realization that this has to be controlled during the development process is a next step.

I would advise companies not to use vulnerability scanners. Too many steps are required to install the tools and to keep them working. Manual use of these tools would be an achievable solution. However, integrating these tools into the build pipeline is not feasible for many companies. This is because many tools are not designed for integration.

A.1.3 Requirements of a security test approach

Tools should also be usable for integration tests. Companies do not use the tools because of logistic reasons. They do know the tools exist, but they do not know how to use and integrate them. Thus, a tool would need to be easily integratable, installable, and configurable. Developers should be able to use a tool with a reasonable efficiency.

A.2 Interview #2

A.2.1 Experience with dynamic vulnerability scanners

I have mixed experiences with tools, one of which is IBM AppScan. All these tools have a high percentage of false positives, only 10 to 20% is correct. In order to increase performance, you need to configure it for the system you want to test. This is not a problem when testing during the development process, since you would only have to do it a couple of times. On the other hand, issues that these tools do not find are usually more interesting. When doing manual tests, you usually come across problems that are far worse than the issues these tools find. Therefore, the tools are useful as a start, but they should not be relied upon.

Furthermore, it is difficult for tools to find vulnerabilities specific for a system. SQL Injection is an easy example. The methods to abuse that vulnerability are commonly known and can be easily automated. However, log-in pages always contain pieces of self-written code. Even if you use a framework, you will always need to glue components together and that is were 100 things could go wrong. Every system has something unique and tools cannot anticipate on this.

I think tools can be used during the development process, but only in a supportive role, and mostly by penetration testers. I have seen them being used very rarely.

If you know the tool and the system you want to test, then you can tune the tool to your system. However, this requires a lot of effort for very few results. Therefore, it is a high investment to use such tools. I have the feeling this is a big impediment for companies. Maybe it is better to use them manually once a year, but then you do not have the continuous assurance.

Connecting different tools into CI is difficult. You have to connect the tool to the build pipeline, new versions of the tool and they might return a slightly different output format. My experience is that it is often required to put effort into keeping the tools running.

A.2.2 State of security testing in companies

In general, security is important for companies. However, this does not mean that a commensurate amount of time, attention, and money is put in it. Usually, there is no structured approach. This leads to security by accident. Developers quite often know little to nothing about security. Also, there is always a battle between security and deadlines. That makes it that there is no structured approach for dealing with security.

Even though using the tools does not require security knowledge, solving the found problems does. Therefore, a generic web developer still would be clueless on what to do. Searching for a solution will only take time, but it does not become clear immediately.

A.2.3 Requirements of a security test approach

Not much is needed for writing integration security tests. Usually you can use your favorite tool to send requests and receive responses. This way, it is easy to test if you are blocked after a number of failed log-in attempts. I would use other tools for trying to test for SQL Injection and similar kinds of vulnerabilities. Also, I would like to write regression tests for security bugs.

Using multiple tools for testing similar vulnerabilities would probably improve the results. On the other hand, this requires knowledge of many tools. This would take time. So for every tool, easy integration and good usability is required.

A.3 Interview #3

A.3.1 Experience with dynamic vulnerability scanners

For my profession, I came in contact with those kinds of tools a lot of times. Some of them are really helpful, because they can take away a lot of manual pen testing from the pen tester. However, they also lack a lot of testing. I found out it is really hard for a security test tool to have the right context to determine whether a security issue is present. They try to fuzz or use algorithms to cover that, but good tools only find about 20% of the issues.

Those 20% are useful of course, but there are also a lot of false positives. Scanning large websites will therefore produce many false positives. This takes quite a lot of time to process.

Also, when scanning large websites, scanning takes a long time. This is because the tool does not possess the right context. Tools will fully scan static pages, even though they can't be interacted with. The more context you have, the more precise you can attack.

Another problem with trying automated scanning is that it takes more time than one would think. I tried this at another company and it failed horribly. To get good results during development, you need to adjust the configuration of the tools to match the current version of the website under test. This still takes a lot of time.

What I would suggest to developers is to use the tools, because they can take away a lot of manual testing when used tactfully. If you do not use it tactfully and let the tools run over everything, the added value is going down because you will get a lot of false positives, which you will have to analyze.

Tools are good at finding SQL injection and XSS vulnerabilities. Some can also find command injections or CSRF vulnerabilities. However, finding more advanced vulnerabilities is much harder for the tools. Tools cannot test application logic. That is a big flaw. I would create special unit tests to test specific application functionalities on logic flaws.

When I did pen testing at my previous job, I would always suggest having a meeting with the technical person to get all the context of the application. Sometimes, using the knowledge of the developer can help me create the context. That is why developers should write security tests, because they know the context of their application best.

A.3.2 State of security testing in companies

Normally, developers have limited time to deliver a project. Then you have the challenges of getting all the functions working and delivering those on time. Usually, security is not mentioned in there. Some developers are aware of security, but they do not have the time to implement it properly. That is why security is rarely tested. The knowledge is there, but their environment limits them.

A.3.3 Requirements of a security test approach

I think it would be helpful if you could create a configuration that tells the tools which technologies are used in the background in which places. This would save the tools from testing xpath injections in a place where only XML technologies are used.

A.4 Interview #4

A.4.1 Experience with dynamic vulnerability scanners

In the past, I have worked with SkipFish and ZAP. My experiences with those are that scans can take a long time to run, and only easy mistakes are found. Using knowledge of the application can get you a lot further than using tools that do not have that information. A vulnerability scanner knows nothing about the context of a web page. If a form is found, it does not know what the fields mean. Therefore, only random data is used. Or the check-box to accept the terms of condition is not being set, making the next page impossible to access. A manual tester will never forget this, but a tool might.

Using the tools in the development process could be useful as a safety net. Especially because it is automated so it will not take much time to run the tests. However, I have the feeling it is more important to use the latest development technologies. When you develop an application that needs authentication, why not use facebook authentication for example. They have thought a lot about it, they have experience with it. That way, you do not have to create your own authentication system, which could introduce many vulnerabilities.

A downside of using these scanners is that it can contaminate the application with useless data. Therefore, companies would need to set up a separate testing environment.

A.4.2 State of security testing in companies

Security is important for companies, but it is not always being tested. If it is tested, it is usually done by a separate security team. However, functionality is usually more important than security. Security is not explicitly asked for.

A.4.3 Requirements of a security test approach

I would like to be able to set the context of a website. Also, I would like to see what payloads are used by tools to test for certain vulnerabilities. It is not always clear what the tools do exactly.

A.5 Interview #5

A.5.1 Experience with dynamic vulnerability scanners

I have only heard of such tools, but never actually used them. I do know if you want to use the tools in CI, than you would need to configure build steps that run those tools.

For one of our customers, security is very important. They have a tool running that tries to generate errors inside the application and analyses the output. However, it cannot log in to the application, or check for privilege escalation.

Development frameworks take care of many possible vulnerabilities like SQL Injection. Therefore, SQL injection is impossible if you use the newest technologies.

A.5.2 State of security testing in companies

We do think security is important, but we have only been attacked twice in the last six years. Those attacks were not successful, so we have never been hacked. Therefore, we do not give it a lot of attention.

Currently, we do not test for security at all. Our customers are not wealthy, and creating and running security tests takes time and therefore cost money.

We also believe it is more important to test for application specific problems. I trust the framework we use that unauthorized people cannot log in. However, making sure that authorized users only have access to the pages they are authorized for, that is something application specific and therefore requires custom tests. We do not test this due to a lack of knowledge on how to test it, and it costs a lot of extra time and money. Also, customers do not ask for these tests.

A.5.3 Requirements of a security test approach

I do think it is useful to integrate automated scanners in the development process. However, current software development frameworks like Microsoft's .Net Framework has vulnerabilities like SQL Injection and XSS covered. Having a way of automating application specific tests would be very helpful and would save a lot of time.

It does not really matter if a scan takes a long time. You are not performing the scan but a machine is. If it takes long, maybe you do not want to start the scan after every commit, but only when you merge to a staging environment. Even if the scans take hours, as long as you do not have to do it yourself, it does not matter.

The use of the tools should be simple. It should be easily configurable. The best option would be to have only one tool that tests everything instead of many tools that each only test a bit. However, using more than one tool might also find more issues. If that is the case, than more would be better, but it should be easy to use.

Finally, something should be done with false positives. You do not want the same false positive popping up every time you run the tests. Something should be found for that.

Appendix B

Reference Codes

Ease of Use	
Interview #1	I found that SQLMap is very difficult to use
Interview #2	So for every tool, easy integration and good usability is required.
Interview #5	The use of the tools should be simple. It should be easily configurable.

Ease of Installation & Maintenance	
Interview #1	Too many steps are required to install the tools and to keep them working.
Interview #2	My experience is that it is often required to put effort into keeping the tools running. So for every tool, easy integration and good usability is required.

Ease of Context Configuration

Interview #2	In order to increase performance, you need to configure it for the system you want to test. This is not a problem when testing during the development process, since you would only have to do it a couple of times. If you know the tool and the system you want to test, then you can tune the tool to your system. However, this requires a lot of effort for very few results.
Interview #3	To get good results during development, you need to adjust the configuration of the tools to match the current version of the website under test. This still takes a lot of time. I think it would be helpful if you could create a configuration that tells the tools which technologies are used in the background in which places. Also, when scanning large websites, scanning takes a long time. This is because the tool does not possess the right context. Tools will fully scan static pages, even though they cannot be interacted with. The more context you have, the more precise you can attack. When I did pen testing at my previous job, I would always suggest having a meeting with the technical person to get all the context of the application. Sometimes, using the knowledge of the developer can help me create the context. That is why developers should write security tests, because they know the context of their application best.
Interview #4	A vulnerability scanner knows nothing about the context of a web page. If a form is found, it does not know what the fields mean. Therefore, only random data is used. Or the check-box to accept the terms of condition is not being set, making the next page impossible to access. I would like to be able to set the context of a website
Interview #5	The use of the tools should be simple. It should be easily configurable.

Ease of Integration	
Interview #1	Tools like SQLMap are not created for integration. For that, you want to be able to put it into the build pipeline, but that is not possible. However, integrating these tools into the build pipeline is not feasible for many companies. This is because many tools are not designed for integration.
Interview #2	Connecting different tools into CI is difficult. You have to connect the tool to the build pipeline.

Handling Different Output Formats	
Interview #2	You have to connect the tool to the build pipeline, new versions of the tool and they might return a slightly different output format.

Priority	
Interview #2	In general, security is important for companies. However, this does not mean that a commensurate amount of time, attention, and money is put in it. Also, there is always a battle between security and deadlines.
Interview #3	Normally, developers have limited time to deliver a project. Then you have the challenges of getting all the functions working and delivering those on time.
Interview #4	However, functionality is usually more important than security. Security is not explicitly asked for.
Interview #5	We do think security is important, but we have only been attacked twice in the last six years. Those attacks were not successful, so we have never been hacked. Therefore, we do not give it a lot of attention. Currently, we do not test for security at all. Our customers are not wealthy, and creating and running security tests takes time and therefore cost money. We do not test this due to a lack of knowledge on how to test it, and it costs a lot of extra time and money. Also, customers do not ask for these tests.

Security knowledge

Interview #1	A lack of knowledge of security vulnerabilities could be a cause, but I do not think this has to be an issue. Basic stuff like SQL Injection is easy to explain and everyone will encounter these at least once in his or her career. Generally, companies do not use vulnerability scanners. This is due to nescience. Companies do not make a conscious choice for only using penetration testers to perform security tests. This is the current concept of security. Nescience is not the enemy here, since people know security is important. However, the realization that this has to be controlled during the development process is a next step.
Interview #2	Developers quite often know little to nothing about security.
Interview #5	We do not test this due to a lack of knowledge on how to test it

Effectiveness

Interview #1	The downside of this approach is that this does not test security for all input fields. Therefore, you do not have 100% test coverage. We do not even know how much coverage we actually have. What would be better is a tool that automatically tests every input field. I found that SQLMap is very difficult to use and only tests a small part. It is nice for testing SQL Injection, but that is not all you want to test for. However, SQLMap finds vulnerabilities and that makes it useful.
Interview #2	On the other hand, issues that these tools do not find are usually more interesting. When doing manual tests, you usually come across problems that are far worse than the issues these tools find.
Interview #3	Good tools only find about 20% of the issues. Those 20% are useful of course.
Interview #4	My experiences with those are that scans can take a long time to run, and only easy mistakes are found.

Test Speed

Interview #1	Furthermore, a possible downside of integrating these tools into the build pipeline could be that, if running the tests takes a lot of time, it increases the building process. This should be fast.
Interview #3	Also, when scanning large websites, scanning takes a long time.
Interview #4	My experiences with those are that scans can take a long time to run.
Interview #5	It does not really matter if a scan takes a long time. You are not performing the scan but a machine is. Even if the scans take hours, as long as you do not have to do it yourself, it does not matter.

False Positives

Interview #1	False positives could also be a problem, but there is no way to prevent this. It is better than nothing.
Interview #2	All these tools have a high percentage of false positives, only 10 to 20% is correct.
Interview #3	Those 20% are useful of course, but there are also a lot of false positives. Scanning large websites will therefore produce many false positives. This takes quite a lot of time to process.
Interview #5	Finally, something should be done with false positives. You don't want the same false positive popping up every time you run the tests. Something should be found for that.

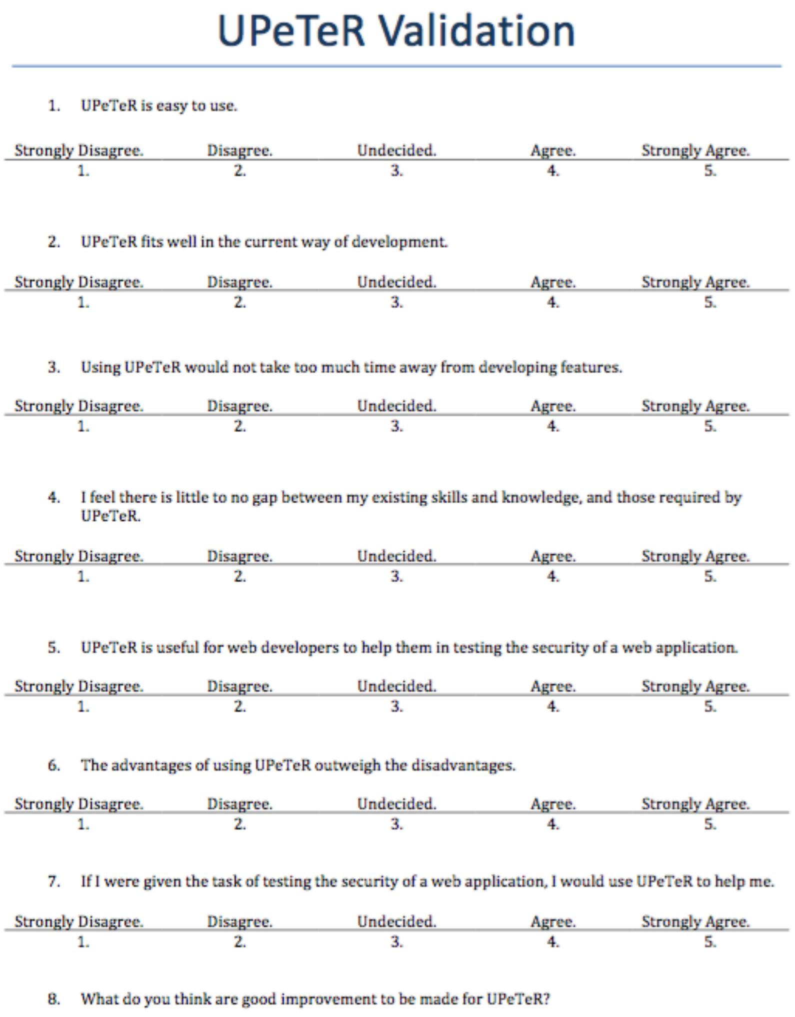
Data Contamination

Interview #4	A downside of using these scanners is that it can contaminate the application with useless data. Therefore, companies would need to set up a separate testing environment.

Understanding	**Tool Functionalities**
Interview #4	Also, I would like to see what payloads are used by tools to test for certain vulnerabilities. It is not always clear what the tools do exactly.

Appendix C

Acceptance Questionnaire



UPeTeR Validation

1. UPeTeR is easy to use.

Strongly Disagree.	Disagree.	Undecided.	Agree.	Strongly Agree.
1.	2.	3.	4.	5.

2. UPeTeR fits well in the current way of development.

Strongly Disagree.	Disagree.	Undecided.	Agree.	Strongly Agree.
1.	2.	3.	4.	5.

3. Using UPeTeR would not take too much time away from developing features.

Strongly Disagree.	Disagree.	Undecided.	Agree.	Strongly Agree.
1.	2.	3.	4.	5.

4. I feel there is little to no gap between my existing skills and knowledge, and those required by UPeTeR.

Strongly Disagree.	Disagree.	Undecided.	Agree.	Strongly Agree.
1.	2.	3.	4.	5.

5. UPeTeR is useful for web developers to help them in testing the security of a web application.

Strongly Disagree.	Disagree.	Undecided.	Agree.	Strongly Agree.
1.	2.	3.	4.	5.

6. The advantages of using UPeTeR outweigh the disadvantages.

Strongly Disagree.	Disagree.	Undecided.	Agree.	Strongly Agree.
1.	2.	3.	4.	5.

7. If I were given the task of testing the security of a web application, I would use UPeTeR to help me.

Strongly Disagree.	Disagree.	Undecided.	Agree.	Strongly Agree.
1.	2.	3.	4.	5.

8. What do you think are good improvement to be made for UPeTeR?