CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Modular Blackbox SQL Injection Vulnerability Web Scanning

Master's thesis in Computer science and engineering

MIRIAM DEGERMAN, DENNIS DUBREFJORD

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Modular Blackbox SQL Injection Vulnerability Web Scanning

MIRIAM DEGERMAN
DENNIS DUBREFJORD

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Modular Blackbox SQL Injection Vulnerability Web Scanning
MIRIAM DEGERMAN, DENNIS DUBREFJORD

Supervisor: Benjamin Eriksson, Dept. Computer Science and Engineering
Advisor: Martin Forssén, Recorded Future
Examiner: Andrei Sabelfeld, Dept. Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Modular Blackbox SQL Injection Vulnerability Web Scanning
MIRIAM DEGERMAN, DENNIS DUBREFJORD
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The use of web applications has increased heavily the last couple of decades. In line with this, an increasing amount of sensitive data is stored on web servers. Furthermore, SQL injections are one of the most common web application security risks. It can have devastating consequences, as it can cause confidential data to be read, modified and deleted. It could even allow an attacker to gain administrative privileges on the server database and compromise individual machines or entire networks.

A popular approach to finding web vulnerabilities is using autonomous web vulnerability scanners. In order for a scanner to be successful, it needs to be good at both crawling the web and detecting vulnerabilities when presented with possible attack vectors. For the most part, these two components are integrated to some degree. Our hypothesis is that web vulnerability scanners would benefit from using a modular approach instead. By allowing for easy exchange of crawler and detection module used in a scanner, the scanner could be optimised for specific tasks, whether that be finding SQL injections or other vulnerabilities. It could also be adapted to various types of web applications as different crawlers specialize on different areas.

To test the hypothesis, we have developed a modular design that can be used to combine crawlers and detection modules. We have also implemented a scanner using the modular design as a proof of concept. The results show that the modular approach benefits from the advantages of both crawler and detection module used and it outperforms state-of-the-art web vulnerability scanners in both code coverage and vulnerabilities found. Moreover, the modular scanner was the only scanner that was able to find three previously unknown vulnerabilities in the web application WSPortal.

Keywords: Computer science, engineering, master thesis, SQL injection, web scanning, web vulnerabilities, modular, modularity.

Acknowledgements

Both of us would like to express our gratitude to our supervisors Benjamin Eriksson at Chalmers University of Technology and Martin Forssén at Recorded Future, whose assistance and guidance has helped us stay on track and ensure a high quality throughout the project.

Miriam Degerman & Dennis Dubrefjord, Gothenburg, June 2021

Contents

List of Figures

List of Tables

Glossary

AJAX AJAX, short for *Asynchronous* JavaScript and XML, is a set of web development techniques using many web technologies on the client-side to create asynchronous web applications.

Blackbox A web vulnerability scanner is called blackbox if it tests the web application from the point of view of a potential attacker, meaning it has no access to server-side code.

Crawler A crawler is a program that traverses the web and downloads web documents in a methodical, automated manner.

DOM The Document Object Model (DOM) is a cross-platform and language-independent interface that treats an XML or HTML document as a tree structure wherein each node is an object representing a part of the document.

Fuzzer A fuzzer is a tool that generates invalid, unexpected or random input and feeds it to a system. During the process it monitors the response of the system and behavior to find which, if any, inputs cause unexpected behaviour.

Scanner According to the definition used in this paper, a web vulnerability scanner consists of two modules: a web crawler and a detection module (or somewhat simplified - a *fuzzer*).

UNIX UNIX is a family of general purpose operating systems that has significantly influenced most operating systems of today.

UNIX pipe In Unix-like computer operating systems, a pipe is a mechanism for inter-process communication using message passing.

1

Introduction

As we rely more and more on web applications for our digital lives, security becomes increasingly important. According to the leading organization for web security *Open Web Application Security Project* (OWASP), the most common web application security risk is code injections [16], such as cross site scripting (XSS) and Structured Query Language (SQL) injection. Due to the complexity of modern web applications it can be hard to locate such vulnerabilities. It is of interest for both the research community and data security industry to develop and investigate methods to find vulnerabilities in web applications, considering that they immediately affect our daily lives. Web applications are for instance often used for banking, online shopping and social media.

A web vulnerability scanner is a computer program designed for finding and assessing weaknesses in web applications. A fuzz tester (or fuzzer) is an automated software tool that iteratively generates invalid, unexpected and/or random inputs with which it tests a specified computer program. The technique is surprisingly effective despite its seemingly naive approach [11]. Web vulnerability scanners usually consist of two components: a crawler that traverses the web application and a detection module that injects payloads with a fuzzer, analyses the result and determines whether the are any vulnerabilities. However, the crawler and detection module tend to be intertwined to some degree, making it hard to extract or replace the crawling module or fuzzing module from a scanner. As both web crawling and vulnerability detection are complicated tasks, focus must be put on building not only sophisticated detection modules, but also advanced crawlers.

SQL injection is a code injection technique that can read, delete and modify content in databases. A SQL injection is done by placing malicious code in SQL statements via web page input. This can result in unauthorized access to sensitive data such as credit card details, passwords or personal user information, thus affecting data confidentiality. SQL injections can also affect integrity by modifying or deleting database content. The attack can also disrupt operations by sending a large amount of computationally heavy queries, thus affecting availability as well. In other words the attack has potential to affect all three parts of the CIA model (confidentiality, integrity and availability) and can cause severe damage to organizations and individuals. Considering this, and the fact that it is one of the most common web vulnerabilities, makes it a compelling matter to address.

This thesis will explore how scanning of SQL injection (SQLi) vulnerabilities can be improved by allowing for separate web crawlers and detection modules to be combined as desired. By doing so, crawlers and detection modules can be chosen for particular purposes and the joined forces optimized for a specific task, such as finding certain SQLi vulnerabilities.

1.1 Aim of this thesis

The aim of this project is two-fold. Firstly, the aim is to investigate methods for improved web vulnerability scanning by developing a modular approach to web vulnerability scanning. By separating the crawling and detecting pieces, detection modules with different qualities and capabilities can be combined with state-of-the-art crawlers in an optimized way for a specific goal. This will allow users to easily combine the detection module of a scanner they prefer with a crawling component from another scanner.

The second aim is to implement a novel scanner that uses the modular approach. The scanner is evaluated by comparison to state-of-the-art vulnerability scanners on a specific set of web applications.

1.2 Scope of this thesis

This project will focus solely on SQL injections - no other web vulnerabilities will be analysed. However, as an extension it could serve as an interface for other web vulnerabilities as well. The idea is that it should be possible to combine modules (crawlers and detection modules) freely, including detection modules dealing with other web vulnerabilities.

Furthermore, the project will not cover a specific type of SQL injection vulnerability, called second order SQL injections. These are notoriously difficult to find using blackbox web vulnerability scanning. More on this in Section 2.1.8.

In order to make a deep analysis possible, both the web applications and the vulnerability scanners used in the evaluation need to be open source. This allows us to measure code coverage, i.e. how much of the server-side code that was executed in each scan. The scanners are advantageously open source as well, thus giving us the possibility to examine how they handle specific cases and why they do or do not find certain vulnerabilities.

1.3 Contribution

This thesis contributes to existing research in the field of web application security by investigating methods for improved web vulnerability scanning. We present a newly developed design and interface for modular web vulnerability scanning. We also present a novel blackbox SQLi vulnerability scanner, named Blackmap, that implements the design and interface. Furthermore, we release this implementation as open source [1]. Modularity is a beneficial feature since it makes scanners extendable and replacement of crawlers and detection modules seamless. An advantage is that outdated crawlers and detection modules can quickly be replaced with newer ones as they are introduced.

Finally, an in-depth comparison of different SQL injection vulnerability scanners, including our modular scanner Blackmap, is presented. In addition to this, we present key attributing features of the different scanners and how they relate to the results to help further understand blackbox scanning.

1.4 Ethical considerations

In computer science, and perhaps particularly in security, ethical considerations must be taken into account when a project is conducted. In our project, we propose an effective design for web vulnerability scanning. Moreover we implement this design in a proof-of-concept scanner. Web vulnerability scanning can be used to locate vulnerabilities in a system, both by security researchers and practitioners trying to find vulnerabilities to remove them, but also by malicious actors searching for vulnerabilities to exploit for personal gain. By furthering the development of web vulnerability scanners, we help both the defenders and the attackers of web applications.

However, the only way to make the web more secure is to locate and remove vulnerabilities, in doing so making web applications harder to breach. Our scanner facilitates this process and thus contributes to securing the web. We believe that the benefits of empowering security researchers and practitioners and gradually making the web more secure outweighs the risk of our tool being used by malicious actors.

We will report any vulnerability we find in actively used software to the vendor in accordance with the Google Project Zero policy. This is the standard policy used in industry today. According to the policy, any vulnerability found should be disclosed to the public 90 days after it is reported to the vendor, regardless of whether the vulnerability is patched or not. By doing so, the vendor has a limited time frame to patch the vulnerability, which has proven to be important for industry improvement. Whilst putting some pressure on the vendor, the time frame also gives the vendor enough time to patch the vulnerability thoroughly [24].

[1] The source code can be accessed at https://github.com/Dubrefjord/ModuleMatcher, https://github.com/Dubrefjord/attack_module_sql and https://github.com/mirdeger/bw

2

Background

In this chapter, we will thoroughly explain what a SQL injection is and show examples of different kinds of SQL injections. Furthermore, we will briefly discuss database management systems, the systems handling the databases that can be compromised by SQL injections, and PHP, a scripting language commonly used on web servers to operate web applications. Thereafter we mention related work in the field of web crawling and dive somewhat deeper into the specific vulnerability scanners that are used in this project.

2.1 SQL injection attacks

SQL injection is the placement of malicious code in SQL statements. The code injection is done via webpage input, e.g. a login form or search bar. SQL, Structured Query Language, is used to communicate with databases using queries. Queries follow SQL syntax and consist of statements and names of databases, tables and columns. Some examples of common statements are `SELECT`, `WHERE` and `ORDER BY`.

By sending queries, a user can retrieve data from a database, insert, update and delete records in a database. It can also create new databases and tables, as well as create stored procedures in a database. A database usually consists of a number of tables which in turn consists of a number of rows and columns.

One of the things that makes web applications useful is interaction and possibility to retrieve, send and update information. This can be done via some form of user input. To enable interaction with a database in a web application, the scripting language PHP is commonly used on web servers in combination with SQL.

To demonstrate how a SQL query can be run, consider an example where a user enters a username and a password in a login form connected to a SQL database on a web application written in PHP. The following piece of PHP code shows a query being stored in the variable `query`, with user input highlighted in red:

```
$query = "SELECT * FROM users WHERE
            username = '" . $_GET['username'] . "' AND
            password = '" . $_GET['password'] . "'";
```

The variable `query` is a string containing SQL statements that together form a query which selects all entries in the table `users` where the username and password corresponds to the input given in a form. If a user submits the username `joe` and password `monkey123`, the query sent to the SQL database would be:

```
SELECT * FROM users WHERE username = 'joe' AND password = 'monkey123'
```

This piece of code is vulnerable to SQL injection attacks, as will be shown in Section 2.1.1.

There are various kinds of SQL injection attacks and different authors and organizations categorize them differently. In this section, we will define them according to Doupé et al. [10], with a complement of out of band attack according to Clarke [3].

In addition to the different kinds of SQL injections, there are innumerable variations on the attacks. They are often used together or sequentially in order to achieve specific goals. In the following subsections, the attacks are explained with examples where user input is highlighted in red.

2.1.1 Tautologies

The general purpose of a tautology-based attack is to inject code in conditional clauses so that they always evaluate to true. This attack can be used to bypass authentication and extract data.

Example:

```
SELECT * FROM users WHERE username = 'admin' AND password = '' OR
↪  1=1; --'
```

The example shows a SQL query used to authenticate users who try to log in to a web application. The attacker inputs a single quote to escape the password string in the SQL query, and writes `OR 1=1`, which will make the SQL statement always evaluate to *true*. The result of this query is that the password check is replaced with a tautology. Thus the attacker is able to bypass the password check and still authenticate herself as the user *admin*.

2.1.2 Logically Incorrect Queries

This attack can be used to gather information about the type and structure of the backend database by injecting code that makes the SQL syntax invalid. This will make the applications return default error pages, which are often overly descriptive. This attack is often used as a stepping stone for other attacks as it can identify injectable parameters and reveal names of tables and columns. The attacker injects

statements in attempt to cause a syntax error, type conversion error or logical error. Syntax errors can identify injectable parameters. Type errors can be used to extract data and conclude what the data types of certain columns are. Logical errors often reveal names of tables and columns.

Example:

```
SELECT * FROM accounts WHERE username = 'abc'' AND password = 'aaa'
```

This will cause the web application to show an error message like *Incorrect syntax near 'abc'. Unclosed quotation mark after the character string " AND password='aaa' ".* This reveals that there is a column called "password".

2.1.3 Union Queries

When performing union query attacks, the attacker injects code that causes the application to append data to the result of the query, thus extracting more data from the database than intended. This could for instance be data from other tables. The attack is done by injecting a `UNION SELECT` statement into the vulnerable parameter.

Example:

```
SELECT title FROM news WHERE id=1 UNION SELECT * FROM accounts
```

This malicious query would not only return the title with id 1 from the table `news`, but also all entries in the table `accounts`.

2.1.4 Piggy-backed Queries

In this attack, an attacker appends one or more additional queries to the original query. If successful, this attack can cause execution of virtually any type of SQL command. As a result, piggy-backed queries can be used to extract, add or modify data, perform denial of service and execute remote commands. Vulnerability to this type of attack often depends on the database configuration allowing multiple statements in a single string.

Example:

```
SELECT accounts FROM users WHERE login=''; DROP TABLE users -- '
AND password=''
```

The semicolon indicates that the query ends there. The `DROP TABLE` statement deletes the specified table completely, thus destroys valuable information and likely

makes the web application malfunction. As mentioned in Section 2.1.1, the two dashes are the comment operator in SQL. They are put there by the adversary to make the program ignore the closing quotation mark, which otherwise would lead to a syntax error.

2.1.5 Stored Procedure Attacks

A stored procedure is a prepared piece of SQL code that can be saved so that the code piece can be reused multiple times. They are widely used due to the benefits of encapsulation of business logic in a single entity, strong validation, faster execution and exception handling [18]. The syntax for creating a procedure can look like this:

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

The stored procedure is called using the name of it and possibly a parameter list. The SQL queries in a stored procedure can be vulnerable similar to SQL queries outside the stored procedure, in case user input without proper validation is used within it.

2.1.6 Inference

In this type of attack, the query is modified such that an action is executed depending on the result of a true/false condition about values in the database. The attack can be useful on websites that give no usable error messages. Instead, the attacker observes the response of the website and notes in what cases the website behaves differently.

Example:

```
SELECT accounts FROM users WHERE login='joe' AND 1=0 -- ' AND
↪  password=''
SELECT accounts FROM users WHERE login='joe' OR 1=1 -- ' AND
↪  password=''
```

A web application vulnerable to this attack will respond differently to these two queries. As an example, the webpage could return a login error message in response to the first query (as `AND 1=0` will make the statement evaluate to false). At this point, the attacker does not know whether the error message was shown because the attack was blocked or the attack itself caused an error. If the attacker then submits the second query (that always will evaluate to true) and there is no login

error message, the attacker knows that the login parameter is vulnerable to code injection.

Another interesting kind of inference attack are timing attacks. By observing the time delays in the response of the exposed database, the attacker can gain information from it. In order to do this, the attacker structures the injected query as an if/then statement, whose branch predicate depends on something unknown about the database content. In one of the branches, a SQL construct that takes a known amount of time to execute is used, e.g. the `WAITFOR` keyword, which causes a specified delay. By measuring time delays, the attacker can gather small pieces of information about the database step by step.

2.1.7 Alternate Encoding

Many applications have common defense mechanisms such as scanning for known unsafe characters e.g. single quotes or comment operators. Alternate coding attacks are used to bypass these defense mechanisms and are often used in conjuction with other types of attacks. It works by using encoding methods such as hexadecimal, ASCII or Unicode character encoding to send the payload to the database.

Example:

```
www.examplesite.com/index.php?id=1%20%55%4E%49%4F%4E%20%53%45%4C%45%
↪ 43%54%20%32%2C%34
```

The encoding `%20%55%4E%49%4F%4E%20%53%45%4C%45%43%54%20%32%2C%34` is URL encoding for `UNION SELECT 2,4`. The result of typing this into an address bar, provided that the vulnerable site `examplesite` exists, is that the web server of the site decodes the URL and builds the SQL query using it. The result is the following query.

```
SELECT text FROM blogposts WHERE id=1 UNION SELECT 2,4
```

The query is a standard union query, which has been covered in Section 2.1.3. Thus, the attacker was able to bypass the defense mechanisms of the web server and send an unsafe query to the SQL server.

2.1.8 Second Order Injections

Second order injection occurs when submitted values contain commands that are stored instead of executed immediately. In a second order injection attack, an attacker injects a query fragment into a query that is not necessarily vulnerable to injection. Later on that injected SQL command is executed in a second query that is vulnerable to SQL injection.

For instance, imagine a web application where it is possible to register users and then, when logged in, change the password of the user by entering the current and new passwords. Even if the registration form is protected against SQL injections, we can still input a payload as username and hope that the username is used by the server in another query. For example, we can input as username `joe'; DROP TABLE users --`. When updating the password, the following query is constructed.

```
UPDATE users
SET password='abc'
WHERE '%$username%' and password='aaa'
```

Note that the username is not a user input in this query. Instead it is taken directly from the stored value at the web server. The query that is sent to the database is thus:

```
UPDATE users
SET password='abc'
WHERE 'joe'; DROP TABLE users --' and password='aaa'
```

Second order injections are notoriously hard to detect using blackbox scanning. This is mostly because the scanners payloads are constructed with other SQL injection vulnerabilities in mind [19]. Also, a unique difficulty when scanning to find second order SQL injections is that the stored payload can be used in a SQL query at the server at any point in time. Even if the scanner is able to detect that a SQL injection has occurred, it can be hard to link this back to the saving of the payload, especially if many such payloads have been submitted.

2.1.9 Out of Band Attack

So far, all attacks described use the same HTTP(S) channel to both send a request and receive a response. However, in an out of band attack, the attacker uses different channels to launch the attack and gather the results. Out of band SQL injection techniques rely on the database server's ability to make DNS or HTTP requests to deliver data to an attacker.

The features of modern database servers can do more than simply return data in response to a query. They can for instance send automatic e-mails, interact with the file system on the server and open connections to other databases [3]. The functionality is not only helpful for server administrators but also attackers.

One way an attacker could exploit a helpful feature is to inject a payload triggering the server to send e-mails containing sensitive information. However, this can only be done if the feature in enabled in the database management system, which seldom is the case.

Another possible channel for an attacker to use is DNS. For instance, Microsoft SQL Server's command `xp_dirtree` can be used to make DNS requests to a server an attacker controls. The attacker can use this to retrieve data from the database by including it in the request sent to the attacker's server [1]. This example is shown in code below. Consider the following URL, crafted by an attacker:

```
https://example.com/products.aspx?id=1;EXEC%20master..xp_dirtree
%20'%5c%5ctest.attacker.com%5c'+--+
```

When decoded, the URL will look like this:

```
https://example.com/products.aspx?id=1;EXEC master..xp_dirtree
'\\test.attacker.com\' --
```

This will produce the following SQL query:

```
SELECT * FROM products WHERE id=1;EXEC master..xp_dirtree
'\\test.attacker.com\' --
```

Notice the semicolon that symbolises the end of a query. There are now two separated queries that will be executed by the SQL server. The command `xp_dirtree`, which is a stored procedure, is used to display a list of every folder, sub-folder and file for the path given to it. In this case, the path given (`\\test.attacker.com\`) is the domain of the server that the belongs to the attacker. To list all folders, the address of the domain must first be resolved. A DNS query to the attacker's server is made. The attacker, who monitors DNS server logs, can see queries made to the domain. If a DNS query is made, the attacker knows that there is a SQL injection vulnerability exploitable via an out of band vector.

2.2 Crawling

A web crawler is a program that traverses the web and downloads web documents in a methodical, automated manner. The purpose of the crawler can be to collect data or to scan for vulnerabilities by looking for attack vectors. When doing the latter, a goal is to find every input parameter on every page. Crawlers are primarily evaluated based on their code coverage, meaning how much of the web applications functionality they manage to traverse.

In the past, websites were simply a collection of static HTML pages, connected by hyperlinks. The websites were also stateless. Crawling such a website entails following each hyperlink on each page, storing the URLs in the process, and estimating the web application state. A crawler that handles these tasks well is Doupé's Enemy of the State [6].

According to Doupé, modern web applications are more complex [7]. As an example, they usually keep state, allowing users to authenticate themselves to get access

to more features, such as logging into an e-mail account before being able to read the e-mails. Another example is shown in Figure 2.1, where an administrator has the privilege of adding new users, but only when authenticated and logged in.

Dynamic and event driven JavaScript features add more complexity to websites of today. For instance, access to a form could be given only by clicking a button which does not change the server-side state (URL), but only changes the client-side state. In order to traverse such features, the crawler must implement dynamic program analysis. See Figure 2.1 for an example, where a JavaScript event taking place in `admin.php`. The administrator has to click a button in order for a form to add a new user appears.

In addition to state and dynamic events, tracking inter-state dependencies proves to be a challenge for web crawlers. There is an inter-state dependency in a web application if user inputs are connected to the states of a web application. As an example, imagine a web application in which anyone can see the list of users and the administrator can add new users, as in Figure 2.1. After the administrator has added a new user to the database, it will appear when visiting the page that shows all users. The red line represents the inter-state dependency.

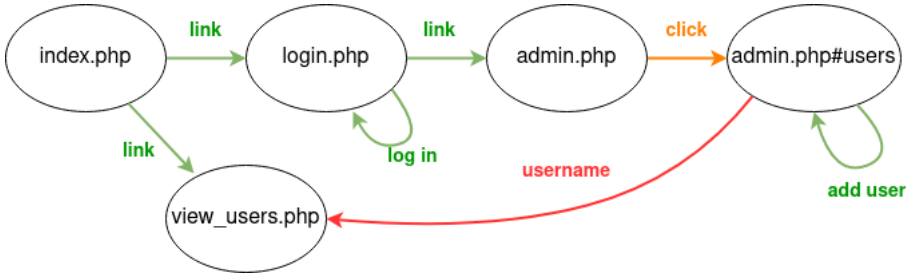

Figure 2.1: Example of a web application with some challenges for a web crawler. There is a JavaScript event is highlighted in orange. Green lines are HTML. The red line represents an inter-state dependency.

The crawler jÄk [20] presents a model for handling dynamic and event driven JavaScript features. It does not only save URLs, but also JavaScript events. While crawling, it also builds a navigation graph which can be used to perform further crawling. However, jÄk does not handle form submissions. Neither does it track inter-state dependencies.

A state-of-the-art crawler which handles client-side states, including form submissions, as well as inter-state dependencies is the one used in Black Widow [8]. Since the crawler of Black Widow is used in the modular scanner implemented in this project, a more comprehensive explanation of it can be found in Section 2.6.1.

2.3 Detection

It is clear that early detection of SQL injection vulnerabilities improves quality and safety of software. The two main approaches to test web applications for vulnera-

bilities are white box testing and blackbox testing. White box testing is based on analysis of the source code of the web application whilst blackbox testing consists of analysis of the application behaviour during execution. In the latter approach, the tester does not know the internals of the web application.

The testing can be done manually or by using automated tools. In the case of white box testing, code analysis tools may be used. In blackbox testing, automated scanning can be used or manual penetration testing can be done. The blackbox technique is widely adopted due to the ease of use, automation, and independence from the web application technology used.

Fuzz testing is one of the most widely adopted techniques to test software correctness and is commonly used in blackbox testing [12]. Two main challenges with this technique are generating fuzz input and identifying vulnerabilities.

There are various methods for generating fuzz input, everything from pre-generating test cases after analysis of the data structure to completely random generation. Usually, combinations of static fuzzing vectors (values that are known to be dangerous) are used [15].

In order to identify vulnerabilities, fuzzers must be able to detect when an execution violates the security policy. Testing for input validation vulnerabilities such as SQL injection vulnerabilities is challenging as it requires understanding of the behaviour of the web application. It is not possible to reliably detect SQL injections from the response of a web application [12]. For instance, in the case of a time based inference attack, the results may be incorrect due to network instability. Furthermore, when doing logical attacks it can be difficult to detect the difference between TRUE and FALSE in the response of a web application. Simply looking at the length of the response is not enough, as the rest of the page might change regardless of the SQL injection result.

2.4 Database Management Systems

There are several different database management systems (DBMS) some of which utilize SQL, Each of them have their own implementation of the SQL language. Some common DBMSs are Oracle, MySQL, Microsoft SQL Server, PostgreSQL, MariaDB and SAP HANA.

Although they all implement the SQL language, there are some differences in syntax between them. When trying to inject SQL commands into a query, it is crucial to know what syntax applies to the specific system that is being exploited, so that the intended behaviour can be prompted.

An example of the difference in syntax is case sensitivity. For instance, PostgreSQL is case sensitive, whilst MySQL is not [4]. Some of the versions also require single quotes around the strings in queries, while others also accept double quotes. The function for getting the current date differ between most of the versions. As do the

function for instructing the SQL-server to sleep, which is important when an time based inference injection is used. This injection is covered in Section 2.1.6.

While these syntax differences do exist between the versions, the principles of the SQL-injection attack is applicable to all of them.

2.5 PHP

Hypertext Preprocessor (originally Personal Home Page tools, PHP) is a popular scripting language used mainly on web servers to drive, operate and power internet sites with dynamic content. Dynamic content can be content generated from a database or the visitors' form data. According to W3Techs' data, PHP is used by 79.1% of all websites with a known server-side programming language [25].

Many web developers are unaware of how SQL queries can be tampered with and miss out on basic prevention techniques [21]. There are however multiple built in ways to prevent SQL injection in PHP, such as functions checking if the given input has the expected data type. PHP has a wide range of input validating functions such as `is_numeric()` and `ctype_digit()` [5].

Another prevention technique that PHP provides is prepared statements. Prepared statements can be used to effectively avoid SQL injections, since they ensure that client-supplied data is treated as content of a parameter and never as a part of a SQL statement. The root of the SQL injection problem lies in the mixing of code and client-supplied data. In successful SQL injection attacks, data is directly inserted into the program allowing the program to be altered. When using prepared statements on the other hand, a small program (the prepared statement) is sent to the server first, in which the data is substituted by some variable. Hence, a SQL injection cannot occur [5].

The execution of a prepared statement consists of two stages. In the first stage a statement template is sent to the database server, where a syntax check is performed and internal resources are initialized by the server. In the second stage, the statement is executed. Parameters values to be used will be sent from the client to the server during this stage. More importantly, they will be bound to the previously created internal resources. Below, a code example for executing a prepared statement in MySQL can be seen [9].

```
/* Prepared statement, stage 1: prepare */
$statement = $mysqli->
    prepare("INSERT INTO test(id, label) VALUES (?, ?)");

/* Prepared statement, stage 2: bind and execute */
$id = 1;
$label = 'PHP';
```

```
$statement->bind_param("is", $id, $label);
// "is" means that $id is bound as an integer and $label as a string

$statement->execute();
```

2.6 Related work

In this section we will discuss some research that has been done in the field of web vulnerability scanning and present the scanners that are used in this project: Black Widow, Sqlmap, Arachni and OWASP ZAP.

2.6.1 Black Widow

Black Widow [8] is a novel blackbox web application scanner which builds on three pillars: navigation modeling, traversing and tracking inter-state dependencies. Given a starting point in the form of a URL, it crawls the web application with a novel JavaScript dynamic crawler and creates a navigation model of the application along the way. The JavaScript crawler is able to explore both the static structure of webpages and dynamic events, which makes it useful in the increasingly dynamic web applications of today. The static structure of a webpage can e.g. be anchors, forms and frames whilst dynamic events can be mouse clicks, hovering the cursor over an element or resizing the browser window.

Besides creating a navigation model, Black Widow captures the sequence of steps required to reach a given page. This enables the scanner to retrace its steps. Furthermore, Black Widow uses a dynamic taint tracking technique allowing us to understand dependencies between different pages. The scanner identifies taint sources such as input fields, probes them with taint values in the form in unique strings and later looks for the taint values to resurface in the HTML document, i.e. it checks for so called sinks [8]. The dependencies between different pages are called *inter-state dependencies* and knowing about them can be useful for finding web vulnerabilities other than SQL injections, such as cross-site scripting vulnerabilities.

2.6.2 Sqlmap

Sqlmap is an open source penetration testing tool that can detect and exploit SQL injection vulnerabilities in an automatic manner, by utilizing fuzzing techniques. It has full support for six different SQL injection techniques: boolean-based blind and time-based blind which both are a kind of inference attack, logically incorrect queries, UNION query-based, piggy-backed queries and out-of-band.

Sqlmap offers an extensive variety of options. The target can be specified in a number of different ways, such as a URL, logfile or configuration file. There are

multiple options that can be used to specify how to connect to the target URL. One can for instance force usage of a given HTTP method (PUT, GET, POST etc.), specify a data string to be sent through POST or set a HTTP cookie header value.

Besides the target URL, Sqlmap requires different parameters depending on the situation. As an example, any parameter dynamically rendered by JavaScript will be missed unless explicitly specified for Sqlmap to examine.

Sqlmap constructs queries based on what attack it is performing. It replaces or appends a SQL statement string to the parameter being tested. The string can contain e.g. a `SELECT` sub-statement, a `UNION` sub-statement, a logically incorrect statement triggering an error message or a statement causing the DBMS to wait a certain number of seconds before responding. Sqlmap then analyses and compares the HTTP responses.

2.6.3 Arachni

Arachni is a modern web vulnerability scanner written in Ruby. It supports all major operating systems (Windows, Mac OS X and Linux) and covers a lot of different use cases with its versatility. There is a simple command line scanner utility, a Ruby library allowing for scripted audits and a multi-user multi-scan web collaboration platform. Arachni has an integrated browser environment and can support highly complicated web applications using technologies such as JavaScript, HTML5, DOM manipulation and AJAX.

Beside SQL injection vulnerabilities, Arachni has functionality to detect reflected XSS, local and remote file inclusion, invalidated redirects and backup files. The tool can learn from the behavior of the web application and adapt to each web application resource [2].

2.6.4 OWASP ZAP

OWASP ZAP (short for Zed Attack Proxy) is web application vulnerability scanner intended for professional penetration testers as well as those new to web application security [17]. It is one of the most active OWASP projects and widely used within web application security research. It has won a number of awards for its proficiency [23, 14, 13]. Some of its features are AJAX crawling, WebSocket support and fuzzing. It has a plugin-based architecture that allows new or updated features to be added via an online marketplace.

3

Methods

In order to assess the benefits of a modular approach to web vulnerability scanning, we introduce a design that defines an interface between scanners and crawlers. Furthermore, we implement a proof-of-concept vulnerability scanner that uses this design. Our modular scanner is built on the crawler part of Black Widow and uses Sqlmap as detection module. We evaluate it by comparison to other successful scanners.

In the following sections, the design of the system is explained as well as the implementation of our proof-of-concept vulnerability scanner.

3.1 Design Overview

The idea with modular web vulnerability scanning is that an arbitrary crawler should be possible to combine with an arbitrary detection module. To make this possible, we have created a wrapper that encapsulates the crawler and detection module and specifies an interface between them.

As can be seen in Figure 3.1, the wrapper handles all communication between the crawler and detection module. The scanner as a whole unit is started by running the wrapper and giving a crawler, detection module and URL as input arguments.

In order for a crawler to be compliant with the wrapper, it needs to implement some functions in accordance with a given specification. For a detection module to be compatible with the wrapper, it needs to be runnable according to our criteria, either by itself of through a layer in between the wrapper and the detection module. The specification and criteria are covered in Section 3.1.2 and Section 3.1.3.

3.1.1 Wrapper: *ModuleMatcher*

The purpose of the wrapper, which we have named *ModuleMatcher* is to define interfaces to the crawler and the detection module. On top of that, *ModuleMatcher* takes care of deduplication of nodes and threaded execution of the detection module with different nodes as input. Implementation of these interfaces allows for quick

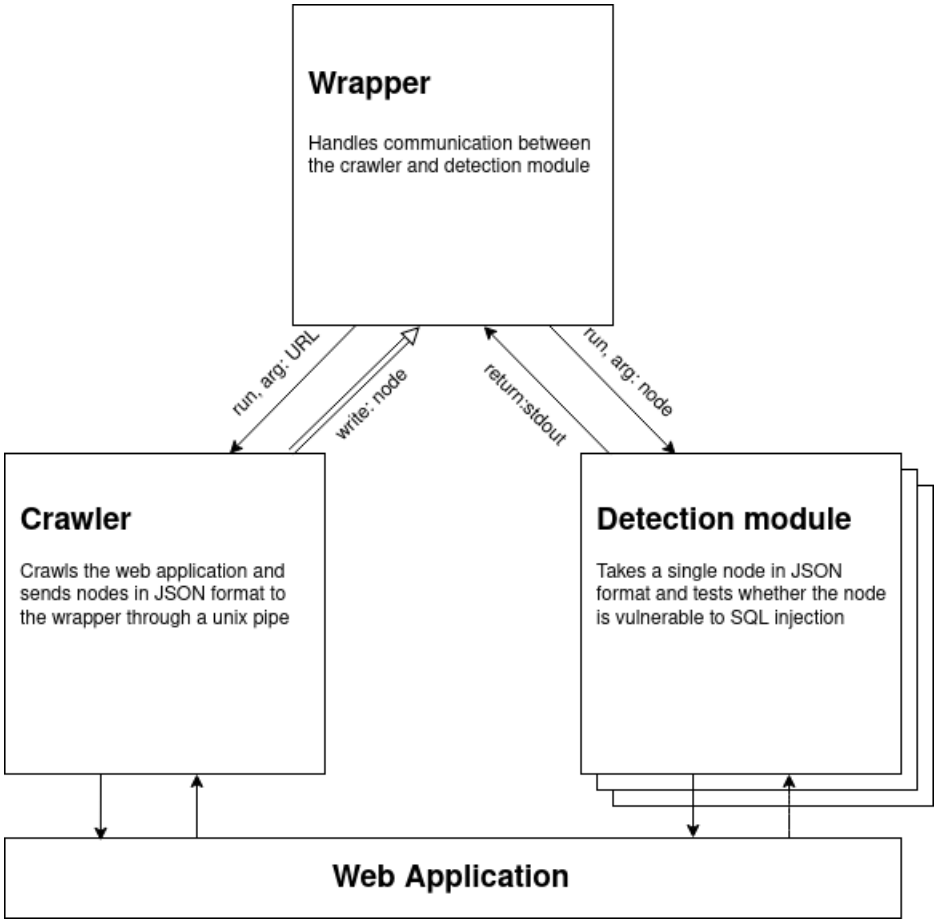

Figure 3.1: An overview of the modular architecture. The pipe-like arrow from the crawler to the wrapper is used to symbolize that nodes are being sent continuously during the crawler execution through a dedicated UNIX pipe, as opposed to everything being sent at the end of the execution. The multiple detection modules illustrate that many instances of the detection module are run in parallel, in order to analyse many nodes at the same time.

```
python3 matcher.py /path/to/crawler
    /path/to/detection_module http://examplesite.com
```

Listing 1: The command used to run ModuleMatcher with a compatible crawler and detection module.

and easy replacements of the modular parts.

To use the vulnerability scanner, *ModuleMatcher* is run with the relative paths to the attack module and the crawler module, as well as the URL as arguments. See Listing 1 for an example of how ModuleMatcher is run.

The wrapper runs the crawler and continuously retrieves so called *nodes* being sent from the crawler. The nodes adhere to a certain specification in the form of a JSON object. The specification of the node formatting can be seen in Figure 3.2. The JSON object contains of key/value pairs for a URL, a list of parameters, a list of data, a list of cookies and a HTTP-method. The field `parameters` is meant for parameters that can be found in the URL of the node, whilst the field `data` is meant parameters but with values assigned to them. This is useful for HTTP-POST data. An element in the field `cookies` contains of a name and a value of a session ID.

```
("url": URL,
 "parameters": [PARAMETER],
 "data": [PARAMETER=VALUE],
 "cookies": [NAME=VALUE],
 "method": HTTP-METHOD)
```

Figure 3.2: JSON-formatting of nodes. The square brackets indicate that there is a list of elements.

To further illustrate how the node specification is meant to be used, imagine a web page containing a login form. It is a HTML form that can be used to send a POST request and has a username field and a password field. An example of how a node constructed by a web crawler visiting this page could look is shown in Figure 3.3.

```
("url": "http://example.com/login.php",
 "parameters": ["username",password"],
 "data": "username=joe,password=123",
 "cookies": ["PHPSESSID=g7vflje6cdq523vl1lk2sgdrh1"],
 "method": "post")
```

Figure 3.3: Example of how a node can be constructed by a web crawler examining a login page.

The nodes are sent from the crawler to the wrapper through a dedicated UNIX pipe. A pipe is a construct which allows a process to write data that another process can read. The communication is one-way and uses a first-in-first-out strategy.

When the wrapper receives a node, it makes sure that the node has not previously been examined. If that is the case, the node is added to a log file and a new thread is created that runs the detection module with the node given as a command line argument. When the investigation of the node is finished, the detection module returns its standard output (`stdout`) to the wrapper, which then prints it to the user. The algorithm used in the wrapper can be found in Algorithm 1.

```
  // save command line arguments
1 path_to_crawler = args[1];
2 path_to_attacker = args[2];
3 url = args[3];
  // create UNIX pipe, with file descriptors for reading and writing
4 crawler_output, crawler_input = pipe();
5 run_in_thread(path_to_crawler, url, "–matcher", crawler_input);
6 attack_threads = [];
7 attacked_nodes = [];
8 for node in crawler_output do
9 |   if node in attacked_nodes then
10|   |   continue;
11|   end
12|   attacked_nodes.append(node);
13|   thread = run_in_thread(path_to_attacker, node);
14|   attack_threads.append(thread);
15 end
16 for thread in threads do
17 |   print(thread.output);
18 end
```

Algorithm 1: Wrapper algorithm

3.1.2 Interface to crawler

In order for a crawler to interface with ModuleMatcher it needs to do two things. Firstly, it needs to save the pipe it receives from ModuleMatcher. Secondly, it needs to send the nodes it finds in JSON format back to ModuleMatcher via the pipe.

More specifically, when the crawler is run by ModuleMatcher, it will receive the command line argument *–matcher*. Upon doing so, the crawler must save the file descriptors from the environment that was passed along from the wrapper. When the crawler finds nodes in the web application it writes them as JSON objects to the file descriptor that was saved from the environment, which represents the UNIX pipe to the wrapper. A newline character is appended to the JSON object, for the purpose of separating the JSON objects. The newline character is used because it is escaped by default within JSON objects, meaning that it can not occur naturally in a data stream of only JSON objects. Thus, whenever it occurs it must be between the objects, making it fit to serve as delimiter.

Thus, to make an existing crawler compatible with ModuleMatcher, the necessary additions are:

- The crawler must accept an argument `--matcher`
- The crawler must be runnable using the following command:
  `python3 path/to/crawler --url "www.example.com" --matcher`
- Upon receiving the argument `--matcher`, the crawler must save a file descriptor from the environmental variables *crawler_write_fd*.
- A crawler started with the argument `--matcher` must save the URL, parameters, POST data (if any), cookies and HTTP-methods when a node is found. These are saved as a JSON object which is written to the file descriptor it previously saved. After a node has been written to the pipe, it must also write a newline character to the pipe.

3.1.3 Interface to Detection Module

The detection module has to do even less than the crawler to be compatible with ModuleMatcher. The only requirement is that it must be runnable using the following command: `python3 path/to/detection_module json_node`. Alternatively, a layer in between ModuleMatcher and the detection module must be runnable with that command, as long as it starts the detection module with the information given in the JSON node.

3.2 Implementation: *Blackmap*

In the prototype built as a part of this project, the structure of which can be seen in Figure 3.4, the crawler used is based on Black Widow [8]. The crawler from Black Widow was extracted and made compatible with ModuleMatcher.

Black Widow was chosen due to its shown proficiency in web crawling. The initial research of the project indicated that it is one of the best open source crawlers. Contrary to the other scanners used in this project, Black Widow has no knowledge related to SQL, making it an excellent candidate to show the power of the modular approach to scanning.

As detection module, we decided to use Sqlmap since it is one of the most widely used, user friendly SQL injection fuzzers. A python script was written to wrap Sqlmap and act as an interface between it and ModuleMatcher. We call the finished prototype *Blackmap* and more detailed description of the implementation of Blackmap is presented in the following sections.

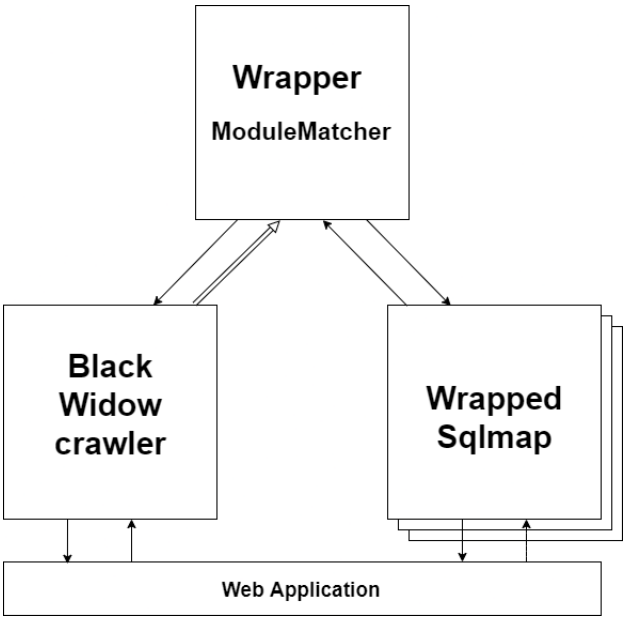

Figure 3.4: The structure of Blackmap. The crawler from Black Widow is used as the crawling component and a wrapped Sqlmap is used as a detection module.

3.2.1 Modification of Black Widow

Since Black Widow originally is a XSS vulnerability scanner, the modification of it began with removal of all functionality regarding XSS. Unlike other crawlers such as Arachni and OWASP ZAP, Black Widow has no options to turn off the XSS functionality. Thus, we removed it manually from the code. After the removal, we had a pure web crawler with no fuzzing technology included.

The crawler was then made compatible with ModuleMatcher according to the specifications in Section 3.1.2. While traversing a web application, Black Widow constructs a navigation graph containing nodes and edges. Each edge contains information about two nodes and for every new edge found, the function `send_node_data()` is called. In that function, information is extracted from both nodes and their information is sent to the wrapper through a pipe. If the edge contains a form, the node responsible for sending the form is extracted. On top of the parameters and cookies, the HTTP method of the form (POST/GET) is included, and if the form is a POST form, the POST form data is included as well. The `send_node_data()` function is presented in Listing 2.

3.2.2 Using Sqlmap as Detection Module

As previously mentioned, Sqlmap is used as detection module for the prototype scanner. To use Sqlmap as detection module, a wrapper between ModuleMatcher and Sqlmap was created.

The attack-script in the prototype uses the fields of the given node to construct a

```
1  def send_node_data(edge, self):
2      cookies = extract_cookies_from_edge(edge)
3      method = edge.value.method
4      data = []
5      parameters = []
6      json_list = []
7
8      for node in [edge.n1, edge.n2]: #extract param from the two
           nodes in the edge.
9        parameters = extract_parameters(node)
10       json_node_data = {"url": node.value.url,
11                         "parameters": ",".join(parameters),
12                         "data": "",
13                         "cookies": ",".join(cookies),
14                         "method": ""}
15       json_list.append(json_node_data)
16
17     if method == "form":
18         data, parameters = extract_data_from_forms_in_edge(edge,
               self)
19         json_node_data = {"url": edge.value.method_data.action,
20                           "parameters": ",".join(parameters),
21                           "data": ",".join(data),
22                           "cookies": ",".join(cookies),
23                           "method": edge.value.method_data.method}
24         json_list.append(json_node_data)
25
26     if self.matcher:
27         for json_node_data in json_list:
28           json_node_data = json.dumps(json_node_data)
29           self.crawler_pipe_output.write(str(json_node_data))
30           self.crawler_pipe_output.write('\n')
31           self.crawler_pipe_output.flush()
```

Listing 2: The send_node_data function that is responsible for creating and sending nodes over the pipe from Black Widow to the wrapper module.

Sqlmap command and run it. The data returned from Sqlmap is then filtered and finally returned to the wrapper. Since Sqlmap places three dashes --- before and after the results, the filter function simply extracts the data encapsulated by that pattern. An overview of the algorithm can be seen in Listing 3.

```
1
2  node_data = json.loads(json_data)
3
4  sqlmap_command = ["sqlmap"]
5  sqlmap_command.append('—answers=keep testing=Y')
6  sqlmap_command.append("—batch")
7  sqlmap_command.append("-v")
8  sqlmap_command.append("0")
9  sqlmap_command.append("—flush-session")
10 if "url" in node_data:
11   sqlmap_command.append("-u")
12   sqlmap_command.append(node_data["url"])
13 if "cookies" in node_data:
14   sqlmap_command.append("—cookie")
15   sqlmap_command.append(node_data["cookies"])
16 if "data" in node_data and node_data["method"] == "post":
17   sqlmap_command.append("—data")
18   sqlmap_command.append(node_data["data"])
19 if "data" in node_data and node_data["method"] == "get":
20   sqlmap_command.append("—forms")
21 if "parameters" in node_data and node_data["parameters"] != "":
22   sqlmap_command.append("-p")
23   sqlmap_command.append(node_data["parameters"])
24
25 attack = subprocess.run(sqlmap_command, capture_output=True, text=
       True)
26
27 print(filter_sqlmap_output(attack.stdout))
```

Listing 3: The algorithm used in the layer between ModuleMatcher and Sqlmap. It constructs a command that runs Sqlmap based on the information given in the JSON node. It then runs the command, filters the output of the execution and prints it.

The attack-script takes a node in JSON format as input. It then extracts the different fields and tries to fuzz them in order to find SQL injections vulnerabilities. The results of the fuzzing is printed to stdout, which is returned to ModuleMatcher.

3.3 Evaluation

To evaluate our approach we compare it with existing state-of-the-art scanners. The evaluation compares two metrics: the number of found vulnerabilities and server-side code coverage. The code coverage is interesting to investigate because it could be the case that the detection module run in parallel interferes with the performance

of the crawler, since they both interact with the same web application.

The other scanners evaluated are Arachni, OWASP ZAP, Sqlmap. These were chosen because they are frequently evaluated in academic papers on the topic, signaling that they are reasonably good scanners. They are also all open source.

Detection Module	Version
Blackmap	1.0
Arachni	1.5.1
OWASP ZAP	2.10.0
Sqlmap	1.5.4.8

Table 3.1: Table of the scanners being evaluated.

To evaluate and compare the scanners, they were all run on a common set of web applications, shown in table Table 3.2. The scans were allowed to run for a maximum of eight hours.

One of the web applications in the set is our own implementation, produced to test some particular cases. Another web application in the set, WackoPicko [7], is specifically developed for web vulnerability scanning assessment. The remaining three applications are ordinary web applications used for blogging and shopping. Two of them, WordPress and osCommerce, are of recent version that are currently being used in production. WSPortal on the other hand is of an older version not in use anymore and contains one known SQLi vulnerability.

Web application	Version
SimpleWebApp	N/A
WackoPicko	(2018)
WordPress	5.7
osCommerce	2.3.4.1
WSportal	1.0

Table 3.2: Table of the web applications used in our evaluation.

To measure server-side code coverage, the PHP module Xdebug [22] was used. Xdebug is a debugging tool for PHP that can return detailed data about what lines of code have been executed in the application.

3.3.1 Our own test application *SimpleWebApp*

In order to test some likely scenarios of a web application, we created a simple web application *SimpleWebApp* implementing those cases. We release this application open-source for other scanners to practice and improve with. The cases contained in the web application are the following:

- When form contains multiple vulnerable parameters.

- When some dynamic client-side content needs to be generated in order to find a vulnerability.
- When a user has to be registered and authenticated in order to explore all webpages of a web application.

An overview of the web application can be seen in figure Figure 3.5. The figure also shows which steps are needed to be taken in order to reach the different webpages.

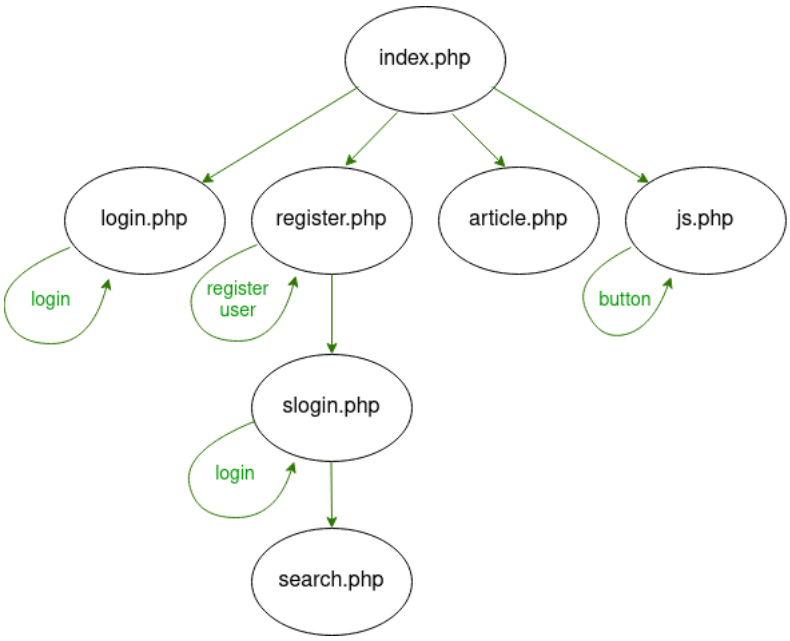

Figure 3.5: Overview of SimpleWebApp. In total, the application contains five SQL injection vulnerabilities. Two vulnerabilities are found in login.php, one in search.php, one in article.php and one in js.php

In the first case, an input form contains multiple vulnerable parameters. The point of this test is to find out whether the scanner being tested reports only one vulnerable parameter or continues exploring parameters within the same form after finding the first vulnerability. If one parameter is vulnerable, other parameters can possibly be vulnerable as well. This case is tested in `login.php`.

In the second case, when some dynamic content needs to be generated in order to find a vulnerability, the point is to test the ability of a scanner to handle dynamic content and events. In modern web applications, it is common for client-side code to require chaining, such as hovering a menu to see all links. This case is tested in `js.php`, where a button needs to be clicked in order for a vulnerable input field to be rendered.

In the third case, a single parameter in the URL is injectable. This is a very simple injection that every non-faulty vulnerability scanner should be able to find. This case is tested in `article.php`, where the vulnerable parameter `id` is accessible in the URL.

In the fourth case, when a user has to be registered in order to explore all webpages of a web application, the point is simply to test the ability of the scanner to register a user. Many web applications require a login to explore all webpages, hence finding all potential vulnerabilities. This case is tested in `register.php`, where a user can be registered. This user can then be used to log in at `slogin.php` and finally `search.php` can be accessed, which contains a vulnerable search bar.

4

Results

In this chapter, we present the results of running the chosen scanners against the specified set of web applications. In Section 4.1 the server-side code coverage provoked by running the different vulnerability scanners is presented. Afterwards, the found vulnerabilities are presented in Section 4.2. Finally, the results are analysed in Section 4.3.

4.1 Code coverage

While running the scans, the server-side code coverage was tracked. The code coverage of Sqlmap, ZAP and Arachni compared to the code coverage of Blackmap on each web application is presented in Figure 4.1. Note that Blackmap outperforms every single scanner on every single web application. When comparing our findings with the numbers presented in the Black Widow paper [8], they look very similar. Thus, the results indicate that the detection module does not notably affect the crawler's ability to crawl the web application.

Also, Table 4.1 is presented, showing how many lines of code were covered in each scan. On average Blackmap covered 279% more lines of code than the other scanners.

Table 4.1: Total lines of code covered in each scan.

Scanner	WSPortal	osCommerce	SimpleWebApp	WackoPicko	WordPress
Sqlmap	53	1876	69	446	54973
Arachni	53	6060	74	513	25947
ZAP	99	5144	136	152	44783
Blackmap	659	10879	142	766	55196

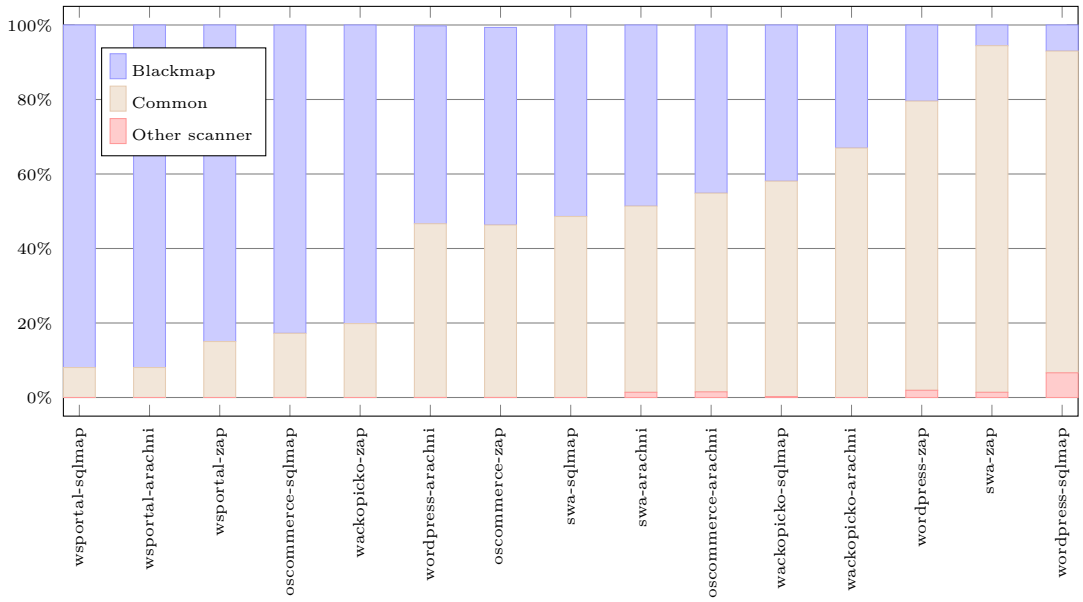

Figure 4.1: Each bar compares Blackmap to one other scanner on a web application. The bars show three fractions: unique lines found by Blackmap (blue), lines found by both scanners (yellow) and lines uniquely found by the other scanner (red).

4.2 Found vulnerabilities

The web vulnerability scanners Arachni, ZAP, Sqlmap and Blackmap were run against the web applications SimpleWebApp, WackoPicko, WordPress, osCommerce and WSPortal. In some of the web applications, the scanners were able to find vulnerabilities. These were tested manually to make sure they were actual vulnerabilities and not false positives. The vulnerabilities found by each scanner on SimpleWebApp, WackoPicko, and WSPortal are presented in Table 4.2, Table 4.3 and Table 4.4 respectively. Note that no tables are presented for osCommerce or WordPress. The reason is that no scanner was able to find a SQL injection vulnerability in those applications.

Table 4.2: Vulnerabilities found by the scanners when run on SimpleWebApp. Note that only Blackmap is able to find all of the present vulnerabilities.

SimpleWebApp Parameter	/login.php username	/login.php password	/article.php id	/js.php js	/search.php username
Sqlmap	✓	-	✓	-	-
Arachni	-	✓	✓	-	-
ZAP	✓	✓	✓	-	✓
Blackmap	✓	✓	✓	✓	✓

Table 4.3: Vulnerabilities found by the scanners when run on WackoPicko. As can be seen, all of the scanners find the vulnerability in login.php.

Wackopicko Parameter	/users/login.php username
Sqlmap	✓
Arachni	✓
ZAP	✓
Blackmap	✓

Table 4.4: Vulnerabilities found by the scanners when run on WSPortal. On this web application, Blackmap is the only scanner that finds any vulnerabilities. Moreover, the vulnerability found in editadmin.php, alertip_post.php and deleteadmin.php are previously unknown.

WSPortal Parameter	content.php page	editadmin.php username	alertip_post.php ip	deleteadmin.php username
Sqlmap	-	-	-	-
Arachni	-	-	-	-
ZAP	-	-	-	-
Blackmap	✓	✓	✓	✓

4.3 Analysis

In this section, the results will be analyzed and interpreted. The primary tool of the analysis is the in-depth code coverage data of the scanners, showing which lines they have provoked the web server to run. The goal is to interpret why a particular scanner fails to detect a vulnerability that another scanner manages to detect.

4.3.1 SimpleWebApp

When running the scanners on SimpleWebApp, the findings vary greatly. In this section, the reasons behind the discrepancies will be covered.

4.3.1.1 login.php

When scanning `login.php`, all of the scanners found at least one of the two vulnerable parameters username and password. Coverage-wise, all of the scanners provoke the same code to be run at the server-side. However, Blackmap and ZAP find both the username and the password vulnerabilities, whereas Arachni and Sqlmap only find one each. This implies a fuzzing deficiency in Arachni and Sqlmap. Perhaps they are configured to stop testing a form when they have found one vulnerable parameter, making them miss potential other vulnerable parameters.

It is noteworthy that Blackmap uses Sqlmap as its fuzzing component and still finds both vulnerabilities. This is the case because Blackmap manually overrides Sqlmaps default behaviour of terminating when it has found a vulnerability by adding *"–answers=Keep testing=Y"* to the command used to initiate the Sqlmap fuzzing.

4.3.1.2 js.php

When scanning `js.php`, only Blackmap succeeds in finding the vulnerable parameter `js`. Analyzing the executed lines of code of the web server when being scanned by Arachni and Sqlmap reveals that they fail in the same fashion. They both miss out on sending the necessary parameter name `js` in their POST request.

The analysis reveals that line 8 in listing 4 is run when the web app is scanned by both scanners individually, whereas the lines below it which contact the data base management system is not run by either. This implies that the condition in the if-statement evaluated to false, meaning that the `js` parameter was not included in the post request. The `js`-parameter is available to the user on the web app, but the user must issue a JavaScript event, a button press, to generate the form which reveals the name of the parameter. Thus, the fact that Arachni and Sqlmap does not find this parameter is a failure of their crawling components.

```
8       if(isset( $_POST['js'])){
9       $link = mysqli_connect("localhost", "root", "", "sqli");
10
11
12      $query =
13      "SELECT * FROM users WHERE
14      username = '" . $_POST['js'] . "'";
```

Listing 4: A code snippet from the js.php-file being run on the web server.

```
11  if(isset($_POST['username']) && !empty($_POST['username']) && isset(
        $_POST['password']) && !empty($_POST['password']) ) {
12
13    $link = mysqli_connect("localhost", "root", "", "sqli");
14
15    // Not super secure but probably enough here.
16    $u = addslashes($_POST['username']);
17    if( !filter_var($u, FILTER_VALIDATE_EMAIL) ) {
18      echo "Bad email!";
19      die();
20    }
21    $p = addslashes($_POST['password']);
22
23    $query =
24      "INSERT INTO users (username, password)
25      VALUES ('" . $u . "', '" . $p . "')";
```

Listing 5: A code snippet from the register.php-file being run on the web server.

Blackmap and ZAP both find the `js` parameter and manages to provoke the web server to send queries to the database management system. Yet, ZAP does not find the vulnerability. Since the crawling component finds the input form, we draw the conclusion that the reason ZAP does not find the vulnerable parameter is due to a poor fuzzing component.

4.3.1.3 search.php

To find the page `search.php`, the crawler has to register a user through `register.php` and log in as the same user at `slogin.php`. At the `slogin.php` page after the crawler is logged in, there is a link to `search.php`. The analysis of the scanners shows that Sqlmap and Arachni both fail to register a user.

Sqlmap provokes the web server to run line 11 in listing 5, but never gets past it. Since Sqlmap fails to register a user, it cannot find `search.php`. Arachni gets past the if-clause on line 11, but does not manage to enter a valid email in any POST request. This is evident since lines 17, 18 and 19 are run, but never any lines below them. Thus, both Sqlmap and Arachni fail to find the vulnerability due to insufficient crawling components.

```
1 <?php
2 # This creates a session / cookie
3 session_start();
4 if(isset($_SESSION['auth'])) {
5   echo "You are logged in!<br>";
6 } else {
7   echo "You are <b>NOT</b> logged in!<br>";
8 }
```

Listing 6: A code snippet from the beginning of the register.php-file being run on the web server

An interesting finding in the analysis of the scanner ZAP was that it provokes the web server to run a line in `register.php` that Blackmap does not.

In listing 6, the web server runs line 5 only when being scanned by ZAP. This shows that the ZAP scanner loops back to the `register.php` page after it has logged in with the registered user, which is something Blackmap does not do. Imagine that a form is made available to logged in users if they visit the `register.php`-page. In such a case, ZAP would find the form, but Blackmap would not. Thus, in that specific instance, the crawling component of ZAP would outperform the crawling component of Blackmap.

4.3.2 WackoPicko

Even though all scanners find the vulnerability in `login.php`, it is interesting to analyse the code coverage regardless. For instance, it is interesting to note that Blackmap is able to provoke the execution of every single line of code that is covered by the other scanners, while also finding more lines than each of them. In that sense, Blackmap fully outperforms the other scanners when it comes to code coverage.

Another interesting thing to note is that Arachni seems to have gotten stuck in an infinite loop in the `calendar.php` page, as over 90% of the requests to the web server has been made from this page. If the scanner were to not get stuck, perhaps it would have been able to cover more code which could lead to more vulnerabilities found. Similarly, if it would have got stuck in the `calendar.php`-section before exploring the login-page, the vulnerability in `login.php` would never have been found.

Besides the vulnerability in `login.php`, there is a second order SQL injection vulnerability that no scanner was able to find. The vulnerability can be found manually by registering a user at `register.php`, placing a payload in the `name` field. Afterwards, one must navigate to the page `similar.php` where the payload is executed in the SQL server.

4.3.3 WSPortal

Once again we can see that Blackmap completely fully outperforms the coverage of the other scanners. This leads to it discovering four SQL injection vulnerabilities that none of the other scanners can find. Moreover, three of the found vulnerabilities are previously unknown vulnerabilities.

Sqlmap, Arachni and ZAP have difficulties crawling this web application since it often embeds pages in other pages using HTTP iFrames. Sqlmap, Arachni and ZAP don't handle HTTP iFrames and thus miss a lot of content.

4.3.3.1 content.php

This is the vulnerability that was known prior to this project. To find the vulnerable page variable in `content.php`, the crawler must first navigate to `addpage.php` and add a page. Doing so adds a link to the newly added page, which resides at `content.php?page=1`. Moreover, both `addpage.php` and `content.php` are embedded within HTML iFrames. The crawling compontents of Sqlmap, Arachni and ZAP does not seem to be able to handle content embedded within HTML iFrames, since they do not find the `addpage.php` page. Since they can not add a page through `addpage.php`, there is no way for them to discover the `content.php` page or the associated and vulnerable parameter `page`.

4.3.3.2 editadmin.php

In many ways, this vulnerability is similar to the vulnerability in `content.php`. The crawler must visit `edadmin.php` where it can find a link to `editadmin.php`. At `editadmin.php`, the crawler can send a POST form to edit the admin data. The vulnerable parameter `username` is not shown as a form entry field on the client side, but is included in the POST request. Both `edadmin.php` and `editadmin.php` are embedded within HTML iFrames. Thus, Sqlmap, Arachni and ZAP fail to find the vulnerability because they do not handle content embedded in HTML iFrames.

4.3.3.3 alertip_post.php

To find this vulnerability, the crawler must simply go to `alertip_post.php` and submit a POST form containing the parameters `ip` and `message`, where the `ip` parameter is vulnerable. The tricky part in discovering this vulnerability is that upon clicking the link to `alertip_post.php`, the file content is embedded in an HTTP iFrame. Once again Sqlmap, Arachni and ZAP fail to find the vulnerability because they do not handle content embedded in HTML iFrames.

4.3.3.4 deleteadmin.php

Much like in editadmin.php, the crawler must navigate to `edadmin.php`. From here, there is a link to `deleteadmin.php` where the vulnerable GET parameter `username` is appended. Again, both `edadmin.php` and `deleteadmin.php` are embedded in HTTP iFrames. The scanners with crawling components unable to handle HTTP iFrames will not find this vulnerability.

4.3.4 WordPress

Since no vulnerability has been found by any of the scanners, an in-depth comparison is hard. Blackmap does not fully outperform the other scanners in code coverage when scanning WordPress, but note that this may be due by the fact that the scans were terminated after 8 hours. If Blackmap - and the other scanners for that matter - would have had time to finish scanning, perhaps the result would have been different.

A line of code run by only one scanner does not mean that the other could not find the right command to trigger an execution of the line. It might simply mean that the scanner traverses the web application in a different order and hence does not reach the place where the code is run within the allotted 8 hours. However, it is interesting to note that in the 8 hours the scanners were allowed to run, Blackmap did beat the other scanners in coverage.

4.3.5 osCommerce

The code coverage analysis shows that Blackmap vastly outperforms Sqlmap, Arachni and OWASP ZAP in code coverage. However, In similarity with WordPress, no vulnerabilities have been found in osCommerce.

An analysis of the osCommerce source code reveals that it utilizes two defensive mechanisms to defend itself from SQL injection vulnerabilities, both of which are essentially strategies to sanitize user inputs. The first sanitizes the input of the user, when the input is supposed to be an integer. This is done by simply typecasting the input to an integer. Thus, if the input contains anything other than an integer, the typecast will result in an error and the query will not be sent to the database. Note the blue type cast by `customer_id` below.

```
tep_db_query("update " . TABLE_CUSTOMERS . " set customers_password
↪  = '" . tep_encrypt_password($password_new) . "' where
↪  customers_id = '" . (int)$customer_id . "'");
```

The other defensive mechanism used in osCommerce is used to sanitize input that is supposed to be a string. This is done by sending the input to the PHP function

`mysql_real_escape_string`, the purpose of which is to escape any character that could alter the interpretation of a SQL query by placing a backslash (\) in front of them. The output of the function is then placed in the query.

5

Conclusion

In this chapter, we will discuss the project from a holistic view, as well as summarize the project with some concluding remarks.

5.1 Discussion and future work

Even though the results clearly indicate that Blackmap outperforms other SQL injection vulnerability scanners, they could be more precise and fair with further testing. For this project we have only used open source scanners and web applications, which substantially limits the set of testable software. By also including proprietary software, thus adding more scanners and more target web applications, the tests could be made more conclusive.

In the following sections, we will discuss the project more deeply. We will start with a discussion of the methods, then implementation followed by results and lastly some general limitations and opportunities for future work.

5.1.1 Methods discussion

In its current form, ModuleMatcher only handles a few specific pieces of information which can be sent from the crawler to the detection module. These are, per node; a URL, parameters, data (values to the parameters), cookies and HTTP(S)-method (e.g. a GET/POST/FORM). There are however more pieces of information that could come in handy, especially if this project would be extended to include other web vulnerabilities than SQL injections. For instance, when scanning to find stored XSS vulnerabilities, the crawling component must identify inter-state dependencies between different pages in the web application. This information must be sent to the detection module so that it can make its inputs on one page, while checking the behaviour of another page. As presented in Section 2.6.1, Black Widow has the capabilities to find these dependencies, but currently ModuleMatcher does not have support for handling such information.

We used Xdebug to track the server-side code coverage when running the scanners on the web applications. The method worked well, but we noticed that it some-

times generated segmentation faults. We were able to deduce that the faults were generated only when specific pages were put under heavy load from the scanners. Our method to work around the problem was to limit the sending rate of the scanners, effectively reducing the load on the pages. Additionally, we checked the log files after the rate limited scans to confirm that no segmentation faults had been generated. This workaround was enough for our needs, but ideally the issue should be investigate in detail to solve the underlying problem.

Currently, the wrapper module does not handle scheduling of the execution of the crawler and the instances of the detection module. As long as the whole web application is scanned, that is irrelevant. On bigger web applications however, scanning all of it is not always feasible. A great addition to the wrapper would be to allow for a command line argument to be given, stating for how long the scan should run. With this information, the wrapper could make intelligent scheduling decisions to make sure as many nodes as possible are examined within the given time frame. For instance it might be wise to halt the execution of the crawler towards the end of the scan in order to allow the detection modules to examine the nodes that have already been found.

Finally, the wrapper only supports one crawler and one detection module. An interesting option would be to allow the user to input a list of crawlers and a list of detection modules that could be run in parallel. Having several crawlers crawling the web application, each sending nodes, could mean a better code coverage in total since the crawlers make up for each others weaknesses. However, this outcome is not guaranteed. When all of the crawlers investigate the web application at the same time, it might be the case that they interfere with each other and thus get worse code coverage. The same could happen on the detection module side. If two detection modules, possibly looking for different vulnerabilities, interact with the same node on the web application it is possible that they would interfere with each other, making them both perform worse. Thus, sending a list of crawlers or detection modules places a lot of responsibility on the user to avoid this negative interference. An interesting opportunity for future research would be to investigate under which circumstances crawlers and detection modules can be combined with positive outcomes.

5.1.2 Implementation discussion

In this project, we have taken some steps to make Blackmap more efficient. However, we have not measured what timing delays are introduced by using the ModuleMatcher structure instead of an integrated design. We suspect, however, that the main contributions of time spent are made by the crawler and detection module. Those components are present in an integrated tool as well, thus the time difference should not be substantial. Measuring the timing overhead introduced by ModuleMatcher and trying to make it more efficient is a great opportunity for future work.

A possible area of improvement is how the modular scanner is run. At the moment, the wrapper module assumes that the crawler and detection module are runnable using Python. It was done that way since all files handled in this project were written in Python. Any crawler or detection module not written in Python would only need a minimal python wrapper around it to interface with the wrapper module. An improvement would be to allow the user to input the command to execute the crawler and detection module instead of giving their paths (e.g. the user could type `python3 crawler/blackwidow.py` instead of only `crawler/blackwidow.py`). In the wrapper, when executing the modules, the command provided by the user could be used instead of requiring that the path must be prefaced with `python3`.

The wrapper uses the option `pass_fds` to send the file descriptors of the pipe to the crawler. However, this option does not work on Windows systems. An improvement could be to remove this option and instead send the file descriptors on the command line.

5.1.3 Results discussion

From our analysis it is clear that existing web vulnerability scanners have weaknesses in both crawling components and detection modules. Because of their integrated designs, replacing the weak components with stronger ones can be a challenge. Thus, outdated components may not be updated and this could be a main reason as to why the prominent scanners of today fall short when compared to Blackmap.

While Blackmap also has areas of improvement, one crucial strength is the modular design that makes exchanging the components of a scanner seamless. For instance, when an updated version of Black Widow is released, the process of updating the version that is compatible with ModuleMatcher is almost automatic. Thus, the easy process of making a specific crawler compatible with ModuleMatcher only needs to be conducted once.

5.1.4 Other limitations and future work

An opportunity for future research is to extend the modular web vulnerability scanner design to other common web vulnerabilities as well. If a renown fuzzer for the specific vulnerability exists, it is possible to easily combine it with a powerful crawler using the ModuleMatcher interface, the same way we combined the crawler from Black Widow and the fuzzer from Sqlmap to create Blackmap.

Another opportunity for future research is to investigate modular blackbox web scanning to find second order SQL injection vulnerabilities. As was noted in Section 4.3.2, none of the tested scanners, including Blackmap, is able to find the second order SQL injection present in the application. The difficulty of finding such vulnerabilities using a blackbox method is covered briefly in Section 2.1.8.

5.2 Conclusion

A novel, modular approach to web vulnerability scanning, ModuleMatcher, has been developed. Using it, we have developed the SQL injection web vulnerability scanner Blackmap. Blackmap vastly outperforms competing tools both when it comes to code coverage and found vulnerabilities. For example, among the tested scanners, Blackmap was the only one to find three previously undiscovered vulnerabilities in WSPortal. Further, the analysis of why existing scanners fail to detect the vulnerabilities found by Blackmap shows that the tools have both weak crawling components and fuzzing components. This further cements the need of a modular design of web vulnerability scanners, which would make it easy to replace the weak component when a stronger one is invented.

Bibliography

[1] Acunetix. Sqli part 6: Out-of-band sqli. https://www.arachni-scanner.com/features/framework/crawl-coverage-vulnerability-detection/. Accessed: 03.06.2021.

[2] Arachni. Crawl coverage and vulnerability detection. https://www.acunetix.com/blog/articles/sqli-part-6-out-of-band-sqli/. Accessed: 03.06.2021.

[3] Justin Clarke. *SQL Injection Attacks and Defense, Second Edition*. Elsevier, 225 Wyman Street, Waltham, MA 02451, USA, 2012.

[4] Datacamp. Sql server, postgresql, mysql... what's the difference? where do i start? https://www.datacamp.com/community/blog/sql-differences. Accessed: 2021-04-01.

[5] PHP Documentation. Sql injection. https://www.php.net/manual/en/security.database.sql-injection.php. Accessed: 2021-03-16.

[6] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 523–538, Bellevue, WA, August 2012. USENIX Association.

[7] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In Christian Kreibich and Marko Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *Proceeding of the 42th IEEE Symposium on Security & Privacy*, IEEE SP 2021. IEEE, 2021.

[9] The PHP Group. Prepared statements. https://www.php.net/manual/en/mysqli.quickstart.prepared-statements.php. Accessed: 03.06.2021.

[10] Rasoul Jahanshahi, Adam Doupé, and Manuel Egele. You shall not pass: Mitigating sql injection attacks on legacy web applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 445–457, New York, NY, USA, 2020. Association for Computing Machinery.

[11] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[13] MaxiSoler. 2013 top security tools as voted by toolswatch.org readers. `http://www.toolswatch.org/2013/12/2013-top-security-tools-as-voted-by-toolswatch-org-readers/`. Accessed: 27.04.2021.

[14] MaxiSoler. 2014 top security tools as voted by toolswatch.org readers. `http://www.toolswatch.org/2015/01/2014-top-security-tools-as-voted-by-toolswatch-org-readers/`. Accessed: 27.04.2021.

[15] OWASP. Fuzzing. `https://owasp.org/www-community/Fuzzing`. Accessed: 23.05.2021.

[16] OWASP. Owasp top 10. `https://owasp.org/www-project-top-ten/`. Accessed: 12.12.2020.

[17] OWASP. Owasp zap. `https://owasp.org/www-project-zap/`. Accessed: 27.04.2021.

[18] Paladion. Are stored procedures safe against sql injection? `https://www.paladion.net/blogs/are-stored-procedures-safe-against-sql-injection`. Accessed: 28.04.2021.

[19] Muhammad Parvez, Pavol Zavarsky, and Nidal Khoury. Analysis of effectiveness of black-box web application scanners in detection of stored sql injection and stored xss vulnerabilities. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 186–191, 2015.

[20] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. JÄk: Using dynamic analysis to crawl and test modern web applications. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, page 295–316, Berlin, Heidelberg, 2015. Springer-Verlag.

[21] Chen Ping, Wang Jinshuang, Pan Lin, and Yu Han. Research and implementation of sql injection prevention method based on isr. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pages 1153–1156, 2016.

[22] Derick Rethans. Xdebug - debugger ad profiler tool for php, 2019.

[23] Matt Sarrel. Bossie awards 2015: The best open source networking and security software. `https://www.infoworld.com/article/2982962/bossie-awards-2015-the-best-open-source-networking-and-security-software.html#slide8`. Accessed: 27.04.2021.

[24] Project Zero Tim Willis. Policy and disclosure: 2020 edition. `https://googleprojectzero.blogspot.com/2020/01/policy-and-disclosure-2020-edition.html`. Accessed: 21.06.2021.

[25] W3Techs. Usage statistics of php for websites. `https://w3techs.com/technologies/details/pl-php`. Accessed: 2021-03-16.

A

Appendix 1

Scanner configuration

1) *Sqlmap*: The following command was used to run Sqlmap.

```
sqlmap.py -u [url] --crawl=400 --batch
↪  --answers=skip=N,continue=Y,proceed=X --flush-session
↪  --forms -v 1 --cookie [cookie]
```

2) *Arachni*: The following command was used to run Arachni.

```
time arachni [url] --checks=sql* --browser-cluster-pool-size=1
    --plugin=autologin:url=[loginUrl],parameters=
    "[userField]=[username]&[passField]=[password]",
    check="[logout string]"
```

3) *ZAP*: To scan a web application with ZAP, open the ZAP GUI. Within it, open the embedded web browser and navigate to the web applications login page and log in. From here, run an Active Scan against the target.

4) *Blackmap*: The following command was used to run Blackmap.

```
time python3 path/to/ModuleMatcher/match.py path/to/Blackmap/crawl.py
    path/to/attack_module_sql/sql_attack.py [url]
```

B

Appendix 2

SimpleWebApp code

index.php

```
1 <a href="login.php">login</a><br>
2
3 <a href="article.php?id=1">Article 1</a><br>
4 <a href="article.php?id=2">Article 2</a><br>
5 <a href="js.php">js</a><br>
6 <a href="register.php">register user</a><br>
```

login.php

```
1 <body>
2 <?php
3 if(isset( $_GET['username'])){
4   $link = mysqli_connect("localhost", "root", "", "sqli");
5
6
7   $query =
8     "SELECT * FROM users WHERE
9     username = '" . $_GET['username'] . "' AND password = '" . $_GET
          ['password'] . "'";
10
11   echo "<br><pre><code>";
12   echo $query;
13   echo "</code></pre>";
14   $result = mysqli_query($link, $query);
15   if (!$result) {
16     echo "<br>";
17     echo "<br>";
18     echo "<br>";
19     echo mysqli_error($link);
20   }
21
22   if($result) {
23     $found = 0;
24     if ($row = $result->fetch_row()) {
25       $found = 1;
26       echo "<h1>Success! Welcome: ".$row[1]."</h1>";
27     }
28     if(!$found) {
```

```
29          echo "<h1>Fail!</h1>";
30        }
31      }
32  }
33  ?>
34
35  <br>
36  <hr>
37  <br>
38  <form action="login.php">
39  <input type="text" name="username" value="<?= @$_GET['username'] ?>"
        ><br>
40  <input type="text" name="password" value="<?= @$_GET['password'] ?>"
        ><br>
41  <input type="submit">
42  </form>
43  </body>
```

article.php

js.php

```
1  <body>
2  <?php
3  //
4  //
5  // This example should be solveable by Black Widow but not only
       SqlMap
6  //
7  //
8  if(isset( $_POST['js'])){
9    $link = mysqli_connect("localhost", "root", "", "sqli");
10
11
12    $query =
13      "SELECT * FROM users WHERE
14      username = '" . $_POST['js'] . "'";
15
16    echo "<br><pre><code>";
17    echo $query;
18    echo "</code></pre>";
19    $result = mysqli_query($link, $query);
20    if (!$result) {
21      echo "<br>";
22      echo "<br>";
23      echo "<br>";
24      echo mysqli_error($link);
25    }
26
27    if($result) {
28      $found = 0;
29      if ($row = $result->fetch_row()) {
30        $found = 1;
31        echo "<h1>Success! Welcome: ".$row[1]."</h1>";
32      }
```

```
33      if(!$found) {
34        echo "<h1>Fail!</h1>";
35      }
36    }
37  }
38  ?>
39
40  <br>
41  <hr>
42  <br>
43
44  <div id="form">
45  </div>
46
47  <script>
48  function render() {
49    // Base64 encoded HTML form
50    document.getElementById("form").innerHTML = atob('
          PGZvcm0gYWN0aW9uPSJqcy5waHAiIG1ldGhvZD0iUE9TVCI+
          PGlucHV0IHR5cGU9InRleHQiIG5hbWU9ImpzIiB2YWx1ZT0iIj48YnI+
          PGlucHV0IHR5cGU9InN1Ym1pdCI+PC9mb3JtPg==');
51  }
52  </script>
53
54  <button onclick="render()">Render!</button>
55
56  </body>
```

register.php

```
1  <?php
2  # This creates a session / cookie
3  session_start();
4  if(isset($_SESSION['auth'])) {
5    echo "You are logged in!<br>";
6  } else {
7    echo "You are <b>NOT</b> logged in!<br>";
8  }
9
10
11 if(isset($_POST['username']) && !empty($_POST['username']) && isset(
       $_POST['password']) && !empty($_POST['password']) ) {
12
13   $link = mysqli_connect("localhost", "root", "", "sqli");
14
15   // Not super secure but probably enough here.
16   $u = addslashes($_POST['username']);
17   if( !filter_var($u, FILTER_VALIDATE_EMAIL) ) {
18     echo "Bad email!";
19     die();
20   }
21   $p = addslashes($_POST['password']);
22
23   $query =
24     "INSERT INTO users (username, password)
25     VALUES ('" . $u . "', '" . $p . "')";
```

```
26    $result = mysqli_query($link, $query);
27
28    $loginstr = 'Welcome! Login: <a href="slogin.php">Login</a>';
29
30  }
31
32  ?>
33
34  <html>
35  <body>
36
37  <?php echo @$loginstr; ?>
38
39  Register!<br>
40  <form action="register.php" method="POST">
41  <input type="email" name="username">
42  <input type="password" name="password">
43  <input type="submit">
44  </form>
45
46  </body>
47  </html>
```

slogin.php

```
1  <?php
2  # This creates a session / cookie
3  session_start();
4  if(isset($_SESSION['auth'])) {
5    echo "You are logged in!<br>";
6  } else {
7    echo "You are <b>NOT</b> logged in!<br>";
8  }
9
10
11
12 if(isset($_POST['username']) && !empty($_POST['username']) && isset(
       $_POST['password']) && !empty($_POST['password']) ) {
13
14    $link = mysqli_connect("localhost", "root", "", "sqli");
15
16    // Not super secure but probably enough here.
17    $u = addslashes($_POST['username']);
18    $p = addslashes($_POST['password']);
19
20    $query =
21      "SELECT * FROM users WHERE username = '".$u. "' AND password = '
           ".$p."'";
22    $result = mysqli_query($link, $query);
23    if ($result) {
24      if ($row = $result->fetch_row()) {
25
26        // file_put_contents("query.log", "Login as " . $u . ":" . $p
              . "\n", FILE_APPEND);
27
28        $loginstr = '<h1>Login success! Search: <a href="search.php">
```

```
            Search</a>';
29        $_SESSION['auth'] = 1;
30      }
31    }
32
33  }
34
35  ?>
36
37  <html>
38  <body>
39
40  <?php echo @$loginstr; ?>
41
42  Login!<br>
43  <form action="slogin.php" method="POST">
44  <input type="email" name="username">
45  <input type="password" name="password">
46  <input type="submit">
47  </form>
48
49  </body>
50  </html>
```

search.php

```
1   <?php
2   # This creates a session / cookie
3   session_start();
4   if(isset($_SESSION['auth'])) {
5     echo "You are logged in!<br>";
6   } else {
7     echo "You are <b>NOT</b> logged in!<br>";
8   }
9
10
11
12  if( !isset($_SESSION['auth']) ) {
13    echo "Please login!";
14    die();
15  }
16
17  if(isset($_POST['username'])) {
18
19    $link = mysqli_connect("localhost", "root", "", "sqli");
20
21    // No security here! SQLi possible!
22    // $u = addslashes($_POST['user']);
23    $u = $_POST['username'];
24
25    $query =
26      "SELECT * FROM users WHERE username LIKE '".$u. "'";
27    //file_put_contents("query.log", $query . "\n", FILE_APPEND);
28    $result = mysqli_query($link, $query);
29    if ($result) {
30      if ($row = $result->fetch_row()) {
```

```
31          echo "<h1>Found user! ".$row[1]."</h1>";
32        }
33      }
34      if (!$result) {
35        echo mysqli_error($link);
36      }
37
38  }
39
40  ?>
41
42  <html>
43  <body>
44
45  Search for a user!<br>
46  <form action="search.php" method="POST">
47  <input type="text" name="username">
48  <input type="submit">
49  </form>
50
51  </body>
52  </html>
```