UNIVERSITY OF AMSTERDAM

GRADUATE SCHOOL OF INFORMATICS
System and Network Engineering

# *Automated vulnerability scanning and exploitation*

Thijs Houtenbos thijs.houtenbos@os3.nl
Dennis Pellikaan dennis.pellikaan@os3.nl

July 12, 2013

**Supervisors**
Bart Roos bart.roos@ncsc.nl
Jop van der Lelie jop.vanderlelie@ncsc.nl

University of Amsterdam
Graduate School of Informatics
Science Park 904
1098XH Amsterdam

# Abstract

Automated vulnerability scanning is often used in professional development environments to find critical security issues. But what if those techniques are applied to scripts available on the internet? Many scripts are shared on sites like Sourceforge and GitHub, but security might not have been a priority during their development. Using a completely automated approach, a large number of these scripts were downloaded, analysed for vulnerabilities, tested, and finally websites using these scripts were searched for. Combining the information gathered during all these steps in this approach, a list can be generated of web servers running vulnerable code and the parameters needed to exploit these.

Because each of these steps is completely automated, it is possible to continuously download new or updated scripts, and find vulnerable systems that are open to the internet.

During this project, more than 23,000 scripts were downloaded of which more than 2,500 were identified as containing possible vulnerable code. Using the Google search engine, it was possible to find almost 8,000 installations of these vulnerable scripts.

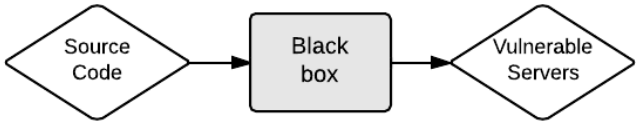
# Contents

# 1 Introduction

Many web applications that are connected to the internet are not written by the person hosting the application. Web developers do not always have the required skills to develop certain functionalities that they need for their websites. An answer to this high demand of generic web scripts is provided by large community driven sources, such as Sourceforge. They provide the web with thousands of open source web applications. Anyone is free to upload their code on this website and it automatically becomes available to millions of users to download the source code. Not everyone using these scripts is able to analyse how the functionality is programmed and what possible security risks are introduced by running these scripts on their servers.

Current research mostly focuses on the highest recall of vulnerabilities in a specific application. The methods used are often very efficient and range from detecting very simple vulnerabilities to more complex vulnerabilities. The tools that are often created can be used by software developers to improve the security of their applications.

*Figure 1.1: Concept*




This research project is not about finding complex vulnerabilities in one specific application, but it targets simple vulnerabilities in a large number of applications of which the source code is readily available on the internet. Figure 1.1 visualises the concept behind this research, in which large amount of source code enters the black box, and it returns a list of vulnerable web servers.

# 2 Research questions

The main research question for this report is the following:

***How feasible is an automated approach to compromise servers using a known source code attack on a large scale?***

This question can be split up in to the following sub-question:

- How can a large amount of source code be audited in an automatic way?
- How can exploits be automatically generated for discovered vulnerabilities?
- How can installations of vulnerable scripts be found and how can this process be automated?
- How can the previous steps be combined to create a system to automatically audit scripts, create exploits for found vulnerabilities and run the exploits on found installations?

# 3 Related work

## 3.1 Common vulnerabilities

Much effort has been devoted to classify and categorise vulnerabilities. Common Vulnerabilities and Exposures (CVE) [6] was launched in 1999 to address the problem of different information security tools using their own databases with their own name for security vulnerabilities. At the time it was hard to tell when the different databases were referring to the same vulnerabilities, and there was no effective interoperability between the different databases. CVE is now the industry standard for sharing vulnerabilities and their exposure names. CVE allows for good insight in current vulnerabilities and possibly existing exploits, which can often be found on websites, such as the Exploit Database [1]. Exploit sources, such as the Exploit Database, often link exploits to vulnerabilities or exposures in the CVE database. The CVE database gives insight in the types of vulnerabilities that currently have a prominent role in web applications, and it provides a starting point for selecting the vulnerability categories that are relevant to this research.

Related to CVE, the Open Web Application Security Project (OWASP) was created to make software security visible, such that users and developers can become familiar with these vulnerabilities and that they can make an educated decision on how to solve these problems. Different from CVE is that the OWASP top 10 project focuses on the general aspect of common vulnerabilities. OWASP provides, amongst other material, literature to educate users and developers about these top 10 vulnerabilities and how to deal with them. OWASP revises their top 10 every few years. The latest top 10 was updated on June 12, 2013 [2]. Compared to the previous top 10 list of April 19, 2010 [1], an interesting observation shows that on both lists the number one vulnerability is *Injection*. Injection is a general term, which means that it is possible for an attacker to inject code into the web application to change the behaviour of the application. Often, injection is thought of only as SQL Injection, but other attack vectors are also possible, such as shell injection. Because both CVE and OWASP aim at openness, they allow for good insight in the vulnerability categories that are interesting for targeting in this research.

Automatically finding vulnerabilities in web scripts are commonly approached in two different manners. First, there is the black box method, which looks at the application from a functional point of view without looking into its internal structure [3]. Second, there is the white box method. White box testing is done by looking at the source code of the application, which requires an understanding of the application. A survey of these techniques is made by *Vigna et al* [7]. The techniques described in that paper are useful for a web developer or auditor to test the security of a single application. However, these methods often require some form of interaction with the user of these tools. Automatically analysing source code requires a different approach in which it is allowed to have false positives, but that there is an extra step to verify the results. Within this research, a solution is provided to automatically verify if a potential vulnerability can be exploited.

[1] `http://http://www.exploit-db.com/`

## 3.2 Automatic generation of exploits

The automatic generation of exploits depends greatly on the category of the vulnerability.

SQL Injections are often targeted by attackers, and the automatic generation of SQL Injections is thoroughly researched. The automatic generation of SQL Injections can be divided into two categories. First, their goal is to expose vulnerabilities. Research by *Kieżun et al.* [10] shows a method to automatically generate SQL injections that expose the vulnerability when it exists. The goal however is not to automatically compromise web server that are prone to SQL Injections. Another problem with their approach is that they require of fully installed and working application. Their strategy is to query the web server with malicious code and monitor the output of the system. From there on they can deduce if the web server is prone to SQL injections. However, to automatically analyse large amounts of web scripts, it is not possible to also automatically install the web scripts as fully functional applications. The steps that need to be taken can vary enormously between different scripts. Therefore, their methods do not fit the process in this research to automate the analysis of large amount of source code.

Automatically compromising web servers is the second category of the automatic generation of SQL Injection exploits. A popular penetration testing tool is sqlmap [2]. Sqlmap not only generates SQL injections to expose vulnerabilities, but it actually tries to take over the database and execute commands on the operating system of the database server. Sqlmap's approach to find SQL Injection vulnerabilities is similar to the research done by *Kieżun et al.*, as described above. This makes sqlmap not suitable for automatically finding SQL injections without installing the scripts, but when an SQL Injection is found, it makes it suitable for automatically compromising web servers using these vulnerable scripts.

## 3.3 Identify websites

Search engines are often used to find vulnerable websites that make very common mistakes. Search engines crawl the internet for websites and store information such as the title of the web page and the Uniform Resource Locator (URL) used to access the web page. The information that can be found in the URL may hold valuable information with regards to finding vulnerabilities. The Google search engine is known to be used intensively for finding vulnerable websites. It is used so intensively that the term *Google hacking* now refers to very process described above. Research done by *Billig et al.* [4], uses the information that Google indiscriminately stores for finding vulnerable websites. In their research they show how to form URLs to efficiently find websites. The URLs often not only contain the location and the file name of the website, but also the parameters that are passed to the web server. Gathering the parameters that are found in vulnerable web scripts allows for the automatic generation of search queries.

Because of the usefulness of Google for finding vulnerabilities, another initiative was started, called Google Dorks [3]. Google Dorks serves as an open portal for users to add Google search queries that lists vulnerable web pages. Very simple design mistakes by popular web applications can result in thousands of vulnerable hits on Google. However, Google does not allow for automatic crawling of their search engine, and Google blocks the automated search with a CAPTCHA when it detects this. Google also provides an Application programming interface (API) to use their search engine, but the API is linked to a user account and limits the number of queries to 100 per day. This hard limit

[2] `http://sqlmap.org/`
[3] `http://www.exploit-db.com/google-dorks/`

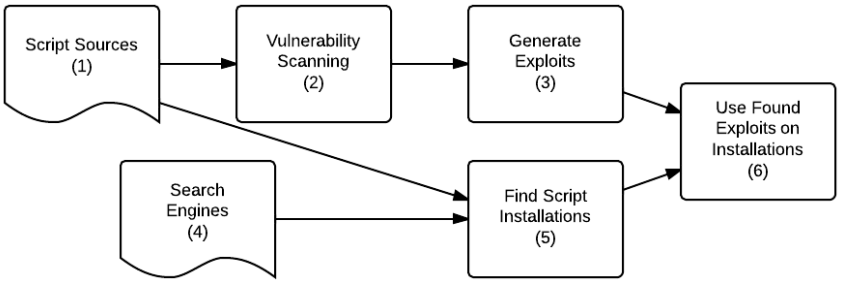
does not apply when using the Google web interface, but when Google suspects that an automated process is crawling their website, then the user needs to solve a CAPTCHA challenge. To overcome this problem, *Pelizzi et al.* [12] provide a solution that allows for uninterrupted search, but it still requires the user to manually solve the CAPTCHA challenge. Although their method does not provide a way to query the search results as fast a possible, it is adequate enough for this research.

# 4 Process

The goal of this research project is to demonstrate whether open source scripts available for download on the internet can be used to automatically find and exploit a large number of scripts. This automated approach can be seen as a black box system. The user of the black box provides the system with source code as input and could get access to vulnerable systems as the output. The processes executed inside the black box consist of several independent components. Linked together these complete the process. A schematic overview of the different components in the black box are shown in Figure 4.1. This process is different from other approaches in that it does not target any specific server, but rather focuses on the complete process of finding exploitable code and an installation base of servers running this code.

## 4.1 Process components

*Figure 4.1: Process overview*




### 4.1.1 Script sources

The process starts with the gathering of script sources. Since the entire approach is automated, the results improve as the amount of different source code packages increases. Popular script sharing sites can be used as input to collect a large number of scripts.

### 4.1.2 Vulnerability scanning

All script files in the source code are automatically analysed for any potential vulnerabilities. Depending on the type of program these can vary. For web applications there is a higher chance of problems in sanitizing user input, creating possibilities for injection. In desktop programs this focus can shift towards lower level problems, such as buffer overflows.

### 4.1.3 Generate exploits

The amount of vulnerabilities found in the analysed source code can be very large, depending on the number of input scripts. Part of the process is to exploit the vulnerabilities in an automated way. A local installation can be used to test behaviour of the program.

#### 4.1.4 Search engines

Once exploits for vulnerable code are created, an overview of locations where this code is running is required to use these exploits. Search engines crawl and index as much of the internet as possible to give users the ability to find specific websites. The database of these search engines can be used to gather this overview of installation locations of vulnerable scripts.

#### 4.1.5 Find script installations

The search results that are produced by search engines are validated to determine if the vulnerable script is actually installed on the remote server. Several methods that can be used to perform this comparison accurately is described later in this paper.

#### 4.1.6 Use found exploits on installations

The final component of the process is to run the created exploits on the installed vulnerable scripts. Depending on the amount of installations, the amount of found exploits and the efficiency of the previous components it is suspected that a large number of servers can be compromised. During the research no servers not taking part in this research project are attacked.

# 5 Approach and methods

During this research project a proof of concept of the black box system is made and tested. The source code of the proof of code is not publicly available, but it is available upon request. Contact information can be found on the front page of this paper.

The implementation of the several components are discussed in this chapter. For the proof of concept the focus is placed on PHP scripts and three types of vulnerabilities. PHP is a scripting language very popular among web developers. The easy learning curve makes it attractive for new developers who can share their scripts without performing a security review.

## 5.1 Script sources

Two sources are used to gather a large amount of scripts. Sourceforge[1] is a site offering free hosting for open source projects. More than 440,000 projects are available on Sourceforge. The site features tags to indicate the platform and languages used. Of those projects, more than 33,000 indicated that the PHP scripting language is used. There is an API available to request information about projects and the available files, but it cannot be used to filter for PHP projects.

GitHub[2] is a site extending on the great popularity of the git version management software. It provides web access to the managed code and an issue tracking system. Searching for 'PHP' results in more than 50,000 individual projects, or repositories as it is called in git.

## 5.2 Identify exploitable vulnerability categories

Automatically exploiting vulnerable websites requires for the exploit to be trivial. Within this research the focus goes out to what is referred here as "the low hanging fruit". These types of vulnerabilities are simple mistakes that are still made by web developers. As described by *Scholte et al.* [13], SQL injections and cross-site scripting vulnerabilities are still increasing. Their research focused on web vulnerabilities that were categorised in the CVE database from 2004 until 2009. The OWASP Top 10 Project [16] identifies similar vulnerabilities. The goal of the top 10 project is to raise awareness about application security. Beside the more generic top 10 list, OWASP also provides a top 5 that specifically focuses on the PHP scripting language [3]. A research that is done by *Fonseca et al.* [9], is by looking at attacks from the attacker's perspective, as opposed to that from a defensive perspective. They identified other high risk vulnerabilities that are often used by attackers. Their research aims at the most effective result, which often includes the execution of their code on a web server. Based on the CVE statistics, the OWASP PHP top 5 and the research done by *Fonseca et al.*, a subset of these categories is selected to be further used in this research, as shown in table 5.1. Other categories, such as cross-site scripting (XSS) and cross-site request forgery (CSRF), are also very popular, but they are not suitable for taking over web servers. These types of vulnerabilities are targeted at clients visiting the website, and require a different approach on how to automatically detect them, as described in [5].

[1] `http://sourceforge.net/`
[2] `https://github.com`
[3] `https://www.owasp.org/index.php/PHP_Top_5`

Table 5.1: Vulnerability categories

| Category |
|---|
| SQL injection |
| File Inclusion |
| Command Injection |

### 5.2.1 Parameters

When requesting a web page, several dynamic parameters can be passed to the web server. There are two main request types called GET and POST. Each type is explained shortly using an example request to website `http://www.example.com` and a parameter *user* set to value *admin*.

With a GET request type the parameters are passed in the request URL, behind the name of the web page. This combines to `http://www.example.com/?user=admin`.

In a POST request the parameters are passed as body in the request of the website. The normal script URL is requested (`http://www.example.com`) and the parameter body is sent after the headers (*user=admin*).

To identify the input parameters in this paper, the PHP notation is used. Meaning $_GET and $_POST represent arrays containing respectively the GET and POST request parameters, and $_GET[user] represents the value of parameter *user* in a GET request.

### 5.2.2 SQL Injection

Web pages are now often dynamically generated on request using scripting languages like PHP. To store the data used in these sites, a connection to a database server can be used. A popular database engine used in combination with PHP is MySQL. To be able to query the database from inside a script special run-time functions are available. These functions are provided by loading a module in PHP. Typically used functions for simple database questions are listed in Table 5.2.

Table 5.2: PHP functions for MySQL databases

| Function | Use |
|---|---|
| mysql_connect() | Initialize a connection to the database |
| mysql_select_db() | Select the database to use |
| mysql_query() | Execute a query on the database |
| mysql_fetch_assoc() | Return one row of query results |

To craft a page to a specific user, the input parameters can be used in the database queries. By combining these parameters and queries in an improper way, a possibility for exploitation is created.

```
1) mysql_query("SELECT * FROM ‘Users‘ WHERE ‘ID‘ = ’$_GET[user]’");

2) http://www.example.com/?user=admin
   mysql_query("SELECT * FROM ‘Users‘ WHERE ‘ID‘ = ’admin’");

3) http://www.example.com/?user=admin’ or 1=1--
   mysql_query("SELECT * FROM ‘Users‘ WHERE ‘ID‘ = ’admin’ or 1=1-- ’");
```

When the request parameter is used directly in the query (1) the content of the query can be altered by the user. Normally, the parameter contains only normal characters and it works as expected (2), but with specially designed parameter values the query can be adjusted in such a way that different data is returned from the database (3).

### 5.2.3 Command Injection

PHP [4] allows scripts to run code on the web server. When a dynamic parameter is passed to the execution function, and no, or insufficient input validation is done, then this may allow an attacker to execute malicious code. Command Injection is explained in detail in [14, 15]. An overview of the PHP functions that are prone to command injection is shown in table 5.3.

### 5.2.4 File Inclusion

Besides the functions that allow for executing programs outside of the scripting language, PHP also has functions that allow inclusion of other PHP files, or directly interpreting input data as PHP code. Web developers sometimes allow user input via parameters to these functions to allow for a dynamic feel of their web applications. An overview of the file inclusion functions that are looked at in this research, is shown in table 5.3. A survey done by *Ami and Malav* [11] explains this type of vulnerability and others more in depth.

*Table 5.3: PHP functions prone to command injection and file inclusion*

| Command Injection | File Inclusion |
|---|---|
| backtick operator | include |
| exec | require |
| passthru | include_once |
| popen | require_once |
| proc_open | eval |
| shell_exec | assert |
| system | preg_replace |
| pcntl_exec | |

## 5.3 Vulnerability scanning

To identify possible vulnerable scripts in the downloaded projects, all files are scanned for known common programming problems. Using the regular expressions in Appendix B, all files with extension *.php* are scanned on a line-by-line base. All lines matching one of the expressions are marked for further analysis. The regular expressions include detection for the three described categories. Custom functions outside the matched line can be used in the script to escape any user input parameters. This and other run-time script changes make that the results of the scanning process can include false positives. This calls for a need to automatically verify if the identified line is indeed vulnerable and if the exploitation can be done without user intervention.

## 5.4 Automate the exploitation

When one or more possible vulnerabilities are found in a script, it needs to be determined if the detection was correct or that it was a false positive. In this case, a false positive does not necessary mean that a script is not vulnerable, but that automatically exploiting the vulnerability did not succeed. A common scenario of not being able to automatically detect if a script is vulnerable, could be that the vulnerability itself is found in a part of an application that can only be accessed after the user has applied the right credentials. This type of scenario is very complex to automate. The process of validating each found vulnerability is discussed below.

### 5.4.1 SQL Injection

As part of the vulnerability scanning process the scripts are analysed for a combination of query functions and direct parameter usage inside those func-

[4] `http://www.php.net/manual/en/ref.exec.php`

tions. This results in potentially vulnerable scripts. To be able to exploit such a vulnerability it first has to be verified that the scanning results are correct and the code can be misused.

To perform this verification a test environment is created with specifically designed database functions, overriding the default ones listed in table 5.2. By overriding these built-in functions the complete query can be analysed for possible injections in an automated way.

By calling each possibly vulnerable identified script with the detected parameters set to a distinguishable value, the results were verified. All queries executed by the script are collected using the custom database functions. By comparing the distinguishable parameter value with all values inside the database query, a match can be made and analysis can be done on any escaping or sanitising. When values are not escaped or sanitised, the input parameter matches the query values exactly, and injections to alter the query are possible.

### 5.4.2 Command Injection

Potentially found vulnerabilities for command injection functions (table 5.3) also have to be verified. Some of these functions are implemented at such a core level in the scripting language the they cannot be overridden during run time, so the method used to verify SQL Injections does not work for them.

The vulnerability scanning script collects several details about its results. These include the identified file, function name, line number, the parameter type and the parameter name. Using this information, a new test environment is created where the indicated line (1) is analysed before execution. The found command execution function is replaced with a specially created log function, while no changes are made to the parameters (2).

```
(1) shell_exec ($_GET['cmd']);
(2) log_function ($_GET['cmd']);
```

After replacing just the name of the injectable function, the script is called with the identified parameter set to a distinguishable value and the process is completed in the same way as with the SQL Injections. Because only the name of the called function is changed and only one dynamic request parameter is tested at the same time, the flow of the script is not influenced by this change and the results are reliable.

### 5.4.3 File Inclusion

With potential file inclusion vulnerabilities some of the same issues arise as with command injection. These functions are defined at a core level and cannot be overridden at run time either. To overcome this, the same approach is used as with command injection.

```
(1) require_once "includes/" . $_GET['page'];
(2) log_function ("includes/" . $_GET['page']);
```

An extra point of attention is that some of the functions are so called 'control structures'. This means that they do not necessarily have to be called as functions, but that they can also be used as statements (1). Where the parameters for a function are between parentheses, this is not always the case with statements, and this has to be taken into account when replacing the statement name with the logger function (2).

After the called function is replaced, the same verification method as with SQL Injections is used.

## 5.5 Search engines

The Google search engine is used to find installations of vulnerable scripts. For this research Google is chosen as the most effective search engine by comparing possible advanced search operators and the number of results. This section describes how search queries are constructed and what limits are encountered while performing a large amount of search queries.

### 5.5.1 Find websites using the Google search engine

Search engines are designed such that the results relate as much as possible to the search query that the user provides. With advanced operators[5] the Google database can be queried on information other than the page content. This can produce many results, but not necessarily accurate. Getting only accurate results from Google is not very likely, therefore there are two ways in which the search queries are constructed. The first approach is to find sites that have the vulnerable script file name in combination with the GET parameter that is used in that file. Second, sites are found by their titles that are found in the scripts. Both approaches are described in this chapter.

#### Find websites with URL parameters

By using the Google search engine, it is possible to specify what a URL must contain. The *allinurl* search operator allows for search results that must contain specific text in the URLs. Of the websites that contain vulnerabilities it is known in what file name the vulnerability exists and if the vulnerability is triggered by the GET parameter. For example, when a vulnerable file `page.php` uses the GET parameter `page_id`, this could be translated into the following search query:

```
allinurl:"/page.php?page_id="
```

Google returns all URLs that it has indexed that contain the file name in combination with the GET parameter in the URL. However, the results do not take the path before the file name into account. It is therefore possible that Google returns the following results:

```
http://example1.com/page.php?page_id=
http://example2.com/admin/page.php?page_id=
http://example3.com/somesite/somefolder/page.php?page_id=
```

#### Find websites with titles

Another operator in the advanced Google search queries is *allintitle*. Similar to the *allinurl* operator, this operator searches for websites having all words of the search term in the HTML <title> element of the website. All projects containing possible exploitable scripts are analysed for page titles.

The results for the title scanning process are listed in table 5.4. The amount of titles found per project greatly differs. For more than half of the projects between 1 and 10 titles were found but there are also projects with more than 2000 titles. Since it cannot automatically be determined what titles give the best results before querying the search engine, projects with a large amount of titles are excluded from the search.

To efficiently query the search engine a selection is made of what titles to include in the search process as a trade-off between number of search queries and amount of projects covered in the search process.

[5] `https://sites.google.com/site/gwebsearcheducation/advanced-operators`

Table 5.4: Titles collection

| Produced titles | Scripts | | | Titles | | |
|---|---|---|---|---|---|---|
| | Abs. | Rel. | Rel. cum. | Abs. | Rel. | Rel. cum. |
| 1 - 10 | 1,219 | 59 % | 59 % | 4,267 | 7 % | 7 % |
| 11 - 25 | 356 | 17 % | 76 % | 5,737 | 9 % | 16 % |
| 26 - 50 | 238 | 12 % | 88 % | 8,378 | 14 % | 30 % |
| 51 - 100 | 129 | 6 % | 94 % | 8,918 | 15 % | 45 % |
| 101 - 150 | 36 | 2 % | 96 % | 4,399 | 7 % | 52 % |
| > 150 | 80 | 4 % | 100 % | 29,268 | 48 % | 100 % |
| **Total** | **2,058** | **100 %** | **100 %** | **60,967** | **100 %** | **100 %** |

As a result, only projects with twenty-five titles or less are included in the search. This includes more than 75% of the projects in the search process while only querying the search engine for 16% of the found titles, greatly reducing the total amount of search engine queries.

### 5.5.2 Selective results

Some of the automatically generated queries are so generic that a large amount of items in the search engine's database are a match. To prevent the gathering of lots of false positive, result filtering is done depending based on the total amount of results available, as indicated by the Google search engine. All search queries where the search engine indicates more than 750 results are flagged as too generic. During the search process this initial indication on the estimated amount of results as returned by the search engine turned out to be a very unreliable source for the actual number of search results.

A better indicator is found in the links to the number of pages at the bottom of the search result page. With each page returning ten results, this can be used to calculate another estimated amount of results for the search query. For some queries, the indicated amount of results was set to numbers close to 50,000, while the amount of result pages indicated less than 100 results. The number of pages turns out to be much more reliable and is eventually used as the filtering selector.

### 5.5.3 Throttling

Google has built in rate limiting integrated into their search engine. Especially for the advanced search queries, which are more resource demanding, these limits are well within reach when using automated scripts to execute the searches.

*Pelizzi et al.* [12] describe the use of proxy servers and a throttling system to automatically perform a large number of search queries. Their paper does not describe the exact limits for the throttling system and their approach is to circumvent the throttling by manually solving the presented CAPTCHA images.

During this research attempts were made to extend on this and perform a large number of search queries without interruption and without user interference. After several attempts an optimum was found in the amount of queries executed over time and the blocking behaviour, allowing for a continuous search process. Using a total of twenty-six different IP-addresses distributed across two servers it was possible to run over 20,000 queries per day.

As an addition, the use of IP version 6 addresses for the search queries was investigated. By using a random address from a 64-bit subnet for each query it was not possible to circumvent the rate-limit blocking. The blocking behaviour for the entire 64-bit IPv6 network could be matched with that of a single host on IPv4, indicating the single host IPv4 limits are equal to the limits for an entire 64-bit network in IPv6.

## 5.6 Identify websites using vulnerable scripts

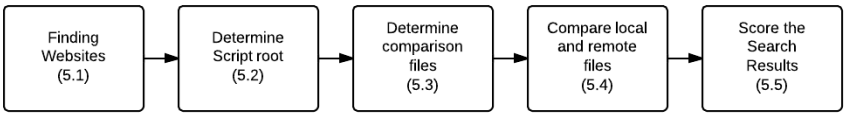



*Figure 5.1: Finding installed scripts*

When all vulnerable scripts are identified, the next step is to identify the websites that are using these scripts. By zooming in on step 5 of figure 4.1, this part of the whole process is explained more in depth. Section 5.5.1 discusses the methods that are use to find websites that are using the identified vulnerable scripts (5.1). The next step (5.2), is to determine the installation root of the script, which is described in section 5.6.1. By knowing the installation root of a script, it is possible to deduce the relative path of files within that script. When the installation root of the script is determined, the frequency of each file type that is used within that script, is used to determine what file types are more common. For each script, a selection of maximum six files is made to be used for validation (5.3). The most common file types are used to support the selection process. Section 5.6.3 discusses the comparison method that is used to compare the selected files with the remote host (5.4). Finally, in section 5.6.4, the method is described of how the score is calculated to indicate the accuracy of the search result (5.5).

### 5.6.1 Script installation directory

When the Google search results are gathered, it is still uncertain if these results contain false positives. When common file names and parameters are used, such as `login.php?id=`, it is very likely that many results are not related to the vulnerable script. The installation root of the vulnerable script needs to be determined before something useful can be said about the files that reside within this directory or subdirectories. Two methods are designed to determine the installation root of the vulnerable script. The first method is a more deterministic approach, and it uses the absolute path of the local vulnerable PHP file when the search results are found with the *allinurl* Google search operator. The second method is more probabilistic approach, which is used when the first method does not suffice. The results of the second method are less likely to be successful, but it may still hold positive results. The second method goes through all the directories in the vulnerable script and tries to find an index file or a PHP file in the subdirectory that is closest to the root directory. This directory is then considered to be the installation root.

#### Deterministic approach

Of the files that are selected to validate the installation of vulnerable scripts, it is still unknown in what location the file is located on the remote server. The installation directory of the web script, greatly depends on where the web master places the files. In a shared hosting environment it is very common that the directory where the files are stored have a prefix that is specific to that website, for example: `http://example.com/script/` or `http://example.com/username/webapps/script/`. To solve this problem, the absolute path of the vulnerable PHP scripts is used to find the possible root of the installation. When the Google search results are found by using the *allinurl* operator, the results also include the PHP file name that was used to find the URLs. The installation root is found by first removing the file name part of the URL and the local file path, and then comparing the subdirectory names from right to left. When the local subdirectory does not match with

the remote subdirectory, then it can be concluded that this is where the root of script is. An example of this process is shown in table 5.5.

*Table 5.5: Example of finding the script installation root*

| Step | Local path | Remote path |
|---|---|---|
| 1 | /script/admin/login.php | /example.com/user/www/admin/login.php |
| 2 | /script/admin/ | /example.com/user/www/admin/ |
| 3 | /script/ | /example.com/user/www/ |

In this example it can be concluded that the local installation root is `/script/` and the remote installation root is `http://example.com/user/www/`. As shown in algorithm 1, the installation root always starts where the local and remote subdirectory do not match.

**Algorithm 1**: Deterministic method to find the script installation root

```
Output: Script installation root
initialization;
local_dir = split ('/', remove_filename (local_path));
remote_dir = split ('/', remove_filename (remote_path));
j = count(remote_dir);
for i ← count(local_dir) to 0 do
    if local_dir[i] <> remote_dir[j] then
        local_install_dir = join ('/', local_dir);
        remote_install_dir = join ('/', remote_dir);
        exit for;
    end
    splice (local_dir, i, 1);
    splice (remote_dir, j, 1);
    j − −;
end
```

Both the local installation root and the remote installation root are important to further determine if the script is installed on the remote server or not. This is explained in section 5.6.2

### Probabilistic approach

When the method described above does not have any results, or the script is not suitable for that particular method, then a more optimistic approach is used to find the installation root.

Determining the root of the script is done by looking at the name of the files. The first attempt is to look for a file with the name `index.php`, `index.html` or `index.htm`. When one of these files is found, then the directory wherein this file resides is considered to be the installation root of the script. When none of these file names exists within the script, a second search attempt is started to look for a file with the *php* extension. The directory in which a file is found with this extension is considered to be the installation root of the script.

For each find attempt the directories are traversed one level at a time, starting at the root of the script in which it is downloaded. This means that the find process goes through each layer of subdirectories and only moves up one level when no files are found. This way it prevents the find algorithm from going into many subdirectories. It is not expected that the installation root lies deep within the script directory structure. Table 5.6 explains this problem with an example.

*Table 5.6: Example for traversing subdirectories*

| Step | Level | Path |
|---|---|---|
| 1 | 1 | /backup/ |
| 2 | 1 | /www/index.php |
| 3 | 2 | /backup/www/index.php |

In the first step, when the level is one, the first subdirectory is search through is `backup`. Because within this directory there is no index file, the find process continues on to the next step, which is also at level one. In step two the process is finished, because it has found an `index.php` file in `/www/`. If the find level would not limit itself to the current search directory, then the find process would continue in the `backup` directory and very likely identifies the wrong file. Algorithm 2 describes this process more formally.

**Algorithm 2**: Probabilistic method to find the script installation root

```
Output: Script installation root
initialization;
for i ← 1 to 10 do
    foreach files do
        if filename (file) = 'index.[(htm[l]?|php)' then
            install_root = dirname (filename);
            return;
        end
    end
end
for i ← 1 to 10 do
    foreach files do
        if filename (file) = 'php$' then
            install_root = dirname (filename);
            return;
        end
    end
end
```

The reason this method has probabilistic characteristic is that there is no known other method to validate the result of this process. The outcome of this process is taken as a basis for further validation.

### 5.6.2 Common file types

The next step is to validate if the Google search results indeed contain the websites that are using the vulnerable scripts. Because PHP files are server side scripts and cannot be compared by simply downloading them from the web servers, another approach is used to validate the Google search results. Most web applications have more files than only PHP scripts. Images, text files and client-side scripts, such as JavaScript, are often part of a web application. Because of the copy and paste behaviour that many web developers have, it is very likely that these files are also installed on the web server. A selection of these files is made from the vulnerable scripts to see if they exist on the remote web servers, and if they exist, to what extend they compare. The comparison process is explained further in section 5.6.3.

A frequency table of the most commonly used file types is generated by going through each file of all vulnerable scripts and count the number of occurrences per file type. Based on their frequency and the file type, a subset of this list is selected and is used for validating the Google search results. Of the file types that are selected it is known that web servers normally do not block access to them and that they are not expected to change frequently, such as JPG, PNG and GIF. But also CSS, JavaScript and HTML files are not excluded, because of their popularity. For each script, a maximum of six files is selected. The installation directory of each script is searched through, selecting one file for each file type. This way it assured that the validation process does not focus only on one file type. When all file types have been searched for and there are less than six selected files, this process is reiterated until there are six selected files, or until there are no more files to be found in the installation directory. It could also mean that there may not be any files to select.

### 5.6.3 Compare files in the installation root with the remote host

Comparing local files with remote files is done in two ways. First, an MD5 hash is calculated for the local file and the remote file and then both hashes are compared. Second, when the hashes do not match, the files are compared for similarity when the files contain text, such as HTML and JavaScript. The differences between both files is looked at from a local point of view and a remote point of view. This means that it is looked at how much text of the local file is found in the remote file, and how much text of the remote file is found in the local file. Both values show a percentage of how much text of one file occurs in the other file. The method used to calculate this is explained in the research done by *Grune and Huntjens* [8]. The local file score and the remote file score are later used to calculate a final score value that indicates how likely it is that the remote server has the vulnerable script installed.

Each URL that is found during the Google search process is contacted with a request for the files that are selected. After which it is tried to download each selected file and compare the remote file with the local file. Depending on the number of hashes that match with the local file, a score is assigned to a URL. URLs containing a score above a certain threshold are considered positive search results, all other URLs are considered false positives. The scoring algorithm is explained following section.

### 5.6.4 URL validation score algorithm

When the Google search results are validated, the total score of how accurate the URL is needs to be determined. There are three different states in which a search result can be after it is validated.

1. The search result could not be validated, because the installation root of the script could not be determined.
2. The search result could not be validated, because there are no files available to compare with the remote host.
3. The search result is validated based on file comparison and each file has a score assigned to it.

In the first case, the search result cannot be validated and therefore it is considered to have a good change of not being correct. In the second case the script does not contain any of the common file types. Very small scripts may only be intended to be included as part of larger applications, or have very simple functionality. This makes it very hard to validate the correctness of the search result. The last case allows for files to be compared with the remote host. As explained in section 5.6.2, each file has several scores assigned to it. These subscores are used to calculate a final score, which indicates the accuracy of the search result.

For each script there is a maximum of six files that are selected to validate if the vulnerable script is installed on the remote host. When there is a hash match between a local file and a remote file, both the *local score* and the *remote score* have a value of 100. When there is no hash match, then both scores can be different, as explained in the previous section. Comparing differences between files is only done when the files contain text, and thus are not binary. When there is no hash match and the file contains binary data, then both scores are set to 0. The final score is calculated as shown in equation 5.1.

$$Score = \frac{\sum_{i=1}^{N} S_i}{N} + \sum_{i=1}^{N} S_i * \frac{1}{6} \tag{5.1}$$

$N$ = Total number of selected files
$S_i$ = $\frac{LocalScore_i + RemoteScore_i}{4}$

Within this equation it is taken into account how many files can be used for validating the search result. When there is only one file available and it has a hash match with the remote file, meaning a *local score* of 100 and a *remote score* of 100, then the final score is 58. This way, validating a search result with only one file does not get a perfect score. When three files are available and all of them have a hash match with the remote file, then the final score is 75. When more than one file is used for validating a search result, it increases the changes of being correct. Therefore, it is considered that three or more files give an accurate indication if a vulnerable script is installed on the remote host.

When the score is less than 50, it is considered to be a false positive. When the score is equal or greater than 50 and less than 75, the search result is considered to be plausible. A score of 75 or higher is considered to be accurate. A score of 100 means that six files are available and all have a *local score* and a *remote score* of 100. This is considered to be a perfect score.

## 5.7 Automate the compromisation

Once the vulnerabilities in the scripts are found and the installations of those scripts are identified, then the last step is to automatically complete the process of exploitation. The way vulnerabilities can be exploited depends on the category.

### 5.7.1 SQL Injection

After the search process is completed, all parameters are available to run automated SQL Injections. The open source penetration testing tool sqlmap is designed specifically for this purpose. It can be called by passing a URL of the web page and the vulnerable parameter and can then perform automatic analysis of the page and complete the exploitation. Access can be gained to the database behind the website, and depending on the configuration of the server, shell access on the system is also possible.

### 5.7.2 File Inclusion

File Inclusion vulnerabilities are harder to exploit. With Local File Inclusion vulnerabilities, the attacker first needs to upload a file to the server before it can be included. Because of the low number of found File Inclusion vulnerabilities and the complexity of automatically completing the exploitation of this type of vulnerability it has not been automated during this research. To execute an attack, specific script and system parameters need to be analysed by the attacker before a successful attack can be launched.

### 5.7.3 Command Injection

Command injection vulnerabilities are the easiest to exploit from the three categories discussed in this paper. The calling of a page allows direct execution of shell commands on the remote server, running as the same user as the web server is running. With direct shell commands execution it is possible to start processes listening for incoming shell connections or install custom programs on the remote machine.

# 6 Results

## 6.1 Collect scripts

Over a period of two weeks, a total of 23,291 PHP scripts were downloaded. 9,668 scripts were downloaded from GitHub and 13,623 scripts were downloaded from Sourceforge. The SourceForge website claims that they have over 33,000 PHP scripts, but many of those scripts did not actually contain any PHP files. Some of those projects were incorrectly tagged as being PHP projects, or the projects were just simply empty. Both sources provided a website that allowed for automatic crawling for scripts. GitHub however, uses rate limiting to prevent users from hammering their servers. It appeared to be the case that GitHub only checked the number of requests in a specific time frame, and therefore this could easily be solved by introducing a 60 second pause between every 20 requests. This allows to continuously crawl GitHub for new repositories. GitHub provides advanced search features that can be set in the GET parameters of the URL when requesting the GitHub web page. This way it is possible to continuously ask GitHub for the latest updated repositories and it provides an interesting method to constantly analyse the source that is added to GitHub.

## 6.2 Identify vulnerabilities

Of all 23,291 PHP scripts, $\approx 11\%$ contained one or more vulnerabilities that were found using the regular expression as listed in appendix B. Table 6.1 shows the number of vulnerabilities found per category. The total number of vulnerabilities found exceeds the number of vulnerable scripts. This is because a single script can have vulnerabilities of one or more categories.

*Table 6.1: Vulnerabilities per category*

| **Category** | **Frequency** | |
|---|---|---|
| | Absolute | Relative |
| SQL Injection | 2,333 | 84.59 % |
| File Inclusion | 376 | 13.63 % |
| Command Injection | 49 | 1.78 % |
| **Total** | **2,758** | **100.00 %** |

An interesting observation that can be made is that of the 9,668 GitHub scripts, only $\approx 5\%$ is identified as having a vulnerability. Whereas, of the 13,623 Sourceforge scripts, $\approx 18\%$ is identified of having a vulnerability. This difference does not necessarily mean that the source code on GitHub is better. Because the source code that was pulled from GitHub were recently updated repositories. The source code of Sourceforge still contains old code that is not maintained any more, but it is still used by web developers.

## 6.3 Generate exploits

Section 5.4 describes the methods that are used to automatically determine if a vulnerability is automatically exploitable. Of over 14 % of all found vulnerabilities it is certain that the vulnerability is exploitable. This means that the script is indeed vulnerable, but it cannot be concluded with absolute certainty that each installation of an automatically exploitable script can be compromised.

This depends on the environment in which the script is installed. However, in normal conditions, these vulnerabilities are automatically exploitable.

*Table 6.2: Automatically exploitable vulnerabilities*

| **Category** | **Frequency** | |
|---|---|---|
| | Absolute | Relative |
| SQL Injection | 334 | 85.20 % |
| File Inclusion | 49 | 12.50 % |
| Command Injection | 9 | 2.30 % |
| **Total** | **392** | **100.00 %** |

## 6.4 Find websites

Of the 2,552 vulnerable scripts, 2,217 were suitable for creating Google search queries. The scripts that could not be used for the generation of search queries only contained PHP files that did not produce a title or used a GET parameter. Of those 2,217 scripts it was possible to construct 22,469 Google search queries. Table 6.3 shows how many search results were produced by finding sites with a title and finding sites with the vulnerable PHP file name in combination with the associated GET parameter.

*Table 6.3: Google search results*

| **Category** | **Total** |
|---|---|
| Search results found by title | 71,622 |
| Search results found by GET parameter | 47,535 |
| **Total** | **119,157** |

## 6.5 Validate search results

Validating search queries is done by comparing common files that are locally available with those on the remote system. Section 6.5.1 describes the effectiveness of both methods that are used for finding the installation root of the script. The frequency of common file types in the vulnerable script, and of which is known that web server normally allow access to them, is discussed in section 6.5.2. Finally, the results of validation process of the search results is discussed in section 6.5.3.

### 6.5.1 Installation root

The installation root was first tried to be determined with the file name of the vulnerable script in combination with the Google search results associated to that scripts. All search results that were found with the file name of the vulnerable script were used in this process. When the installation root could not be determined, then a probabilistic method was used to determine the installation root. Table 6.4 shows the effectiveness per method for determining the installation root of each search result. When the deterministic method was not successful in determining the installation root, then in all other cases the probabilistic method was successful. Therefore, in all cases each vulnerable script contained at least an `index.htm`, `index.html`, or a PHP file.

*Table 6.4: Effectiveness of determining the installation root*

| **Method** | **Total** |
|---|---|
| Deterministic | 9,813 |
| Probabilistic | 109,344 |
| **Total** | **119,157** |

### 6.5.2 File types

The frequency per file type is counted for all 2,552 vulnerable scripts. In total there were 3,117,901 files. A large amount of these files were related to how git manages repositories, those files can safely be ignored. Table 6.5 shows the most commonly used file types of all vulnerable scripts. Of these file types it is known that most web servers do no block access to them by default, and that they are not expected to change regularly. Therefore, these file types are good candidates for validating the Google search results. More than 25% of all files found in the vulnerable scripts were PHP files, but those files cannot be used for comparison, as explained in section 5.6.2.

*Table 6.5: Most common file types*

| **Rank** | **File type** | **Frequency** | | |
|---|---|---|---|---|
| | | Absolute | Relative | Relative cumulative |
| 1 | gif | 405,056 | 12.99 % | 12.99 % |
| 2 | png | 358,119 | 11.49 % | 24.48 % |
| 3 | js | 237,648 | 7.60 % | 32.08 % |
| 4 | html | 158,764 | 5.09 % | 37.17 % |
| 5 | jpg | 156,490 | 5.02 % | 42.19 % |
| 6 | txt | 52,477 | 1.68 % | 43.87 % |

### 6.5.3 Search result scores

Table 6.5.3 shows the final results of the search results that were returned by Google. The number of search results that have a score of zero is still very high. These results are very likely false positives. None of the files that were used for comparison matched with the local files. A large amount of scripts do not have any of the common file types that can be used for validating the search results. These type of scripts are expected to be very small or to be still in development.

*Table 6.6: Search result scores*

| **Score** | **Frequency** | | |
|---|---|---|---|
| | Absolute | Relative | Relative cumulative |
| 0 | 58,520 | 49.11 % | 49.11 % |
| 1 - 24 | 3,490 | 2.93 % | 52.04 % |
| 25 - 49 | 1,969 | 1.65 % | 53.69 % |
| 50 - 74 | 6,511 | 5.47 % | 59.16 % |
| 75 - 99 | 1,018 | 0.85 % | 60.01 % |
| 100 | 324 | 0.27 % | 60.28 % |
| No common file types | 47,325 | 39.72 % | 100.00 % |
| **Total** | **119,157** | **100.00 %** | **100.00 %** |

Of the results with a score of zero or higher, it is also looked at which search operator proved to be most effective. Table 6.7 shows the search results related to the Google search operator. Although the majority of the search results that could be validated were produced with the *allinurl* Google search operator, the effectiveness of the search operator does not show to have more accurate results than the *allintitle* search operator.

*Table 6.7: Search results related to the Google search operator*

| **Score** | **allinurl** | | **allintitle** | |
|---|---|---|---|---|
| | Absolute | Relative | Absolute | Relative |
| 0 | 38,837 | 84.54 % | 19,683 | 76.02 % |
| 1-24 | 2,002 | 4.36 % | 1,488 | 5.75 % |
| 25-49 | 928 | 2.02 % | 1,041 | 4.02 % |
| 50-74 | 3,643 | 7.93 % | 2,868 | 11.07 % |
| 75-99 | 450 | 0.98 % | 568 | 2.19 % |
| 100 | 79 | 0.17 % | 245 | 0.95 % |
| **Total** | **45,939** | **100.00 %** | **25,893** | **100.00 %** |

# 7 Conclusion

This research focused on an automated approach for analysing web scripts for vulnerabilities, finding the websites that are using these scripts, and finding methods for compromising these websites. The different processes that were defined to automate this approach were combined into a *black box* system.

Large scale automated source code analysis is possible when the type of vulnerability is trivial. As described in section 5.3, source code can be audited automatically using regular expressions. In total, approximately 11% of the audited scripts contained one or more possible vulnerabilities. For more reliable results, this can be extended to static code analysis, which takes statements spread over multiple lines into account.

In section 5.4, a method is described to automatically confirm the possibility for exploitation of given parameters by replacing built-in functions of the scripting language. By comparing the input of the parameters that are used to call the script with the output of the replaced functions, any escaping or sanitising can be detected. Using this method, it was possible to automatically verify almost 14% of the found vulnerabilities.

To find installations of the vulnerable scripts, the Google search engine was used. It was possible to tune the scraping of the search script in such a way that the search process could run completely automatic, without any user intervention to circumvent rate limiting. A total of more than 22,000 search queries were constructed from the vulnerable scripts, which resulted in almost 120,000 search results. The search results that were collected using the methods described in section 5.5 are not very accurate. Therefore, a method to verify if the results match the original script is included in section 5.6. After verification of the search result, almost 8,000 search results were considered to be a match with the vulnerable script.

By running the complete system on the collected scripts, it was confirmed that the proposed automated approach to analyse available scripts is considered feasible. Especially the automated aspect, which makes it possible to run the system on a large scale, makes it interesting to use such a system. With a continuous flow of new scripts, it can run unattended for an extensive time to increase the results.

# 8 Future work

The components that are defined in figure 4.1 all proved to have their own challenges. For each component their is room for improved and further work. Therefore, this chapter describes future work related to these components.

## 8.1 Component improvements

### 8.1.1 Collect scripts

Automatically downloading scripts from GitHub and Sourceforge has proven to be a relative easy task. Other similar sources may also allow for automatically crawling of their websites. This could easily result in more source code to be analysed, and this might expose more installations with vulnerable scripts.

### 8.1.2 Identify vulnerabilities

The number of vulnerabilities found during this research show that there are still many vulnerable scripts on the internet that are used by many web developers. The proof of concept that is part of this research only focuses on the PHP scripting language. Other popular scripting languages, such as ASP.NET, could allow for many more vulnerable websites to be exposed. This research focuses on simple vulnerabilities, the so called "low hanging fruit". The black box system that is shown in this research allows for a more complex vulnerability detection system. It is also interesting to look into other vulnerability categories that may yield many more vulnerabilities.

### 8.1.3 Generate exploits

The method provided within this research to automatically determine if a vulnerability is exploitable has proven to be very reliable. However, this method does not take into account simple control statements, which can determine when a certain section of code is evaluated or not. By improving the automatic verification of the found vulnerabilities it is expected the results can be improved greatly.

### 8.1.4 Find script installation

The method used in this research to find vulnerable websites by using the Google search engine, has proven to be a difficult task. The number of false positives is still very high and further research in this field should allow for more accurate search results. When GitHub is used for requiring source code, an interesting approach could be to analyse the git log of a specific web application to find the website that is using the application. This however narrows the scope to a single installation of the vulnerable script. The git log often also contains contact information. This information could be used to automatically inform the author about the vulnerabilities that are found.

# 9 Acknowledgements

We would like to thank the National Cyber Security Centre (NCSC) for giving us the opportunity to do our research there and providing us with an open and pleasant atmosphere to work in. We would especially like to thank Bart Roos and Jop van der Lelie for their continuous constructive criticism and their enthusiasm. We very much appreciate all that you have done for us.

# Bibliography


[1] OWASP top 10 - 2010. The ten most critical web application security risks. Technical report, OWASP The Open Web Application Security Project, 2010.

[2] OWASP top 10 - 2013. The ten most critical web application security risks. Technical report, OWASP The Open Web Application Security Project, 2013.

[3] Lauri Auronen. Tool-based approach to assessing web application security. In *Security, Seminar on Network Security*, 2002.

[4] Justin Billig, Yuri Danilchenko, and Charles E. Frank. Evaluation of google hacking. In *Proceedings of the 5th annual conference on Information security curriculum development*, InfoSecCD '08, pages 27–32, New York, NY, USA, 2008. ACM.

[5] Christian Korscheck. Automatic detection of second-order cross-site scripting vulnerabilities, 2010.

[6] Common vulnerabilities and exposure. `http://cve.mitre.org`.

[7] M. Cova, V. Felmetsger, and G. Vigna. Vulnerability Analysis of Web Applications. In L. Baresi and E. Dinitto, editors, *Testing and Analysis of Web Services*. Springer, July 2007.

[8] Dick Grune and Matty Huntjens. Detecting copied submissions in computer science workshops.

[9] Jose Fonseca, Marco Vieira, and Henrique Madeira. The web attacker perspective - a field study. *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, 0:299–308, 2010.

[10] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09, Proceedings of the 31st International Conference on Software Engineering*, Vancouver, BC, Canada, May 20–22, 2009.

[11] Parvin V. Ami, S. C. Malav. Top five dangerous security risks over web application. 2013.

[12] Riccardo Pelizzi, Tung Tran, Alireza Saber. Large-scale, automatic xss detection using google dorks.

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

[14] R. Sekar. An efficient black-box technique for defeating web application attacks .

[15] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. *SIGPLAN Not.*, 41(1):372–382, January 2006.

[16] The open web application security project. `https://www.owasp.org`.

# A Acronyms and abbreviations

**API** Application programming interface
**CSRF** cross-site request forgery
**CVE** Common Vulnerabilities and Exposures
**OWASP** Open Web Application Security Project
**URL** Uniform Resource Locator
**XSS** cross-site scripting

# B PERL regular expressions for finding vulnerabilities

The tables in this section show the regular expressions that are used to find the vulnerabilities in the web scripts. The regular expressions are formed such that they work well with the PERL scripting language, which was used for analysing the web scripts. When the column Ignore contains a "Yes", then that specific line of code is ignored to lower the possibility of false positives.

*Table B.1: Comments*

| Regular expression | Ignore |
|---|---|
| ^# | Yes |
| ^// | Yes |
| ^/* | Yes |
| addslashes | Yes |

*Table B.2: SQL Injection*

| Regular expression | Ignore |
|---|---|
| wpd->prepare | Yes |
| wpdb->prepare | Yes |
| wpdb->get_var | Yes |
| mysql_real_escape_string | Yes |
| select\ +.*\ from\ .*=.*\$_get | |
| select\ +.*\ from\ .*=.*\$_post | |
| insert\ +into\ +.*\$_get | |
| insert\ +into\ +.*\$_post | |
| delete\ +from\ +.*\ +where\ +.*=.*\$_get | |
| delete\ +from\ +.*\ +where\ +.*=.*\$_post | |
| update\ +.*\ +set\ +.*=.*\$_get | |
| update\ +.*\ +set\ +.*=.*\$_post | |
| mysql_query\ *\(.*\$_get.*\) | |
| mysql_query\ *\(.*\$_post.*\) | |

*Table B.3: File Inclusion*

| Regular expression | Ignore |
|---|---|
| ^include\ +.*\$_get | |
| ^include\ +.*\$_post | |
| ^require\ +.*\$_get | |
| ^require\ +.*\$_post | |
| ^include_once\ +.*\$_get | |
| ^include_once\ +.*\$_post | |
| ^require_once\ +.*\$_get | |
| ^require_once\ +.*\$_post | |
| ^eval\ *\(.*\$_get.*\) | |
| ^eval\ *\(.*\$_post.*\) | |
| ^assert\ *\(.*\$_get.*\) | |
| ^assert\ *\(.*\$_post.*\) | |

*Table B.4: Command Injection*

| Regular expression | Ignore |
|---|---|
| ^\`.*\$_get.*\`; | |
| ^\`.*\$_post.*\`; | |
| ^exec\ *\(.*\$_get.*\); | |
| ^exec\ *\(.*\$_post.*\); | |
| ^passthru\ *\(.*\$_get.*\); | |
| ^passthru\ *\(.*\$_post.*\); | |
| ^popen\ *\(.*\$_get.*\); | |
| ^popen\ *\(.*\$_post.*\); | |
| ^proc_open\ \(.*\$_get.*\); | |
| ^proc_open\ \(.*\$_post.*\); | |
| ^shell_exec\ *\(.*\$_get.*\); | |
| ^shell_exec\ *\(.*\$_post.*\); | |
| ^pcntl_exec\ *\(.*\$_get.*\); | |
| ^pcntl_exec\ *\(.*\$_post.*\); | |
| ^system\ *\(.*\$_get.*\); | |
| ^system\ *\(.*\$_post.*\); | |