Securing Web Applications

Jens Thomas Vejlby Nielsen
jtvn10@student.aau.dk
Computer Science 9^{th} Semester Student Project
Aalborg University

Aalborg Universitet
Cassiopeia
Computer Science
Selma Lagerlöfs Vej 300
Phone 96 35 97 31
Fax 98 13 63 93
http://cs.aau.dk

Title: Securing Web Applications

Project Period:
9^{th} semester

Project Group:
des907e14

Written by:
Jens Thomas Vejlby Nielsen
jtvn10@student.aau.dk

Supervisor: René Rydhof Hansen

Number of prints: 2

Pages: 27

Completed on: 8. January 2015

Abstract:

Securing web applications is difficult, and often developers are unaware of the lacking security. Here different vulnerabilities are presented, and how they can be prevented in the PHP programming language; namely SQL Injections, Cross Site Scripting, Cross Site Request Forgery, Code Injection and HTTP Header Injection.
The problem of securing web application plugins when a secure core is present, and how to stop plugins from affecting the core if they are vulnerable is presented, along with proposed solutions. Additionally, proposals for how to ensure some security level is presented.

The content of this rapport is freely accessible, but publication (with sources) is only allowed with written consent from the authors.

Preface

This project is the result of a 9^{th} semester project conducted by a student from the Department of Computer Science, Aalborg University. The theme of this report is Software Security, and the focus will be on web application security. The goal of the project is to analyse the problem with developing secure software for the web, and propose a solution for how to heighten the security. The proposal will be the basis for further work in the specialisation.

- Citations will be on the form [x], where the number represents the place in the bibliography in the back of the report.
- Code written in the report will be in the `ttfamily` font family.
- Broken lines are represented by a ↵.

I would like to thank René Rydhof Hansen for supervision and guidance during the project.

Contents

1 Introduction

In recent years more and more applications are being transferred to the web. As such, the attack surface which malicious attackers use also switches to web applications, as shown by a study conducted in the first half of 2012. This study shows that 47% of all attacks was on web applications [13].

The benefit of web applications is that the applications are more accessible, and there is no system requirements for using the application aside from a web browser. In addition, the developers will only need to maintain a single code base for all installations, and patching the application will be significantly more painless. An alternative would be to have individual client installations, and having to update any and all of these. However, since the web application runs on the Internet, the clients will always have the newest version of both the application and the data it contains.

The connection to the Internet is also a drawback for the web application; if the server which hosts the application crashes, all clients will loose access, the same is the case for data loss. Having everything centralised in a single web application also means that the users of the application will have to trust the web application to manage the confidentiality of the data contained within the application, along with the integrity of the data.

Another drawback with the availability of web applications on the Internet, is that then malicious attackers will have a single target, which will have to be as secure as possible. In a more decentralised solution, where each user has his or her own copy of the application with locally stored data, the malicious attacker would have to attack each user individually.

In web applications, the core problem is that often the clients are trusted to supply benign input. This is not always the case, and as such an applica-

tion can be vulnerable to malicious attacks. Creating secure applications is no easy task, and much research has been devoted to this task.

In this project the different security vulnerabilities will be presented, along with a proposed solution as to they can be prevented.

Chapter 2 will describe the analysis of the problem domain along with the different vulnerabilities. Chapter 3 will describe the problem, and Chapter 4 will describe the proposed solutions. Finally, Chapter 5 will conclude the project.

2 Analysis

This chapter will briefly introduce the used terminology, followed by a description of the core problem which is being worked with, and finally the analysed technologies.

A *Web Application* is an application running on a web server that takes the input supplied by the client and uses it to construct the output which is served to the client. A web application can be *vulnerable*, which means that if it receives unexpected input it can malfunction and either expose protected data, crash or other unintended action. In this context, the *client* is often a web browser, but can also be other types of applications, such as cURL.

In this report, a *vulnerability* denotes the class of the same vulnerabilities. An example of a vulnerability class is Cross Site Scripting (XSS), which allows a malicious attacker to embed scripts into the requests, which can lead to legitimate users executing malicious JavaScript.

Client and *User* are used interchangeable in the report, and both terms mean a legitimate user of the application.

2.1 The core problem

The core problem with web application security is that clients can submit arbitrary data to the application. In addition, the client can interfere with any piece of data that has been transmitted, and can avoid any client side validation. Examples of maliciously attacking a web application could be:

- Changing the price of a product.
- Submitting invalid passwords, that is, bypassing the security check.

- Modifying the HTTP header, for example by changing the session token.
- Only sending some of the query parameters, for example to avoid checks on the server side.
- Altering input which will be processed by a back-end database to get malicious access.

Additionally, although security awareness has increased in recent years, most web applications are still developed in-house and as such may contain unknown defects. Even in the case where the code consists of well-tested components, they are often bolted together using custom code [16].

Not only that, but often the development is focused on creating functional, stable code, along with adhering to a deadline, as opposed to the more intangible security aspect. In addition, most web applications are created by developers who lack the sufficient knowledge to design a secure application, and as such the detection of security problems is left till the end of the development cycle. Small organisations will only be able/willing to pay for a few man-days of security consultance, and as such only a small-scale penetration test will be conducted. This small test will only be able to identify the most obvious vulnerabilities [16].

The problem with developing code without a focus on security is that such code is more prone to security errors, which can be hard to detect. One of the ways the security can be improved is to use a well-established core which makes basis functionality easily accessible to the programmers. One example of such a core could be Wordpress, which is a blogging and Content-Management (CMS) system [11].

2.1.1 PHP and MySQL

A lot of different programming languages and databases exist for web development, but in this project the focus will be on PHP and MySQL.

PHP is a hugely popular programming language for web development [6, 9, 10], and MySQL is often the database used for web applications [7, 2]. This can be attributed to the fact that LAMP, MAMP and WAMP (Linux/Mac/Windows Apache Mysql PHP) stacks has become widely available, and makes it easy to set up a working environment for developing web applications, thus making it easy for developers to get started.

Almost any existing PHP content-management system or framework has support for using a MySQL database, and a lot of web hosts make installation of such frameworks or CMS systems easily available to the user through

automated installation scripts [8]. This makes it easy to deploy the systems for not so tech savvy users.

PHP is dynamically typed, interpreted and does not provide protection against vulnerabilities. As such, it is the developers responsibility to sanitise any and all input supplied to the web application. PHP does, however, supply sanitisation functions, that when used correctly will sanitise the input.

The developer will also need to make sure that the supplied input is of the correct type. An user could for example submit a string where an integer were expected, and this could lead to the web application being exploited.

PHP also makes dynamic execution of code possible, both supplied as a string and in a file. This makes it possible to supply for example a sort function as input. It can, however, also pose as a risk, as described in section 2.2.

PHP has 3 different classes for interacting with a MySQL database, the now-deprecated `mysql`, the successor `mysqli` and the object-oriented approach `PDO`. Of the 3 approaches, `PDO` can supply a generic database interface, and can be used for different databases apart from MySQL. This also makes it possible to change the database.

2.2 Vulnerability types

Trusting user supplied data is the primary reason for vulnerabilities in web applications, and as such it is necessary to sanitise the input which has been supplied by the users. Failure to properly sanitise input can lead to a web application being vulnerable, and this vulnerability can then be exploited by a malicious attacker.

Another way a web application can be vulnerable is to denial of service attacks, in which an application is brought offline or slowed down considerably. However, in this kind of scenario, the access is only limited temporarily, and no data has been compromised. In this project, Denial of Service attacks are not considered further.

The remainder of this section will discuss the most common attacks conducted against web applications [5].

2.2.1 SQL Injection

SQL injections is an attack on the web application with the objective of conducting malicious actions on the database. These can include extracting data, bypassing logins, modifying the database (delete/update/insert), create users and shutting down the database. SQL injections can be quite severe, as the damage it can cause is only limited by the attackers skill and imagination.

An example of a vulnerable piece of PHP code can be seen in Listing 2.1. This code will work perfectly fine for regular users, but a malicious user might exploit it to bypass the login. As such, if the malicious attacker inputs `´ OR ´1´=´1´--` as the password, then the SQL statement will look like in Listing 2.2, and as such the user will be permitted to login without supplying a valid username or password. This is the case since the first predicate will fail, but 1=1 will always evaluate to true. Often this will allow the attacker to login as the site administrator, as it is often the lowest ID in the database, and as such it is that row which will be returned.

```
1 $result = mysqli_query("SELECT * FROM users WHERE username=´" . ↩
      $_POST['user']."´ AND password=´" . $_POST['pass'] . "´");
2 if ($result->num_rows != 0) {
3   //Logged in
4 }
```

Listing 2.1: Vulnerable login method

```
1 SELECT * FROM users WHERE username=´´ AND password=´´ OR ´1´=´1´↩
      --´
```

Listing 2.2: Example of exploiting Listing 2.1

Preventing SQL injections, such as Listing 2.1 can be done in two ways using PHP. The first way is illustrated in Listing 2.3, and this makes use of the build-in method `mysqli_real_escape_string`, which will sanitise the input which is dangerous to the database. The second way to avoid SQL injection is to use prepared statements, illustrated in Listing 2.4. In the example, each named parameter is replaced with the value sent through the request. Such that the named parameter `:user` receives the value of the value from `$_POST['user']`, and the password get the corresponding password value. Using prepared statements, the database server will treat every input sent after the binding as data, and as such avoid any potential injections.

There are cases where prepared statements cannot be used. They are limited in that the database server calculates the query plan when the statement is prepared, and as such it is not possible to use prepared statements to con-

duct joins, in selects, among others. In this case the `mysqli_real_escape_string` is the only solution.

In both examples the variable `$con` represents the connection to the database.

```
$username = mysqli_real_escape_string($con, $_POST['user'];
$password = mysqli_real_escape_string($con, $_POST['pass'];
$result = mysqli_query("SELECT * FROM users WHERE username='" .
    $username."' AND password='" . $password . "'");
if ($result->num_rows != 0) {
  //Logged in
}
```

Listing 2.3: Example of escaping the query from Listing 2.1

```
$stmt = $con->prepare('SELECT * FROM users WHERE username=:user
    AND password=:pass');
$stmt->execute(array(
  'user'=>$_POST['user'],
  'pass'=>$_POST['pass']
  );
if ($stmt->num_rows != 0)
  //Logged in
```

Listing 2.4: Prepared statement for escaping Listing 2.1

A limitation to protect against SQL Injections in PHP is that the `mysqli_query` can execute a single query on the database, as such it protects against attacks where the attacker can use an arbitrary query. There is, however, times where it is desirable to use multiple queries, and for this the `mysqli_multi_query` can be used, and an application using supporting multiple queries can be seen in Listing 2.5. If a request is sent containing `'; DROP TABLE admin;` as the username, then the admin table will be dropped, as seen from the executed query in Listing 2.6.

```
$result = mysqli->multi_query("SELECT * FROM users WHERE
    username='" . $username."' AND password='" . $password . "'"
    );
```

Listing 2.5: Example of multiple SQL statements

```
SELECT * FROM users WHERE username=''; DROP TABLE admin;--AND
    passwords=''
```

Listing 2.6: Multiple SQL query injection

Protecting against the conduction of multiple queries is done by sanitising the input in the same way as for single queries. However, the usage of multiple queries should be done with case, as it is possible to split the query in most cases.

2.2.2 Cross Site Scripting

Cross-Site Scripting (XSS) is a class of injection attack used to inject malicious scripts into a web application. These scripts will be executed at the client, and can access cookies, session information and other sensitive information. In addition to extracting information, the scripts can also modify the page the client is visiting, for example to display malicious content.

In the worst case, an XSS attack can be used to hijack a legitimate user's session, install trojans on the client computer, use the client to perform unwanted actions, such as changing the user's password or transmitting sensitive information back to the attacker.

There are several different categories for Cross Site Scripting attacks, but generally categorised into two categories; stored and reflected. There is a third category called DOM based XSS, which is a form of XSS where the entire data flow from source to sink occurs only at the client [15]. This means, that the server's response never contains the attackers script in any form.

An example can be seen in Listing 2.7, where the legitimate use is to write the name of the user. However, sending the request `?user=<script>alert(document.cookie)</script>`, would instead be executed and display the cookie to the user. Note that the server never transmits the script, or even knows of the scripts existence. Instead of displaying the cookie to the user, a more malicious action could be to transmit it to the attacker, so that he/she can hijack the session, for example.

```
1 <html>
2 <title>Logged in</title>
3 ...
4 Welcome
5 <script>
6 var pos=document.URL.indexOf("user=")+5;
7 document.write(document.URL.substring(pos,document.URL.length));
8 </script>
9 ...
```

```
10 </html>
```

Listing 2.7: Vulnerable DOM

Stored XSS

Stored XSS is persistently stored on the web application, such as in a database, and then delivered to the client when the specific page is requested. This could for example be a user requesting the comments in a forum, where a script could be injected.

An example can be seen in Listing 2.8. The vulnerability is in line 4 and line 7 where the input is first stored in the database, and then printed back to a client without any sanitisation. For example by sending the following POST request `?key=12&vuln=<script>alert("Alert!")</script>`. This harmless example will open a dialogue box writing "Alert!", but could contain an arbitrary script. Had this been in a comment thread, the malicious attacker could have used the scripts to extract sensitive information from all users reading the comment.

```
1  <?php
2  ...
3  if (isset($_GET['vuln'])) {
4    storeInDB($_GET['key'], $_GET['vuln']);
5  }
6  else {
7    echo getFromDB($_GET['key']);
8  }
9  ...
10 ?>
```

Listing 2.8: Stored XSS

Reflected XSS

Reflected XSS is where the injected script is reflected off the web server itself, such as in search results or in the URL. The difference is that the script is not stored on the server, and as such is non-persistent.

An example can be seen in Listing 2.9. Here, if `?name=<script>alert("1")</script>` is appended to the URL, then the server will deliver the script back to the client, and as such it will be executed.

```
1  <?php
2  ...
```

```
3 echo $_GET['name'];
4 ...
5 ?>
```

Listing 2.9: Reflected XSS vulnerability

A more malicious example can be seen in Figure 2.1. In this example, the user will log in to a site, after which the malicious attacker will send a crafted URL to the user. When this URL is pressed, the server will deliver the malicious script to the client. This script will then be executed on the client, and could for example be used to send the session token to the attacker, which the attacker can use to hijack the user's session, and as such appear to be the legitimate user.

This attack could also be conducted in the stored context, in which the user will request the page with the malicious script, as opposed to requesting the attacker's URL.

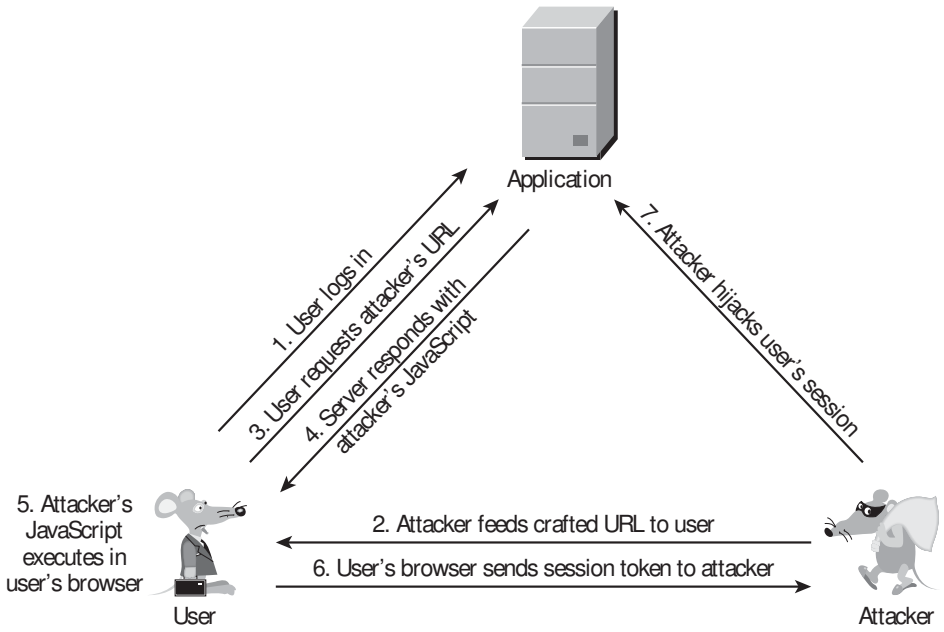

Figure 2.1: Reflected XSS attack [16, p. 436]

Preventing XSS

Preventing XSS is done by sanitising all input before outputting it back to the client. In PHP, the `htmlentities` or `htmlspecialchars` can be used to sanitise the input before delivering it to the client. A method using `htmlspecialchars` has been proposed by the OWASP project [4], and can

be seen in Listing 2.10. The method will escape all HTML special characters, such that the script `<script>alert("Alert!")</script>` will be replaced with `&lt;script&gt;alert(&quot;Alert!&quot;)&lt;/script&gt;`. This script is harmless, and the browser will interpret it as HTML special characters, thus avoiding Cross Site Scripting vulnerabilities. It would, however, be better to use a HTML framework which automatically escaped all data sent to the client, as a single missed escape could lead to the whole application being compromised.

```php
<?php
//xss mitigation functions
function xssafe($data,$encoding='UTF-8')
{
   return htmlspecialchars($data,ENT_QUOTES | ENT_HTML401,
       $encoding);
}
function xecho($data)
{
   echo xssafe($data);
}
?>
```

Listing 2.10: Preventing XSS [4]

2.2.3 Cross Site Request Forgery

A Cross-Site Request Forgery (CSRF) is an attack, where a client is forced to execute unwanted actions on web applications where the client is currently authenticated. This will exploit the trust the website has to the user. Since the browser automatically transfer the credentials associated with the web application, such as the IP address and session cookie, the application will be unable to detect that the request is forged and not legitimate.

As such, CSRF attacks are used to conduct actions as an authenticated user, as opposed to the theft of information. Examples include changing the client's password, sending E-mails, purchasing items or transferring funds. An example of a CSRF attack can be seen in Listing 2.11.

```
Content on malicious site
<img src="https://bigbank.com/transfer.php?account=153569&amount
    =100000" width="0" height="0" border="0">
```

Listing 2.11: Cross Site Request Forgery example

In the example, an image is embedded on a malicious website (or a benign, using an XSS vulnerability), and this image will be loaded whenever a client visits the web page. This image will make a request to transfer funds from one bank account to another, and even though the image will be invalid the action will be performed, often without the client's knowledge. This attack is possible only if the user previously logged in on the bank web site, and has an active session. The user will then have an active authentication cookie, which the client's browser will transfer to the bank automatically.

This is possible through other attack vectors other than simply adding an image to a site. A malicious newsletter could be delivered, for example where the clicking on a link will perform an unintended action.

Detecting CSRF is difficult in practice, and there is no way to mitigate CSRF automatically in PHP. However, the OWASP project does propose the OWASP PHP CSRFGuard [3] for prevention of CSRF, but it has not been verified by professionals.

The theory behind mitigation of CSRF is simple

- Every request that does anything should be CSRF mitigated.
- While CSRF mostly happens on GET requests, it will as easily happen on POST requests, so secure both.
- Re-authenticate for sensitive operations (change mail, transfer funds).
- If an operation might be CSRF vulnerable, a CAPTHA could be added to provide protection.
- Add CSRF tokens for added security.

Not all of the above will be practical to add, for example adding a CAPTCHA to every sensitive operation will greatly inconvenience the end user. Also, detecting which operations *need* to be CSRF mitigated can be difficult, and it could be desirable but impractical to secure any and all operations.

Using unique one-time tokens can improve the security, and the PHP CSRFGuard does that. One function creates an unique token, which once a client connects can have three states; either the session is inactive, in which there is no CSRF risk, the token was found but not the same/not found, and as such the token validation fails, finally the token can be found and match the checked token, which will lead to a successful validation.

This is, however, not yet mature, and as such should be used appropriately.

The above example illustrates the dangers of CSRF attacks, along with the difficulty of detecting the attack. This can make it very dangerous, especially if coupled with other kinds of vulnerabilities, for example if a large payment solution, such as Google Wallet or PayPal was vulnerable to CSRF attack, and a website with a huge amount of traffic were vulnerable to a XSS injection, then the visitors to the site could transfer huge amounts of funds without knowing.

2.2.4 Code Injection

Code injection attacks happen when an attacker is able to inject arbitrary code into a web application. This code is then executed/interpreted by the application, and the malicious attacker is only limited by what the programming language is able to do. The worst case of this attack is a complete compromise of the web application, and unrestricted access to the underlying system. It can be used to get access to documents which is only accessible to logged in users, or protected by other means. Simply by asking the system to supply the document, that is, the disclosure of sensitive information.

In PHP, there is a number of ways to perform this attack;

- Using a dynamic evaluation function, such as `eval()`, allows the execution of an arbitrary string as code. To exploit this as a vulnerability the web application need to not sanitise the supplied user input before interpreting the code.
- Uploading a file without checking the file extension can make it vulnerable if the malicious attacker is able to execute the uploaded script, for example from a public location on the web server.
- PHP allows dynamic inclusion of source files, so if the user is dynamically choosing which files is executed, for example as query parameters, then a malicious user might be able to include a malicious file. In PHP it might even be possible to include files from other domains.

An example of a piece of code vulnerable to Code Injection can be seen in Listing 2.12. This example makes use of the `eval()` function in PHP to interpret any code simply by requesting `file.php?sort=echo "Hello World!";`. This harmless example will simply write back ”Hello World!”, but it could also have been used for more malicious purposes only limited by

the programming language. A legitimate use case for the below snippet of code could be if it was possible for the user to supply a sorting algorithm, for example from search results, but the developer did not sanitise the input before interpreting it using `eval()`.

If the host is a running on a *NIX platform, the `eval()` function might be used for a command injection attack, in which Operating System commands are executed. One such attack can make use of the PHP functions, `passthru()`, `system()`or `exec()`, for executing external programs. Of the three functions, the `passthru()` function displays the raw output when executed. This attack could for example be used for information retrieval, such as requesting `file.php?sort=passthru('cat /etc/passwd');` to get the list of all users.

```
1 <?php
2   eval($_GET['sort']);
3 ?>
```

Listing 2.12: File vulnerable to Code Injection attacks

The file upload can be exploited, for example in the case of an image uploading facility. If the image file extension is not checked or properly sanitised, then a malicious attacker might be able to upload a malicious script and use it for malicious purposes. This attack is possible because the application blindly trust the clients, but this does not automatically make the web application vulnerable to Code Injection. If the scripts are never uploaded to a context in which it can be executed, then there will be no vulnerability, for example if the malicious script gets stored in a database or in a folder which is not publicly available.

File inclusion can in some cases be exploited to read arbitrary files at the web host. An application making use of client supplied parameters in the query string for the selection of pages, for example in a Model-View-Controller pattern might desire dynamic inclusion of files, so that every file need not be statically coded. An example of a vulnerable piece of code can be seen in Listing 2.13.

```
1 <?php
2   ...
3   require "$_GET['view']" . '.php';
4   ...
5 ?>
```

Listing 2.13: File vulnerable to File Inclusion

In the above example, if the attacker submits the query `?view=/etc/passwd%00`, where `%00` is the NULL special character, then it might be possible to get the list of users on the host. This attack is, however, dependent on the host, and in some cases it is possible to instruct the web browser to disallow the inclusion of files not in the web root.

Preventing Code Injection can be done most simply by disallowing the usage of dynamic evaluation functions, such as `eval()`, disallowing dynamic include of PHP files, such that included PHP files are not directly supplied by the client. An example could be of using a `switch` to select the appropriate files such as in Listing 2.14. In the cases where dynamic inclusion is required the input should be properly sanitised.

```
<?php
  ...
  switch($_GET['view']) {
    case 'login':
      require 'login.php';
      break;
    ...
    default:
      require 'frontpage.php';
      break;
  }
  ...
?>
```

Listing 2.14: PHP switch case

For avoiding uploading of malicious files, for example images, the files should be properly sanitised. One example of sanitisation is forcing the extension to be of a certain type. This approach does not, however, ensure that malicious files cannot be uploaded, just that they cannot be executed directly. This can, be circumvented in the case where there is a vulnerability where the include function can be used.

2.2.5 HTTP Header Injection

Header Injection attacks can be used for session hijacking, but it can also be used to redirect a user to a malicious site controlled by the attacker, for example by making an exact mirror of an online banking application.

Much like Cross Site Scripting attacks, the problem with Header Injection attacks is that the web application trust the client to supply valid and not

malicious input. An example of a vulnerable piece of code can be seen in Listing 2.15.

```
1 <?php
2   header(Set-Cookie: ' . $_GET['name']);
3 ?>
```

Listing 2.15: Header Injection vulnerability

In the above example, if the request containing `?name=user\nLocation: myevilbank.com` is sent, then the client will be redirected to a site controlled by a malicious attacker as opposed to the legitimate site. This attack could trick the legitimate user into transfering funds to a different account, or be used to get the credentials for the users bank account.

Preventing Header Injections is done by properly sanitising the client input using the `urlencode()` function. This function will ensure that the characters are properly escaped, thus preventing the manipulation of the HTTP header.

3 Problem

The problem that this project will try to solve is how to create secure web applications. As described in Chapter 2 this is a huge challenge, with no apparent easy solution. Plenty of work has been devoted to statically detecting vulnerabilities, often with varying results. However, code bases is getting increasingly larger, and as such harder to analyse, and in some cases even impossible.

In addition, a lot of security vulnerabilities could be prevented if there was a secure core available, such as for example a Content Management System (CMS) such as Wordpress. This is the case since most vulnerabilities found in CMS systems are not in the core of the systems, but in the plugins which is developed for the system. Not only that, but often the plugin developers are slower to fix the mistakes which are present in the plugins, as opposed to correcting the core [13]. Additionally, CMS systems often supply functions for the sanitisation of input, and encourage the usage of these for the sanitisation of data.

The relative low amount of vulnerabilities in CMS cores, and the speed in which they are found and corrected, leads to the following assumptions;

- We can assume the core of a CMS system to be secure.
- The functions which the CMS system supplies for sanitisation correctly sanitises the input.

However, if there exist a plugin which is vulnerable, then how is it known that the core will remain secure? One approach could be to introduce a sandbox, in which the plugins are running. Another approach could be to

only allow the plugins to use certain functions, and then use a runtime checker to validate whether or not a given plugin is allowed to do the action it wishes.

An approach which does the following is the Google Native Client (NaCl) [1]. NaCl compiles C/C++ source code to an executable format that can be executed directly in the Google Chrome browser using the NaCl runtime component. However, to increase security, NaCl only allows the access to system resources, such as files, through whitelisted API's. This ensures that each individual application is operating in an isolated environment. This validation is enforced by statically analysing the code before execution, such that unsafe code is not executed. However, a limitation of NaCl is that it currently only runs in Google Chrome and Chromium.

Another approach could be to limit the features of the PHP language to contain only secure constructs, or even force the developer to declare which security policies a given plugin is allowed to use, such aswhat the Paragon programming language requires [12]. Paragon follows *secure by design*, which means that programs written in Paragon are secure relative to the policies they adhere to. The policies can cover for example file access, such that a program is only allowed to access the files it has permission to. In the context of plugin management, this could mean that a given plugin were only allowed to access files that it had created itself.

As such, the objective of this project is to explore the following;

- How can consequences of a vulnerable plugin be mitigated?
- How can some security level in a plugin be guaranteed?

4 Proposed Solutions

When a secure core is assumed to exist, how can it be guaranteed that the different plugins does not interfere with each other? This is proposed partly because developers will need to manually use the code analysis tools. However, many web application developers does not think about security, and as such might not even be aware of their own shortcomings. It is hard to patch code for SQL injections, if a developer does not know what it is, never mind the more complex vulnerabilities.

A proposal done in this project will be how to create secure code, and how to validate the code as being secure. A proposal is by constructing a number of *rules* that each plugin will need to honor to be validated. One such rule, for example to be declared at the highest security level could be to not make use of any dynamic constructs, as these are known to cause vulnerabilities, thus disallowing the usage of `eval()` and dynamic includes.

To get different strictness of the security rules, a tiered architecture (illustrated in Figure 4.1) could be used. The idea behind this is that a plugin will get a security level in accordance to which rules it adheres to. The more rules it adheres to, the lower in the tier. Take for example a banking plugin, where the bank has a secure core it wishes to expand with additional functionality. The bank wishes for the plugin to be extremely secure as it works with very sensitive data.

In a different context, a developer might be making a plugin for Wordpress for E-commence. The developer wishes to market it as highly secure, but since there is currently no proof of any security, the developer will be unable to prove any kind of security. However, given a tiered architecture, the developer will be able to place lower in the tier to signify that the plugin

adheres to stricter security rules. In this way, the users of the plugin will know that the plugin at least adheres to the specified rules.

These rules for the lowest tier (which signifies the most security) could be very strict, and for example only allow a subset of the PHP language, with all potentially dangerous constructs disallowed, such as for example dynamic inclusion, evaluation, home-made sanitisation functions etc. The developer might for example also be limited to prepared statements only, and all output and input must go through the sanitisation functions provided by the core.

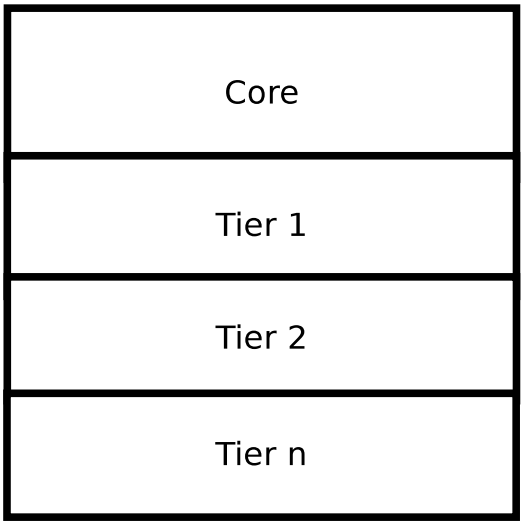

Figure 4.1: Tier architecture

Another consideration will be how to validate the security of a plugin, if it makes use of components from another plugin? Can this new plugin then get the same security? For example with the tiered architecture above, how will the security be validated for such a plugin? One solution might be to choose the plugin to be on the same tier. This will have the benefit that plugins can extend each other without compromising the validated security. However, this can have the impact that the plugins might interweave so strongly that analysing them reliably becomes impossible because of the increased complexity.

4.1 THAPS

THAPS [14] is a vulnerability scanner based on symbolic execution, used to find vulnerabilities in PHP web applications. The symbolic execution is

used to model all execution paths in a given web application. This execution path is then analysed to detect where client input can reach a critical point in the application. Every time data reaches a critical point without being properly sanitised, THAPS will generate a vulnerability report. The authors state that the usage of symbolic execution will reduce the number of false positives, and the conducted experiments confirm this statement.

THAPS takes the dynamic features of PHP into account by extending the Zend Engine, which is used to execute PHP code on the host. This gives THAPS a higher code coverage, as it is possible to analyse the dynamic part of web applications. In addition, the dynamic analysis can reduce the number of branches which is analysed by only analysing what is actually included in a particular execution run.

To conduct this dynamic analysis, THAPS crawls the page to find the links which will then be analysed. This crawl is used to analyse the specific execution which happens when the web application is visited. The analysis itself is a two-step process; first all links are extracted, and then followed. The second phase tests the file directly in the web directory. This means that the first approach tests if the web application is secure when browsed as the developer intented. The second phase tests if the application is secure when unintended use is conducted on the web application.

The interaction between the dynamic and static analysis parts can be seen in Figure 4.2.

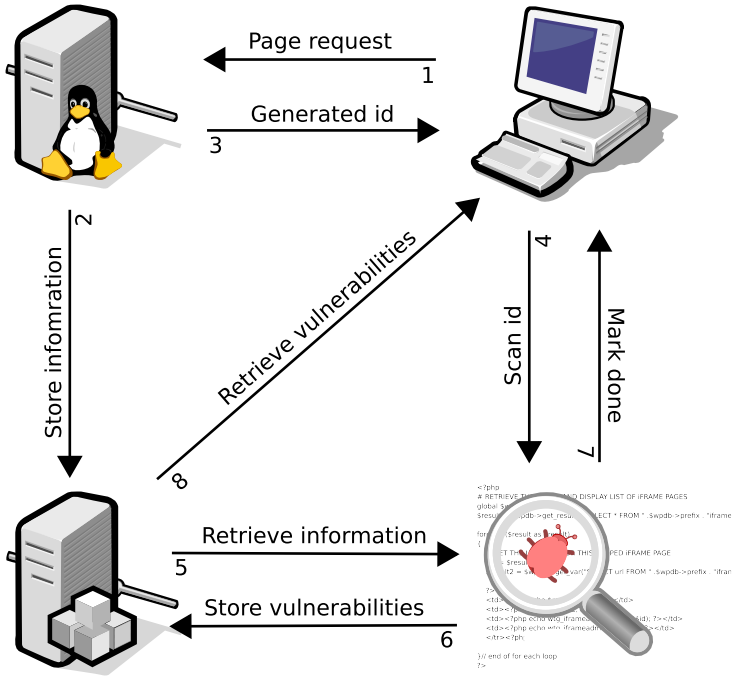

Figure 4.2: THAPS architecture [14]

4.1.1 THAPS for plugins

Analysing large code bases can take a very long time, and contain a lot of branches; far more than what static analysis can handle. A CMS system such as Wordpress is huge. Version 3.2.1 of the Wordpress core consists of 13.918 build-in and six user-defined functions, just for rendering the front page. Such a huge code base can be difficult, if not impossible, to analyse. As such, the authors of THAPS assume that there exist a secure core, which is also assumed to be the case in this project.

Given that there exist a secure core, developers will only have to check their own work, that is, analyse the plugins they develop for the CMS systems. This can significantly reduce the time and complexity of analysing the web applications.

Plugin developers might use plugins supplied by the framework, and THAPS needs to be aware of this so that it will know that the code is secure. To handle this, THAPS include Framework Models, which they use to identify the functions supplied by the framework, so that THAPS knows that these will properly sanitise data. The development of these models takes time, but the developers state that a model for Wordpress was done in a few

days, but that this is dependent on the coverage, and how complex the system is. In addition, the models can be expanded as required, for example when new functions are used.

This templating approach makes it possible for THAPS to analyse Wordpress plugins, and the authors successfully manage to analyse the 300 most popular Wordpress plugins, along with 75 randomly chosen. The analysis itself can analyse the plugins separately, without analysing the Wordpress code base, thus reducing the time required to analyse the plugins.

The plugins were analysed quite quickly, with only a few ones failing to complete analysing within an hour, this despite the fact that the analysis time is exponential to the size of the analysed application.

In addition, THAPS managed to find security vulnerabilities in the plugins, with a relative low amount of false positives. The developer will, however, have to manually confirm the existence of the vulnerabilities, along with manually fixing the errors.

As such, if THAPS is used to scan the added plugins, it can assist with mitigation of the vulnerabilities in the plugins, but it might be possible to extend THAPS to handle the analysis of plugins to find out which functions it makes use of, and detect if a given plugin is allowed to use this function, and as such find out which security rules it adheres to. This could be used to place the plugins into the tiers.

5 Conclusion

This project describes the difficulty of creating secure web applications, and the core problems often faced by the developers, namely the sanitisation of the client supplied input. PHP and MySQL are chosen as the database and programming language of choice because of the wide availability and accessibility. Different vulnerability types is described; SQL Injection, Cross Site Scripting, Cross Site Request Forgery, Code Injection and HTTP Header injection, and how they can be prevented in the PHP programming language.

The problem of securing web applications is described, namely how it can be done without using the traditional approach of conducting code analysis on the whole application, as this can be too large to analyse. In the end a few assumptions is made; that there exist a secure core which we can rely on to be secure, and that the sanitisation functions provided by the core correctly sanitises the input if used appropirately. Several approaches has been explored for how to more clearly define the problem, both the Google Native Client and the Paragon programming language for securing the application, along with the THAPS vulnerability scanner.

Finally, proposals for how to solve the problem of securing web application plugins, and how a vulnerable plugin can be mitigated is presented. In addition, a proposal for how to ensure a security level in a plugin is explored.

Bibliography

[1] *Google Native Client*, Accessed 7. January, 2015. URL https://developer.chrome.com/native-client/.

[2] *Jelastic, Software Stacks Market Share*, Accessed 7. January, 2015. URL http://blog.jelastic.com/2013/10/10/software-stacks-market-share-september-2013/.

[3] *OWASP PHP CSRF Guard*, Accessed 7. January, 2015. URL https://www.owasp.org/index.php/PHP_CSRF_Guard.

[4] *OWASP PHP Security Cheat Sheet*, Accessed 7. January, 2015. URL https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet#XSS_Cheat_Sheet.

[5] *OWASP project 2013 Top 10*, Accessed 7. January, 2015. URL https://www.owasp.org/index.php/Top_10_2013-Top_10.

[6] *Programming Language Popularity*, Accessed 7. January, 2015. URL http://langpop.com/.

[7] *Scalebase, The State of the Open Source Database Market*, Accessed 7. January, 2015. URL https://www.scalebase.com/the-state-of-the-open-source-database-market-mysql-leads-the-way/.

[8] *Softaculous*, Accessed 7. January, 2015. URL http://softaculous.com/.

[9] *TIOBE Software: Tiobe Index*, Accessed 7. January, 2015. URL http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[10] *Usage Statistics and Market Share of PHP for Websites*, Accessed 7. January, 2015. URL `http://w3techs.com/technologies/details/pl-php/all/all`.

[11] *Wordpress.org*, Accessed 7. January, 2015. URL `https://wordpress.org/`.

[12] Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *Programming Languages and Systems: 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, pages 217–232. Springer, 2013.

[13] IBM. *IBM X-Force 2012 Mid-year Trend and Risk Report*. 2012. URL `http://www-935.ibm.com/services/us/iss/xforce/trendreports/`.

[14] Torben Jensen, Heine Pedersen, Mads Chr. Olesen, and René Rydhof Hansen. Thaps: Automated vulnerability scanning of php applications. In Audun Jøsang and Bengt Carlsson, editors, *Secure IT Systems*, volume 7617 of *Lecture Notes in Computer Science*, pages 31–46. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-34209-7. URL `http://dx.doi.org/10.1007/978-3-642-34210-3_3`.

[15] Amit Klein. Dom based cross site scripting or xss of the third kind. *Web Application Security Consortium, Articles*, 4, 2005.

[16] Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley, 2011.