# Aalborg University

**Department of Computer Science**

**Abstract:**

This report presents an analysis of: Common web application vulnerabilities, a number of techniques to detect vulnerabilities, and tools based on those techniques.
The vulnerabilities are analyzed with regards to their nature, what damage they can cause, and how they can be prevented in web applications. The technique analysis discusses different approaches that can be taken in order to detect vulnerabilities in web applications. Finally 13 tools have been tested and compared against three real web applications and a test application, and the four tools that gave the best results have been tested further to identify their properties.
The result of the analysis is a proposal of a tool that addresses the shortcomings of the analyzed tools. The tool is targeted web application developers to be used during and after development to test the application for common vulnerabilities.

**Title:**
A Study of Web Application Vulnerabilities and Vulnerability Detection Tools

**Topic:**
Security in Web Applications

**Project Period:**
September 1st - January 5th, 2012

**Project Group:** sw901e11
Torben Jensen
Heine Pedersen

**Supervisors:**
René Rydhof Hansen
Mads Christian Olesen

**Number of appendices:** 2

**Total number of pages:** 37

**Number of pages in report:** 30

**Number of reports printed:** 5

# Preface

This report substantiates the result of Software Engineering group sw901e11's 9th semester project at the Department of Computer Science, Aalborg University. The report is documentation for the project made in the period from 1st of September until 5th of January 2012.

The topic of this semester project is "Security in Web Applications" and the goal is to analyze existing vulnerability tools and determine their limitations. The analysis is the basis for a proposal to solve these limitations.

The reader is expected to have understanding of programming corresponding to a student that has completed the 8th semester of Software Engineering.

Unless otherwise noted, this report uses the following conventions:

- Cites and references to sources will be denoted by square brackets containing the initials of the authors surname, and the year of publishing. The references each corresponds to an entry in the bibliography on page 31.
- Abbreviations will be represented in their extended form the first time they appear.
- When a person is mentioned as *he* in the report, it refers to *he/she*.

Throughout the report, the following typographical conventions will be used:

- References to classes, variables and functions in code listings are made in `monospace` font.
- Omitted unrelated code is shown as "..." in the code examples.
- Lines broken down in two are denoted by a ↲.

The code examples in the report are not expected to compile out of context.

Appendices are located at the end of the report.

# Contents

# 1 Introduction

People today are using their computer a lot differently than just five years ago. More and more applications are moving online as web applications replace desktop applications. This has some clear advantages, for both the developers and the users, as distribution of an application and the availability of it depend only on the computer in use being connected to the Internet. The users' data can also be stored on servers connected to the Internet and hence the data will be available anywhere. Additionally the developer, and the supporter, is always aware of which version the user is using and hence support will be easier.

But these advantages do not come without drawbacks. If the host is offline or the user does not have access to the Internet then the service is unavailable and the user will not be able to access his data. The user also has to trust that the web application protects his data with regards to both availability, that his data will not disappear, and confidentiality, that his data is only available to those he specifies.

Especially the last part, protecting the users' data with regard to integrity, is getting more attention because the data is valuable to malicious people. The information does not have to be credit card details, as information about your habits and email addresses has high value as well.

To prevent malicious access to the data, the system on which the web application is executed has to be secure, but the application itself also has to be secure. The security of the application is in the hands of the developer and hence he has to be aware of all the pitfalls and how to avoid them. This can be a difficult task, even in small applications, and to ease the task security testing tools have been created which can check the application for security flaws.

This project studies the testing of security in web applications by discussing techniques that can be used by the tools and some of the vulnerabilities that web applications might suffer from. Additionally a selection of these tools have been tested against three applications, with known security problems and one test application which exposes the flaws in the simplest way, to identify shortcomings of the tools and finally a proposal of a new tool which deals with these shortcomings is made.

The next chapter will discuss some of the terms that the reader is expected to know when reading the rest of this report.

# 2 Prerequisites

To understand the terms used in this report this chapter introduces how the problem domain is specified by us. Additionally the common techniques used for testing are discussed with regard to security testing, and finally the most common web application vulnerabilities are outlined.

A *web application* is an application that runs on a web server and changes output dependent on the input provided by a client. A *client* is usually a web browser, but could also be other types of applications. If a web application is *vulnerable* it means that it during normal operation will not malfunction but when receiving specific input will crash, expose private data, or take other unintended actions.

A *vulnerability* describes a class of the same problems. One of such classes is SQL injections which allow the attacker to change the flow of the SQL queries in the application. Common types of vulnerabilities, including SQL injections and cross site scripting, is described in Section 2.2 along with other vulnerabilities. Common to the vulnerabilities are that they are preventable.

A problem, which is not classified as a vulnerability by us, that a web application might suffer from is *session hijacking*. When a user authenticates with an application the application usually uses a cookie to identify the user, but as cookies often are readable by any user of the computer and rarely are deleted, a unique session id is used instead of the actual credentials which instead is stored on the server along with the session id. A session hijacking attack is when this session id is intercepted by an attacker which then is authenticated to the website as the victim and has access to all his data. This type of problem is not due to improper sanitization and is difficult to prevent completely and hence it can be used as part of an attack.

If an application is vulnerable the cause is most likely in the source code. The developer has to handle user input with care such that it will not be able to change the flow of the application unexpected and open a potential vulnerability. The areas in the code where user input is received is called the *sources* and the points where the input is able to effect the application is called the *sinks*.

To avoid the input from doing any damage it has to be *sanitized* properly. Sanitizing input means to ensure that it does not contain content that can break or change the application's flow in an unexpected manner. It is not trivial to ensure that the input is no longer malicious as different types of attacks exist. With regard to web applications sanitizing the input for SQL injections does not prevent the attacker performing a cross site scripting attack, both attacks are described in Section 2.2.

Because it is difficult to ensure that all input is properly sanitized a technique called *tainting* can be used. Tainting works by keeping track of which variables contain user input, called tainted data, and propagates this tracking during execution. When a sink tries to use tainted data it is possible to take action, i.e. end the execution with an error message preventing a successful attack. To remove the taint tracking the programmer either has to use methods that handle the tainted data or specifically remove the taint tracking. There exist different levels of taint tracking because, as stated earlier, sanitizing for one attack does not protect against others, and these implementations track if the data is tainted for SQL injection or cross site scripting.

When a user accesses a website it is through an *entry point*. Entry points are dictated by the developer and specify where to start execution of the application. However, due to the structure of some languages and frameworks it might be possible to use unintended entry points which might change the behavior of the application and possibly create a vulnerability even though that the application is secure if the intended entry points is used.

To help the developers protect the application against these vulnerabilities, tools have been made to help find the vulnerable parts. These tools we call *security testing tools* and some of these tools will be described in Section 3.2. The purpose of the tools are to identify vulnerable parts of the application under test, but this is not trivial as the tools can only check for known vulnerabilities or patterns that might expose risks. Therefore the tool could potentially not find all vulnerabilities, a so called *false negative*, and give the user a false sense of security. Additionally the tool might not be able to identify that the input has been properly sanitized before use and report a vulnerability which does not exist; a *false positive*. Of course the tools are only considered a help, and any prevented vulnerability is better than none, but they might be considered useless if they report to many false positives and false negatives.

## 2.1 Testing Techniques

Testing applications with regard to security might be forgotten because the functionality is more important, so the tests performed on the application are often targeted towards the functionality. Another factor is that development is rushed by inexperienced management in order to be the first providing a specific service.

Normally when doing tests of a piece of software there are two aspects that influence the structure of the tests. The first is how much information about the internals of the application is available. White box testing means that the source code is available, black box testing where no information is available at all and finally gray box testing where some knowledge about the internals is available, i.e. which algorithms are used. The second aspect is if the application is executed during the tests; *dynamic testing*, or not; *static testing*.

The following sections describe what the two aspects mean to vulnerability testing and how the tests work.

### 2.1.1 White Box Testing

Having access to the source code enables the tester to create tests that are targeted to the application. It is possible to see how data flows through the application and how input can change the control flow to reach other parts of the application. Additionally there exist databases containing descriptions of known vulnerabilities. This information enables the tester to check if the application contains similar vulnerabilities.

### 2.1.2 Black Box Testing

With no information about the internals of the application from the source code, collecting as much information about the application is acquired at other places; the manual, expected input and output, the system the application is executed on, etc. which might help create input that exposes the vulnerabilities. One such piece of information could be in what language the application is written, as the language can have specific vulnerabilities to test for.

Another strategy is to test the environment around the application. The saying: "A chain is not stronger than the weakest link" fits very well as even the best programmers cannot protect the application against privileged users giving away the information to strangers.

A common strategy is to bombard the application with random generated input, a technique called fuzzing, which has shown to be a very effective way of finding vulnerabilities [4].

### 2.1.3 Gray Box Testing

Gray box testing is similar to black box testing, but some information about the internals are available like the data structures or which algorithms are used, and thus the tester is not required to have full access to the source code of the application. This enables the tester to make more specific tests which might not been thought of if it was pure black box testing.

Gray box testing may in some cases include reverse engineering to determine value boundaries or error messages, and if such information becomes available the tester could end up making better choices while doing the tests.

Common to all testing methods are the more information available might make it easier to find the vulnerabilities, but does not imply that they are all found.

### 2.1.4 Dynamic Testing

Testing an application by executing it allows the tester to see what the application actually does on certain input. Other factors from e.g. threaded or timed execution in the application could reveal vulnerabilities not easily found just by reading the source code as this could be very complicated.

Additionally dynamic testing allows finding platform/environment specific vulnerabilities like missing character encoding or numeric representation handling.

### 2.1.5 Static Testing

Static testing of an application is very dependent on the information level. White box level means that it is possible for either the developer or an expert to walk through the code and find vulnerabilities. Tools have been made, e.g. RIPS, CodeSecure, and Yasca, to automate or help this walkthrough of various languages which will generate a list of possible vulnerabilities based on predefined rules.

### 2.1.6 Environment Testing

No information about the internals of the system and not being able to execute it changes the way of testing the application. This means e.g. that the procedures of how users are getting their credentials the first time, and the accessibility to the server hardware should be tested.

The result of environment testing is not directly code related but rather the specification of the system.

## 2.2 Vulnerabilities

Indiscriminate trust in the input data from the users is the origin of vulnerabilities in web applications. Developers of web applications have to be aware of how user input might interfere with the control flow or output of the application. Not doing so enables malicious users to exploit the vulnerability on the page to attack its users.

The focus of this project is to protect the data with regard to confidentiality so attacks such as Denial of Service attacks which only limit the access temporarily will not be considered further.

The following list discloses some of the most common attacks that can be performed and each item will then be described in further details [6].

- SQL injection
- Cross site scripting
- Cross site request forgery
- Header injection
- Code execution attacks

There exists many tools, languages and frameworks for developing web applications, but in this project the focus is on the programming language PHP. This is chosen as PHP is one of the most used programming languages for writing web applications [11, 3] and that we have great experience developing web applications using PHP.

PHP does not, like some languages, provide automatic protection against vulnerabilities and hence, the developer is responsible for checking and sanitizing all input from the user in order to avoid them. Languages like Ruby and ASP.NET with MVC3 has built-in mechanisms that can be used to protect against various vulnerabilities. This gives the advantage that the developer does not have to worry about those specific vulnerabilities, but it may also give problems if the configuration on the production system differs from the testing system in such a way that these mechanisms are disabled. Common to most languages is that there exist frameworks which helps avoid vulnerabilities like SQL injections by changing the way the data is accessed.

Furthermore PHP is a dynamically typed language, so there is no guarantee that variables retain the same type. The attacker could submit strings instead of integers which could result in a SQL injection. Some developers might not be aware of the consequence of dynamically typed languages.

Nearly all web applications utilize a database for storing its data, and we have chosen to use MySQL in our test environment. MySQL is an open source database system which is widely used along with PHP applications. However, other languages and databases provide similar functionality but will not be described in this report. Almost any content management system or framework written in PHP has support for MySQL connections, as it is fairly easy to connect to MySQL databases from within a PHP application. Furthermore most web hosts offers a MySQL database to the customers along with disk space for their web application. Lastly WAMP, LAMP, and MAMP (Windows/Linux/Mac, Apache, MySQL, PHP/Perl/Python) are a popular bundle of software packages that sets up a full working environment of PHP/Perl/Python and MySQL with Apache as the web server on the desired operation system. This makes it easy for developers to start writing web applications with a simple installation phase.

The following sections assume that PHP is used with a MySQL database when discussing ways to prevent the attacks.

### 2.2.1 SQL Injection

A SQL injection is an attack on the application trying to change the statements performed on the database. This way the user input can bypass the limitations of the statement or even change which data that is retrieved. The PHP source code shown in Source Code 2.1 is an example of how an authentication check could contain a SQL injection which allows the attacker to login as any user he wants.

```
if (isset($_POST["username"])) {
   $res = mysql_query("SELECT * FROM users WHERE ↵
       username='".$_POST["username"]."' AND ↵
       password='".$_POST["password"]."'");
   if (mysql_num_rows($res) > 0) {
      ... // Successfully logged in
   }
}
```

**Source Code 2.1:** Example of a SQL injection vulnerability.

This code might seem legit, but if a user inputs `' OR '1'='1` as password the where clause of the SQL statement is changed such the resulting statement would look like Source Code 2.2.

```
SELECT * FROM users WHERE username='' AND password='' OR '1'='1'
```

**Source Code 2.2:** Example of a SQL injection attack.

This statement will return all users one by one as `'1'='1'` will always be true, and often the first user will be the administrator user and the attacker will be logged in as the site administrator as implementations usually uses the first returned row in the result.

In most standard PHP applications MySQL queries does not allow multiple statements in the same query. However, some APIs does allow this, for instance by using mysqli's `multi_-`

`query()` method. This allows the attacker to execute notable other queries where for example tables in the database can be completely erased.

```php
$stmt = $mysqli->multi_query("SELECT * FROM users WHERE username='".$_POST["username"]."' AND password='".$_POST["password"]."'");
```

**Source Code 2.3:** SQL injection where multiple queries are allowed.

Consider the `$_POST["username"]` variable in Source Code 2.3 contains `admin';DROP TABLE users;--`, and the `$_POST["password"]` variable is empty. The final SQL query is shown in Source Code 2.4, which is a semicolon separated list of individual queries. The first query retrieves all information about the *admin* user. The second query drops the *users* table, that is erase all data in the table plus the structure of the table. Lastly the third query begins with `--` which comments out the rest of the query.

```sql
SELECT * FROM users WHERE username='admin';DROP TABLE users;--' and password=''
```

**Source Code 2.4:** Example of how to inject multiple SQL statements.

Preventing SQL injections in PHP can easily be accomplished in one of two ways. The first way is by sanitizing all user input with the `mysql_real_escape()` function which escapes all characters which are troublesome. The second way is to use parametrized statements where the user input is inserted into the query by the database and not the developer and will be sanitized automatically. The code example in Source Code 2.5 shows how this works.

```php
1  if (isset($_POST["username"])) {
2     if ($stmt = $mysqli->prepare("SELECT * FROM users WHERE username=? AND password=?")) {
3        $stmt->bind_param("s", $_POST["username"]);
4        $stmt->bind_param("s", $_POST["password"]);
5        $stmt->execute();
6        if ($stmt->num_rows > 0) {
7           ... // Successfully logged in
8        }
9     }
10 }
```

**Source Code 2.5:** Example of how to prevent SQL injection attacks.

Each question mark is replaced with a parameter which is bound in line 3 and 4 to a specific type, in this example a string, and when all parameters are bound the final query is executed. This ensures that all user inputs are escaped properly and thus prevents the attacker from exploiting the SQL query.
When a SQL injection is exploited the worst case scenario is dependent on the application. Most common is if the data is of any value to the attacker the worst case is that all your data

is acquired by the attacker, but if it is of no value then the worst case is that all data is deleted. Sometimes both would be the worst case scenario. If the system would not expose data but still has a SQL injection vulnerability the worst case might evolve into one of the other vulnerabilities like a cross site scripting vulnerability created by a SQL injection.

### 2.2.2 Cross Site Scripting

Cross Site Scripting (XSS) is an attack against the users of a web application where they unknowingly navigate into an infected page, which makes their data vulnerable for the attacker. The basics of a XSS attack is that malicious JavaScript could be injected into a page which then is executed in the user's web browser when the user visits the page. Typically the JavaScript either sends data, the user's cookie for session hijacking or parts of the visited page, to a site owned by the attacker by appending an invisible image to the page with the data in the URL, or by changing the target of forms on the page such that when the user submits them the data is sent to the attackers site.

An example of how vulnerable code could look like is shown in Source Code 2.6. As seen the problem is simply that user provided data is put into the document without any way to distinguish it from the other parts. The input is not sanitized in any way which makes it possible to exploit the site by injecting malicious code.

```
<body>
...
<?php
echo $_GET["type"];
?>
...
</body>
```

**Source Code 2.6:** Example of a simple cross site scripting vulnerability.

A really simple attack of this flaw is to access the page with the following appended to the URL `?type=%3Cscript%3Ealert(%22vulnerable%22)%3C%2Fscript%3E` which is the same as `?type=<script>alert("vulnerable")</script>` with URL encoding. This quite harmless attack simply creates a popup saying "vulnerable" when visiting the page. A more critical attack could be to retreive a user's cookies for the specific site. For example if a message board is vulnerable to XSS, a regular user, and in this case an attacker, could inject the code shown in Source Code 2.7.

```
<script>
document.write("<img src=\"http://attack.com/?"+ document.cookie ↲
    +"\">");
</script>
```

**Source Code 2.7:** Cross site scripting exploit that sends the user's cookies to the attacker's website.

This JavaScript inserts an image from a website managed by the attacker. The URL for the image includes the cookies available from the current website. The attacker could in some cases use some of the cookies to authenticate on the page as the exploited user.

To prevent XSS attacks the user input has to be sanitized or encoded to prevent the data from being evaluated by the user's browser. In PHP this can be done by using the `htmlentities()` function on user input which converts all dangerous characters into HTML entities which by the browser is interpreted as a text character and not a control character. The usage of the `htmlentities()` function is shown in Source Code 2.8.

```
<body>
...
<?php
$type = $_GET["type"];
echo htmlentities($type);
?>
...
</body>
```

**Source Code 2.8:** Usage of the `htmlentities()` function.

If the `$type` variable contains `<script>` the output of the `htmlentities()` function is `&lt;script&gt;` which the browser interprets as special HTML characters, and not as regular HTML source code, and thus the output that is visible for the user is the text `<script>`.

The worst case scenario for a XSS attack is difficult to specify but it can generally be divided into two groups: The user's session is hijacked or the user is tricked into an unwanted action. If the session is hijacked the attacker is able to act as the victim and perform actions on his behalf. If the user is tricked then the possibilities are practically endless; installing malware or Trojans, showing commercials, or even doing a phishing attack.

### 2.2.3 Cross Site Request Forgery

In combination with XSS, Cross Site Request Forgery (CSRF) exploits the trust a website has to a particular user, where the user unknowingly makes a request to a website that trusts the user. For example the attacker could embed an image URL which is a malicious request to a page that trusts the user. This is illustrated in Source Code 2.9 which could be a XSS exploit on a message board.

```
Sample text in the message board post.
<img src="http://stocksellingcompany.com/buy?stock=google&quantity=100">
```

**Source Code 2.9:** Example of a simple cross site request forgery vulnerability in combination with XSS.

The attacker embeds an image with a request to a stock selling company where it buys 100 Google stocks on behalf of the user. The image on the message board would not be visible though, as the request is not a valid image, but the request still completes. In most cases the user is not aware of the situation and unknowingly bought 100 stocks. The attack is possible as the user previously logged in to the stock company's website and did not log out. The user then has an authentication cookie which automatically authenticates him to the website next time he, even unknowingly, makes a request.

As seen in the example above, the whole idea is that the URL requested on another page is trusted, so XSS vulnerabilities are not the only way to make a CSRF attack. Fake newsletters or spam e-mails is just examples of other approaches to exploit CSRF, as e-mails could embed a direct link to a particular page.

CSRF is not easy to detect and there is no universal solutions to prevent these attacks, however, there are some techniques which should limit the risk. First of all a "Remember me" option should be avoided. This reduces the possibility that the website does not accept arbitrary requests from the user, however, long-lived sessions are still a risk. Secondly adding a unique token to each request could limit the CRSF risk. The token should have a timeout that marks it invalid after a certain amount of time and only be valid once.

The worst case scenario of this attack is if any high-traffic website has been compromised for XSS. If Slashdot is vulnerable for XSS and PayPal is vulnerable for CSRF, the attacker could transfer money to the attacker's account without the user's knowledge and acceptance, thus the users has been robbed without knowing how.

### 2.2.4 HTTP Header Injection

HTTP header injections are classes of attacks which can result in the before mentioned XSS and other types of session hijacking, but it can also be used to redirect a user to a malicious page on another server without his knowledge.

The source of the problem is the same as the one for XSS, trusting user input and inserting it directly in the headers without sanitizing it. Source Code 2.10 shows an example of vulnerable code.

```php
<?php
header("Set-Cookie: type=".$_GET["type"]);
?>
```

**Source Code 2.10:** Example of a header injection vulnerability.

To exploit this vulnerability the attacker has to end the current header line by passing new line characters and then he is able to inject any header line he wants i.e. a location line which redirects the user to the site specified. If the content of the `$_GET["type"]` variable is tainted with the following data: `admin\r\nLocation: http://imaginarybank.com`, then the final header is changed to:

```
Set-Cookie: type=admin
Location:   http://imaginarybank.com
```

which redirects the user to another page, in this example to an imaginary bank's website. This could be an exact clone visually of a real bank, which could trick the user to transfer money to another account than he expects.

Just like the other attacks HTTP header injections can be prevented by sanitizing the input, this time by using the function `urlencode()`. This prevents the attacker to manipulate the header.

The worst case scenario is if the attacker makes an exact copy of the victim's online banking website and the attacker has full control of this fake virtual bank, to which he has been redirected. The attacker could modify the source code to transfer money to his own account for each transfer the victim conducts. It is possible for the attacker to hide behind this website because the victim fully trusts his online banking service, and is not aware of the attacker's interference and thinks he has transferred money to the regular account.

### 2.2.5 Code Execution Attacks

A powerful attack on a web application is a code execution attack. This way the attacker is able to inject code into the application and this way be able to send data to himself without the users being able to detect it. There are in general three methods to perform this type of attack in PHP:

**Reflection** PHP allows dynamic evaluation of code in a string using several methods, one is the `eval()` language structure. If user provided data is used as an argument for these methods the site is vulnerable as there are no restrictions on these methods.

**File Upload** Uploading a file to the web server can be vulnerable as all PHP files are considered executable, so if the file uploaded to a public available location and no checks are performed the attacker might be able to upload his own script and execute it.

**File Inclusion** To avoid having all PHP code in one file, another language structure allows loading files when needed. If the user data is used to determine which files are loaded it might even be possible to include files from another domain or files from outside the application.

Common to these methods are that they are easy to prevent as one could use the user input in control structures that decides which code or files are executed, instead of using it directly as executable code. However, sometimes that is not possible and a strict user authentication and trust to the users are required.

In some web applications users are allowed to upload files, for example an avatar as a profile picture. If the developers blindly trust the files the users are uploading, these files could include PHP code. This is not necessarily a vulnerability if the code is never executed, but it becomes dangerous if the uploaded file is directly available. If the file `attack.php` is uploaded to the website and the file is accessible by for example requesting `http://page.com/uploads/attack.php` the attacker could easily run malicious code on the host. Source Code 2.11 shows the content of `attack.php` where `eval()` is used to run whatever PHP code the attacker requests.

If the attacker requests `attack.php?code=echo "Hello";` the page writes Hello back. In PHP there exists several functions to execute system commands, among these the `passthru()` function, which executes a command and displays the raw output. If the host

```php
<?php
eval($_GET["code"]);
?>
```

**Source Code 2.11:** Content of a file which exploits a code injection attack.

runs on a UNIX like platform and the attacker wants a list of all users on the system, he simply requests `attack.php?code=passthru("cat /etc/passwd");`.

Reading arbitrary files on the host is a critical vulnerability. The file inclusion attack can in some cases do this. Consider a web application which includes other PHP files to switch content, for example a page using `?page=contact.php` to show a page containing contact information. Such code is shown in Source Code 2.12 where the input parameter `page` is not validated. If an attacker rewrites the URL to `?page=/etc/passwd`, a list of users on a UNIX like system is retrieved.

```php
1 <?php
2 ...
3 if (isset($_GET["page"]) {
4    include $_GET["page"];
5 }
6 ...
7 ?>
```

**Source Code 2.12:** File inclusion vulnerability.

Even if the developer limits the inclusion to only allow `.php` files, there might still be a vulnerability. If line 4 is substituted with `include $_GET["page"]."".php"`, the attacker can possibly request with a NULL metacharacter, which would result in this request: `?page=/etc/passwd%00`. This, however, depends on the configuration of the web server. The `include` statement will read characters until the NULL meta character is found, which excludes the need for `.php`.

Code execution attacks can in some cases grant access to a company's or government's intranet through the requested commands. If the intranet is accessible the options for the attacker are practical limitless and can in the worst case reveal sensitive documents.

# 3 Findings

Many tools exists that test PHP applications for vulnerabilities, and each of them have different approaches for accomplishing this task. We have chosen 13 of these tools and tested their ability to find XSS and SQL injection vulnerabilities in four web applications. The reason for choosing XSS and SQL injection is that they are the most common exploited vulnerabilities [6], and common to almost all the tested tools is that they have the functionality to find such vulnerabilities.

In this chapter the tested tools, what their abilities are, and the current state of the tool, are elaborated. The test result is presented to give an overview of how many vulnerabilities that were found during testing, and finally the properties of the most promising are derived.

## 3.1 Test Environment

One of the four web applications is a small custom made application, and the rest are a combination of real world applications. The web applications are:

**TestApp** is a simple script written by ourselves, source code is available in Appendix A. This application contains four obvious vulnerabilities: Three variations of XSS injections and one SQL injection. This gives a baseline of the tools' abilities to find very simple vulnerabilities in non-complex applications, as we know exactly which vulnerabilities the tools should find.

**WordPress** is a widely used blog system with support for third-party plugins. WordPress is an open source application which previously have had vulnerabilities in both the core of the system and in some of the plugins. The tools are tested against an older version, 3.2.1, of WordPress along with a plugin which contains a known SQL injection vulnerability. The vulnerability exists on an older version of the plugin, hence the WordPress installation have to match the version the plugin depends on, however, there is no guarantee that this version does not contain any other vulnerabilities.

**Moodle** is an open source course management system. Many places of study use this system to manage courses and meetings. It is a fairly comprehensive system which according to exploit databases has a smaller amount of known vulnerabilities compared to WordPress, however, the version, 1.6.2, the tools are tested against contains at least one SQL injection vulnerability. Moodle could potentially contain confidential personal information, such as grades, and hence the system should be secure.

**DoubtfulSystem** is a web application developed by a local newly established company. It is developed without the use of any content management system or framework. The application is rather large and is currently under heavy development. This application is analyzed because it is a real world example of developing without security in focus. We know the application contains several vulnerabilities, as the source code was examined and several security holes was found, however, we might not have found all vulnerabilities in the application.

DoubtfulSystem started as a semester project at Aalborg University. The developers saw potential in this project and decided to form a company based on the idea. The focus of their education is to understand software developers and learning to estimate software projects, and not the art of software development itself, hence they only have had basic programming lessons without having any security related topics. The reason the application's real name is not mentioned is that the application already is in production and hence disclosing the vulnerabilities could expose the system unnecessarily. We found the application interesting to examine because we believe that many new companies publish vulnerable web applications without having focus on security. In this case the developers knew that vulnerabilities existed but not how to prevent them and decided not to take any action at first.

## 3.2 Tested Tools

This section elaborates the 13 tested tools. A description of each tool specifies the applicability and the current state. A discussion of the result of each tool is presented, which clarifies what vulnerabilities they were able to find and what vulnerabilities they lack finding.

### 3.2.1 PHP Taint

PHP does not have any taint support in the mainstream version, however, a fork of the development tree offers this functionality [12]. This fork is a preliminary implementation for dynamically checking tainted variables, which only has a 0.5%-1.5% runtime overhead, depending on CPU used. The goal for PHP Taint is to help the developers to find and eliminate vulnerabilities before an attacker can exploit them. This tool is not a vulnerability scanning tool but is more alike an automatic protection against the most widespread attack techniques. Currently there is support for code execution attacks, XSS, and SQL injections. An internal setting in PHP's configuration file allows PHP Taint to either; give a warning and continue executing code, or to stop execution thus preventing the vulnerability to be exploited, when tainted data reaches a sink.

Even though taint checking might protect web applications against exploits, PHP Taint requires modifications to the core of PHP. Many changes have already been made to the core, and the developer proposed this fork to be patched into the mainstream development. The main developers did, however, discard this proposal back in 2006 as it may lead to false sense of security [5]. The last release of PHP Taint was in June 2008 and still got some loose ends, however, it still got potential in the sense of helping the developer to write secure code.

#### Test Results

Even though the implementation is based on an old version of PHP, PHP Taint does a remarkably good job in tracking tainted variables. All tested sinks were denied in all test cases, however, it had some problems with PHP's error control operator. PHP allows expressions to be 'silenced' which mean that any error messages an expression yields is ignored. This is done by prepending the @ sign in front of the expression, which resulted in a blank page and no log information either. Without this information, the taint warning or error message, the usefulness of the tool is affected.

As PHP Taint is no scanning tool, the potential vulnerabilities are not found until the exploit is performed, however, PHP Taint still denies execution of the exploit. Furthermore no false negatives were found during testing but some false positives were found were custom sanitizing was used.

### 3.2.2 Yasca

The development of Yasca started in 2007 by Michael Scovetta and is capable of scanning source code written in several languages, among them PHP. Yasca is primarily written in PHP but make use of external libraries written in other languages. It is a command line tool that statically analyzes the source code for detecting vulnerabilities by using pattern matching. It is easy to extend the tool by writing new regular expression patterns within the plugin directory that resides in the root of Yasca, however, the user of the tool does not necessarily have to write his own expressions as Yasca includes the most common ones.

Moreover, Yasca is capable of taking use of other tools, such as Pixy, by writing plugins that executes the tool in the background and fetches the results. The most notably plugin is called *Grep* which uses external files to scan for certain patterns. One example of these is `Injection.FileInclusion.grep` which checks for file inclusion vulnerabilities. However, some of the vulnerabilities are currently written for Java Server Pages (JSP) rather than PHP, which means not all vulnerability are found in our test cases, though Yasca supports finding XSS and SQL injections.

The last stable version was released 4th of June 2010 and the last source code change was committed 31th of December 2010. No forum posts or bug reports were submitted to the project website since the last commit.

#### Test Results

Yasca found SQL injections in TestApp and in the source of DoubtfulSystem, however, vulnerabilities were found in neither Moodle nor WordPress. These applications might be too complex as Yasca were not tracking any variables.

It found a small amount of false positives, where some of the reported vulnerabilities were found in source code comments, which obviously had no effect on the execution of the web application.

### 3.2.3 Metasploit Pro

As a part of the Metasploit Framework, this professional edition is a black box vulnerability scanner which offers penetration testing. The framework was created by HD Moore in 2003 and he is now Chief Security Officer at Rapid7, which acquired Metasploit in 2009. This resulted in two additional proprietary editions called Metasploit Express and Metasplot Pro.

Metasploit is one of the most used vulnerability exploitation tools and provides large databases with information about newly found exploits, where users is able to download and execute exploits on the target host [7]. Metasploit contains tools to detect system information about the target host, to determine if the host is susceptible for certain exploits. This information can be gathered using OS fingerprinting and port scanning tools such as the well-known UNIX tool *nmap*.

The framework is an open source platform which allows developers to write their own tools or exploits. Currently the framework includes fuzzing tools in order to discover vulnerabilities, instead of just offering exploits to known bugs.

Metasploit Pro includes several forms of interaction, including two command line interfaces (CLI), a web-based interface and finally a native GUI. The web-based interface simply executes CLI commands and writes back the result to the user. This eases the usage of the framework as the CLI can be quite comprehensive to use.

#### Test Results

The tested version was Metasploit Pro which was the only edition that included the web-based interface. It found all XSS vulnerabilities in TestApp but found no SQL injections. Metasploit Pro was not able to detect any vulnerability in the rest of the tested applications.

### 3.2.4 PHP-Sat

This tool is yet another tool for statically analyzing PHP source code. The development of PHP-Sat started at Google's Summer of Code 2006 by PhD Eric Bouwers, because a lot of students were not aware of the security problems involved when programming PHP applications. No development has occurred since November 2009 and never reached a stable release build, however, the tested version was the latest committed SVN revision.

PHP-Sat reads a configuration file with a set of rules of what procedures that are tainted sources and sensitive sinks. Initially the configuration file was only capable of detecting XSS but support for SQL injection detection was added in order to detect these vulnerabilities.

#### Test Results

PHP-Sat found all XSS vulnerabilities in TestApp, and was able to successfully track variables and detect if the input was sanitized. It also found the SQL injection vulnerability, however, it suffered from the same problem as PHP-Taint, where the @ sign can be used to silence the sink. Even though it found the SQL injection vulnerability, it could not detect if the input was tainted as it reported the vulnerability after properly sanitizing the input.

In WordPress it reported no actual vulnerabilities, however, it reported many false positives. Moodle and DoubtfulSystem failed as PHP-Sat did not support complex object-oriented code.

### 3.2.5 PHPIDS

PHPIDS, short for PHP Intrusion Detection System, is an open source tool written in PHP by M. Heiderich, C. Matthies, and L. H. Strojny back in March 2007. It is mainly a security layer for PHP applications which recognizes attacks and reacts when intrusion is detected. PHPIDS is able to detect XSS, SQL injection, HTTP header injection, remote file execution and several other attacking techniques. This tool needs to be initialized before any other PHP code.

There exist modules and plugins to larger content management systems and frameworks. For example the PHPIDS module for Drupal, a popular content management system, adds the security layer to a running Drupal instance. It allows administrators on Drupal to log the attacks and even send mails if an intrusion is detected. Furthermore PHPIDS extensions achieves the same in WordPress and Zend Framework applications.

PHPIDS applies regular expressions to detect known intrusion patterns and because of this some unknown patterns could attack the site without being detected. Furthermore administrators could wrongly assume that their application is secure which leads to a false sense of security.

The developers actively maintain PHPIDS and the latest stable version was released in August 2011.

#### Test Results

As a small amount of PHP code should be injected in order to be protected with PHPIDS which required a small PHP configuration change. It was not possible for us to perform exploits for the vulnerabilities in TestApp, so all XSS and SQL injections were successfully found. The one known vulnerability in Moodle was successfully obstructed. The WordPress plugin was installed in order to protect the web application for vulnerabilities and successfully detected when the vulnerabilities were exploited, but it did not avoid the attack, it only logged the intrusion and mailed the result to the administrators. Though one of the known vulnerabilities in DoubtfulSystem was not hindered.

### 3.2.6 Wapiti

Wapiti is a dynamic black box testing tool that uses partly a database of known exploits and a fuzzer targeted PHP, ASP, and JSP for finding vulnerabilities. The latest update was in January 2010 so the database of known exploits is outdated but the vulnerabilities are still relevant. The tool is developed as a part of the Romulus project which is a project trying to improve the web software development and increase its quality.

The tool works by attacking the site with different requests and based on the response it determines if the attack was successful. It is possible to provide an authentication cookie and specify that the tool should not request the log out functionality.

#### Test Results

The tool was able to find both the SQL injection and the XSS vulnerabilities in TestApp but when tested on WordPress it found no vulnerabilities, but returned many false positives because WordPress returns a HTTP 500 response when faced with the wrong input. The tool did not find anything on Moodle and did not return any false positives either. The test of DoubtfulSystem returned a lot of problems which all was different versions of the same problem: A vulnerability allowing the attacker to send emails to whomever he wants with content he specifies. Wapiti did, however, not find any of the other known vulnerabilities in DoubtfulSystem.

### 3.2.7 RIPS

RIPS is a static white box analysis tool for PHP, which is able to find vulnerabilities in PHP source code. It was released during the month of PHP security in 2010 but has been developed since regularly [10].

The tool uses the internal tokenizer of PHP and analyzes the specified source files for vulnerabilities by detecting if untrusted data reaches a sink. It has different settings which specify what data is trusted and which is not, and if sanitizing should result in trusting the data. It generates results that show the vulnerable sink and the lines of code that the data flows through as well as the conditions which has to be true in order to reach the sink. The tool has some missing functionality with regard to PHP functionality. Object-oriented support has not been implemented and if the application is dynamically loading code RIPS can have trouble determining which files to include.

#### Test Results

RIPS found all the vulnerabilities in TestApp, but did not find any in Moodle. RIPS did find one XSS vulnerability in WordPress allowing the attacker to redirect a user to any site of his choice. In DoubtfulSystem RIPS found many of the known bugs but not all. The shortcomings with regard to WordPress, Moodle and DoubtfulSystem might be due to the missing functionality of RIPS as all the applications are using those.

### 3.2.8 CodeSecure

CodeSecure, earlier known as WebSSARI, is a commercial tool that does static white box testing of the application under test. The tool is able to scan the source code of the application either by doing a checkout from a versioning system or packaged in a file and automatically scans it when required.

CodeSecure has a public available version that allows for analyzing a maximum of 10.000 lines of code and it is possible to request a trial version which provides a temporary full version with unlimited lines of code. We did, however, not get an approval for a trial version, but decided to test TestApp with the tool anyway because some scientific papers are based on the work from WebSSARI [2, 13].

#### Test Results

Due to the license limitation of CodeSecure, it was only able to scan TestApp and DoubtfulSystem. It found all the vulnerabilities in TestApp, but found nothing in DoubtfulSystem.

### 3.2.9 Secubat

Secubat is a dynamic black box testing tool to identify SQL injection and XSS vulnerabilities on webpages. It was developed as a proof of concept in 2006 by S. Kals, E. Kirda, C. Kruegal, and N. Jovanovic from Secure Systems lab, Technical University of Vienna but has been developed until January 2010.

The tool works by using different plugins which are targeted specific types of vulnerabilities. The current version does, however, consist of plugins with simple databases of vulnerabilities which is used to exploit the forms on the website.

#### Test Results

The result from scanning TestApp resulted in six reports, but they were in fact the same vulnerability detected several times. On the other applications Secubat did not find any vulnerability but did not report any false positives either.

### 3.2.10 Pixy

Pixy is a research project by N. Jovanovic, C. Kruegel, and E. Kirda, whom also made Secubat, and is a static white box tool. The last update to Pixy was made in July 2005.

The tool detects SQL and XSS vulnerabilities in PHP source code by determining if user input reaches a sink without sanitization. The result is the type of the problem along with the location of the sink, if the problem is specific or unconditional and graphs showing where the vulnerability is.

#### Test Results

Pixy did find all the vulnerabilities in TestApp but due to the fact that Pixy has not been updated it does not support object-oriented code which made it unable to test neither of the bigger applications.

Additionally it is worth mentioning that Pixy does have problems with character encoding of the files and one of the tested web applications needed to be converted before Pixy eventually had to give up on the object-oriented code.

### 3.2.11 N-Stalker

N-Stalker is a dynamic black box commercial tool able to detect several different vulnerabilities like XSS, SQL injection, and code injection. It is also able to detect AJAX functionality in JavaScript and test the targets of this as well. The tool is regularly updated and uses profiles, i.e. a XSS profile and a Webserver Infrastructure profile, covering different types of security checks. Additionally N-Stalker has functionality to record how to log into the site and hence allow the tool to test password protected pages also.

The free version of N-Stalker only provides the Full XSS Assessment profile with relevance to our testing and as they did not provide us with an evaluation license, SQL injections are not found by this tool.

#### Test Results

N-Stalker found one XSS vulnerability in TestApp.

### 3.2.12 Acunetix

Acunetix Web Vulnerability Scanner is another commercial tool for dynamic black box testing of a website. It is able to detect XSS, SQL injections, and other vulnerabilities along with generating regulatory compliance reports. Both a database and a fuzzer are used to identify the vulnerabilities. The tool is regularly updated and the version used was from October 2011.

The tool does, like the other commercial tools, not provide full functionality in the free version, as it is limited to XSS vulnerabilities, and like the others they did not provide a trial license.

#### Test Results

Acunetix did find the XSS vulnerabilities in TestApp, but it did not find anything in any of the other applications.

### 3.2.13 Skipfish

Skipfish is a dynamic black box tool which has been developed by Michal Zalewski from Google. It is able to detect many web related vulnerabilities including SQL injections and XSS. The tool is continuously developed with the last release in August 2011.

The tool works by generating a site map by crawling the website and a database of common used paths and identifies possible vulnerable parts of each page. Subsequently it attacks all the parts in different ways and generates a report which contains all the relevant information about the potential vulnerability, including the response of the attack.

#### Test Results

The results of the test were that Skipfish found both the SQL injection and the XSS vulnerabilities in TestApp. The test of the other applications did not return actual vulnerabilities but just false positives which were easily ruled out.

## 3.3 Joined Results

The results for all tested tools are clarified in Table 3.1, to gain an overview of the tools.

The results from the tests have been used to select the tools that generated the most useful results: PHP Taint, CodeSecure, Wapiti, and RIPS. Even though Yasca had good result in DoubtfulSystem it did not find any XSS vulnerabilities in any of the applications, and failed on the TestApp as the only tool, it was left out. The results from these tools with regards to WordPress and Moodle will be covered in the following section and the result of DoubtfulSystem in the section after that.

| | | **TestApp** | | | **WordPress** | | | **Moodle** | | | **DoubtfulSystem** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **FV** | **VV** | **FP** | **FV** | **VV** | **FP** | **FV** | **VV** | **FP** | **FV** | **VV** | **FP** |
| D | PHP Taint | 4 | 4 | 0 | 1① | 1① | 0① | 1① | 1① | 0① | 1① | 1① | 0① |
| S | Yasca | 0 | 0 | 0 | 3 | 0 | 3 | 3 | 0 | 3 | 11 | 10 | 1 |
| D | Metasploit Pro | 2 | 2 | 0 | | ② | | 0 | 0 | 0 | 0 | 0 | 0 |
| S | PHP-Sat | 6 | 3 | 3 | 3 | 0 | 3 | 10 | 0 | 10 | | ② | |
| D | PHPIDS | 4 | 4 | 0 | 1① | 1① | 0① | 1① | 1① | 0① | 0① | 0① | 0① |
| D | Wapiti | 6 | 4 | 2 | 27 | 0 | 27 | 0 | 0 | 0 | 67 | 15 **?** | 0 **?** |
| S | RIPS | 4 | 4 | 0 | 144 | 1 **?** | 14 **?** | 21 | 0 **?** | 3 **?** | 79 | 10 **?** | 5 **?** |
| S | CodeSecure | 4 | 4 | 0 | | ③ | | | ③ | | 0 | 0 | 0 |
| D | Secubat | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | Pixy | 4 | 4 | 0 | | ② | | | ② | | | ② | |
| D | N-Stalker | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D | Acunetix | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| D | Skipfish | 2 | 2 | 0 | 0 | 0 | 0 | 5 | 0 | 5 | 0 | 0 | 0 |

**FV** = Found vulnerabilities.
**VV** = Verified vulnerabilties.
**FP** = False positives.
**D** = Dynamic tool.
**S** = Static tool.
**?**= Partial result of 15 entries (if available).
①= Was not tested thorough, see the test description in Section 3.3.
②= Failed to scan application.
③= Not tested due to missing license.

**Table 3.1:** Summary of all tested tools.

## 3.3.1 Moodle and WordPress

None of the tested tools, besides PHP Taint, found the vulnerabilities in either Moodle or WordPress. The vulnerability in Moodle was in the blog module of the system, where an SQL injection allowed exposing the password hash of an admin user [8]. In WordPress the actual vulnerability was in the SCORM Cloud plugin and not the core of WordPress [9]. The vulnerability was a SQL injection in some Ajax functionality which could be used to execute arbitrary SQL statements on the database.

The reasons that the tools did not find the vulnerability in Moodle were two things: The dynamic tools were not aware that there were a tag parameter and hence they did not try to exploit it, and the static tools had trouble handling the object-oriented code and could not track the tainted data through to the vulnerable `mysql_query()` sink.

Almost the same reasons were relevant to the WordPress vulnerability however the dynamic tools did not find the vulnerability as none parsed the JavaScript and identified the `ajax.php` file as an entry point and hence they did not try to exploit it.

### 3.3.2 Test Case Script

Due to the fact that only PHP Taint was able to prevent, and not detect, all vulnerabilities in the DoubtfulSystem's code, which we consider fairly simple compared to both Moodle and WordPress, a test case script was made, see Appendix B. All the tools were tested against this script in order to identify the limitations of the tools with regards to PHP routines. This script contained various vulnerabilities which was used as test cases, and each of these test cases are elaborated below:

**Unknown tainted variable**
Since the black box tools does not know about the source code, this test is basically a trap no black box tools are able to find. The test simply checks if a specific user variable contains data, but there are no references to this variable on the website, thus the test is called *Unknown tainted variable*. The variable is printed back to the user without being sanitized.

**Conditional sanitizing**
This tests sanitizes tainted user input when several conditions are met, but if none of these are met, the user input is not sanitized and is written back to the user with tainted content.

**Wrong escaping for XSS**
Chapter 2 states that tainted user input should be proper sanitized before used. In this test the `escapeshellcmd()` function is used when printing the user input instead of the proper `htmlentities()` function. This test tries to trick the static white box tools to think that the user input is properly sanitized, as the user data is not used as a shell command.

**Wrong escaping for SQL**
Like the above test, this test escapes one field correctly, but another field is sanitized using the improper `htmlentities()` function which paves the way for a SQL vulnerability.

**Silenced SQL query**
Some tools detect SQL injections by identifying an error message in the response. To get those error messages it submits input that could change the SQL statement such that it becomes invalid. PHP's error control operator can silence warnings and errors, so no messages occur on unexpected behavior. This operator is used in this test.

**Hidden field SQL query**
Forms typically contains several input fields and some of them can be of the type `hidden`, which is not a visible field on the rendered page but is yet another field that gets posted when submitting the form. This field can change value as all other form fields. This test contains a hidden field which is not sanitized after being posted and thus an SQL injection is possible.

**Tainted function returns**

To test the traceability of the white box tools, this test executes a function with tainted input data and returns the same data. The tools should recognize if the input data is sanitized when evaluating the returned result.

**Unreachable code**

This test is made to trick the static white box tools. Unreachable code cannot be executed by dynamic tools, but the static tools might mark it as a vulnerability. This is not actually the case as this is not a vulnerability, however, small modifications in the source code can make the code executable, thus making it a vulnerability.

**Object-oriented code**

As many of all the tested tools failed on object-oriented code, a test was created with objects containing tainted user input. This test verifies if the tools are able to track the tainted variables on object-oriented code.

**Regular expression**

Custom sanitization routines are difficult to evaluate. This test replaces all characters that are not numbers and not in the alphabet with an empty string. Static tools can have difficulties detecting if the output of the regular expression is properly sanitized, whereas dynamic tools can evaluate the output at runtime.

**Sanitizing source**

This final test sanitizes all user input data for XSS before actually using them. All `$_GET` variables are assigned a new sanitized value, which is done in order to trick the static white box tools, as they might not detect that all these variables have been properly sanitized before use.

The reports from these tools have been analyzed to determine the shortcomings of the tool and the technique it is using. These shortcomings along with the common shortcomings of the tools will be discussed in the rest of this chapter. Table 3.2 shows the summary of what vulnerabilities the tools found.

## 3.4 Properties of PHP Taint and Wapiti

Two of the tools that generated the best results were PHP Taint and Wapiti which both are dynamic tools. However, PHP Taint is white box and Wapiti is black box and as it turned out they have different properties.

PHP Taint is able to detect the specific sink that is vulnerable and specify the line of the source code and what type of vulnerability that was found, but this requires that the taint detection works correctly. This means that all the sources and sinks have to be known by the tool, but also that the propagation through the code is correct otherwise false positives or negatives will be generated. PHP Taint fails in the test case script on the regular expression test and identifies it as a vulnerability because it cannot determine if the regular expression properly sanitizes the value, however, PHP Taint provides a `untaint()` function that removes the taint tracking and hence will not report any vulnerabilities.

| | | Unknown tainted variable | Conditional sanitizing | Wrong escaping for XSS | Wrong escaping for SQL | Silenced SQL query | Hidden field SQL query | Tainted function returns | Unreachable code | Object-oriented code | Regular expression | Sanitizing source |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D | PHP Taint | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | | ✓ |
| D | Wapiti | | ✓ | ✓ | ✓ | ✓ | | ✓ | | ✓ | ✓ | ✓ |
| S | CodeSecure | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | |
| S | RIPS | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | |

**D** = Dynamic tool.
**S** = Static tool.

**Table 3.2:** Summary of what vulnerability test cases the tools successfully passed.

Additionally PHP Taint does not trigger the application itself, the application has to be requested by the developer or any other user with access to the part under test. This makes it difficult to test all parts of an application.

Wapiti, on the other hand, is able to crawl the website and detect sources and vulnerabilities. This approach will automatically perform a test of all the pages of the application but it might not cover the whole application, as all functionality has to be exposed on the website. There is also the possibility that Wapiti will not detect the vulnerabilities as it has to identify the error in the response and generate the correct input which in many cases was the problem.

Wapiti has the advantage that it is language independent, which allows the same tool to be used on all web applications as vulnerabilities like XSS are not language dependent.

Wapiti and the other tools that used a dynamic black box technique does not or poorly support custom authentication. That is authentication based on sessions within the application. They rely on the user providing a cookie that is already logged in, and that all URLs that would log out the user is excluded.

Another problem with the crawling approaches is that they might end up in an endless crawl. This is due to some pages like a calendar where the links change into unique links every time. Some tools allows for defining the depth the crawler should crawl and other allows for excluding specific URLs, however both have its limitations and may result in overlooking vulnerabilities.

Common to both tools is that they require at least one request per code path to determine if there is a vulnerability, and Wapiti might need several in order to generate the right input. This requires that the code needs to be executed several times and will make the test more time-consuming.

The following list is a summary of the properties of the dynamic tools we have tested.

- They can test a website as seen by the users.
- They can be language independent.
- They can return many false negatives.
- They can be slow.
- The black box tools have the following in common:
  - They do not use module files as entry points unless they are instructed manually.
  - They will not find all functionality if it is not present directly on the website.
  - They handle pages with custom authentication bad.
  - They might crawl endlessly on calendars and similar pages.
- The white box tools does not:
  - Examine the website itself.
  - Generate a report for all pages at once.

## 3.5 Properties of CodeSecure and RIPS

Two of the further tested static tools were CodeSecure and RIPS which uses the same approach, namely analyzing the source code statically, and both of them passes and fails the same test cases. They were able to find hidden functionality on the web application which was not sanitized at any point. Furthermore they found the vulnerability where malicious code could be injected in hidden input fields. The *Unreachable code* test was marked as a vulnerability, which is not an actual vulnerability, but developers using these tools will be aware of improper sanitization no matter if the code is executable or not.

When the source code was statically analyzed some false positives occurred. One example of this is custom sanitization where the tools failed detecting that a string is successfully sanitized using regular expressions, however, regular expressions should be used with care and thoroughly tested before use. Detecting proper custom sanitization is hard to accomplish, however, work has been presented to verify the sanitization by using an automata-based approach [14, 1].

One pitfall was when the source of the vulnerability was replaced with a sanitized value with regard to XSS. The *Sanitizing source* test is shown in Source Code 3.1.

```
foreach ($_GET as $k => $v) {
   $_GET[$k] = htmlentities($v);
}
echo $_GET["value"];
```

**Source Code 3.1:** Sanitization where the source is replaced with an untainted value with regard to XSS.

In the source code above all `$_GET` variables have been properly sanitized with regards to XSS. The static tools failed to detect that the user input was sanitized, as the tools do not verify the actual values of the variables because the validation is not made at runtime. Hence line 4 was marked as a false positive vulnerability.
Because the evaluation is not performed at runtime the static tools cannot determine dynamic content. If another file is included based on the input from the user, the static tools do not know which file to include. The tools can, however, scan all files located in the web application and find the vulnerabilities in each file. But there are some problems with this approach. Firstly the tainted variables could have been sanitized before use in the current scanned file, thus leading to false positives. Secondly the control flow can change dependent on the user input. For instance some files might not be included when certain data is received from the user, thus again leading to false positives.
One of the major drawbacks of these tools is the lack of certain language features. They both failed tracking object-oriented code where a class property contained tainted user data and written back to the user. Major PHP applications, such as Moodle and WordPress, uses objects frequently, and the tools will not detect vulnerabilities appearing when working with objects.
The following list summarizes the properties of the two static tools:

- They can find hidden vulnerabilities.
- They fail to verify custom sanitized user data.
- They cannot detect if the source is replaced with a sanitized value.
- They are unable to track variables in dynamic scenarios.
- They have difficulties with object-oriented code.

## 3.6 General Properties of all tested Tools

There are some properties that are common to most of the tools. All tools examined, besides PHP Taint, are either dynamic black box testing or static white box testing. It seems that none of the tools explore the possibilities of gathering information from different combinations of techniques and utilizing the advantages from them to overcome some of the disadvantages. Some research has, however, been done trying to combine the static and dynamic approaches with successful results but the tool is not publicly available [1].
Also the level of configuration of the tools seems very limited with regard to information level. That is if the user of a crawling tool knows that a URL pattern is exposing the same functionality, like a calendar, it is not possible to limit the requests to only visit the same functionality once. Users of the static tools cannot be instructed to find specific patterns in the language that exposes risks.
The lack of configuration possibilities and the amount of dynamic black box tools that tries to exploit known vulnerabilities, and PHP Taint which is used as runtime protection, gives the impression that the tools are created as hacking tools instead of helping preventing the vulnerabilities. Some of the tools that are targeted prevention are commercial and as none of them provided a trial license the quality of those has been impossible to measure.

# 4 Conclusion

Throughout this report several testing techniques, vulnerabilities, and scanning tools were examined. The testing techniques were discussed with regards to web applications in order to understand how the security detection tools work. The white box, gray box, and black box testing procedures were described where the essence was to perform tests based on the amount of internal knowledge about the target to exploit. Dynamic and static testing described the testing approach, where dynamic testing means that the application is executed, while under static testing it is not.

Some of the common vulnerabilities; SQL injections, cross site scripting, cross site request forgery, HTTP header injections, and code executions, have been described to gain an overview. Each contains a description of how the vulnerability arises, how to exploit it, how to protect against the attack, and finally the worst case scenario if the vulnerability is exploited. There is no single way to protect a web application against all vulnerabilities at once, however, if the developer codes with discipline it is possible to avoid the obvious vulnerabilities.

We tested 13 tools against four web applications: Moodle, WordPress, a simple script called TestApp, and finally an application called DoubtfulSystem; a local newly established company's website. All applications contained known vulnerabilities, either cross site scripting or SQL injections. Out of the tested tools the two most promising dynamic testing tools and the two most promising static testing tools were further examined. PHP Taint, Wapiti, CodeSecure, and RIPS were the tools that gave the best results, and for these tools we made more advanced test cases to enlighten their exact strengths, weaknesses, and limitations, and to find the exploit scenarios they could not prevent. This resulted in an evident set of properties for the dynamic and static tools.

## 4.1 Future Work

Throughout this report we found that static and dynamic tools had each their strengths, weaknesses, and limitations. No tool tries to use multiple techniques or combinations of techniques to overcome the limitation. As stated in Section 2.1 more information makes it easier to test an application effectively. Therefore we propose a modular tool that allows the modules to access all data available, that is data generated from other modules that has already been executed, or maybe use the existing Metasploit framework if it fits. This allows us to create a tool that is able to use a combination of static analysis and dynamic testing thus giving more information and hence an improved scan report should be possible.

We have planned to develop an initial set of modules in order to find vulnerabilities in the web applications we have tested against during this project. The nature of the modules can be divided roughly into three groups: Information gathering, vulnerability detection, and report generation. This division is merely a convenience to the user when using the tool and not a technical requirement and it is also used in the following description of our planned modules.

**Vulnerability detection** The vulnerability detectors are the key modules of the tool, without one of those the tools will not be able to detect any vulnerabilities.

- **Taint detection** Through our testing we found that PHP Taint was the tool that did the most accurate detection of the vulnerabilities. Therefore changes will be made in the reporting part of PHP Taint such that the information will be available to the tool. Our other modules will be created with this type of vulnerability detection in mind.

**Information gathering** This group of modules is responsible for gathering information about the application to be used by the vulnerability detectors.

- **Alternative entry points** The tested dynamic tools crawled through the website and maybe used a predefined list of common files in web applications, but none tried to use the separate files of the application as entry points. This module should try to identify all the possible entry points of the application.
- **Input variables** All of the vulnerabilities we have covered were due to improper sanitizing of user input and hence all possible sources of user input in the application should be identified by a module and used as attack points.
- **Code coverage/control flow identification** As PHP Taint only detects vulnerabilities in the executed code of each request, a module should analyze the code of the web application and detect how user input influences the control flow and identify the possible values of the input variables such that all code is executed.
- **Crawler** A crawler should be implemented as this should perform the requests such that PHP Taint is able to do the vulnerability scan.

**Report generation** This group of modules generates the actual report of a scan. The output format could be of any type. We have not thought of how the reports of our first module should be, however, this is of minor concern now.

We believe that the before mentioned combination of modules will allow us to detect the vulnerabilities in the four applications we tested against that the other tools had problems detecting. Of course more modules could, and should, be developed targeted other languages or being language independent.

# Bibliography

[1] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications, 2008.

[2] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D. T., AND KUO, S.-Y. Securing Web Application Code by Static Analysis and Runtime Protection. *WWW'04 Proceedings of the 13th internal conference on World Wide Web* (2004).

[3] LLC, D. Programming language popularity. `http://langpop.com/`, October 2011.

[4] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM* (1990).

[5] PHP.INTERNALS MAILING LIST. Run-time taint support proposal. `http://news.php.net/php.internals/26979`, December 2006.

[6] PROJECT, T. O. W. A. S. OWASP Top Ten Project. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[7] SECTOOLS.ORG. Top 125 Network Security Tools. `http://sectools.org/tag/sploits`.

[8] SECURITYFOCUS.COM. Moodle Blog Module SQL Injection Vulnerability. `http://www.securityfocus.com/bid/20395`.

[9] SECURITYFOCUS.COM. WordPress SCORM Cloud Plugin 'ajax.php' SQL Injection Vulnerability. `http://www.securityfocus.com/bid/49484`.

[10] SEKTIONEINS. The month of php security. `http://www.php-security.org`, 2010.

[11] SOFTWARE, T. TIOBE Programming Community Index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, October 2011.

[12] VENEMA, W. Taint support for PHP. `https://wiki.php.net/rfc/taint`, June 2008.

[13] XIE, Y., AND AIKEN, A. Static Detection of Security Vulnerabilities in Scripting Languages. *Usenix Security Symposium* (2006).

[14] YU, F., BULTAN, T., AND IBARRA, O. H. Symbolic string verification: Combining string analysis and size analysis. In *in Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS* (2009), pp. 322–336.

# A Source Code of TestApp

```php
<?php
ini_set("display_errors", "on");
error_reporting(E_ALL);

if (isset($_GET["hej"])) {
   $a = $_GET["hej"];
   echo htmlentities($a);
}

if (isset($_POST["bab"]))
   echo $_POST["bab"];

if (isset($_POST["submit"])) {
   $cn = mysql_connect("localhost", "root", "baconbab");
   mysql_select_db("test");
   $username = mysql_real_escape_string($_POST["username"]);
   $password = $_POST["password"];
   $q = mysql_query("SELECT * FROM users WHERE username='". $username ."' AND password='". $password ."'");
   $res = mysql_fetch_assoc($q);
   echo ($res ? "Welcome ". $_POST["username"] : "Failed.");
   mysql_close($cn);
}
?>
<title><?php echo @$_POST["bab"] ?></title>
<hr/>
<a href="?hej=rap">A test link</a><hr/>
<form action="" method="post">
<input type="text" name="bab" value="Some text"/>
<input type="submit" value="Send a text"/>
</form><hr/>
<form action="" method="post">
Username: <input type="text" name="username"/><br/>
Password: <input type="password" name="password"/><br/>
<input type="submit" name="submit" value="Login"/>
</form>
```

# B Test Case Script

```php
<?php
// Setup
ini_set("taint_error_level", E_WARNING); // PHP Taint specific
ini_set("display_errors", "on");
error_reporting(E_ALL);

// Helpers
define("VAL", @$_GET["val"]);
function t($i, $m="GET") {
   return ($m == "POST" ? isset($_POST["type"]) && ($_POST["type"] == ↵
       $i) : isset($_GET["type"]) && ($_GET["type"] == $i) );
}

// Hidden vulnerability test
if (t("hidden"))
   echo VAL;

// Condition test
if (t("cond1")) {
   $a = htmlentities(VAL);
   echo $a;
} elseif (t("cond2")) {
   echo htmlentities(VAL);
} else {
   echo VAL;
}

// Wrong escape method test
if (t("escape")) {
   $b = escapeshellcmd(VAL);
   echo $b;
}

// Function test
function a_test($an_input) {
   return $an_input;
}
if (t("func")) {
   echo a_test(VAL);
} elseif (t("func")) {
   echo VAL; // Unreachable code test
}

// OOP test
class DummyClass {
   private $a;
   public function __construct() {
      $this->a = VAL;
   }
   public function __toString() {
      return $this->a;
   }
}
if (t("oop")) {
   $d = new DummyClass();
   echo $d;
}

// Regex test
if (t("regex")) {
   $c = preg_replace("#[^a-z0-9]#i", "", VAL);
   echo $c;
}

// SQL injection test
if (t("login", "POST")) {
   $cn = mysql_connect("localhost", "root", "baconbab");
   mysql_select_db("test");
   $username = mysql_real_escape_string($_POST["username"]); // OK
   $password = htmlentities($_POST["password"]); // VERY WRONG
   $q = mysql_query("SELECT * FROM users WHERE username='". $username ↲
       ."' AND password='". $password ."'");
   $res = mysql_fetch_assoc($q);
   mysql_close($cn);
}

// Silenced SQL injection test
if (t("loginsilence", "POST")) {
   $cn = @mysql_connect("localhost", "root", "baconbab");
   @mysql_select_db("test");
   $username = @mysql_real_escape_string($_POST["username"]); // OK
   $password = htmlentities($_POST["password"]); // VERY WRONG
   $q = @mysql_query("UPDATE users SET password='". $password ."' ↲
       WHERE username='". $username ."'");
   $res = @mysql_fetch_assoc($q);
   @mysql_close($cn);
}
```

```

// Hidden SQL injection
if (t("loginsilencefaked", "POST")) {
  $cn = mysql_connect("localhost", "root", "baconbab");
  @mysql_select_db("test");
  $id = $_POST["raprap"]; // VERY WRONG
  $username = mysql_real_escape_string($_POST["username"]); // OK
  $password = mysql_real_escape_string($_POST["password"]); // OK
  $q = mysql_query("UPDATE users SET password='". $password ."', ↵
      username='". $username ."' WHERE id='". $id ."'");
  $res = mysql_fetch_assoc($q);
  mysql_close($cn);
}

// Overwrite source
if (t("overwrite")) {
  foreach ($_GET as $k => $v)
    $_GET[$k] = htmlentities($v);
  echo VAL;
}
?>
<form action="" method="post">
<input type="hidden" name="type" value="login">
Username: <input type="text" name="username"/><br/>
Password: <input type="password" name="password"/><br/>
<input type="submit" name="login" value="Login"/>
</form><hr/>
<form action="" method="post">
<input type="hidden" name="type" value="loginsilence">
Username: <input type="text" name="username"/><br/>
Password: <input type="password" name="password"/><br/>
<input type="submit" name="login" value="Login"/>
</form><hr/>
<form action="" method="post">
<input type="hidden" name="type" value="loginsilencefaked">
<input type="hidden" name="raprap" value="1">
Username: <input type="text" name="username"/><br/>
Password: <input type="password" name="password"/><br/>
<input type="submit" name="login" value="Login"/>
</form><hr/>
<a href="testcase.php?type=cond1&val=a_value">Cond1</a><br/>
<a href="testcase.php?type=cond2&val=a_value">Cond2</a><br/>
<a href="testcase.php?type=escape&val=a_value">Escape</a><br/>
<a href="testcase.php?type=func&val=a_value">Func</a><br/>
<a href="testcase.php?type=oop&val=a_value">OOP</a><br/>
<a href="testcase.php?type=regex&val=a_value">Regex</a><br/>
<a href="testcase.php?type=overwrite&val=a_value">Overwrite</a><br/>
```

**Source Code B.1:** Test cases for the four selected tools.