Evaluating the Most Efficient Method of Website Vulnerability Detection

Student Name: B.D. Holmes

Supervisor Name: Dr M.J.R. Bordewich

Submitted as part of the degree of **Computer Science** to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University.

***Abstract* –**

Background: Websites are becoming an increasingly popular medium for interactive services created by a spectrum of different people from individuals to huge businesses. Since anyone can create a website, mistakes in coding often go unnoticed. This can lead to vulnerabilities that other users can exploit.

Aims: The aim of this project is to investigate the best method of identifying any vulnerabilities that are present in a given website. These will then be presented to the user with an in-depth analysis of any issues found, along with potential solutions. We will evaluate two methods, one of source code analysis, the other of simulated injections.

Method: Source code analysis involves lexically parsing the source code of a website itself, tracing user entered variables and other data that could potentially be malicious. Simulated injections involve submitting tailored user input to a website and determining if it was successfully validated.

Results: Both methods successfully discovered new vulnerabilities in external scripts. Source code analysis discovered more issues than simulated injections; however both provided different information useful to fix the problems.

Conclusions: Both methods of vulnerability detection were implemented successfully and both highlighted security flaws in external scripts and provided potential solutions. The source code analyser is currently limited to analyzing the PHP programming language and should be extended to support others in the future. The results show that both methods are useful for vulnerability detection, each with different strengths and weaknesses. Neither method is clearly better than the other; therefore it is prudent to say the most effective approach would be to use them simultaneously.

Keywords – XSS (cross-site scripting), XSRF (cross-site request forgery), SQL (structured query language), Injection, GUI (graphical user interface), Vulnerability, JS (Javascript).

I. Introduction

The internet is constantly growing. Each day thousands of new websites are born. The expanding demand for digital media requires an increasing number of website developers. Since the web is still a relatively young concept, the number of website development training courses is quite low and their nature continually being refined. The web is very unstructured and HTML (hyper-text markup language, the main language used to describe the layout of a webpage) can be poorly written yet still work to a certain degree. Many different types of media are constantly being added to the internet including video, flash, applications and so on. It is also relatively easy for an individual to create their own website for a low cost or even free. All of this means that new websites are constantly being created without the developer having a full understanding of the code they are writing, but they can get away with this as many other people cannot tell a poorly written website from a robust one. This leads to many companies and individuals purchasing websites that are vulnerable to different types of attack which could completely compromise their system, database and user experience. In fact there have been occasions where whole companies have collapsed after a malicious user has exploited a hole in the company's website [1]. Even Google has been the victim of these exploits, most recently the e-mail accounts of some human rights activists were hacked which has in part lead to Google pulling out of China [2]. The main aim of this project was to create a system that can detect these vulnerabilities before they pose as a real threat.

The project looks at two different methods of detecting website vulnerabilities: source code analysis and simulated injections. Source code analysis involves lexically parsing the actual source code of a given website in order to determine if there are any issues that need addressing. Simulated injections involve injecting harmless scripts into a website and determining whether or not they were successfully sanitized. This is done by first running a web crawler on the website to gather all of the available pages, parsing out HTML forms that accept user input and then forging a specific injection script for each form. The outcome of this is an application that can be provided with the link to a website and its source code and then run the two methods on the given input and produce a summary report of all the different flaws it may find, along with potential solutions for each. This study aims to answer whether or not security issues of a web based system can be accurately detected and eradicated through source code analysis and if it is a more effective method than a simulated attack. To evaluate the methods, they were run on simple freely available PHP applications from a cross section of genres in order to determine which is best at identifying security issues and their specific nature.

II. Related Work

There are numerous areas of concern when it comes to website vulnerabilities. Below we will look at the most serious of these and discuss their possible implications, how they occur and how they can be prevented.

A. Cross-Site Scripting Attacks (XSS)

XSS attacks are the most common and serious website attacks, with some researchers claiming that up to 68% of websites could be vulnerable [3]. These attacks mainly occur when a user enters malicious input into a website form which it then uses without being correctly validated. They can also occur through the manipulation of queries sent to webpages as a string following a question mark at the end of the URL. Some webpages use this data to determine which content to display; if it is not validated correctly then issues may arise. Non-persistent vulnerabilities will immediately cause the next page to react to the malicious input, meaning only the user that injects the code will be affected. However, by social engineering, or through an XSRF attack, an innocent user could be tricked into submitting this input which may then compromise their experience. Persistent vulnerabilities are more dangerous, causing the malicious input to be stored in a database on the target website; any user that visits the compromised webpage will then be subjected to the attack.

An XSS attack can be formed in a large variety of different ways, ranging from simply changing the syntax of the vector (malicious input) in order to avoid specific sanitization methods to altering the character encoding of the vector causing the target script to interpret it in a different manner [4]. For instance, in 2005 a vulnerability in Google was found whereby modifying the query string to use UTF-7 character encoding when being redirected to an external website caused the search engine to directly output the query string, enabling a user to enter a script and have it executed [5]. A typical XSS vector contains HTML tags in order to alter the content on the target webpage in some manner, for instance this might simply change the way something looks on the page or more seriously load a Javascript script from an external location. For examples of the different kinds of vectors that are used see ***Table 1*** (page 14). To demonstrate how simple it is to execute this form of attack, ***Figure 1*** shows how an unprotected form can be misused.

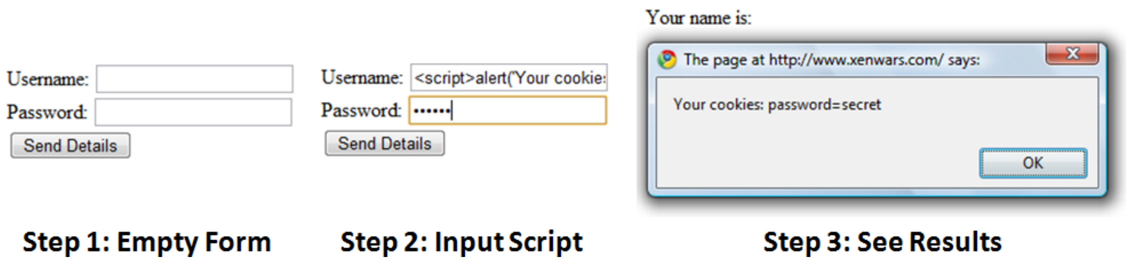

Figure 1. An example of how a user may execute an XSS attack and the results produced.

Despite the frequency at which these vulnerabilities occur, they can be surprisingly easy to deal with. Almost all user input can be sanitized by simply converting HTML characters such as ‘<’ and ‘>’ into their corresponding ‘entity’ values, in this case ‘&lt;’ and ‘&gt;’. These values are interpreted by web browsers to display the same original character, but are no longer parsed as parts of the HTML. In the PHP programming language, the ‘htmlentities’ function takes any variable as input and converts all HTML characters it contains into their entity value equivalents [6]. Of course, in some scenarios this may not be appropriate, such as messages on online bulletin boards that allow some text formatting. In these scenarios more imaginative custom sanitization methods must be put in place restricting the user from entering malicious code.

B. Cross-Site Request Forgery Attacks (XSRF)

XSRF attacks are less common. These can be used in tandem with an XSS attack or standalone. They involve requesting an external website’s services, commonly through the use of a HTML image tag, HTML inline frame or Javascript script which automatically loads the content of a supplied link. These services may involve the sending of specific details, for example to a bank website a ‘transfer to’ bank number and a ‘money’ value, which the target website may then execute automatically without the user knowing [7]. In this sense, XSRF attacks are a form of the ‘confused deputy’ problem, where a webpage is deceived by another user in order to abuse its authority [8].

Once a form has been discovered that is vulnerable to this kind of attack there are two options that a hacker can take. The first option would be to personally setup an external website and embed a hidden script that causes the vulnerable form to be automatically submitted when a user visits your website. This would require you to trick the user into visiting your website and also for them to have logged in or shared some personal details with the target website; otherwise it will have little effect on the user if they are uninterested in the target website and its services. The second option would be to look for an XSS vulnerability on the target website that allows the injection of an XSRF vector; the inserted vector could target the vulnerable form, meaning any user that visits the compromised page would automatically submit the form with the malicious input detailed in the vector. Since in this situation the vector targets the same website, it is likely any users caught by this will have shared sensitive information with the website that could be violated. Furthermore, an XSRF vector embedded in the target website could send sensitive data from that website to an external website, such as a users session details. This would then allow the hacker to visit the target website using the same details, tricking it into thinking they are the original user. ***Table 1*** (page 14) shows some vectors that can be used in an XSRF attack, and ***Figure 2*** shows the necessary steps to execute the attack.

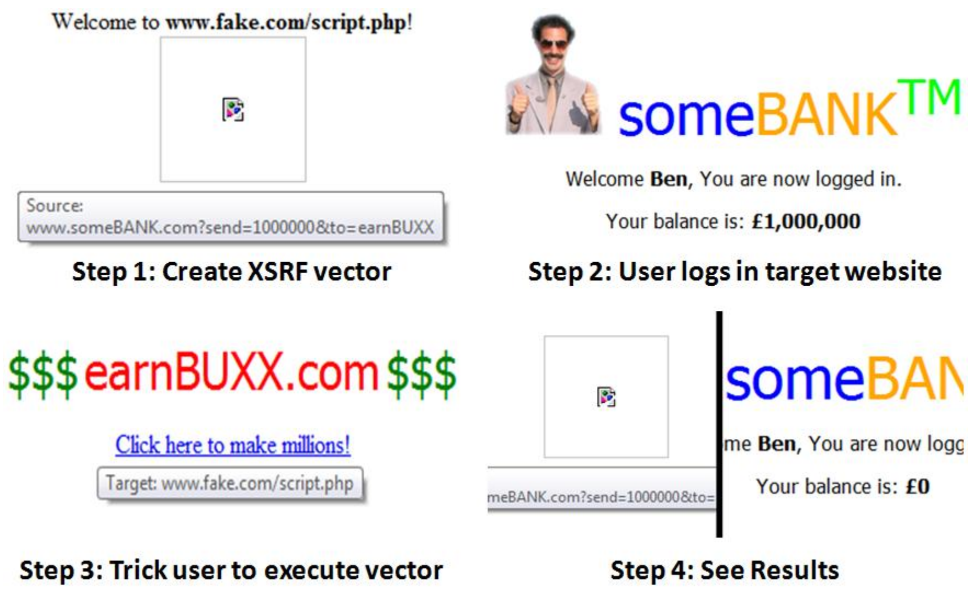
Welcome to www.fake.com/script.php!
Source: www.someBANK.com?send=1000000&to=earnBUXX
Step 1: Create XSRF vector
someBANK™
Welcome Ben, You are now logged in.
Your balance is: £1,000,000
Step 2: User logs in target website
$$$ earnBUXX.com $$$
Click here to make millions!
Target: www.fake.com/script.php
Step 3: Trick user to execute vector
meBANK.com?send=1000000&to=
someBAN
me Ben, You are now logg
Your balance is: £0
Step 4: See Results

Figure 2. An example of how a user may execute an XSRF attack and the results produced. Initially a malicious user creates a webpage containing an XSRF vector targeting a bank website. Another user logs in the targeted bank website. The malicious user then tricks the other user into visiting the compromised webpage. Because the user was logged in on the target bank website, the vector executes successfully and the user's bank account is emptied.

Unlike XSS, the prevention and detection of XSRF vulnerabilities can be more difficult. The most basic method to guarantee that a form is protected is to generate a unique token as one of the form elements every time the form is loaded, and also store this as a user cookie. When the form is submitted, if the token is omitted or is different to the cookie value then simply ignore the data received. Because XSRF attacks are blind, there is no way for the vector to determine what the unique token will be, so unless the token is guessable the attack will always fail. A more secure method would be to store the token in a server side database so that if the attacker somehow has access to the user's cookies they still cannot determine the token value. Similarly, the use of a Turing test on a form, which is a test to verify if a user is human or a machine, can prevent XSRF attacks [9]. One example of such a test is the use of a CAPTCHA image [10]. Another approach which is somewhat easier to implement is to check the referrer header of all data received from a form or query; if the origin of the data is not the same domain as this one then it can be ignored. This method is flawed however, as referrer information is modifiable. By limiting the lifetime of user cookies, or only using session data, an XSRF attack can be averted as when the vector is ran the authentication on the target website may have ended. Developers should also ensure that no 'crossdomain.xml' files exist on the website, as this can allow flash to access pages on this website as an authenticated user [11].

C. Structured Query Language Injections (SQL)

SQL injections are also fairly common, again involving the use of incorrectly validated user input. Whenever a database is queried using data that has been entered by a user, there is a risk that the entered data may contain malicious code that can alter the nature of the query. This can lead to disastrous effects such as the deletion or modification of database records.

The actual format of an SQL injection varies depending upon the effect the malicious user wishes to attain, however all injections come as one of two basic types. The first and most common format, but also the most restrictive, requires the user to first interrupt the query being injected by using a single or double quote, and to then enter their own SQL to complete the query in their own way. The reason the quote interrupts the query is because the query is represented as a simple string by the webpage, so by adding an extra quote the webpage regards the string to either have ended or have a variable being appended to it. The second format also interrupts the query like the first, but then by the use of a semi colon allows the attacker to execute their own completely custom queries. The reason the first format is more restrictive is because the attacker can only manipulate the type of query they are hijacking, so if it is a 'select' query they can only choose which data is extracted from the database, whereas the second format can append any query such as updating or deleting data. On most PHP systems, the second format will not work as multiple queries in one command are disabled by default [6]. ***Table 1*** (page 14) shows some vectors that can be used as SQL injections, and the same steps as in ***Figure 1*** can be used to deploy them.

The prevention of SQL injections, like XSS attacks, is quite straightforward. A limited number of characters, predominantly ' and ", must be escaped by prepending a backslash to them, causing the query to read them literally instead. In PHP, the 'mysql_real_escape_string' function does this for all dangerous characters automatically, without modifying any data stored in the database [6].

D. Javascript Injections (JS)

A more advanced method of website hijacking involves the use of Javascript injections. Javascript is a client side scripting mechanism understood by most modern browsers which allows the modification of data on any given website on the fly. If an XSS vulnerability exists, a malicious user may be able to enter their own custom scripts which either interfere with existing scripts or modify the webpage in some way so that it no longer behaves in the way intended by the original developer. External script files can also be loaded in this way, meaning any text length restrictions can be bypassed in order to inject more code. Javascript vulnerabilities can be very serious as it is possible to access a user's cookie information. This could then be sent to an attacker and be used to access the website that set the cookies under the guise of the tricked user. Javascript may also already exist on a webpage to control different aspects of user interaction. It is possible to modify any variables and algorithms used by this Javascript, the simplest way being to simply enter your own custom Javascript in the address bar. If any sensitive information is contained or manipulated by this Javascript then it is at risk of being tampered with by a user. To prevent this, sensitive information should not be created or used by Javascript, and any information that is created should be thoroughly validated by a server side script (such as PHP). Similarly, any AJAX calls (requests made to other webpages via Javascript) that are made should also be validated in the same manner to prevent XSS and XSRF vulnerabilities.

E. Denial of Service Attacks (DoS)

The purpose of a denial of service attack is to saturate a resource so that it no longer functions correctly. This is primarily involved in computer networking, and more recently has been used against websites, web services and individual users. An attack involves a malicious user repeatedly sending requests to the target, saturating its bandwidth and preventing it from functioning correctly. This project does not address these kinds of attack as they are not concerned with website security and little can be done to prevent them.

F. Existing Solutions

Most literature on the topic agrees that this field of study is of upmost importance in the current world of internet growth and usage. According to one study, 70% of recently reported software vulnerabilities were associated with web applications, with 27% of those involving SQL injections and 24% XSS attacks [3]. The most serious of these attacks often involves the use of Javascript or a combination of multiple vulnerabilities, giving an attacker close to complete control over other users' data. To demonstrate just how serious this issue is, the website xssed.com maintains an archive of fresh XSS attacks found every day on many popular websites [12].

Some research has been undertaken into the use of client-side web vulnerability detection and eradication [13]. This involves scanning data sent and received from a webpage via a client side application. Although this method was effective at detecting possible security flaws, it also caused a large number of false positives. This is because when a user visits a webpage, any request to an external website could potentially be a malicious request aimed at submitting details on the user's behalf without their consent. Furthermore, any script that is executed on a webpage could be flagged as dangerous, especially if these scripts interact with the user's cookies or other data. Because of this, almost every visit to a new webpage would require the user to evaluate whether or not they wished to fully display the page. Although ultimately this was effective at preventing security issues with the user's data, it also meant that they would have to go through an additional page of screening for every page they requested. This was deemed impractical, as users would rather risk exposing their data than go through such a process for every webpage they visited.

Alternatively, a server-side method for XSS detection has also been investigated [14]. This involves automatic verification of data entered into a webpage before the actual script has chance to use this data for any real operations. This method is effective, as although it may not correctly detect malicious input, it will validate all input, meaning almost no existing flaws will be exploitable. It also means that the developer need not be concerned with the security to a large extent; however this may lead to complacency and un-portability. Unfortunately this approach produces a processing overhead; no matter what data is entered it still gets evaluated and therefore incurs extra strain upon the server every page load. For large websites this may be unacceptable as usage latency is a large factor in whether or not users will continue to use the service; research suggests a user will leave a retail website if it does not load after only four seconds [15]. Furthermore, the verification process may modify the input data permanently which for some applications may be unacceptable.

Many researchers have found the same problem in their work on detecting and preventing XSS vulnerabilities, and this again comes in the form of Javascript [16]. The Javascript interpreter is sufficiently versatile that there are many different ways that a script can be invoked. The most common, via a HTML '<script>' tag, is often the only one considered by developers as being dangerous. Javascript scripts can also be executed by simply using the ‘javascript:’ prefix which many string validators may accept as valid input. Further to this, many of the characters required in the scripts such as ‘{’ and ‘;’ can be replicated by using their Unicode equivalents which must also be detected if the webpage is to have adequate protection [4].

There has also been a lot of research into the different vectors that can be used to bypass sanitization functions, many of which exploit the different encodings used by the web, such as URL encoding, base64 encoding and different character sets [4]. Many of these vectors are specific to individual web browsers, as each browser interprets webpages in their own way. Patches for browsers vulnerable to these forms of attack are often released quickly, however many people still use older vulnerable versions of their browsers, such as IE 6 [17].

The primary mechanism used to prevent website attacks is user input validation. This is already an established software engineering process in wider applications of computing, but is often overlooked during web development. According to Perry and Evangelist, 66% of software application errors arose from interface problems [18]. Without validation, any user can enter commands that can alter the default operations of the application causing errors and data corruption [19]. In security critical environments, the user could use this to gain unauthorized access to private resources. Hayes and Offutt developed a method of using pre-defined specifications to reject any user input that doesn’t match; this allowed for dynamic test case generation and evaluation. This method was useful for validating simple, restrictive input fields; however complex fields require significantly more detailed specifications for them to be useful.

One existing method of vulnerability detection, an application known as Ardilla, uses similar methods to my source code analysis in order to detect potential security flaws [3]. By monitoring variables that are produced by some form of user input, known as 'tainted' variables, their path through the code is monitored and subsequently flagged if used in an SQL query. Ardilla also detects second order (persistent) attack vectors by tracing user input through any database systems in use. This works by recording the database fields that are affected by tainted variables and then picking them up again when they are extracted to determine if and when they are used, and if in fact they are validated correctly. This method is useful for detecting both XSS and SQL attacks however it is not useful for detecting XSRF attacks as these can be executed without using malicious data and are more concerned with the structure of the website itself. Since Ardilla detects vectors in user input as opposed to finding vulnerabilities in the code it is essentially an automated level of validation; this is useful but does not solve the underlying issues creating the vulnerabilities.

A team in Chicago has implemented a client side XSS protection tool called BLUEPRINT, which comes in the form of a web browser plugin designed to work with all popular modern browsers [20]. This tool detects scripts on pages visited and attempts to filter any potential threats. It also restricts cross site connections through Javascript and POST forms to help prevent unsolicited XSRF attacks. This approach filters both HTML before it has loaded and Javascript at runtime. Although the tool can effectively prevent attacks, it also adds to the loading overhead for every webpage visited. Furthermore, it may prevent legitimate scripts and webpages from working correctly.

The W3C (World Wide Web Consortium), who control many standards for web based scripting provide their own HTML validation service [21]. This validator takes any URL as its input, detects the type of document and determines whether or not it adheres to the current HTML standards. If it does not, the details of each error are shown with potential fixes. Since this validator is always up to date I decided to use this within my project, allowing a user to see the validator results for any webpage that is crawled.

III. Solution

My solution is a tool called “Website Security Analyser”, or WSA for short. The purpose of the application is to provide website developers with an easy to use interface for automatically detecting vulnerabilities in websites both existing and under development. This is provided in the form of a Microsoft Windows program developed using C# .NET which presents the user with a three-step process to help identify XSS, XSRF, JS and SQL Injection vulnerabilities in any website of their choice. The first step is to use the in-built web crawler on the chosen website to gather all of the webpages it contains and their contents. This also highlights any broken or external links that may exist on the website. The second step is to simulate a preset list of injections on the chosen website; all successful injections are detected and presented to the user with details on exactly which part of the website is vulnerable. The final step asks the user to select the source code for the chosen website; these files are then loaded and analysed for any code that could cause security flaws. Once all the steps are complete the results of each step are presented to the user with in-depth details of each step of the process.

A. Architecture & Design

When beginning the design of my system I decided to adopt an evolutionary process model. This allowed me to continually make changes to my design that would also be reflected in the tool as the duration of the project went on. At first I created a list of objectives and requirements that my tool should meet; these objectives were maintained and met once the project was completed. I also outlined the key components that would constitute the system along with preliminary system and class diagrams indicating how they would relate to each other. I decided to make large use of the singleton pattern; this involves creating a class that is only instantiated once. This allowed me to create various utility classes for different aspects of the tool that held functions and global statuses orientated around one particular area, such as the injection manager. I also ensured that the core functionality of the tool was separated from the interfaces. This means that only interface code is contained within the interface classes and the main algorithms are held in separate classes providing a much higher level of abstraction and data independence as well as making refactoring and maintenance processes easier in the future. To implement the system I chose to use the Visual Studio IDE making the creation of GUI’s much simpler. The main focus of the tool is on websites developed using the PHP programming language; I picked this because it is the most commonly used website programming language [22]. My interfaces were initially created quickly to demonstrate the functionality of the tool and provide an easier way to go about development and debugging. As the project evolved different parts of these interfaces were phased out and replaced with newer versions as seen in the final version of the tool; the first major change was to the results interfaces, providing a much more intuitive and clear way of displaying the data. Once this was complete I moved on to upgrading the core system interface to use the more intuitive three step process and also provide separate tabs to manage other components of the system. Furthermore I created a test website to help test the tool during development; this website was comprised of a large number of simple PHP webpages, each designed to test a specific aspect of the system.

Figure 3 shows the main components of the system and how they all relate to each other.

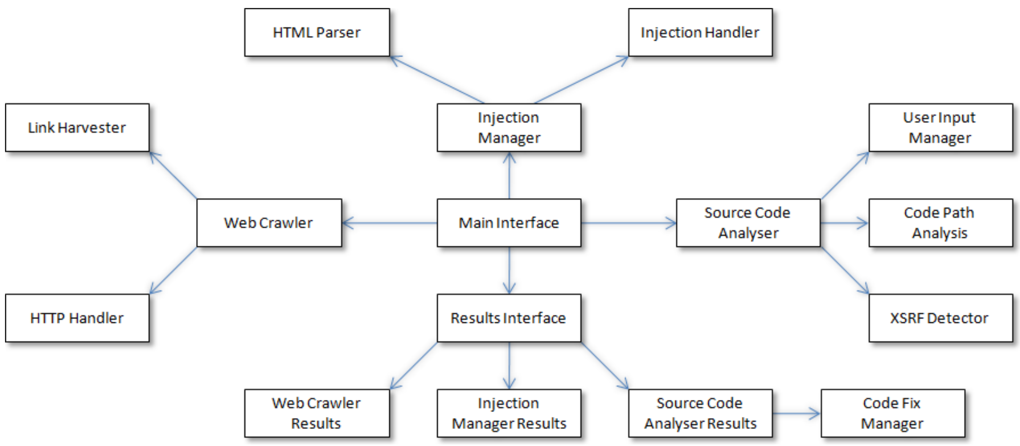

Figure 3. System architecture diagram.

B. Web Crawler

To begin, the web crawler takes the URL of the website to crawl and adds it to a list of links to check. A list of checked links is also maintained to ensure the same link is not visited twice; this could lead to an infinite loop if two webpages both link to each other. Two links are considered to be the same if they both point to the same webpage but use different protocols, for instance ‘http’ and ‘https’. Broken and external links lists are also created for user reference; these lists store links that could either not be evaluated or deviate from the target website. Each link in the links to check list is then crawled for more webpages in turn. Initially we check if this link is an e-mail link; these come in the form ‘mailto:e-mail’ and are irrelevant to our search. If one is found it is added to the checked links list. Next we attempt to grab the actual contents of the current link by using a HTTP GET request with headers set to mimic Google Chrome, ensuring that webpages allow access by the tool. Any cookies that are set are stored by the tool and sent along with each request so that the tool gets exactly the same content as a real user would. If the page has moved or redirects then it is recorded and the original URL is modified to match the latest version. We then check if any errors occurred whilst grabbing the webpage contents, and if so the type of error is recorded and the link is added to the checked and broken links lists. An error may occur if the link is not in a valid format, the tool does not have an internet connection or a specific HTTP error was returned, e.g. webpage not found [23]. Links found that do not contain the domain of the original URL are considered to be external; these links are added to the checked and external links list and subsequently ignored.

Finally, if the link has passed all of these conditions then we know that it is a valid webpage that is part of the originally entered website. We save the contents of this webpage which is then used by the HTML parser to find any HTML forms that it may contain. If a form is found, the name, method (HTTP GET or POST) and action (the page the form sends the details to) are parsed and saved. Any items within the form, such as textboxes, drop-down menus, checkboxes and so on, are also parsed and their name, type, value and maxlengths are stored. Since correct HTML is not strictly enforced, any of these fields may be missing; retrieved values are later used by the injection manager. Now that we have all the details about this webpage we parse the contents for links and their values. Every link found first has to be reconstructed into a full URL as links can appear in four different formats. They can come as a full URL, in which case we do not need to reconstruct the link. They can simply specify query data, such as ‘?login=true’, in which case we take the current URL and append the query. They can also come as a path relative to the current directory by simply specifying the name of the file or a partial directory path to the file. If the website has a ‘base HREF’ specified, we take this base and append the relative path; otherwise we take the current URL, remove the file name from the end and add the relative path in its place. The last format is a path to the file which is relative to the domain of the website, in which case we simply take the website domain and append the path. Each link found is added to the links to check list if it is not already in the checked or links to check lists. The current link is then removed from the links to check list and added to the checked links list. The contents of this webpage are disposed as an optimization technique to preserve RAM since they can contain a lot of information we no longer require. Once the links to check list is empty all of the results are calculated and stored.

C. Injection Manager

Once the web crawler has finished gathering all the webpages on the chosen website, the injection manager may take over and begin to inject the pages found. The tool first loads all the injections into the system ready for use; this includes the injection vector, the output signature and the type of injection. The signature is simply a string that the tool looks for in the content of the page the form is submitted to; if the signature is present then the injection was successful, otherwise it failed. We begin by iterating over every webpage found by the web crawler. If a webpage does not contain any forms, we assume that it does not accept any user input and it is recorded as safe. We process every form found individually; if the form does not contain any items it is ignored, as there are no input fields to inject. Any form missing a method field defaults to using the GET method, whereas if the action field is absent the form is submitted to the same page it was found on. This was implemented to mimic the actions of modern web browsers.

The injection process commences by injecting the form currently being processed with each vector used by the tool (see ***Table 1***, page 14). Any form item that has a unique name, and is of type 'textfield', 'textbox', 'hidden' or 'drop down menu' has its input value injected. Form items without a name are not used by the webpage, and items with a different type to those considered cannot be injected with custom data. Even though drop down menus have specific options to choose from, any data can be sent and it is up to the developer to validate this. Often a webpage will not check if the option chosen from a drop down menu is valid leading to a vulnerability allowing users to choose their own options. Furthermore, if the item has a ‘maxlength’ value that is lower than the length of the injection string, we still attempt the injection as this is a constraint on how the webpage behaves, data of any length can still be entered and the developer must validate this.

Once the input for each form item has been determined, we reconstruct the URL that we are going to submit the form to; this is based on the current webpage URL and the form action and done using the same method as the web crawler. If this form uses the GET method then the data being sent is appended on the end of the reconstructed URL as a query; this URL is then requested and the response stored. If the form uses the POST method, the data is streamed to the reconstructed URL and the response stored. It is important that the correct HTTP method is specified when submitting the form as the target website expects to receive the data in a specific format. The response is then parsed for the injection signature; if it is found then the injection is recorded as successful. We repeat this process for every injection, form and webpage found by the web crawler; once complete the results are calculated and stored.

D. Source Code Analyser

The source code analyser inspects the code of the webpages provided for potential vulnerabilities. Initially individual source files or whole directories can be loaded; currently these source files must contain PHP code. This allows the user to quickly load large numbers of scripts or to choose specific files. When a file is loaded, the code, name and language type is stored; by recording these details we can provide the user with higher quality results.

To begin, the source analyser inspects each source file in turn, initially checking for input variable validation; this can detect variants of all types of vulnerability for both form and query input. The first step is to split the code into tokens; there are many different tokens, each representing one individual piece of code such as a variable or function call. All tokens have a unique identifier indicating what the token represents, for instance the token 'T_WHITESPACE' represents spaces, tabs, etc. Some tokens also hold a value, such as the name of the variable, along with the line number of code that the token is found on. A complete list of the different token types can be found on php.net [24]. To actually split the code into tokens we use the 'token_get_all' function provided by PHP [6]. I chose to use this since it is developed by the PHP team and uses the Zend lexical scanner in order to parse the code into tokens perfectly, much like the actual PHP interpreter. This also means that if any changes are made to the PHP language the token parser will remain up to date. Since the function is implemented in PHP, I created a small PHP script stored on my Apache web server to run the token parser; this increases the portability of the tool as the user does not require a local PHP installation. The script takes source code as input, applies the 'token_get_all' function and returns the tokens in a parsable format.

If tokens were found in this source file, we inspect each one for input variables. If a variable is found, we determine if it is being assigned to, and if it is we store it as the 'current assignment variable'. We also check the value of the token, if it corresponds to '$_GET' or '$_POST' then it is added to the list of input variables. We do this because these values are used by PHP to store data received from forms and queries. If a current assignment variable is set, this is also added to the input variables list as it has now been assigned an input variable and may be tainted. Once an end of line character is reached we reset the current assignment variable as assignments cannot span over two lines of code. Once each token has been parsed, code path analysis is run on each input variable found.

During the analysis we keep track of whether we are in an echo; the PHP command used to output a string to the webpage. We also maintain a list of the functions that are currently being applied and record the current variable that is being assigned to. These statuses allow us to reason about any variables discovered in order to determine if they are the culprits of any security issues. We begin by once again scanning each PHP token discovered in the source file being analysed. If a semicolon token is found, the current assignment variable and echo statuses are reset; these instances cannot traverse multiple lines. If an opening bracket or function call is found it is added to the function list, whereas if a closing bracket is found the most recently added function is removed. This acts as a call stack to ensure we know which functions are being applied when we find a variable. If the current input variable is found, and it is being assigned to, we check if the function stack contains any sanitization functions from a list of pre-recorded PHP functions; this could occur in a scenario such as 'inputVariable = sanitizationFunction(otherVariable)'. Any sanitization functions found are recorded against the input variable, along with whether or not it is involved in a database query. This allows us to later calculate if the variable has been sufficiently validated and whether or not it could be used in an SQL injection. Conversely, if the current input variable is found and is being assigned to another variable we check if the input variable is fully sanitized. If it is not, the variable being assigned to is added to the list of input variables to ensure it also undergoes code path analysis, as it has now been tainted. This could occur in a scenario such as 'otherVariable = noSanitization(inputVariable)'. If the current input variable is found when there is no assignment occurring, but the echo status is set, we must once again record any sanitization functions in the function stack against the input variable; this is because the input variable is being directly output to the webpage. The same process is used if a database query has been created involving the current input variable. If a variable is discovered when there is no assignment occurring, we must check if this variable is initiating an assignment in order to determine the nature of situations involving future tokens. Once all of these scenarios have been checked this token has been successfully parsed and the next token is selected.

Once all of the tokens have been parsed we store the results for this input variable and calculate a code fix suggestion. This is done by looking at the sanitization functions that have been applied to the input variable and determining if they are sufficient to prevent any attacks. If they are not sufficient and the input variable poses as an XSS threat then code using the input variable is created which is sanitized using the ‘htmlentities’ PHP function; this function prevents all HTML from being interpreted by the web browser, however it may not be suitable in all situations in which case custom sanitization should be applied [6]. If the variable poses as an SQL threat then code using the input variable is created which is sanitized using the ‘mysql_real_escape_string’ PHP function; this function prevents all SQL injections from occurring and does not modify the data [6]. The next input variable is then selected. If this input variable was formed from another input variable (i.e. through an assignment), we set the sanitization functions applied to this input variable to be the same as the ones applied to the input variable it was formed from. This is because any sanitization functions that were used on the original input variable have effectively been transferred to the variable that was formed from it.

Once all of the input variables have been traced we calculate the results and move on to check for XSRF vulnerabilities in the forms in this source file. To do this we first count the number of forms in the source code, and also the number of hidden fields used in these forms. If the number of hidden fields is smaller than the number of forms this means that there is a form that does not make use of a hidden field to submit a unique token to prevent XSRF attacks. This is only a rough method of XSRF detection as there are other precautions that can be used to prevent the attacks, however the use of a unique token is a major prevention mechanism that is used, and so by accurately detecting if this is implemented or not allows us to give an informed warning as to whether or not an XSRF vulnerability may exist. Once this is complete, the results of code path analysis and XSRF detection are stored and the next source file is selected. We continue doing this until all of the source files have been scanned, at which point the final results are calculated and stored.

E. Interfaces

The main application interface consists of a tab based window providing three different options - website controls, results and log. The Website Controls tab provides the majority of the functionality for the tool in the form of a user-friendly three step process. As the user completes each step the results will be made available and the next step will be enabled. If at any point the user wishes to restart the steps, the ‘Restart Steps’ button in the file menu allows them to do this. I chose a step based environment as it is a good software engineering principle to provide the simplest user experience possible whilst retaining the same functionality. The initial interface design allowed a user to simply perform any task at any point; however this could lead to confusion when attempting a task that requires a previous task to be completed. The first step requires the user to enter the URL of the website they wish to analyze, prompting the web crawler to begin. The next step allows the user to initiate the injection manager, which simulates each of the pre-defined injections on every webpage found by the web crawler. The final step requires the user to load the source files of the website being analyzed; this can be done by loading a whole directory or individual files, giving the user flexibility over the files that are scanned. Once this is complete, the final results are calculated and the ‘all results’ form is launched. The results tab provides a list of all the results that have been generated by the tool. Saved results can also be loaded into this list using the 'Load Results' file menu option. This allows the user to revisit details about previously analyzed websites that may otherwise be lost once the program is terminated. The log tab provides a text box containing in-depth details of what has happened during the runtime of the tool. This is useful to check for further details if any errors occur during the analysis of a website. The ‘Help’ menu provides extra support.

Whenever a complex process is being executed, a process box is launched telling the user the current status of the process and gives them the option to cancel it at any time. This was implemented to prevent the user from thinking the program had crashed or was not working correctly, as well as giving them an indication of the length of time they may need to wait.

The ‘all results’ form provides an intuitive interface to view the results from each step in a separate window, along with the option to save them. When designing the reports I decided that highlighting significant results in order to attract user attention, whilst still displaying all outcomes, was the main objective. To do this I implemented a traffic light display which users should find instantly intuitive. The web crawler results window provides the user with a tree view of the webpages found; this allows the user to see the link hierarchy, making it easier to browse for specific webpages than a regular list. Each page is colour coded according to whether it was successfully browsed (green), broken (red) or external (purple), so the user can easily identify specific pages. The overview screen gives an indication of the success of the web crawler, along with vital statistics about the pages and forms found. When an individual webpage is selected, a similar screen to the overview is presented, but with details specific to the selection. The interface also provides the user with the ability to launch the W3C HTML validator which tells the user about any HTML errors on this webpage and how they can be fixed. If any forms were found on this webpage, in-depth details of their structure can also be viewed, acting as a debugging aid.

The injection results window provides a tree view containing all of the webpages injected by the injection manager. Each page is colour coded according to whether all injections failed (green), most injections failed (orange) or most injections succeeded (red); this allows a user to immediately identify vulnerable pages. The overview screen gives an indication of the success of the injections, along with vital statistics about the injection manager outcomes. Selecting an individual page displays a similar screen to the overview about the selected page. It also generates a box for each injection used; this gives information about the injection and how successful it was.

The source scanner results window provides the user with a tree view of the files that were analysed; this allows the user to easily navigate the files as if they were in their original directory structure. Each file is colour coded according to whether no vulnerabilities were found (green), low risks were found (orange) or high risks were found (red). The overview screen gives an indication of the amount of overall vulnerabilities found, along with statistics about their types. When an individual file is selected, a screen similar to the overview page is shown pertaining to this file. A box is also generated for each vulnerability found, giving details about the type of vulnerability, where in the code it occurred and a code snippet that can be used to fix it. This enables the user to then immediately find and patch the issues in their scripts.

IV. Results

In order to evaluate which method is best at detecting security issues in webpages, I selected six scripts from open source PHP script repositories on the internet. These scripts were created by different people with whom I have had no contact; to keep the test fair I did not look at the scripts and the only modifications made were to install them on my web server. Each script is from a different category of programs to get a representative sample of the different scenarios that PHP can be used for, ensuring that both methods were tested on different types of code and design. The scripts are lightweight, being made up of only one to five source files; this is beneficial as it means that superfluous source files containing content and design rather than functionality were minimized.

The injection manager and source code analyser were run on each script in turn and the results correlated against each other. ***Table 2*** shows which injections were successful on which script file, and ***Table 3*** shows how many vulnerabilities were identified by the source code analyser and their respective severities. The different injections that were used by the injections manager are shown in ***Table 1***.

Table 1. Injections used by the injection manager.

Injection Type	Injection String	Description
XSS	<b>bens website vulnerability scanner test injection</b>	Attempts to input some very basic HTML into a form. If the form accepts this HTML and then prints it out directly then it may be possible to insert more malicious HTML.
JS	<script>javascript:alert('bens website vulnerability scanner test injection');</script>	Attempts to input a basic Javascript alert message. If the form accepts the script and prints it out directly then the script will be executed. A user could submit a more malicious script to take advantage of this.
JS	<script language="javascript">alert('bens website vulnerability scanner test injection');</script>	Similar to the previous injection but uses slightly different notation in an attempt to avoid primitive sanitization techniques.
SQL	x' AND \`fake\` = CRASH	Attempts a harmless SQL injection which is designed to cause an error to occur on the page if it is successful.
SQL	x" AND \`fake\` = CRASH	Similar to the previous injection but uses slightly different notation in an attempt to avoid primitive sanitization techniques.
SQL	x'; SELECT * FROM \`fake\`;	Attempts a harmless SQL injection designed to target systems that allow multiple queries in one request.
SQL	x"; SELECT * FROM \`fake\`;	Similar to the previous injection but uses slightly different notation in an attempt to avoid primitive sanitization techniques.
XSRF	<img src="http://www.externalwebsite.com?bankDetails=12345"/>	Attempts a fake XSRF attack on a HTML form. If the form accepts this injection and prints it out directly then a user visiting this page may inadvertently send the bank details '12345' to externalwebsite.com.

TABLE 2. INJECTION MANAGER RESULTS.

		Injections							
		XSS #1	*JS #1*	*JS #2*	*SQL #1*	*SQL #2*	*SQL #3*	*SQL #4*	*XSRF #1*
Scripts	*Guestbook Script [25]*	✗	✗	✗	✗	✗	✗	✗	✗
	Login Script [26]	✗	✗	✗	✗	✗	✗	✗	✗
	Search Script [27]	✓	✓	✓	✗	✓	✗	✓	✓
	Upload Script [28]	✗	✗	✗	✗	✗	✗	✗	✗
	Contact Script [29]	✓	✓	✓	✗	✗	✗	✗	✗
	Forum Script [30]	✓	✗	✗	✓	✗	✓	✗	✓

The injection manager found a total of thirteen vulnerabilities in all of the scripts. Six of those were found in the search script, four in the forum script and three in the contact script. No vulnerabilities were discovered in the guestbook, login or upload scripts.

TABLE 3. SOURCE CODE ANALYSER RESULTS.

		Vulnerabilities		
		XSS	*XSRF*	*SQL*
Scripts	*Guestbook Script*	✗	1 Medium	✗
	Login Script	✗	1 Medium	✗
	Search Script	4 Medium	1 Medium	3 High
	Upload Script	✗	1 Medium	✗
	Contact Script	5 Medium	✗	✗
	Forum Script	5 Medium	1 Medium	26 High

The source code analyser found a total of forty eight vulnerabilities. Thirty two of those were found in the forum script, eight in the search script, five in the contact script, and one in the guestbook, login and upload scripts.

Both the source code analyser and the injection manager identified vulnerabilities in the same scripts. The overall figures would suggest that source analysis is the more effective method of vulnerability detection; however, if we look closer it becomes clear that it is not as straight forward as this. Source analysis detects potentially dangerous sections of code, many of which are direct results of other vulnerabilities; this means that multiple vulnerabilities may be detected which all stem from the same security issue. Likewise, the injection manager injects a number of similar vectors which may all be successful due to an individual vulnerability; the individual vectors may also be successful on more than one input field or form but we only record that the injection was successful, not how often.

In order to evaluate the results in a more significant manner we can determine the number of individual vulnerabilities that each method detected. For the injection manager, the XSS, JS and XSRF injections all target inputs that are not correctly HTML validated so we can class these as an individual vulnerability. In contrast, the XSS and SQL vulnerabilities in the search script found by source code analysis are all derived from the same form input, and the XSS vulnerabilities discovered in the contact script are also all as a direct result of one form not being validated. This means in essence both methods discovered the same core security issues in both the search and contact scripts. Source code analysis discovered significantly more vulnerabilities in the forum script than the injection manager did. This is largely due to the amount of SQL queries used, each of which contains a vulnerability, and also leads to further vulnerabilities in the script. Although the injection manager also discovered SQL vulnerabilities, they only concerned one aspect of the script, whereas the source code analyser found them in almost all aspects of the script. The reason for the contrast is that the majority of vulnerabilities are persistent and occur on a different page to the one that was injected; this confused the injection manager into thinking they were unsuccessful. Both methods successfully discovered the same XSS flaws. Source code analysis also discovered the possibility of XSRF vulnerabilities on the forms of every script tested, with the exception of the contact script.

In contrast to the evaluation of the tool on external scripts, I also ran it on the test website that I created. This allowed me not only to measure the different vulnerabilities that each method detected, but also the deliberately created vulnerabilities they did not detect. ***Table 4*** shows the results.

TABLE 4. TEST WEBSITE RESULTS.

Test Website Page	Vulnerability Type	Injection Manager	Source Code Analyser
contact.php	None (Sanitized)	XSS	None
lines.php	XSS (GET)	None	XSS
bookings.php	XSS (Persistent, POST), XSRF, SQL	XSS, SQL	XSRF, SQL
production.php	XSS (POST), XSRF	XSS	XSS, XSRF
info.php	XSS (Query)	None	XSS

All vulnerabilities in the test website (excluding XSRF and Query Injection) were detected by the injection manager, whereas the source code analyser detected them all with the exception of a persistent XSS attack. Although the results indicate the injection manager found a vulnerability on the contact page, this actually refers to the lines page, since the vulnerable form that submits the data to the lines page is found on the contact page.

V. EVALUATION

Both methods of my solution have been a success; unknown vulnerabilities in external scripts have been identified encompassing threats from XSS attacks, XSRF attacks, Javascript attacks and SQL injections. All intentionally created vulnerabilities in the test website were detected by a combination of both methods, and no false positives were reported. Furthermore, code fixes have been suggested for these vulnerabilities with in-depth information indicating where they have occurred and what to change; the potential risk of each issue is calculated and detailed statistics about the webpages, files and their respective contents are displayed to the user. The functionality to carry out these tasks is presented in an intuitive manner and results displayed in a user-friendly interactive format. Since the source code analyser is token based we can use the same process for every token found to determine whether or not a vulnerability exists. This means any working PHP code can be used as input since syntax is irrelevant as the code is always parsed into the same types of token.

Both methods of vulnerability detection have strengths and weaknesses but I believe the best results are obtained when using them together. The injection manager produces more information about the type and format of any exploits that could be used on a given script and guarantees that any successful injections indicate a vulnerability is present. It is also very maintainable as new vectors can be added with ease, allowing the tool to stay constantly up to date with developing internet technologies. Moreover, source code analysis gives more information about the specific issue in the code itself, such as the names of the variables involved and the line numbers of the suspect code. Additionally, it can identify forms that may be susceptible to XSRF and differentiate between vulnerabilities from different form items, unlike the injection manager which only indicates if the whole form is susceptible or not. Another major advantage is that it can detect vulnerabilities from both form input and query strings. Query strings are often used solely to control a webpage and ensure it displays specific information, meaning no user input is required; this does however mean a user can modify these queries and so if left unvalidated they can be exploited. The injection manager does not inject query strings, as unlike forms, it is often unclear when or where a query string may be found. A limitation of both methods is forms which allow a user to upload a file do not get tested correctly; these forms can be abused to upload malicious programs and scripts if not specifically validated.

In reality the injection manager may be more useful as source code analysis can report false positives for some vulnerabilities. This is because the developer can implement sanitization methods in a plethora of different formats making it close to impossible to accurately detect every sanitization used. Instead, by maintaining a list of the most commonly used functions we minimize the chance of a sanitized variable being flagged. The form XSRF detection is also unreliable due to the nature of the randomness used to prevent such attacks; the majority of forms are protected by the use of unique tokens, however since they can come in any format they are difficult to detect. The current method checks if a form has a hidden item as this may be used to submit a token; this is flawed since hidden items can be used for any kind of data, and tokens can also be sent in other ways such as through the form action, cookies or session data. Furthermore, if a new method of vulnerability prevention is developed or a new type of vulnerability emerges it will require an additional amount of complicated coding to be added to the parser for it to remain up to date.

Javascript is an important aspect of modern websites; as such we must ensure that any related security issues are considered. The injection manager can inject these kinds of attacks and accurately gauge their success. However, source code analysis does not currently have the capability to detect such problems as it cannot parse Javascript code. To rectify this a Javascript parser could be implemented that works in much the same way as the PHP parser. There are further Javascript vulnerabilities that can be exploited through the modification of existing Javascript code during the runtime of a webpage; these issues are beyond the scope of this project as they would require the implementation of a real time Javascript emulator.

The system can be run on websites of any size as optimization techniques used ensure that irrelevant data is constantly being disposed. However, the web crawler currently does not differentiate between the type of webpages that it visits; this means the crawler will attempt to parse files such as images, documents and programs for links even though they clearly will not have any. This is not a critical issue but it does mean the user must wait longer when the crawler encounters one of these files. In very rare cases the web crawler can get stuck in an infinite loop. This can happen when a broken link is found that still gets rendered by the webpage being crawled; subsequent links on that page may also be broken due to a mismatch in the directory structure and local links being used, meaning each time one of these broken links is visited it links to a new broken link. The developer must first fix their broken links or modify their server settings before running the tool on the website again.

Another downside of the complexity of the PHP programming language and the many different purposes it can be used for impacts on the code fix suggestions. Whilst it is able to determine the variable that needs to be sanitized and the line number upon which it is found, it is much harder to determine the context that it is used within. Because of this a developer will almost always be in a better position to choose how to deal with the vulnerability, unless they are unaware of how to fix them. This means that the code suggestion manager is limited in the scope of its answers; it determines the most effective sanitization method to use for the particular type of vulnerability and offers that as the solution. In the majority of cases this is an acceptable solution, however in some scenarios the developer may wish to use their own custom sanitization methods.

Another limitation of the source code analyser is caused by the ambiguity of some of the tokens used by the PHP parser. For instance, when maintaining the function call stack we must add a function every time it is called, and remove a function whenever it ends. Function calls are ended with a right parenthesis, which can also be used to end mathematical equations and logical statements. If a function was popped from the stack each time this token was located a stack underflow could occur if the parenthesis was not used to end a function. In order to avoid this, left parentheses are treated the same way as functions, ensuring they are pushed and popped from the stack correctly. Unfortunately this method may still fail, along with other areas of the source code analysis, if there are syntax errors in the source code; however since the webpage will not work it is not vulnerable to attack.

The tool currently only partially supports the detection of persistent XSS attacks. Source code analysis has no means to detect such an attack; input variables are considered to be safe from XSS attacks if they are not directly output on a webpage. However, they can still be recorded in a database and extracted on another webpage without adequate XSS sanitization. The injection manager can detect such an attack if the results are displayed on the same webpage that processes the user input; otherwise it has no indication of where the results are displayed or if they were successful. For the source code analyser to detect such issues, all input variables stored in a database could be logged, along with details on the field they were stored in and any sanitization used upon them. Then in subsequent source files the code could be checked for these inputs being extracted from the database, and continue to check if they are correctly sanitized before being output. Since the injection manager's attacks are blind, the only solution for it to be able to detect such attacks would be to predefine the webpages that show the results of every form submission on the target website. This is impractical since it would be easier for the user to simply check the results manually.

When embarking on a project of this magnitude it is essential to organise each step of the process and continually maintain this as the development cycle unfolds. To help with this I initially created a specification containing a list of the objectives that my solution should cover, along with a Gantt chart indicating at which points in time I should be working on which parts of the project. I tried to allocate at least one day a week solely for project work, as well as having weekly supervisor meetings; this meant I was consistently working on the project preventing it from being pushed to one side over more immediate assignments. The meetings were very useful for this as objectives would be set for the next week and also gave me the opportunity to rectify any issues that may have occurred whilst they were still fresh in my mind. I also ensured that I kept my design and literature review documents up to date and important notes were stored in my project log. This was very useful as it meant that problems I met and information that I found surrounding the subject were not forgotten and could be revisited. I also kept a record of recent news articles reporting these kinds of attacks to help highlight how imperative this field of research is [1][2].

VI. Conclusion

Overall I believe that my project has been a success; a tool has been created implementing two different methods for detecting web vulnerabilities and my research enabled me to reason about these issues and develop strategies to help detect them. The tool itself has identified several previously unknown vulnerabilities in scripts available on the internet and has offered suggestions on how they could be solved. I have also been able to reach a conclusion to my research question; yes it is possible to accurately detect these vulnerabilities through source code analysis, as my tool demonstrates. The most efficient way of doing this is unclear; however source path analysis and simulated attacks are strong contenders. By combining the two methods a powerful tool has been created allowing a developer to identify with confidence whether or not their website is secure.

Due to the nature of these attacks and the constantly evolving web, the tool will soon be out of date; however, it has been designed to be easily maintainable and adding new detection methods should take little effort. One improvement that could be made to this would be the use of ‘definition’ files for both the sanitization functions and the injections that can be used. These files could be updated without having to recompile the tool, allowing the user to add their own custom sanitization functions and any new injection vectors that are discovered. By adding an automatic update service the tool could be constantly kept up to date.

I have found that the most prominent forms of vulnerability are XSRF attacks; few people know of their existence or danger, yet any form can be susceptible. Although my tool can detect some forms that may be susceptible, the approach is primitive and should be improved in the future. The injection manager could also be improved by injecting individual form items one at a time, with valid data entered into the other fields; this would allow the tool to pinpoint the exact item that caused any vulnerabilities to arise. Both methods should also be updated to take upload forms into account; currently neither method specifically checks if an uploaded file is valid. The injection manager could attempt to upload fake files, and the source code analyser could check for code used to validate uploaded files. A major improvement that should be made in the future is the support of more web languages, such as ASP and JSP. The tool has been designed with this in mind and should only require new modules to be written for the source code analyser.

Preventing these kinds of attacks is imperative; anyone with an internet connection can access your website, and unless these security holes are closed they could be maliciously exploited damaging your website, business and users’ experience. Many developers do not implement adequate protection when creating their websites due to either a lack of awareness or knowledge of the issues at hand; what can seem like a harmless web form or query could inadvertently jeopardize all aspects of their website. Often these attacks are most destructive when combined; for instance a simple HTML modification will most likely not cause excessive trouble, however couple that with Javascript and XSRF vectors and users sessions could be completely hijacked or private website data stolen. Both methods of identifying such vulnerabilities in a website are instrumental with each having their own strengths. Source code analysis can detect forms and queries vulnerable to XSRF attacks and provide tailored code fixes specific to individual problems. Conversely, simulated injections produce significantly fewer false positives and can easily be adapted to reflect the latest web exigencies. By using this tool and research, developers will be able to identify these problems before public release, have a better understanding of how such issues can be fixed and ultimately ensure that their websites are secure for both the owners and users alike.

REFERENCES

[1] Emery, D. (2009). *Hackers 'destroy' flight sim site* [online]. [Accessed 02nd January 2010]. Available from: <http://news.bbc.co.uk/1/hi/technology/8049780.stm>.

[2] Shiels, M. (2010). *New era for internet security amid increased attacks* [online]. [Accessed 2nd March 2010]. Available from: <http://news.bbc.co.uk/1/hi/technology/8544413.stm>.

[3] Kiezun, A., Guo, P. J., Jayaraman, K. & Ernst, M. Automatic Creation of SQL Injection and Cross-Site Scripting Attacks, *Massachusetts institute of technology, Cambridge*, September 10, 2008.

[4] RSnake (1995). *XSS (Cross Site Scripting) Cheat Sheet* [online]. [Accessed 28th August 2009]. Available from: <http://ha.ckers.org/xss.html>.
[5] Amit, Y. (2005). *Google.com UTF-7 XSS Vulnerabilities* [online]. [Accessed 28th March 2010]. Available from: <http://www.securiteam.com/securitynews/6Z00L0AEUE.html>.
[6] Achour, M., Betz, F., Dovgal, A., Lopes, N., Olson, P., Richter, G., Seguy, D., & Vrana, J. (2005). *PHP manual* [online]. [Accessed 25th August 2009]. Available from: <http://php.net/manual/en/index.php>.
[7] Zeller, B. (2008). *Popular Websites Vulnerable to Cross-Site Request Forgery Attacks* [online]. [Accessed 2nd April 2010]. Available from: <http://freedom-to-tinker.com/blog/wzeller/popular-websites-vulnerable-cross-site-request-forgery-attacks>.
[8] Hardy, N. The Confused Deputy: (or why capabilities might have been invented), *Key Logic, Santa Clara, CA*, October, 1988.
[9] Turing, A. M. Computing Machinery and Intelligence, October 1950.
[10] Ahn, L., Blum, M. & Langford, J. Telling Humans and Computers Apart Automatically, *Department of Computer Science, Carnegie Mellon University, Pennsylvania*, February, 2004.
[11] Adobe Systems, Inc. (2008). *Cross-domain policy file specification* [online]. [Accessed 3rd April 2010]. Available from: <http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html>.
[12] *XSSed - XSS (cross-site scripting) information and vulnerable websites archive* [online]. (2007) [Accessed 4th April 2010]. Available from: <http://www.xssed.com/>.
[13] Kirda, E., Jovanovic, N., Kruegel, C. & Vigna, G. Client-side cross-site scripting protection, *Institute Eurecom, France, Secure Systems Lab, Technical University Vienna, Austria & University of California, Santa Barbara, USA*, 2009.
[14] Johns, M., Engelmann, B. & Posegga, J. XSSDS: Server-side Detection of Cross-site Scripting Attacks, *University of Passau & University of Hamburg, Germany*, 2008.
[15] JupiterResearch. RETAIL WEB SITE PERFORMANCE, *Cambridge, USA*, 2006.
[16] Wassermann, G. & Su, Z. Static Detection of Cross-Site Scripting Vulnerabilities, *University of California, Davis*, 2008.
[17] *Browser Statistics* [online]. (2010) [Accessed 4th April 2010]. Available from: <http://www.w3schools.com/browsers/browsers_stats.asp>.
[18] Hayes, J. H. & Offutt, J. Input Validation Analysis and Testing, *Computer Science Department, Laboratory for Advanced Networking, University of Kentucky & Information and Software Engineering Department, George Mason University*, 2005.
[19] Hayes, J. H. & Offutt, J. Increased Software Reliability Through Input Validation Analysis and Testing, *Innovative Software Technologies, Science Applications Intl. Corp. & Information and Software Engineering Department, George Mason University*, 1999.
[20] Louw, M. T. & Kenkatakrishnan, V. N. BLUEPRINT: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers, *University of Illinois, Chicago*, May 2009.
[21] W3C (1997). *The W3C Markup Validation Service* [online]. [Accessed 23rd September 2009]. Available from: <http://validator.w3.org/>.
[22] TIOBE (2010). *TIOBE Software: Tiobe Index* [online]. [Accessed 2nd November 2009]. Available from: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.
[23] W3C (1999). *HTTP: Status Code Definitions* [online]. [Accessed 7th November 2009]. Available from: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.
[24] *PHP: List of Parser Tokens - Manual* [online]. (2000) [Accessed 27th January 2010]. Available from: <http://php.net/manual/en/tokens.php>.
[25] Havia, P. (2003). *Simple PHP Guestbook* [online]. [Accessed 27th November 2009]. Available from: <http://www.havia.net/guestbook/>.
[26] *PHP Login script tutorial* [online]. (2008) [Accessed 27th November 2009]. Available from: <http://www.phpeasystep.com/phptu/6.html>.
[27] *Using PHP to search a MySQL database and return paged results* [online]. (2002) [Accessed 27th November 2009]. Available from: <http://www.designplace.org/scripts.php?page=1>.
[28] *PHP File Upload* [online]. (1999) [Accessed 27th November 2009]. Available from: <http://www.w3schools.com/php/php_file_upload.asp>.
[29] *Contact PHP Email Form Script* [online]. (2010) [Accessed 29th March 2010]. Available from: <http://www.ibdhost.com/contact/>.
[30] *Creating a simple PHP forum tutorial* [online]. (2008) [Accessed 17th April 2010]. Available from: <http://www.phpeasystep.com/phptu/12.html>.