



UNIVERSITY OF
SURREY

Web-Based Static Source Code Analysis

Information Security MSc Dissertation

Shaun Webb

sw01229@surrey.ac.uk

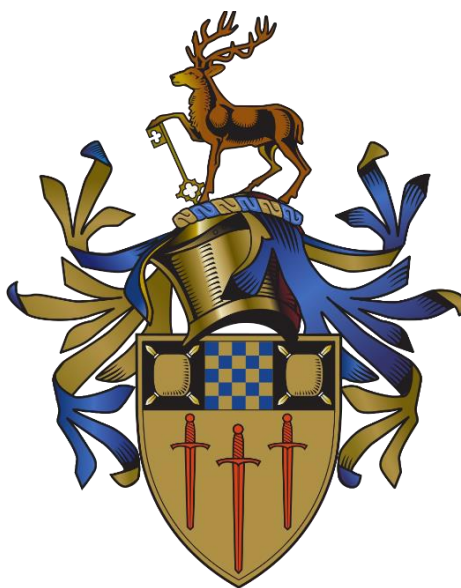


TABLE OF CONTENTS

Abstract	3
Acknowledgements	3
I. Chapter 1 – Introduction.....	4
A. Introduction.....	4
II. Literature Review	5
A. Top ten most critical web application security risks.....	5
B. Defensive programming is not enough	7
C. The use of PHP.....	9
D. Program Analysis	12
1. Program Analysis in the Security Landscape	12
2. Dynamic Program Analysis.....	13
3. Static Source Code Analysis.....	14
III. Development	31
A. Development methodology.....	31
1. Software Development Life Cycle	31
2. Chosen Methodology	32
B. Requirements gathering and analysis	33
1. Functional requirements.....	33
2. Non-Functional requirements.....	36
C. Design	38
1. Components	38
2. Architecture	39
3. Activity diagram UML.....	41
4. UI Design Mock-up.....	41
5. Summary	44
D. Implementation.....	44
1. Choice of languages and tools.....	45
2. System Functionalities	46
E. Testing	58
1. Unit Tests	58
2. UI Automation.....	60
F. Evaluation.....	61
1. Evaluating against the requirements	61
2. Evaluating against other tools.....	66
3. Deployment.....	69

G.	Maintenance.....	70
H.	Summary.....	71
IV.	Conclusion.....	71
A.	Future work	72
1.	Broader Scope	72
2.	Development work.....	72
3.	Static Analysis as part of the build process	73
4.	Advanced detection and verification	73
V.	Glossary.....	74
VI.	References	75

ABSTRACT

Web applications have become profoundly widespread and relied upon for everyday use within our society. This brings many benefits but also introduces numerous implications regarding the security of such applications and the data it manipulates. Cyber security issues are at an all-time high and continue to cause great concern. Understanding the threat landscape and reducing the attack surface is therefore pivotal in moving towards a safer internet. Training developers can be difficult and training them on security related concerns even more challenging. Tools such as static source analysers have a substantial future in web-based software development projects. They can automatically detect software security issues in code even before being deployed to a real environment. In this project a modern static source code analyser is built and evaluated against competitive tools. Research is conducted on how one would build such a tool and then further consideration looks at how they can be better integrated into the software development life cycle.

The project was motivated by first-hand industry experience demonstrating the lack of software verification tools being used.

ACKNOWLEDGEMENTS

I would like to take this opportunity to offer my sincerest gratitude to my dissertation supervisor Dr Lee Gillam for providing professional support and guidance throughout the development of this project. Dr François Dupressoir for offering personal support throughout the academic year as both a personal tutor and programme leader. Finally, thank you to my father, Leonard Webb who has suffered from a traumatic brain injury, dementia, and other issues but has provided me with a lifetime worth of support.

I. CHAPTER 1 – INTRODUCTION

A. INTRODUCTION

Cyber-crime has never been more prevalent and the need for secure systems is critical. Industries from Aerospace, Education, Banking, Healthcare and more rely on software every day. There are massive potential consequences for downtime of critical systems. The WannaCry ransomware attack infected around 200,000 computers across 150 countries, the UK's NHS were also affected causing around £92 million in damage (National Health Executive, 2018). In 2018 alone there were around 137.5 million new malware samples (AV-TEST, 2019). In the UK cybercrime now accounts for more than 50% of all crimes (Zaharia, 2019). With organisations and software being targeted on such a large scale, it evident that more needs to be done to keep our data safe.

Developers are not trained on security issues enough resulting in insecure code vulnerabilities being widespread (Zorabedian, 2017). In the 2017 Application Security Report, published by Cybersecurity Ventures (2018) it was estimated that 111 billion lines of new software code is generated by developers every year. The fast paced and large-scale nature of software makes identifying security issues ever more challenging.

Therefore, it is essential for organisations and governments to attempt to secure their systems and software. Bugs, defects and logic flaws in software are the primary cause of commonly exploited vulnerabilities. Secure coding practices can help prevent these common issues, but they require developers have enough time, training and take advantage of frequent in-depth code reviews. Most common security software issues derive from a small subset of programming issues (OWASP, 2017), other issues are specific to certain languages such as Buffer Overflows in C.

The General Data Protection Regulation (GDPR) was introduced in May 2018, since then, there has been an increase in cyber security awareness and funding for security departments in order to prevent fines being issued for inadequate data handling (Department for Digital, Culture, Media and Sport, 2019).

A range of policies and procedures need to be in place throughout an organisation in order to apply good security principles. The planning stage looks at risk identification, risk assessment and risk control strategies. The defend control strategy is the preferred approach as it attempts to directly prevent the exploitation of the vulnerability. Software development teams can take steps to help reduce vulnerabilities in software before and after deployment.

Web applications are exposed to myriad security vulnerabilities, many of which are related to malicious user string input. Web applications typically accept arbitrary user input through a variety of different sources. Cookies, URL parameters, form fields and more can all be manipulated to send malicious data. The most common web-based examples of such vulnerabilities are SQL injections, which can potentially expose database information, or cross-site scripting, which allows an attacker to execute their own code in a user's browser.

Finding vulnerabilities in code is difficult and may require extensive testing in order to identify and fix them. Large amounts of libraries from a variety of sources are commonly used, this makes finding such vulnerabilities even harder. A good patching policy can help ensure software is up to date and good internal testing with security trained engineers is needed to find most issues.

A program analyser is one type of control that can reduce the number of vulnerabilities. Although program analysis has proved to be effective at reducing vulnerabilities, they have still not gained widespread use, especially in small and medium-sized enterprises (Gleirscher, et al., 2014). Unseen security issues could have potentially been identified and prevented using static analysers. They can be used before deployment or managed in continuous integration of software development projects to aid developers by finding security issues in code.

To facilitate the detection of such vulnerabilities in web-based applications a static source code analyser has been developed for this project that employs techniques such as lexical and taint analysis. These are commonly used in programming language compilers to verify the correctness of the code, in static analysers they are used to detect security vulnerabilities or bugs. Such applications are difficult to develop and even the state of the art can't do it perfectly. They struggle and, in many cases don't even attempt to analyse obfuscated code, furthermore there are even theoretical constraints proven by Rice's theorem that prevent all issues being found in all cases. The effectiveness of static analysers is evaluated and tested against a range of code samples including obfuscated code. Static analyses provide their challenges, but they still have their value within software development projects. Conducting research on how to develop a static analyser and the techniques currently used, improvements in both detection and integration into software projects can be established.

II. LITERATURE REVIEW

A literature review is critical to understand the relevant knowledge and background information supporting the topic. It is crucial to understanding what already exists to aid the development of the project application. It also identifies a gap within the literature and industry that this section tries to address, in relation to web-based static analysis.

A. TOP TEN MOST CRITICAL WEB APPLICATION SECURITY RISKS

The Open Web Application Security Project (OWASP) is an online community that provides articles, methodologies, documentation, tools, and technologies to help improve web application security. Their most known project is the OWASP Top 10.

The OWASP Top 10 is a report that outlines the main security issues for web applications, focusing on the 10 most critical risks. Security experts from all over the world work together to devise the list. OWASP refers to

the report as an awareness document that companies should incorporate into their processes to minimize and mitigate security risks (Cloudflare, 2019).

The OWASP Top 10 2017 list can be seen below with brief explanations of each:

1. Injection

Injection attacks occur when input data from the user is not validated or sanitized and malicious code or commands are passed and executed (typically) on the server. Common injection attacks are SQL, OS command, Object Graph Navigation Library (OGNL) injections and more.

2. Broken Authentication

Broken authentication is where attackers are able to compromise authentication systems. This ranges from the use of default admin passwords, to systems that allow brute force or automated attacks to be performed, and even mismanagement of session IDs.

3. Sensitive Data Exposure

Sensitive data exposure is where data is not securely handled using secure cryptographic algorithms correctly and sensitive data such as user passwords or credit card information is exposed.

4. XML External Entities (XXE)

Web applications that parse XML input may be insecure and can be exploited by attackers. Less complex data formats such as JSON are suggested to avoid this.

5. Broken Access Control

Broken access control covers a range of issues from allowing directory traversal to insecure direct object references where modifying the URL parameters allows attackers to perform privilege escalation and even impersonate other users.

6. Security Misconfiguration

Any level of the application stack can be misconfigured resulting in attackers gaining unauthorized access or knowledge of the system.

7. Cross-Site Scripting (XSS)

XSS is the second most prevalent issue in the list. It involves JavaScript being used by attackers for malicious intent, common examples including sending other users a message with hidden code inside that steals their data. User input should be correctly validated or escaped to prevent this. Cross-site scripting has been included in every OWASP Top 10 list that has ever been devised.

8. Insecure Deserialization

Serialization takes objects from application code and converts them into a format for another purpose. Deserialization is the opposite, it converts serialized data back into objects the application can use. An insecure deserialization exploit is the result of deserializing data from untrusted sources that can result in DDoS and remote code execution attacks (Cloudflare, 2019).

9. Using Components with Known Vulnerabilities

Using legacy or unpatched systems results in systems being exposed to unnecessary risk. A patch management process can be crucial in ensuring software remains updated (Whitman & Mattord, 2014).

10. Insufficient Logging & Monitoring

Effective logging and monitoring systems can help detect breaches as soon as they occur. Organisations need to do more to reduce data breach detection time.

B. DEFENSIVE PROGRAMMING IS NOT ENOUGH

Defensive programming is a technique often used to reduce bugs present in code, like those seen in the Top 10 list. They provide good ways to improve the quality of the code and improve error handling. Good defensive programming makes bugs both easier to find and diagnose. However, defensive programming alone does not guarantee secure software. Security issues can still arise when the defensive code does not go far enough with its checks.

Consider the following JavaScript functions that display a message to a webpage.

```
function print1(message){
    document.write(message);
}

function print2(message){
    if (message == null){
        document.write("Message cannot be null");
    }else{
        document.write(message);
    }
}
```

Figure 1. Vulnerable write message example

For application layer purposes the message might not allow null values, so a check can be done to ensure it is valid (print2 function). Other examples include ensuring the message is over a certain length depending on the application requirements, etc. This type of defensive programming does not account for security related exploits. Software engineers are not trained enough to be aware of secure coding practices (Lent, 2014).

Calling the print2 function as seen below (Fig 2) will cause a basic cross-site script to be executed.

```
print2("<script>alert('XSS');</script>")
```

Figure 2. XSS attack on write message example

As briefly introduced in the Top 10 OWASP issues, cross-site scripting (XSS) is a web-based security vulnerability. It enables attackers to inject malicious scripts into web pages that other users will view. An example of such an attack is to send a message to another user on a web application, when the message is opened by the other user the script is executed and user data from the website could be stolen, content on the web page could even be manipulated. Any type of web application can suffer from a form of XSS, with varying levels of severity. Most web applications use input from a user and output it, unfortunately it is common for the input to not be correctly validated or encoded.

The example given in the print3 function below (Fig 3), the null check is still performed to handle application side error handling, then an additional line is added to ensure the input message is encoded as special characters to prevent a cross-site script attack.

```
function print3(message){
  if (message == null){
    document.write("Message cannot be null");
  }else{
    message = encodeURIComponent(message);
    document.write(message);
  }
}
```

Figure 3. XSS fix for write message example

Instead of the webpage rendering the message as JavaScript code. It changes `<script>` into `%3Cscript%3E` (Figure 4). This means that the page will now view the message as escaped text in the form of special characters instead of it being interpreted as JavaScript code.

```
%3Cscript%3Ealert('XSS')%3B%3C%2Fscript%3E
```

Figure 4. Special characters encoded for XSS write message example

A server-side language may be responsible for displaying the received message, that language will have a built-in function with similar capabilities.

C. THE USE OF PHP

W3Techs (2019) survey indicates that 79% of websites use PHP as their server-side programming language, deeming it the most commonly used language online. PHP even powers WordPress, which accounts for around 25% of the websites in use today (Thor, 2018). A WhiteSource report shows that PHP is 2nd in the total number of reported vulnerabilities per programming language (Goldstein, 2019). The report is based off the total number of CWEs for each language. C originally dates back to 1972 and has been written more than any other language, so it makes sense that it has the highest number of reported vulnerabilities. PHP focuses solely on web-based applications.

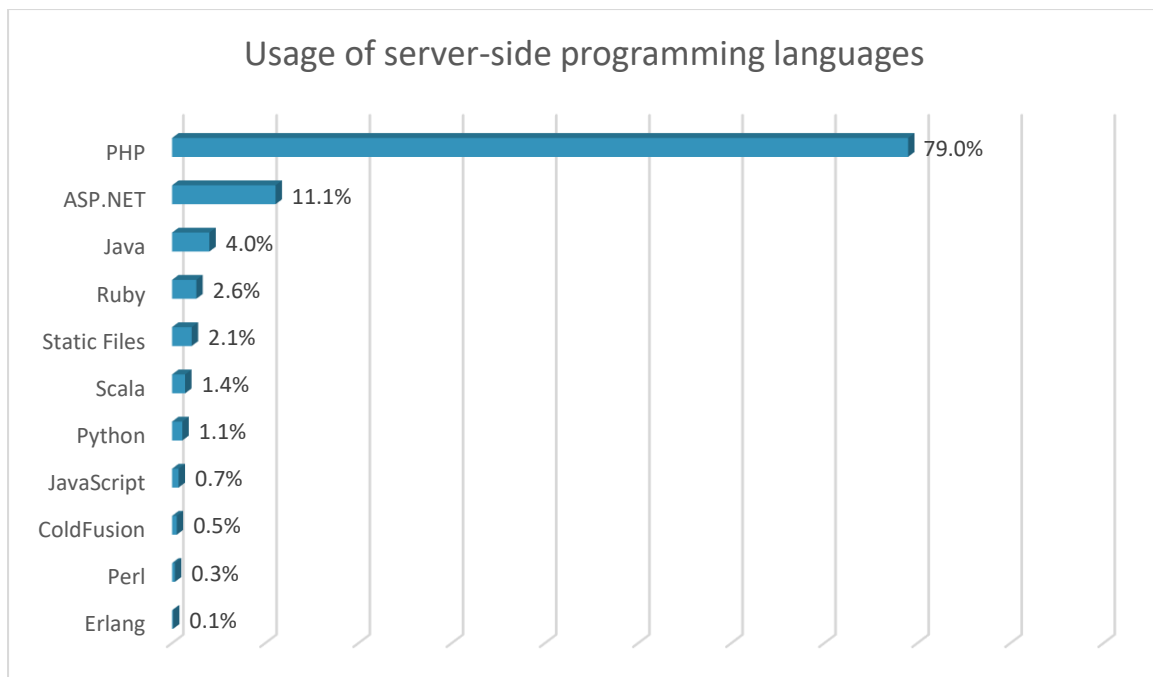


Figure 5. Usage of server-side programming languages

Other server-side languages make up for less than 0.1% and have not been included in the graph.

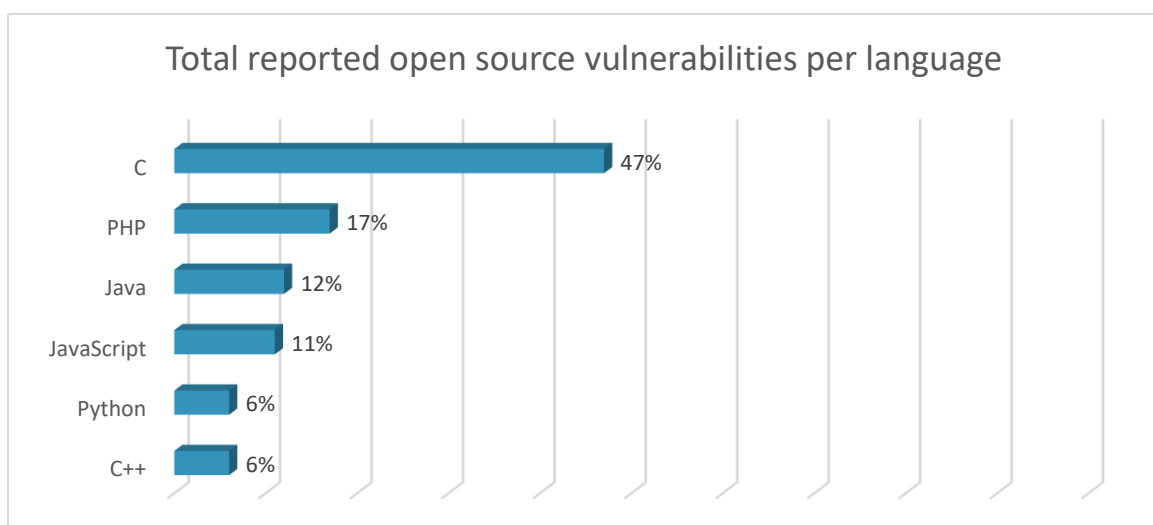


Figure 6. Total reported open source vulnerabilities per language

PHP covers all the OWASP top 10 issues and more, combined with the fact that it is the most commonly used server-side programming language with the 2nd most reported open source vulnerabilities makes it an obvious candidate to target.

PHP has grown and changed overtime rather than deliberately engineered with security goals in mind. This resulted in making writing insecure PHP applications far too easy and common. The most common pitfalls of the language have been laid out in the OWASP Cheat Sheets book (Woschek, 2015), below is a short summary of those pitfalls.

- Weak typing

PHP is weakly typed, which means that the interpreter will predict the data type required. This can cause an incorrect data type such as the string "0" being allowed instead of the integer 0. Instead the use of === should be employed more in place of == to enforce type correctness.

- Exceptions and error handling

Numerous PHP libraries report errors using warnings and do not prevent code execution.

- Configuration - php.ini

PHP behaviour is managed by a php.ini configuration file. This include things how errors are handled, making it very difficult to write code that works as expected in all circumstances.

- PHP functions

Native PHP functions like `mysql_real_escape_string` appear to provide security, but often do not deal with security issues. In later versions of PHP these functions are deprecated and eventually removed but not all organisations have good patch management policies in place, placing them at risk when they continue to use old versions of libraries and insecure code.

- Template language

PHP is essentially a template language that doesn't escape HTML by default, leaving it susceptible to cross-site script attacks.

D. PROGRAM ANALYSIS

We have seen that defensive programming is not enough and secure coding techniques are required. This is a manual process that takes time and requires developers to have extensive security and programming knowledge.

Automated program and source code analysis tools provide a way of scanning code bases for vulnerabilities.

The sections below look at the types of program and source code analysers available, the components which they are comprised of and details on what they can be used for, such as finding XSS vulnerabilities or detecting plaintext passwords left in code.

1. PROGRAM ANALYSIS IN THE SECURITY LANDSCAPE

Understanding the security landscape is important when attempting to target a specific area of security. All layers seen in the defense in depth (Fig 7) below need to be adhered to, this will reduce the overall attack surface an organisation will face. Reducing the attack surface is one of the biggest challenges organisations face (Zorz, 2019).

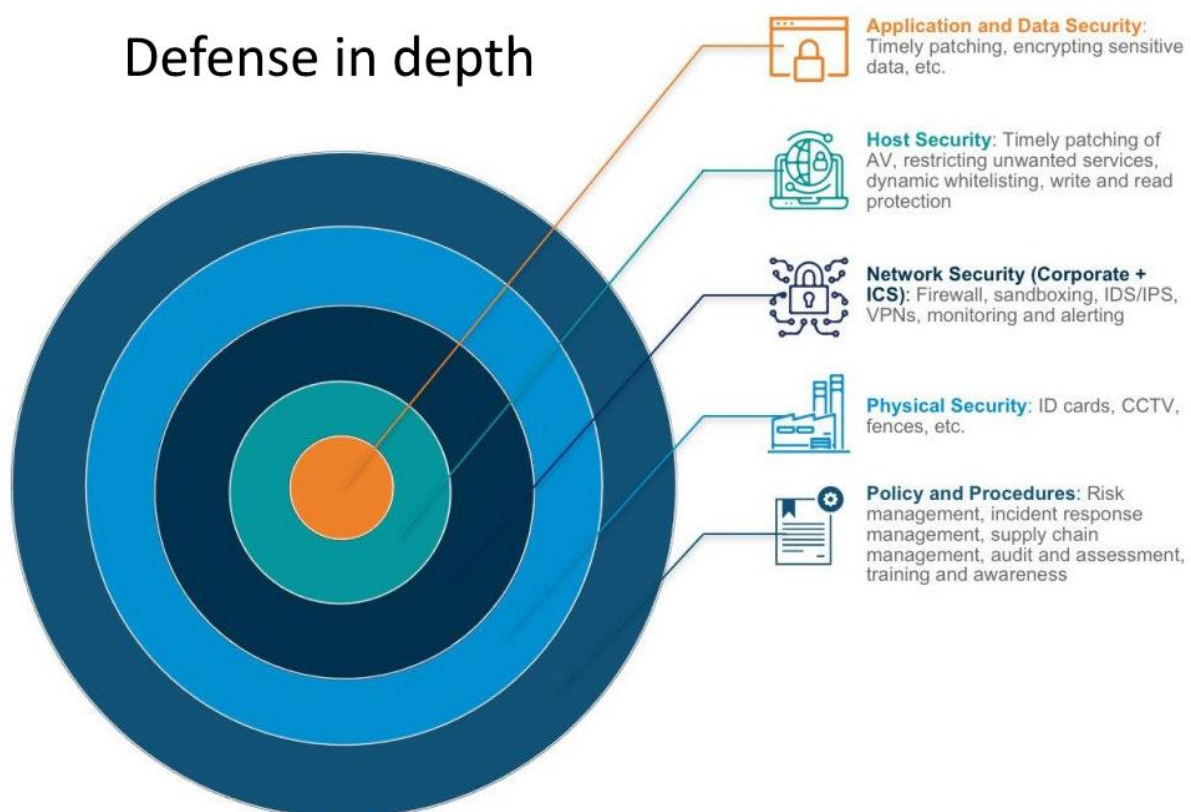


Figure 7. Defense in depth (Modern CISO, 2018)

Program analysis is located at the application layer and looks at a very specific security issue. Application security covers the measures taken to improve the security of an application, techniques to do this include discovering, fixing and preventing security vulnerabilities. Program analysis is a process that can improve application security.

2. DYNAMIC PROGRAM ANALYSIS

Dynamic program analysis is the analysis of software that is currently being executed on a system, with either a virtual or physical CPU.

Dynamic analysis targets the current instructions being executed, and as such must be executed with sufficient test inputs to broaden the range of code coverage that will be monitored. Dynamic program analysis tools may require third-party libraries to be loaded and even recompilation of program code may be needed. Dynamic testing can be performed in myriad ways from unit, integration, system and acceptance tests to the use of third-party tools. Unit tests are the most common method of dynamic program analysis. A summary of the advantages and disadvantages of this type of analysis have been outlined below (Ghahrai, 2018).

A) ADVANTAGES OF DYNAMIC ANALYSIS

- Can detect dependencies that are not possible in static source code analysis as other code bases are being relied upon using reflection, polymorphism or dependency injection. This allows for analysis of applications even when the original code is not accessible.
- Deals with real input data that should closely imitate real usage of the system.
- Runtime environment vulnerabilities can be identified and has access to the environment's full security stack.
- Can identify false negatives that were not caught by static source code analysis.
- Can be used in conjunction with static source code analysis findings.
- Supported by any program regardless of language and environment.
- Much easier to detect vulnerabilities when code obfuscation is used compared to static analysis (Moser, et al., 2007).

B) DISADVANTAGES OF DYNAMIC ANALYSIS

- May negatively impact the performance of the application so analysis should be done during the testing phase and during a suitable time for maintenance to minimize downtime.
- Cannot guarantee full code coverage as the analysis is conducted based on user or automated tests.
- Analysis tools can give a false sense of security to developers and issues can be overlooked.
- False positives and false negatives will be produced as analysis cannot guarantee completeness.

- Dynamic program analysis is only as effective as the rules and data they use to scan with.
- It is difficult to trace the vulnerability back to a specific line in the code, taking longer to debug.
- May be difficult or impossible to implement with Cloud Platform as a Service providers if binaries are proprietary and the environment is out of the developers control.

3. STATIC SOURCE CODE ANALYSIS

Static source code analysis is the analysis of software that is not currently being executed on a system.

It does this by reading and interpreting the source code or compiled code used to make the software. This checks the code against a set of rules and known vulnerabilities and attempts to detect any possible security issues. Static source code analysis tools allow developers to quickly check the security and quality of their code with little interruption or delay. The analysis should be ongoing throughout the early development stages and be used in the Software Development Life Cycle (SDLC) alongside other automation tools such as Jenkins. Security issues found can then be flagged and developers can determine if action is required to update the code. Issues can be flagged as potential security issues and only show the type of security issue it presumes it is vulnerable to, while other tools may offer a more elaborate description that may include sink traces.

A) ADVANTAGES OF STATIC CODE ANALYSIS

- Able to identify the vulnerability at the exact line in the source code.
- Can be used to train software developers on how to write secure code.
- Easy to integrate alongside developers as they write code.
- Allows for a quick turnaround of fixes that are seen in the early stages of the development life cycle.
- Allows for full code coverage regardless of the cyclomatic complexity in a single file.
- Able to detect defects such as unreachable code, unused variables, uncalled functions and more.
- Platform and compiler agnostic as the source code to make the program is being analysed rather than the program itself.
- Can be performed quickly on source code with little to no action required from the developer when testing input values.
- Easy to implement with Cloud Platform as a Service providers as source code can be checked before-hand.

B) DISADVANTAGES OF STATIC CODE ANALYSIS

- Can be time consuming to analyse the results.

- Automated tools still produce false negatives and false positives.
- Difficult to predict what the user will do or what will be passed as input, providing challenges to protect and correctly validate data.
- Can produce a false sense of security that all issues are being addressed. Such tools should only be used as a safety net and not relied upon entirely.
- Difficult to determine vulnerabilities for a wide range of platforms with high level of accuracy as some environments may be vulnerable and others may not, even with the same code. Due to the use of third-party libraries, code from other files being inaccessible, and version detection of other software may result in missed vulnerabilities.
- Difficult to near impossible to detect true code behaviour for interpreted languages when high levels of code obfuscation are used (Moser, et al., 2007).
- Difficult to prove the existence of an issue as code is not executed.

C) CHALLENGES IN STATIC ANALYSIS

Building a static analyser that is accurate, robust and diverse in its feature set is proven to be difficult. Current code plagiarism systems can't even detect code changes that have been obfuscated. Compiler techniques are very similar to static analyser systems. Compilers for modern programming languages are inherently complex. Static analysis attempts to take semantic and syntactic analysis one step further by predicting behaviour and weaknesses in code, libraries and system architecture. Server and client-side code has to be considered for web-based analysers. Currently, even the best tools in the world do not find all security weaknesses in all circumstances and there are even theoretical proofs that show why this is impossible. The section below explains some of the difficulties developers will face when building such systems.

(1) THE HALTING PROBLEM AND RICE'S THEOREM

The halting problem asks whether the execution of a specific program for a given input will terminate. The halting problem was proven to be *undecidable*¹ by Alan Turing (1937). That is, no algorithm can solve it for all programs and all inputs.

This notion of using one algorithm to analyse another is fundamental to the theory of computation (Sipser, 2012).

¹ In computability theory, an undecidable problem is a decision problem for which it is proven to be impossible to construct an algorithm that always leads to a correct yes or no answer.

This complicates any attempt to predict program behaviour, this includes the predictions made in static analysers. We can make predicting almost any programs behaviour equivalent to predicting the termination of a nearly identical program.

The pseudocode below simulates this problem.

```
if program P halts
    call unsafe()
```

Figure 8. Rice's theorem simple code example

In this example, in order to determine whether the `unsafe()` code is ever called the analyser must solve the halting problem.

Rice's theorem proves that non-trivial properties of programs are undecidable (Jones, 1997). A trivial property is one that holds either for all languages or for none (Kumar & Garg, 1994). Therefore, determining security for all types of programs is non-trivial.

Static analysers algorithms do their best to defy the undecidability of the halting problem, they attempt to predict program behaviour. As this is proved to be undecidable, static analysers cannot claim to detect security issues free of false positives and false negatives in all cases (Chess & West, 2007). This implies that static analysis is a computationally undecidable problem.

The main focus of static analysers is to highlight potential security issues or bugs in code. The fact that they are imperfect does not prevent them from having value.

(2) MODERN PROGRAMMING LANGUAGE FEATURES

The following features are just some of the challenges that modern programming languages give static code analysers (Møller & Schwartzbach, 2018).

- Concurrency
- Higher-order functions
- Recursion
- Mutable records, objects, arrays
- Integer and floating-point computations
- Dynamic dispatching
- Inheritance
- Exceptions
- Reflection
- Developer bugs

(3) CODE OBFUSCATION

Code obfuscation is the act of making code difficult to understand. This can be done for a variety of reasons such as to protect intellectual property and attempt to prevent an attacker from reverse engineering proprietary software. Malware commonly employ this technique to hide what the malicious code is really doing to the system and to try and avoid anti-virus detection tools (Sikorski & Honig, 2012). An example of such a project that obfuscates code is PyArmor.

Dynamic analysis will be affected less than static analysis as the obfuscation should not change the underlying behaviour of the program. Disassemblers can be used to dynamically analyse applications, they transfer the binary code into assembly code and the underlying behaviour can be inspected, common tools for this include OllyDbg (Yuschuk, 2014) and IDA Pro (Hex-Rays SA, 2015).

Static analysis needs to read the source code directly and obfuscation can make this task more difficult. Below is a short list of some obfuscation techniques that make static analysis difficult (Singh & Singh, 2018).

- Changing the order of code - code can be re-ordered to disrupt control flow.
- Insertion of Redundant Data - redundant data insertion can be used to trigger false positives making the results difficult to analyse.
- Encryption - code can be encrypted and decrypted upon execution.
- Oligomorphic code - a different variation of the code decryptor is generated each time it is executed.
- Polymorphic code - changes the source code upon each execution but the underlying behaviour of the application stays the same.

Deobfuscation is the act of taking obfuscated code and converting it back into its original form, or at least as closely as possible. Certain obfuscation techniques rename variables and functions so they will look different, but the underlying behaviour should still be the same even after obfuscation or deobfuscation.

A static analyser can try either:

- A. Analyse obfuscated code and determine if the new changes have now made it vulnerability to certain attacks.
- B. Deobfuscate code back into its original format to determine if the original code has security vulnerabilities that are still present in the obfuscated code.

D) TYPES OF CODE ANALYSERS

(1) BINARY OR BYTE-CODE ANALYSIS

Languages such as C are compiled, and binaries are produced that can then be analysed by code analysers. Binaries may add a layer of complexity when reading directly and trying to understand the behaviour, as developers are more likely to be familiar with higher-level source code. Once standard behaviour has been understood, analysing binaries can be easier as techniques like code obfuscation are less of an issue, due to the fact that the underlying behaviour will be the same.

Binaries produced will differ depending on the compiler and the operating system targeted, causing difficulties with broad range of coverage.

In Cloud Computing environments such as AWS Lambda, the Platform as a Service (PaaS) provider is responsible for handling validation and proprietary compilation is done and access to the binaries is not granted, in these circumstances binary code analysis will not work (Zahger, 2017). Manual compilation to simulate the cloud environment is possible but not guaranteed.

(2) SOURCE CODE ANALYSIS

Source code analysis can be applied to both compiled and interpreted languages, allowing for greater coverage of applications to be analysed.

Expert level knowledge of the targeted programming language is essential when creating static code analysers. Difficulties arise with the diverse methods that programmers use to write code, even simple instructions that do the same thing can be written differently. Programs with high cyclomatic complexity and large codebases introduce challenges when attempting to identify possible vulnerabilities. A single source code file may not be vulnerable by itself, however, once combined with other files or third-party libraries the behaviour of the program may change dramatically. Static source code analysis may not even have access to these files to determine the true nature of all the code, resulting in false negatives and false positives.

E) TECHNIQUES FOR STATIC CODE ANALYSIS

In order for static code analysers to be comprehensive, a variety of techniques can be used to maximize its vulnerability identification capabilities. The internals of static analysis tools are similar to that of compilers. Regardless of the analysis techniques used, all static analysis tools that focus on security vulnerabilities are roughly built in the same way. They all accept code, build a model to gain an understanding of what the code is doing, analyse that model with a large dataset of security knowledge, and then finally display the results back to the user.

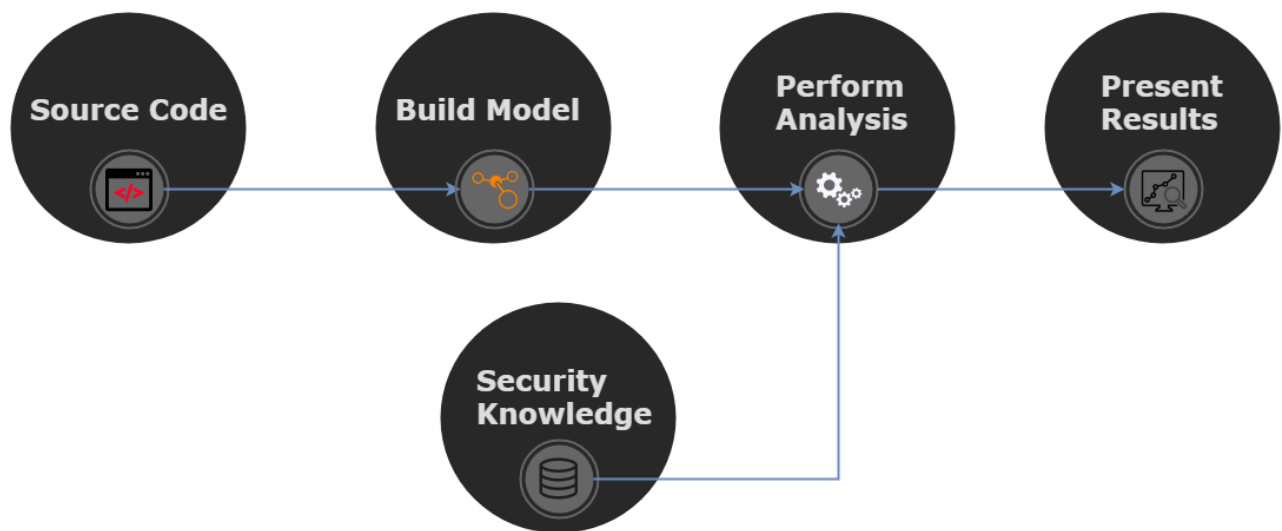


Figure 9. Diagram of the components typically required to create a static code analyser

(1) DATA FLOW ANALYSIS

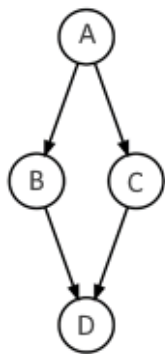
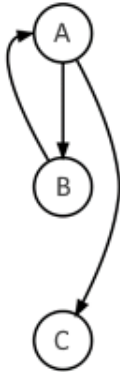
Data flow analysis is used to collect run-time information about data in the code while it is not being executed (Wögerer, 2005). It is commonly used in compilers for optimization. They examine the way data moves through programs. In the case of code analysers this is useful to understand if the variable is being used in an insecure way as the variable itself may not be insecure and the action may not be insecure, but once combined, it results in a vulnerability being exposed.

Data flow analysis is commonly used with control flow graphs to visually determine the paths such data may take through a program during its execution.

(2) CONTROL FLOW GRAPH

A control flow graph can be devised using graph notation to gain a better understanding of all the paths that might be traversed in a program. Each code block is determined by a node and the sequential path in which the execution is done can be followed, branching where conditionals determine execution of different paths. For example, a particular vulnerability may only be present following one route but not the other. Source code analysers require full understanding of the control flow to determine if vulnerabilities exist (Rao, 2019).

Table 1. Control Flow Graph with Code Examples

Control Flow Graph	Code Example
 <pre> graph TD A((A)) --> B((B)) A --> C((C)) B --> D((D)) C --> D </pre>	<pre> if (date("H") < 20) { // A echo "Have a good day!"; // B }else{ echo "Have a good night!"; // C } echo "Finished"; // D </pre>
 <pre> graph TD A((A)) --> B((B)) B --> C((C)) C --> A </pre>	<pre> \$x = 1; while(\$x <= 5) { // A echo "The number is: \$x
"; // B \$x++; // B } echo "Finished"; // C </pre>

(3) TAINT ANALYSIS

Taint analysis is used to check if a variable can be set via user input and traces them to a vulnerability checking for things such as correct sanitization or validation depending on the expected behaviour (Barbosa, 2009). For example, allowing the user to input data via HTTP GET may lead to an insecure direct object reference or even a SQL injection. Determining which variables can be tainted by user input helps the source code analyser gain a better understanding of the security of the code, however, difficulties arise when different files or functions are used as it may be unclear if the returned data is potentially input by the user.

(4) LEXICAL ANALYSIS

Lexical analysis is traditionally used in the first stage of a compiler and are essential for source code analysers. They convert source code into a series of tokens, these tokens make the analysis easier and the parsing engine understands what the syntax means.

In the popular web language PHP the function `token_get_all()` parses the given source string into PHP language tokens using the Zend engine's lexical scanner (php.net, 2019).

An OWASP (2019) example is used below to shows the output of tokenized PHP source code.

Input

```
<?php $user = "Shaun"; ?>
```

Figure 10. Input of code to be tokenized

Output

```
T_OPEN_TAG  
T_VARIABLE  
=  
T_CONSTANT_ENCAPSED_STRING  
;  
T_CLOSE_TAG
```

Figure 11. Output tokenized code

F) FEATURES OF STATIC CODE ANALYSIS

(1) VULNERABILITY DETECTION

The exact nature of which types of vulnerabilities will the analyser detect needs to be understood and developers should be able to answer the following questions.

- What languages and environments will the tool target?
- What types of vulnerabilities can it detect?
- Does it require complete code bases, or will it only work with single files?
- Can it work with only the source code or also the binaries?
- Can it be integrated into an IDE (Integrated Development Environment)?
- Will it support different programming paradigms?
- Who will use the tool and how will they use it?
- Is there a business plan for the tool?
- How will it output the results?

Once the requirements have been understood then the need for the tool will be clear and the techniques explained in "[Techniques for Static Code Analysis](#)" can be used to help design and implement the features.

(2) EVALUATING THE TOOLS EFFECTIVENESS WITH SAMPLE SETS

An effective way of testing and evaluating the effectiveness of the tool, is to use it against relevant data sets. Version control projects such as GitHub and Bitbucket are great platforms to scrape code to test with a static code analyser. The sections below explain the rationale behind why scraping code is important and simple suggestions on how this may be done with examples are given.

(A) TARGETING GITHUB REPOSITORIES

Scraping GitHub for vulnerable repositories can easily be a project on its own. Projects such as GHTorrent mirror millions of public GitHub repositories and have attempted to provide queries (Gousios, 2013), however for an attacker to be able to feasibly search for certain vulnerabilities the sample set needs to be reduced dramatically. If a specific language such as PHP or cloud environment code such as AWS is to be targeted, then only code bases that meet those requirements should be scraped.

A few automation solutions exist that can be easily be integrated into the static source code analyser or even act as a standalone application.

First web pages with relevant repositories need to be found, after that automation scraping tools can be modified to target them.

The official GitHub search page² allows for simple keywords to be searched.

🔍 Search more than 125M repositories

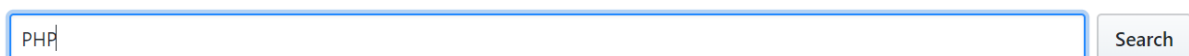
A screenshot of the GitHub search interface. It features a search bar with the text 'PHP' inside, followed by a 'Search' button. Above the search bar, it says '🔍 Search more than 125M repositories'.

Figure 12. GitHub search page screenshot

The resulting URL <https://github.com/search?q=PHP&ref=simplesearch> can be used later with automation tools to recursively scrape each repository. The sorting feature can be useful here as the “Best Match” option usually displays famous repositories that have many people working on them, repositories with this kind of expose are usually less susceptible to vulnerabilities but could still be scanned to be sure.

Another method is to view the topic or language that you wish to search through GitHub topics. The query strings can be set as seen in Figure 13 below.

² <https://github.com/search>



Figure 13. GitHub topic search URL example

The resulting page targets AWS specific projects where the main programming language used is PHP.

Amazon Web Services

Amazon Web Services is a subsidiary of Amazon.com that provides on-demand cloud computing platforms to individuals, companies, and governments, on a subscription basis.



★ Star

[Sign up for GitHub](#) or [sign in](#) to edit this page

Released March 2006

REPOSITORIES **10,736**

Language: PHP ▾

Sort: Recently updated ▾

LEARN ABOUT AWS

[aws](#)

[aws.amazon.com](#)

[Wikipedia](#)

jason4151 / php-crud-rds

A simple PHP application used to demonstrate a connection with an Amazon RDS MySQL database.

[php](#) [aws-rds-mysql](#) [aws](#)

● PHP Updated 2 hours ago

RELATED TOPICS

[amazon](#)

[See more topics](#)

abetomo / Convert-Athena-QueryResults-to-Array ★ 1

Convert AWS Athena QueryResults to Array.

[aws](#) [athena](#) [convert](#) [array](#) [query](#) [results](#)

● PHP Updated 2 days ago

luke908 / php-mini-aws-elasticsearch ★ 1

A simpler alternative to AWS SDK miniES is a PHP interface to AWS API. It is easy to learn, use and modify, unlike ot...

[php](#) [aws](#) [elasticsearch-client](#) [amazon-web-services](#) [amazon](#) [aws-es](#) [aws-sdk-minies](#)

● PHP Updated 5 days ago

Figure 14. GitHub AWS Topic Page for PHP

This URL seen in Figure 13 makes finding potential vulnerable repositories with automated tools much easier as the sample set has been reduced dramatically removing repositories that are outside of our requirements.

(B) AUTOMATED SCRAPING

A range of automated tools exist for web scraping depending on the language or environment a developer wishes to use. A few potential solutions have been briefly highlighted below. Many other variants and techniques can be used instead of the discussed methods.

(I) REQUESTS FOR PYTHON

Requests for Python is a module that simplifies HTTP requests, so minimal code is required.

```
import requests
print (requests.get('https://github.com/topics/aws?l=php&o=desc&s=updated').text)
```

Figure 15. Requests scraping example

Using the simple snippet of code from Figure 15 the HTML for the AWS topic page is obtained. The text then needs to be parsed to obtain the repositories URL, these can be collected and requested again. A method of saving and scanning the targeted code in the repositories needs to be devised using a similar technique.

(II) SELENIUM

Selenium is perhaps the most powerful automated tool, allowing for even UI automation and testing. For our example we only need simple GitHub pages and repositories to be copied so it is cumbersome when used only for these purposes, as it requires integration with a browser on the host system (Mitchell, 2015).

(III) SCRAPY

Scrapy is a python-based tool targeted more towards scraping than Requests as it does much more than simple HTTP GET and POST requests with parsing tacked on. Scrapy provides all of the functions needed to parse data from HTML easily, it automatically preserves sessions, follows redirects, attempts failed requests and more (Scrapy, 2018), making it an easy to use all in one solution for web automation.

(IV) POWERSHELL

PowerShell is a powerful scripting language built into Windows, it can even be used to invoke web requests and perform basic automation (Truher, 2019). The script in Figure 16 was created to demonstrate a single method on how the GitHub topic page and repositories can be scraped.

```

$R = (Invoke-WebRequest -UseBasicParsing -Uri "https://github.com/topics/aws?l=php&o=desc&s=updated") # Invoke request to the GitHub Topic
$links = $R.Links | Where-Object "data-ga-click" | Select-Object href -ExpandProperty href # Get page links

$found = 0
$end = $false #when to end the search

$gitrepos = New-Object System.Collections.ArrayList
foreach ($link in $links) {
    if ($found -gt 1 -and $link -match "https://github.com"){
        $end = $true
    }
    if ($end -eq $false -and $found -gt 1 -and $link -notmatch "/stargazers"){
        $gitrepos.Add("https://github.com" + $link) > $null
    }
    if ($link -contains "/join?source=header"){
        $found ++
    }
}

$gitrepos

```

Figure 16. PowerShell scraping example

(3) FIND PASSWORDS/KEYS IN CODE, FIND ATTACKS

There are a variety of software engineering processes that can identify the previously mentioned issues, such as extensive code reviews, unit tests, manual testing, security framework tools and more. All of these rely on the fact that the developers are trained in security techniques and that the tools code covers and correctly identified the exploit. Common tools use dynamic code analysis, where the application is running live and the program is tested from a user's point of view.

Static source code analysers have the capability to detect a wide range of different exploits including the detection of secret keys and credentials in source code. Insecure use of cryptography is difficult to correctly identify but the use of known weak cryptographic algorithms can be highlighted. Hashed passwords hard coded in source code can be checked against known rainbow tables to verify if the hash is already known and suggestions can be made to the developers or how to correctly use credentials in programming projects or suggest the use of higher entropy passwords.

Source code analysers can be used to check for API keys and passwords left exposed in code. A range of ways to detect sensitive keys in GitHub and code directly are explained in the sections below.

(A) KEYWORD SEARCH

Searching for keywords in GitHub code is made easy by the search feature built into the website. It allows for a specific keyword to be searched in 125 million public repositories (GitHub, 2019).

This method was used by hackers in 2013 to steal SSH keys looked for files that were named `id_rsa` or contained the string "BEGIN RSA PRIVATE KEY" which is used in private key files when generated by `ssh-keygen`. At that time GitHub suspended its search function temporarily (Sinha, et al., 2015). The method is still quite effective albeit a slow and methodical process in determining if the keys work. Targeting the most recently indexed results lowers the chance that the key no longer has permission. Users working on small

projects are likely to think that they will not be targeted and could leave sensitive data hard coded in source code.

The RSA private key search in code returns over 1.5 million results (Fig 17).

<https://github.com/search?o=desc&q=-----BEGIN+RSA+PRIVATE+KEY-----&s=indexed&type=Code>

Showing 1,502,818 available code results ⓘ

Sort: Recently indexed ▼



WitherZuo/WitherZuo.github.io – y

Showing the top seven matches Last indexed 2 minutes ago

```
1  -----BEGIN RSA PRIVATE KEY-----
2  MIIIEowIBAAKCAQEAn1/K/vARVXrpmZ7vPyhwE2wbVzk/3mdMoTBkvszZloKheHN
...
25  imtAdKJh4yf0JDpWr-jDOmrm1KZAMMEf2cDeRbfdNdZ038g4T814K2EN33kktzap4
26  fhP9pWiQCEicIAqDpr4QW+K5Q9x9x/g3zUtLLGnqfKRHeZ2cjizp
27  -----END RSA PRIVATE KEY-----
```

Figure 17. Search result RSA Private key example

<https://github.com/search?q=BasicAWSCredentials&type=Code> can be used to find the keyword

“BasicAWSCredentials”, a common keyword used in code for the AWS SDK (Sinha, et al., 2015).

Showing 40,720 available code results ⓘ

Sort: Best match ▼



kmcowan/fusionj – S3Test.java

Java

Showing the top two matches Last indexed on Jul 6, 2018

```
9  import com.amazonaws.auth.AWSSStaticCredentialsProvider;
10 import com.amazonaws.auth.BasicAWSCredentials;
11 import com.amazonaws.regions.Region;
...
22 public class S3Test {
23
24     public static void main(String[] args){
25         try{
26             /* BasicAWSCredentials awsCreds = new BasicAWSCredentials("AKIAJ46GYE22IBBHTSYQ",
                "j+dj/hw+DsZOTqNxd1QowGJecShTVL7NniYio4D");
```

Figure 18. Search result AWS Credentials example

Other search terms can target specific frameworks that are known to be implemented incorrectly. <https://github.com/search?l=PHP&q=password+.co.jp+cake&type=Code> uses the search terms “password”, “.co.jp” and “cake”. Cake represents CakePHP, a PHP framework commonly used in Japan. Hundreds of emails with working passwords are immediately visible in PHP config files. During the research on this project, working passwords exposed in source code were detected and authors were notified.

Outside of GitHub, code can be simply searched with the built-in find feature supported by most Integrated Development Environments (IDE), a keyword can be performed recursively in directories to find a match.

The keyword search method has a high false positive rate as it does not guarantee that the key or password is used directly in code, and it may reference other parts of code that are not directly related directly. It also relies upon knowledge of the SDK, library or framework to know what to search for, for example the AWS SDK has multiple implementation methods that operate differently, and a single keyword search will not cover all cases. Static analysers can look for these specific keywords in code and highlight potential data leakage.

(B) PATTERN-BASED SEARCH

Another method is to use pattern-based matching to search for keys. The previous method of using the official GitHub search page does not work here as searching by regular expression does not work, so one of the previously mentioned GitHub scraping solutions would have to be used.

Once code has been scraped literal strings can be searched using a predefined regular expression that is known to match the keys used for a particular API provider. A simple pattern-based search can be much more effective than keyword searching but is still prone to false positives. Explicit letter sequences such as “AKIA” in Amazons Web Services Client ID allow for an improved detection accuracy rate.

Viennot, et al. (2014) and Sinha, et al. (2015) provide enough regular expressions to build a table that can be used to detect a variety of service providers API keys in code (Table 2).

Table 2. Regular expression examples for pattern-based search

Service Provider	Client ID	Secret Key
Amazon AWS	AKIA[0-9A-Z]{16}	[0-9a-zA-Z/+]{40}
Bitly	[0-9a-zA-Z_]{5,31}	R_[0-9a-f]{32}
Facebook	[0-9]{13,17}	[0-9a-f]{32}
Flickr	[0-9a-f]{32}	[0-9a-f]{16}
Foursquare	[0-9A-Z]{48}	[0-9A-Z]{48}
LinkedIn	[0-9a-z]{12}	[0-9a-zA-Z]{16}
Twitter	[0-9a-zA-Z]{18,25}	[0-9a-zA-Z]{35,44}

These regular expressions can then be used to identify a variety of API keys in source code and even expanded to find credit card details.

(C) METHODS TO REDUCE FALSE POSITIVES

Improving keyword search is as simple as ensuring the list of keyword candidates are likely to produce matches. This will vary by language, library and SDK used but in conjunction with a large data set useful keywords can be collated. By improving the keywords used in the search, false positives can be reduced but not completely removed.

Pattern-based search is still prone to producing false positives, however combined with simple heuristics they can be improved dramatically.

Viennot, et al. (2014) used a technique to look for a matching Client ID and Secret Key that occurred within 5 lines of each other. This works well when API credentials are hard coded, as they are usually close to one another, but this method fails when an ID and key are far apart.

G) COMPARISON OF THE EXISTING TOOLS

Table 3. Comparison of the existing tools

Tool name	Languages supported	Language developed in	License	What it offers	Notes
Bandit	Python	Python	Apache	Shows severity and confidence of detected issues.	Output not perfectly displayed in PowerShell (Windows). Progpilot shows display better in CLI.
Progpilot	PHP	PHP, Bash	MIT	Command line outputs array with JSON, links to CWE IDs.	Simple to use CLI. Output easy to see issues with CWE.
Flawfinder	C, C++	Python	GPL	CLI or formats for HTML. CWE links for known exploits.	
Codacy.com	Various	Unknown	Proprietary	Webpage scans GitHub repository and produces a report. Cannot see how it does this.	Displays a nice breakdown of potential issues split by category, even analyses cyclomatic complexity.

Facebook Infer	Java, C, C++, and Objective-C	OCaml	Proprietary	Wide range of features. Comments are added in code at specific lines to help developers fix bugs.	No support for web languages such as JavaScript and PHP.
IBM AppScan Source	Various	Unknown	Proprietary	Output shown in a GUI, UML-like, even shows risk assessment matrix for each vulnerability based on severity and likelihood.	Unable to test (Proprietary)
RIPS (Community Edition)	PHP	PHP	GNU	Runs on a PHP webserver. Shows sink trace.	
VCG (VisualCodeGrepper)	C#, C++, Java, PHP	C#	GNU	List of potential issues shown in GUI with description.	Easy to use. No longer maintained. Does not find new issues including potential crypto issues (eg: use of MD5). Does have a high accuracy rate of detecting PHP XSS from initial testing.

Each tool works in a similar manner and results are presented with slight differences. The tools provide the source code to read from, then analysis is performed, and the user is shown the results. Results are presented with the line number, the type of vulnerability and sometimes the related CWE ID.

Some relevant tools detection capabilities are compared later in the [“Evaluating against other tools”](#) section.

H) STATIC ANALYSIS AS PART OF THE CODE REVIEW PROCESS

A static analyser needs to be used correctly to be effective in identifying security issues from software development projects. A static analyser can be integrated into a code review cycle easily. The system can assist

the reviewers and essentially acts as a third-party verifier of the code. The steps below highlight how the static analyser can be integrated into a code review process.

1. Establish Goals

In the initial step a well-defined set of security goals should be devised. This will help prioritize the code that will be reviewed and gain an understanding of the criteria used to review it. Assessing the most likely software security risks the project faces will help when creating goals. The code reviewers need to be well educated and trained on a range of security issues, such as those highlighted in the OWASP Top 10. High-level descriptions can help ensure they have a good understanding of the purpose of the code.

2. Run the static analysis tool

The next step should be to run the project code against the static analyser tool, it is important to remain attentive of the goals that were devised. If the tool allows for specific issues or warnings to be suppressed it should be done during this step. If certain security issues are becoming more common place it would be a good idea to extend the code of the static analysis tool. Adding new rules will allow for detection of new security issues and allow for current issues to adapt to any changes that may occur in the future. These changes may come from updates to the browsers themselves, differences in how the latest ECMAScript version works to understand new JavaScript syntax and more.

3. Review code using the results from the static analyser

The reviewers should go through the results presented by the analyser tool. Vigilance is required as false positives are possible, and issues not shown by the tool may still exist. The tool is simply that, a tool to help find issues that developers may have missed. It does not guarantee security, it simply acts as a safety net, detecting issues developers may have overlooked. Reviewers also need to be aware of semantic and syntactic bugs, not only security issues.

4. Make fixes

When the developers make fixes, it is important that security matters to them. The code changes need to be implemented correctly. Developers should not get trapped into the mentality of quickly trying to fix bugs on a checklist without due diligence, resulting in the same issue reoccurring because it wasn't fixed correctly the

first time. They need time to respond to the feedback from the code review, then plans to correct and secure the code need to be devised and implemented.

III. DEVELOPMENT

A. DEVELOPMENT METHODOLOGY

This section explores different software development methodologies and how they can be employed. Considerations must be done as this type of project brings great difficulty, due to the technological and time limitations. There are only a limited number of similar tools that exist, and they can only solve a subset of the problems. Vulnerability detection with source code obfuscation brings immense difficulty, even the best in the world cannot do it perfectly (Schrittwieser & Katzenbeisser, 2011). Therefore, it is crucial to select a suitable methodology to increase the chances that this development project is successful.

In later sections code that is used for demonstration purposes is presented as such

Example code

Whereas code used in development project directly is presented in a dark IDE theme.

1. SOFTWARE DEVELOPMENT LIFE CYCLE

The Software Development Life Cycle (SDLC) is a process used in the software industry to define each step of the software development stages, from planning, implementing, testing and the maintenance of the project.

The SDLC phases seen in Figure 19 will be used to direct the development of this project, starting with the requirements gathering and analysis phase.



Figure 19. SDLC Phases

The waterfall model was the first process model to be introduced. Each phase must be completed before the next phase can begin and no phase overlaps. Various methodologies have evolved since the first and oldest waterfall model, including agile, spiral, and v-model methodologies (Existek, 2017).

The term agile was popularized by the famous Manifesto for Agile Software Development (Beck, et al., 2001). The agile model understands that every project needs to be handled differently. An iterative approach is taken that allows for software features to be delivered after each iteration. Each build is incremental in terms of features and the final build will have a full set of complete features ready for the end product.

2. CHOSEN METHODOLOGY

An Agile approach was taken due to the limited time constraints of developing an application that is innovative and pushes the boundaries of what is currently possible.

An agile approach breaks the product into small incremental builds, these builds would allow for new features to be worked on. The initial main application will likely take time to setup but once the implementation stage has been reached and once the main backend of the application has been created then it should allow for features to be added in a modular fashion. The new features could target new vulnerabilities or provide additional functionality for the application. This would allow for the features to be added in a realistic manner based on the time constraints of the project. These time constraints will dictate the number of vulnerabilities that can be targeted and would allow flexibility if a feature was to be dropped for another.

B. REQUIREMENTS GATHERING AND ANALYSIS

Requirements gathering is typically the main phase that project managers and stakeholders focus on. It is important to understand what is the problem that the system is trying to solve, who is going to use the system, and what data will the system output.

Once the requirements have been gathered, they are then analysed to ensure they are valid and the possibility of incorporating them into the system is considered. Finally, a requirements specification is formally created that is used to guide the next phase.

1. FUNCTIONAL REQUIREMENTS

Functional requirements describe the features that are required in the software.

The requirements for this project are devised based on the need for improved web security with consideration of the OWASP Top 10 critical issues while also briefly following the NIST 500-268 specification which outlines the main areas that should be focused on when developing source code security analysis tools. The decision was made to focus on static source code analysis over dynamic analysis, due to the prevalence of dynamic tools, and the seemingly great areas of innovation that are available in static source code analysis despite the challenges.

The static source code analysis tool should at a minimum do the following:

- The software must be able to accept as an input compatible source code.
- Identify software security vulnerabilities in source code listed in Table 4.
- Report the security weaknesses that are identified, describe what kind of weaknesses they are, and finally determine the line number of the issue in the code.
- Identify weaknesses despite the presence of coding complexities listed in Table 5.
- Have an acceptably low false positive rate.

Optionally the tool should:

- Produce an easy to digest web-based report.
- Allow specific vulnerabilities to be suppressed by the user so they do not appear in the report.
- Attempt to find hash values original value using rainbow tables.
- Use the Common Weakness Enumeration (CWE) number beside the weakness it reports.
- Support obfuscated code analysis.
- Suggest a secure code alternative for the security issue found.

Table 4. Source Code Weaknesses

Name	CWE ID	Description	Language(s)	Relevant Complexities
Input Validation				
Basic XSS	80	Inadequately filters an input which allows a malicious script to be executed and passed to another client.	PHP, JavaScript	Taint, scope, local control flow, loop structure, obfuscation
SQL Injection	89	Unfiltered input used directly to perform an SQL command.	PHP, SQL	Taint, scope, local control flow, loop structure, obfuscation
Trust of insecure variables				
External Initialization of Trusted Variables	454	In PHP HTTP_ variables can possibly be modified by the client and modified variables could be used for privilege escalation or other attacks.	PHP	Taint, obfuscation
URL Redirection to Untrusted Site	601	Redirecting users to a URL that relies upon a variable such as a HTTP GET parameter puts users at risk of being directed to malicious/phishing websites.	PHP, JavaScript	Taint, scope, local control flow, obfuscation
Data exposure				
Storing Passwords in a Recoverable Format	257	Passwords that are stored in a recoverable format, such as encoding provide no benefit over storing as plaintext.	PHP	Scope, obfuscation

Reversible One-Way Hash	328	Hash functions that are proven to be insecure, through reversibility, collisions and lack of salting that makes rainbow tables easy to use.	PHP	Scope, obfuscation
Use of Hard-coded Cryptographic Key	321	Use of keys hard coded in text, including API keys for providers like AWS.	PHP, JavaScript	Scope, obfuscation
Exposure of Private Information	359	Exposure of private information such as passwords and credit card details	PHP	Scope, obfuscation

Table 5. Source Code Complexities

Complexity	Description	Enumeration
Taint	Attackers can taint user input with malicious data.	HTTP GET/POST parameters/body, URL Path, Cookies
Local control flow	The order in which code is executed creates complexities. Vulnerabilities may only appear when one branch of code is executed and not the other.	If, switch, goto, function calls, loops, recursion, exceptions
Loop structure	The type of loop construct where the weakness is located.	Do, while, foreach, for
Scope	The scope of the control flow related to the weakness.	Local, global, in other files/libraries
Obfuscation	Code that has been deliberately modified to make it difficult for humans to understand. Used to hide malicious code.	Various techniques

Many of the OWASP Top 10 most critical web application security risks are chosen. Targeting the main issues gives the system a higher chance of detecting security vulnerabilities as these can be some of the most prevalent issues, the need for the system is also increased as these are the most critical issues right now. No tool checks for all weaknesses in the CWE. Some of which are hard to define, like leftover debug code (CWE ID 489). Some of the issues vary considerably with each bringing different coding complexities providing unique challenges.

The main issues covered are:

- Injection
- Broken Authentication
- Sensitive Data Exposure
- Broken Access Control
- Cross-Site Scripting (XSS)

Other issues targeted closely match others in the Top 10 list but were not deemed close enough to be counted. The “Using Components with Known Vulnerabilities” issue should be different to “Using Commonly-Known Vulnerable Protocols and Hashing Functions” such as MD5, although libraries which use an insecure cryptographic algorithm may be categorised as a component with a known vulnerability such as collision attacks.

Obfuscation code should be analysed. If the obfuscated code itself cannot be analysed or no issues were found then the system should attempt to deobfuscate it and then retry the analysis.

2. NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements describe how well certain functions should perform in the system.

The main non-functional requirements have been outlined below in Table 6.

Table 6. Non-functional requirements

Requirement Type	Description
Usability	The system should provide a simple to use UI and easy to digest report of the security issues found.

Compatibility	PHP files that also have JavaScript in them should be supported.
Accuracy	The system should aim for a high level of accuracy 70%+, with few false positives.
Scalability	The system should consider the need for scaling using technologies like the cloud.
Performance (speed)	Analysis of code should aim to be completed around (lines of code / 10) in milliseconds. Each case may vary considerably. But in general, the user should not wait for long and feedback should be presented on the screen.
Performance (size)	The system should be able to analyse large files of around 100k+ SLOC.
Maintainability	Once the main backend for the system has been made, it should allow new features to be added easily in a modular fashion. Use of documentation aids this process.
Completeness	The system should be a proof of concept and does not need the full feature set of a complete system, especially given the time constraints.
Portability	The system should work on a range of different platforms from Windows to Linux.
Reliability	The system should not fail, or at least report an appropriate error message.
Security	The system is designed to be ran only by the developers as they are giving full access of the code to the system. Should only be used in an environment independent of other code bases.
Documentation	Documentation should cover both how to use the system and a basic overview of the code which aids maintainability.

C. DESIGN

The design phase of the SDLC creates a high-level design of what the software should accomplish in order to meet the requirements. These designs can range from cost estimations such as COCOMO to MVC design patterns and even architectural designs that describes how parts of the system will communicate.

1. COMPONENTS

Literature review research has shown that static source code analyser components should be comprised of:

1. Source Code Input

Source code should be in a target language with security vulnerabilities exposed so that it can be passed to the next stage. It was shown in Figure 5 that PHP makes up around 79% of the worlds most used server-side programming languages with many vulnerabilities reported. This also helps improve the security knowledge required in the third component. This dictated the reason to use PHP over other languages. JavaScript inside PHP should also be supported. A web-based UI can be used to upload the source code into the program, this allows for greater flexibility when supporting multiple operating systems. Additionally, GitHub repositories can be scraped and checked.

2. Program Model

The static analysis tool needs to be able to parse and transform the code into a program model. The techniques used for this are very similar to compilers. Many static analysis techniques were developed by researchers working on compiler technology (Aho, et al., 2006). Lexical and taint analysis are the 2 most obvious components that should be used. Lexical analysis will make understanding the code easier by tokenizing the source code, allowing for it to be more easily understood. The Zend engine is the open source engine that is used internally to interpret the PHP language. It is possible to use the engines tokenizer component to provide a list of parsed PHP tokens (php.net, 2019). Taint analysis should be done on all of the possible "T_VARIABLE" tokens to determine where its value came from.

If time allows then data flow analysis with control flow analysis could be implemented to better determine and understand the flow and execution of the code. These will improve the accuracy of the analyser but are not required to make the analyser work, as such they have a lower priority and they can be decided upon later.

3. Perform analysis with security knowledge

The source code can be passed into the program and analysis can be performed. The security knowledge can be built using known exploits. The online CWE MITRE database for PHP lists known

exploits and even provides code examples. The main security issues chosen from the Top 10 OWASP list can be found on CWE³.

These code examples can be used when developing and testing the application. To further improve the accuracy of the results certain security issues can be confirmed. For example: AWS credentials found in code can be used to try and connect using the AWS CLI. Hash values can be checked online against rainbow tables to attempt to get the real input value.

4. Present results

The same web-based UI that was used to upload the source code should be used to present the results. The UI should show the following about each security issue it finds:

- Line – The line number and preview of code on that line where the issues were found
- Type – The token type should be displayed which briefly shows what is vulnerable
- Value – The value should highlight a small part of the code to show what is vulnerable
- Description – An explanation of the issue that includes the type of vulnerability
- Severity – A rating that estimates how severe the vulnerability is. 1 is the lowest severity rating and 5 means the highest. Values are determined by the CWE likelihood and impact.
- CWE ID – A CWE ID will help the developer learn more about the issue and better determine how to fix it
- Suggestion – A suggestion will show alternate secure code that could be used instead, useful to train developers.

2. ARCHITECTURE

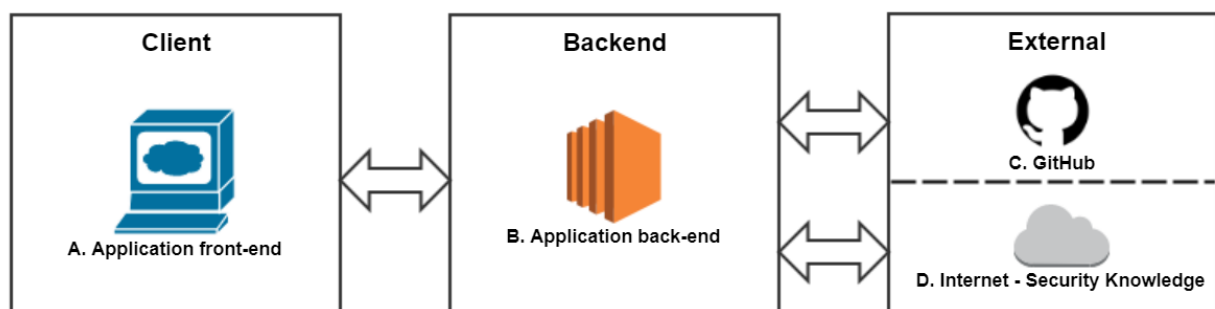


Figure 20. System architecture design

- Client
 - A. Application front-end

Web-based application is accessed by the client and source code selected by the user is uploaded to the backend, or a GitHub repository URL is given and passed to the server.

³ <https://cwe.mitre.org/data/definitions/1026.html>

- Backend
 - B. Application back-end

The backend of the web application is hosted in the cloud such as AWS EC2. Using a cloud environment helps prove the viability of the application in the real-world. The backend then communicates with GitHub if a URL was given or it will use the source code if uploaded directly. The backend communicates with various online sources to aid the program analysis and improve the accuracy of the vulnerability detection, using rainbow tables, etc. A report is then sent back to the client and results are displayed in the web browser.
- External
 - C. GitHub

GitHub is used to scrape repositories that the developer requested. Repository data is given back to the backend and then passed to the client. The client chooses which file should be analysed, then normal behaviour is resumed by the backend to perform analysis.
 - D. Internet – Security Knowledge

Various sources online can be used to back up the program analysis and improve the accuracy of the vulnerability detection.

3. ACTIVITY DIAGRAM UML

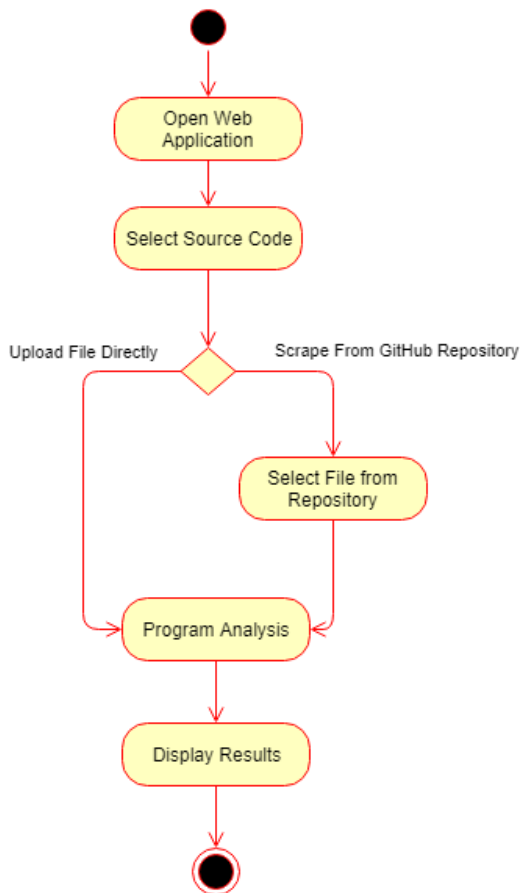


Figure 21. Activity Diagram UML

- **Open Web Application**
The user loads the application.
- **Select Source Code**
The user chooses the file to be uploaded.
- **Conditional**
File is uploaded directly, or the GitHub repository information is downloaded.
- **Select File from Repository**
A list of compatible files found in the repository are listed, the user selects one.
- **Program Analysis**
The program performs analysis on the source code to find vulnerabilities. In some cases, extra steps may be taken to confirm with security knowledge bases online, this is not shown on the activity diagram.
- **Display Results**
The user is presented with the results from the analysis.

4. UI DESIGN MOCK-UP

UI Design mock-ups allow for the user interface to be designed before-hand. Any required changes that are detected at this stage are quick and easy to change.

The user of the system will go to the home screen (Fig 22) and select a file to be analysed by the system. A compatible file (PHP) can be uploaded directly to the server or a GitHub repository URL can be given. If a URL is given, then the repository data is scraped, and the user will be presented with a different screen (Fig 23). Once

a file has been uploaded or selected from the GitHub repository the user is finally passed to the report results screen (Fig 24).

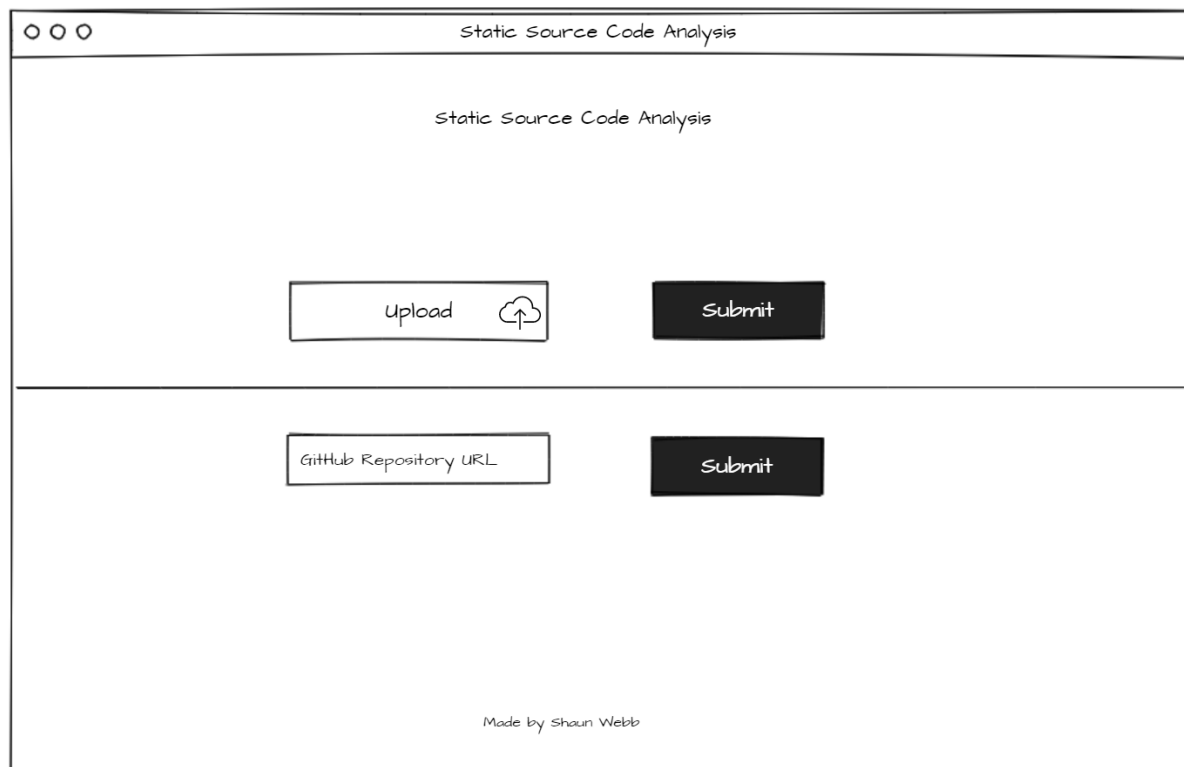


Figure 22. Homepage UI mock-up

The user is shown a GitHub repository screen if they chose to give a URL instead of uploading a file directly. The directories and relevant compatible files for the analyser will be shown, as the system is limited to only PHP files then only files which file extension ends with ".php" will be displayed. If a directory is selected, then the same screen is shown but updated with the relevant files for the selected directory. Once a PHP file is selected this file is then passed to the analyser.

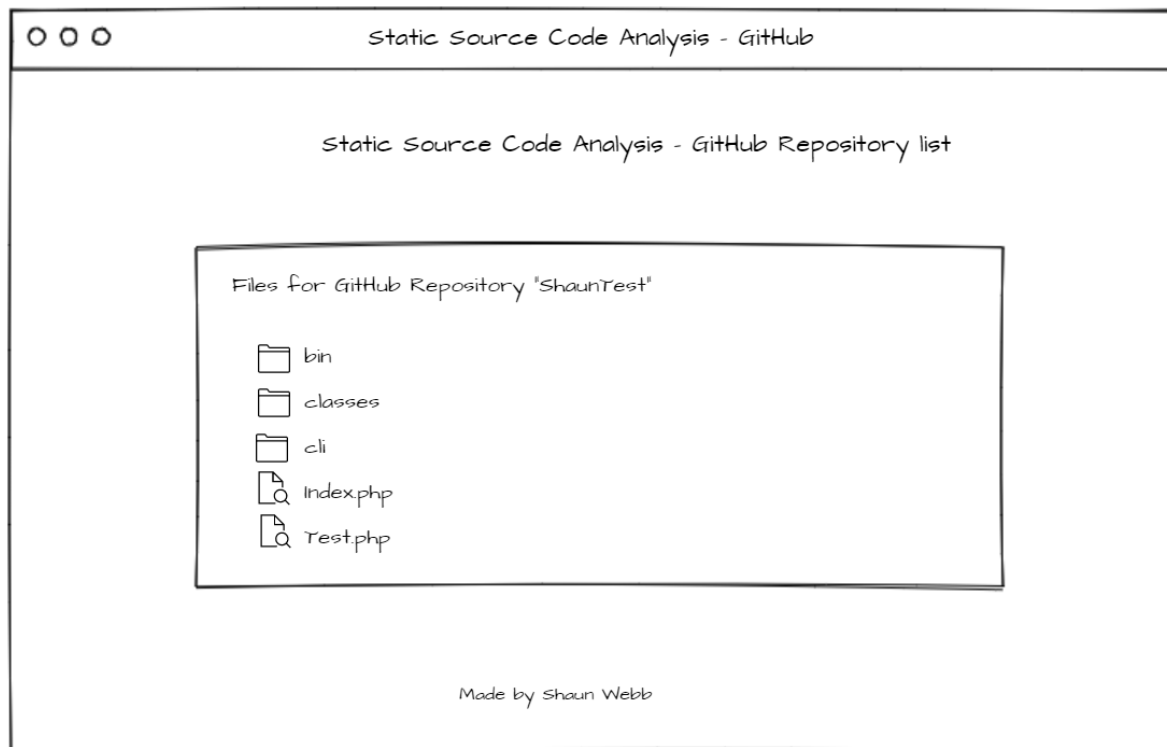


Figure 23. GitHub Repository viewer UI mock-up

The final screen the user is presented with is the results page. The program analysis would have been completed by now and the findings are shown back to the user. The page will display a table showing the vulnerabilities found along with the main points of concern outlined previously. A button to download the report should also be given. Finally, but not so importantly, previously analysed files should be listed and clickable to switch between reports.

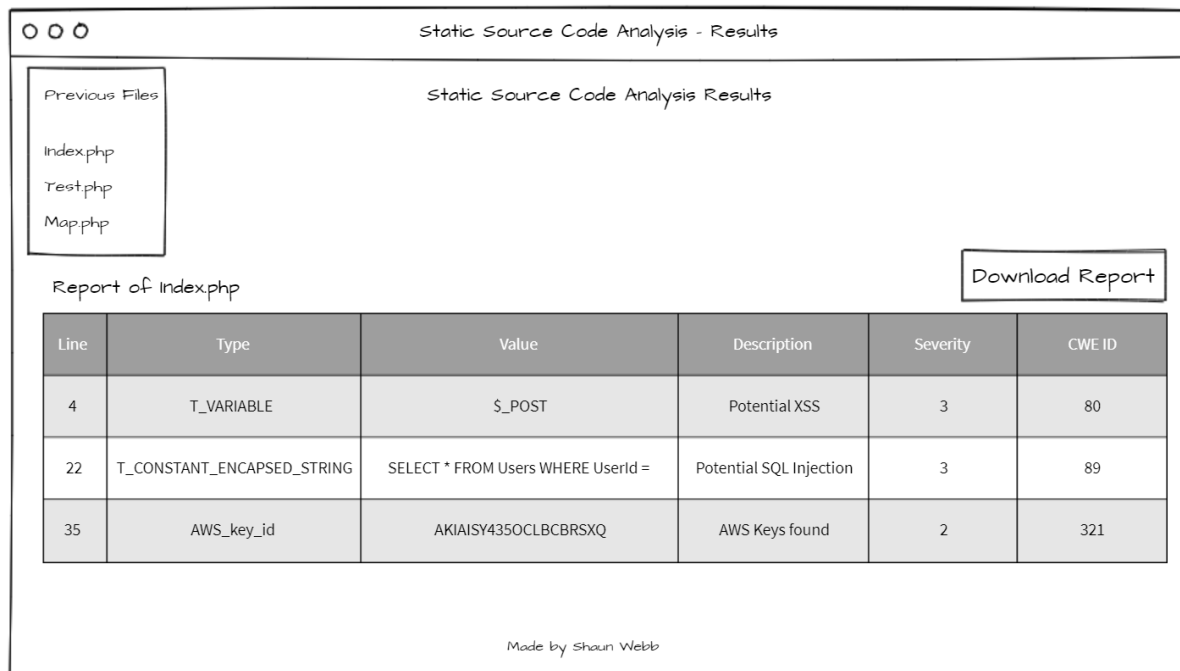


Figure 24. Report screen UI mock-up

5. SUMMARY

In order to identify web-based security issues from the OWASP Top 10 list, an appropriate language was chosen. From the literature review, PHP was found to be the most common server-side programming language, so this is the best candidate.

A web-based system will also be created, that brings together components required for source code analysers to work. These are tried and tested components that have been used in compilers for years. The system is divided into; client-side, backend, and external, all of which make up the architecture of the application. An activity diagram shows the flow of the system using the Unified Modeling Language (UML). Then finally, UI mock-up designs are given for each screen of the system.

D. IMPLEMENTATION

This section outlines the implementation stage of the project in detail. Justifications are made for why tools, technologies and platforms were chosen over others. The implementation is based on the designs previously shown. The main functionalities of the system along with the results are analysed in detail.

1. CHOICE OF LANGUAGES AND TOOLS

As the design decisions lead us to target PHP, it made sense for a web-based solution to be built.

The application was built using the following technologies:

- NodeJS

NodeJS is a server-side JavaScript framework that is built on the Chrome V8 run-time engine. NodeJS with the Express framework was used as it allows for a lightweight server to be up and running easily with minimal code and little overhead. JavaScript inside PHP code also needs to be analysed so this also contributed to the decision to use NodeJS.

- PHP – Zend Engine

PHP is an obvious candidate to target PHP vulnerabilities as it can have dynamic run-time capabilities. However, since the purpose of this project is to review the feasibility of static analysis, the programming environment that the developer has more familiarity with was chosen as parsing large streams of code can become very complicated. The Zend Engine is what PHP uses internally to interpret PHP code, some of the internal functions for the interpreter were used to provide more details about the code such as the lexical analysis tokenizer function (Bakken, et al., 2004).

- AWS EC2

A cloud environment allows for the tool to prove that it will work in an environment that is analogous to that of the real world, this is important to further prove the feasibility of the project. Amazons Web Services is a perfect playground to create virtual cloud environments and run scalable web-based applications. EC2 was used to create an Ubuntu Linux Virtual Machine, acting as the backend for the application.

- HTML – Bootstrap Theme

The front-end system that the user interacts with is done via a web browser. Standard HTML with the Bootstrap CSS theme was used, other libraries such as jQuery were used to simplify code over vanilla JavaScript.

2. SYSTEM FUNCTIONALITIES

This section outlines the main functionalities of each part of the program in detail with small snippets of code. The system is designed for programmers to use as the results given are targeted towards them, but the system is easy to use for a range of skill levels. The application is simplified by the fact that there aren't many different pages to navigate. The main complexity comes in the program analysis part that analyses the code and detects vulnerabilities.

A) HOME SCREEN

The main home screen is simple and easy to use. The user is presented with two options when loading the application. They can either upload a PHP file directly or choose a GitHub repository to be scanned.

Static Source Code Analysis
PHP / JavaScript

Find vulnerabilities in source code before deployment.

Choose File No file chosen

Submit

GitHub

Enter full github repository URL

<https://github.com/TehWebby/SCA>

Submit

Made by Shaun Webb


 UNIVERSITY OF SURREY

Figure 25. Homescreen UI screenshot

(1) UPLOADING A FILE

From the Home screen clicking “Choose File” will bring up a file picker dialog.

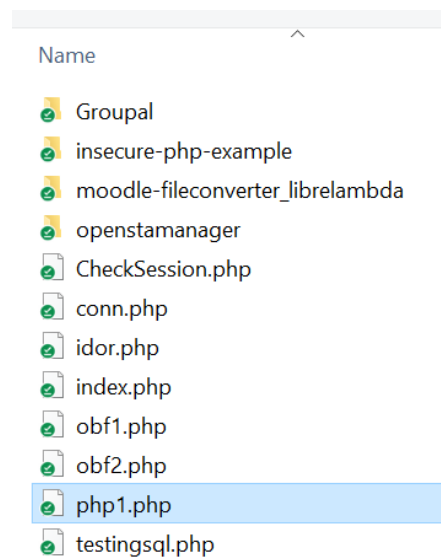


Figure 26. Upload file UI screenshot

Upon selecting a file and clicking the top submit button on the home screen the source code will be uploaded to the back-end server. Only PHP files are supported as the application is designed to target only that language, although it does support JavaScript that is inline in PHP.

B) GITHUB REPOSITORY SCREEN

If a GitHub repository URL is given, then a list of the directories and files will be shown.

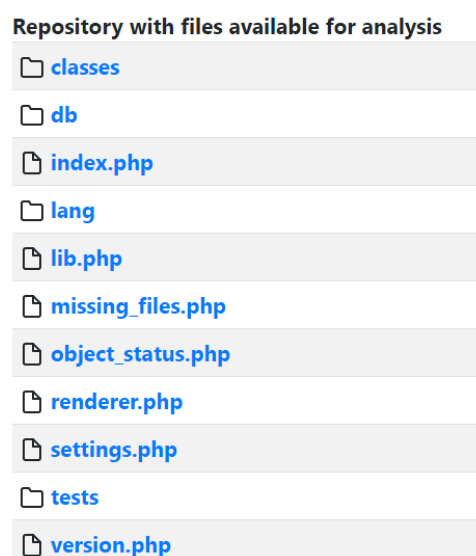


Figure 27. GitHub repository viewer UI screenshot

Upon clicking a PHP file, the system will start the analysis of that file and display the “[results screen](#)”.

C) GITHUB AUTOMATIC SCRAPING

Due to time constraints, this feature was not fully completed but the idea behind it is simple. The idea was to use GitHub’s own search and topic feature to find and target specific repositories automatically. These could be AWS based projects written predominantly in PHP for example. The tool would then scrape a large amount of code from many repositories and perform analysis on them. It could be used to find exploits, credentials and API keys in public repositories.

D) PROGRAM ANALYSIS

Once a compatible file is selected the system then performs program analysis. This stage occurs without the user knowing as this is done on the back-end. Each problem set (Vulnerability detection, Cryptography checking, Sensitive data and Taint analysis) of the analysis is done asynchronously allowing for fast computation.

(1) NODEJS SPECIFIC DETAILS

The coding style chosen is tailored for NodeJS. Typically speaking NodeJS is a single-threaded application, that supports concurrency via the event loop, this allows it to be asynchronous and non-blocking I/O. Simply put this means that code is not always executed in the simple sequential way most programmers think. Callbacks are initiated and data is returned when it is ready via a promise, this allows for other code to keep flowing while computation is being done. See the small example below in Figure 28 and 29 for an explanation of this concept.

Here the code loads a file, prints the file information to the screen then prints the message “hello”. However, this style of programming is not good for NodeJS as it will block the event loop and prevent further execution of any other code. Many NodeJS functions are asynchronous only. This would cause other ongoing executions to hang and wait until previous computation has finished.

```
var fs = require("fs");
var data = fs.readFileSync('data.js', 'utf8');
console.log(data);
console.log("hello");
```

Figure 28. NodeJS sync code example

This example uses the concept of a callback to load the file. It will load the file concurrently and not block the event loop. This means that the program will print the message “hello” before it will print the file information. However, this also means that the variable data cannot be used outside the scope of the readFile call until the callback has returned.

```
var fs = require("fs");
fs.readFile('data.js', 'utf8', function(err, data){
    if(!err) {
        console.log(data);
    }
});
console.log("hello");
```

Figure 29. NodeJS async code example

This asynchronous programming method has been used to optimize the program analysis stage. The Async.js utility was used to better manage the asynchronicity. The below example in Figure 30 is from the developed applications and demonstrates the use of asynchronicity.

```
async.parallel( [
    function ( callback ) {
        // Taint analysis
        arr["is_token_user_input"] = is_token_user_input(result);
        callback(null);
    },
    function ( callback ) {
        // Send Tokenization of data back to the user
        arr["data"] = data;
        callback(null);
    },
    function ( callback ) {
        // Crypto functions check
        arr["crypto"] = checkCrypto(result);
        callback(null);
    },
    function ( callback ) {
        // Sensitive Data check
        arr["sensitive"] = findSensVars(result);
        callback(null);
    },
], function(err, results) {
    // ...
});
```

Figure 30. System async code snippet

Using the asynchronous method, the problem sets are executed independently, resulting in a much faster program. In this case, the program will only be as slow as its slowest function rather than the cumulative execution time of each function. Other functions which require more computation or scan online security knowledge databases are done after the initial source code analysis via AJAX and is explained later in the [“Results Screen”](#) section.

Although NodeJS typically only utilizes a single thread it is possible to take advantage of multi-core CPU systems. Using a cluster of NodeJS processes allow for the load to scale based on the total number of CPU cores available on the server (nodejs.org, n.d.).

An AWS EC2 t2.micro instance is used, although this only has a single core the code is scalable and ready to work with more powerful processors.

(2) PHP ZEND ENGINE – LEXICAL ANALYSIS

The module “exec-php” is used to provide direct access to PHP function calls directly from within NodeJS. The PHP Zend Engine that is used to interpret PHP is written in C⁴. The PHP language provides direct and easy access to some of its functions. The function “token_get_all” is called directly from NodeJS using “exec-php” with the entirety of the source code to analyse being passed as a parameter. The PHP will get executed and it returns a series of tokens back to NodeJS.

The tokens will vary depending on the source code, but they will generally provide the level of detail as seen below in Figure 31.

```
Line 1: T_OPEN_TAG ('<?php ')
Line 2: T_ECHO ('echo')
Line 2: T_WHITESPACE (' ')
Line 3: T_CLOSE_TAG ('?>')
```

Figure 31. System exec-php token example from a simple php file

The array of tokens are then used in the detailed analysis part which attempts to parse and understand the code.

(3) DETAILED ANALYSIS

Once the source code has had gone through the lexical analyser the token data is used and multiple problem sets are analysed in parallel. For each problem being targeted the full set of tokens are iterated through, exposing the line, type and the code. In the sections below, each issue type is explained and some screenshots are taken from the results screen to give better context to certain conditions.

⁴ GitHub PHP <https://github.com/php/php-src/blob/master/ext/tokenizer/tokenizer.c>

(A) Taint Analysis

Taint analysis is a method of checking which variables can be modified by user input.

A list of known good and bad PHP predefined variables are stored. Each token iteration is cross referenced with the bad predefined variable list and exclusions are made for the good predefined variables depending on the situation or context. An example is the header `HTTP_X_FORWARDED_FOR` is secure if it is set properly when behind a proxy but without a proxy, this header can be modified by the user and should not be trusted. A very simple example of a tainted variable can be seen in Figure 32.

```
$data = $_POST['data'];
```

Figure 32. Taint vulnerable code example

This example is much simpler to analyse. However, the difficulty then is transferred to understanding what the expected behaviour of the program is. Is it safe for the user to set the data variable, or should this value be set via the backend? A userID for example should not be set by the client, but many developers mistakenly implement it this way. For this reason, the decision to highlight the potential issue was chosen, this means the code analysis will detect what it thinks could suffer from taint analysis but false positives will exist depending on the intended behaviour of the program.

(B) XSS AND SQL INJECTIONS

The taint analysis feature is then expanded upon to check for both SQL Injection and XSS.

Here is a brief look at a simple XSS check. Once taint analysis proves that an input can be modified by a user, we then need to verify that it could suffer from an XSS attack. A list of XSS protection functions are collated, these are used to see if sanitization is being done on the input. Vulnerable code could be using one of the functions to make it safe, so it is now no longer an issue. However, if the user is calling a function that we do not have access to, and we do not recognize it as an in-built function then we have no way of detecting if the function call will correctly sanitize the input. This is where using dynamic analysis to verify issues would have come into play. Being able to combine static and dynamic would give the analyser improved accuracy, but this was not experimented with for XSS, only with certain sensitive data that is explained in later sections.

Here the use of known sanitization functions calls are checked upon once the variable is determined to be modifiable via user input.

- Check for a list of known XSS protection functions

```
const xssProtectionFunctions = ["htmlspecialchars", "htmlentities"]
```

Figure 33. XSS Function protection list

- If this is true a known XSS protection function was used

```
if (data[i-1][1] == "T_STRING" && xssProtectionFunctions.includes(data[i-1][2])){
```

Figure 34. XSS function protection check code snippet

- If false, the tainted variable could be vulnerable to XSS. Additional checks are done to determine if any other sanitization is attempted before deciding if the code is vulnerable or not.

```
data[i].push(owaspXSS.severity);
if (data[i-1][1] == "T_STRING"){
  data[i].push(owaspXSS.description + ", unless function '"+data[i-1][2]+' validates the input");
}else{
  data[i].push(owaspXSS.description);
}
data[i].push(owaspXSS.url);
data[i].push(owaspXSS.cwe)
arr.push(data[i]);
```

Figure 35. XSS vulnerability check code snippet

This line of source code (Fig 36) is susceptible to an XSS attack, however if customFunction does validate the input correctly it will now be secure. In this example, the analyser does not have access to this function (lets say it's from a different library) so there is no definitive way of determining if the attack was negated using static analysis alone.

```
echo "Hello " . customFunction($_POST['2']);
```

Figure 36. customFunction that may or may not sanitize input example

5	T_VARIABLE	\$_POST	Potential XSS, unless function 'customFunction' validates the input
---	------------	---------	---

Figure 37. Report screen example of detecting non-deterministic function call

Figure 37 shows the issue being presented on the report screen with the line number and a basic description.

(C) CRYPTOGRAPHIC VERIFICATION

For cryptographic-based issues the system simply checks for hash values and encoded messages that are hardcoded in the source code. For example, if the system finds that some text is encoded it will determine the

encoding method such as base64, then decode the message. It is near impossible from the code alone to determine the real context in which the encoded message is being used, it may be used to display a trivial message, but it could also be used for something that requires high level of security. For this reason, the system just informs the developer that it has found the encoded message and was able to decode it, the developer then needs to take initiative to determine if it is ok to leave the encoded message intact.

A similar technique is used for known insecure hash algorithms with proven collisions. Although, this feature has been expanded and is more advanced. For each hash value that is found in the source code the system does some background tasks via AJAX, this ensures that the user experience and results screen is not slowed down because of the tasks. The tasks involve search online databases of known hash values, essentially doing a rainbow table search. These online databases already have huge collections of values and using these is more viable than trying to build a table just for this system.

In this example (Fig 38) the hash value of “2fd4e1c67a2d28fced849ee1bb76e7391b93eb12” was found on line 14.

```
14. $text = "2fd4e1c67a2d28fced849ee1bb76e7391b93eb12";
```

Figure 38. Hash value inside the code to be analysed example

The system then found a match with the website hashtoolkit.com, although the user of the system will not actually see the details shown in Figure 39, as this is done via the analysis tool on the backend. For some sites a web scraping approach was implemented, for others a RESTful API is used.

Decrypt Hash Results for: 2fd4e1c67a2d28fced849ee1bb76e7391b93eb12			
Algorithm	Hash		Decrypted
sha1	2fd4e1c67a2d28fced849ee1bb76e7391b93eb12	Q	The quick brown fox jumps over the lazy dog Q

Figure 39. Hash match with website

The user can then see that the original input value has been found along with the hash algorithms the system thought it qualified for. An information icon can be hovered over for more information (Fig 41).

```
14 hash sha1,ripemd160 2fd4e1c67a2d28fced849ee1bb76e7391b93eb12 The quick brown fox jumps over the lazy dog ⓘ
```

Figure 40. Report screen example finding hash match

Clicking this icon then takes you to the website or database where this information was found, just like as seen in Figure 39 above.

```
14 hash sha1,ripemd160 2fd4e1c67a2d28fced849ee1bb76e7391b93eb12 Hash value has been matched with a rainbow table ⓘ
```

Figure 41. Report screen example finding hash match with tooltip

(D) SENSITIVE DATA

The final type of check looks for sensitive data in source code. The simplest method looks for variables or text with names similar to “password” or “credentials”, once found we can determine from the token data if the code is a variable, comment or something else entirely. A regular expression is used to find credit card details exposed in code, a similar method is used to find AWS keys (Fig 42).

```
{"AWS_key_id": /AKIA[0-9A-Z]{16}/, "AWS_secret_key": /[0-9a-zA-Z/+]{40}/};
```

Figure 42. Code snippet of AWS id and secret key regex

This feature has also been expanded to improve the accuracy of the report, demonstrating a simplified version of the static meets dynamic analysis idea.

This paper proposes, and the project implements a simple verification mechanism, it uses the found credentials and tests them with the API provider. Instead of presenting false positives to the user or developer that is checking for leaked keys, confirming that the keys are valid is very beneficial for them. This verification phase must be combined with one of the mentioned search methods and it is important to try and reduce the total number of false positives first. A simple API call can be done with a user identification call. For Amazons Web Services a simple “get-caller-identify” command can be performed via the command line or in a language supported by the API.

The following AWS CLI command as seen in Figure 43 can also be used in code as seen in Figure 44.

```
AWS_ACCESS_KEY_ID=key AWS_SECRET_ACCESS_KEY=secret aws sts get-caller-identity
```

Figure 43. AWS CLI command

```
var output = exec("AWS_ACCESS_KEY_ID="+key+" AWS_SECRET_ACCESS_KEY="+secret+" aws sts get-caller-identity", {stdio : 'pipe' });
```

Figure 44. AWS CLI command used in NodeJS

If successful, the following AWS response will be like shown (Fig 45).

```
{
  "UserId": "808624997024",
  "Account": "808624997024",
  "Arn": "arn:aws:iam::808624997024:root"
}
```

Figure 45. AWS response on valid match

This simple command can be used programmatically when a pair of keys (Client ID and Secret Key) have been found. Instant verification on the validity of the keys provides perfect accuracy results. Certain restrictions for different APIs need to be considered, such as IP restrictions set in place for credentials, under these circumstances it would not show the credentials as being valid although they may be.

(E) DEOBFUSCATION

Deobfuscating code brings another layer of complexity. There are so many ways in which code could be obfuscated. Simple weak cryptography methods such as rotating the character positions by a few, then reverting when the code is used is a single method of obfuscation that could be used. There are many more possible ways to obfuscate code making deobfuscation extremely difficult to be computed within a reasonable amount of time. We could create an algorithm that attempt to try thousands of variations leaving us with possible spurious code possibilities.

A few simple algorithms were devised to test this but as this is not the main focus of the application the decision was made to use an existing tool and integrate it within the source code analyser. Upon initial inspection the unphp.net tool seems fairly effective at finding many PHP obfuscation techniques but fails to deobfuscate code that used the GOTO statement to move around the code. Perhaps due to the increased complexity of flow analysis that would be required.

When a file is uploaded to the analysis program and no vulnerabilities were detected the user has an option to try and deobfuscate the file.

Report

obf1.php

No issues found in obf1.php

Is the code obfuscated?

Yes, attempt deobfuscation

Upon clicking the button, the source code is sent to unphp.net using their RESTful API. If the site was successful in deobfuscating the code, then it returns original source code back to our analyser. The analyser then puts this code into its program analysis to search for vulnerabilities. It also names the new file by

appending _DEOBF at the end. Then the results are shown. Here we can see what the obfuscated code looked like (Fig 46), then after deobfuscation (Fig 47).

```
eval(str_rot13(gzinflate(str_rot13(base64_decode('LUnHEq04Dv2arn6zI16gc1K+5Bw3Ru
QcWbiEr2KYGcpH2bIkW1RUgqUZrz/b8FjWeKyWP9NLLh/sP2CZRrD8Kca2Lq7 ...
```

Figure 46. Obfuscated code example (partial)

```
echo $_SERVER['HTTP_X_FORWARDED_FOR'];
echo $_SERVER['http_x_forwarded_for'];
```

Figure 47. Deobfuscated code example

After deobfuscating the original code, the issues were reported by the system (Fig 48).

Report

[Download](#)

obf1_DEOBF.php

Potentially Unsafe Code

Line	Type	Value	Description	Severity
2	T_CONSTANT_ENCAPSED_STRING	HTTP_X_FORWARDED_FOR	HTTP_ values can be modified by the client, ensure this is expected behaviour	2
3	T_CONSTANT_ENCAPSED_STRING	http_x_forwarded_for	HTTP_ values can be modified by the client, ensure this is expected behaviour	2

Figure 48. Report screen detecting issues after deobfuscation

E) RESULTS SCREEN

The results screen is important to provide the user with an easy to understand report. Although the main purpose of the static source code analyser is to identify security vulnerabilities, if the developer cannot understand the security issues then they will be unable to fix the issue. Therefore, it is important for the results to display all the information detected and be presented in a comprehensible manner for the user of the system.

A PHP file with purposely made vulnerabilities has created and passed into the analyser. The report has been created and shown below. The report has been split up into parts and is explained in the parts below.

Report

Download

Potential AWS keys have been found. Checking their status...

php1.php

Potentially Unsafe Code

Line	Type	Value	Description	Severity	CWE
4	T_VARIABLE	\$_POST	Potential XSS	2	80
5	T_VARIABLE	\$_POST	Potential XSS, unless function 'customFunction' validates the input	2	80
8	T_CONSTANT_ENCAPSED_STRING	HTTP_X_FORWARDED_FOR	HTTP_ values can be modified by the client, ensure this is expected behaviour	2	454
10	T_VARIABLE	\$_POST	Potential XSS	2	80
57	T_VARIABLE	\$_POST	Potential XSS	2	80
60	T_VARIABLE	\$_GET	Potential XSS	2	80
57	T_CONSTANT_ENCAPSED_STRING	SELECT * FROM Users WHERE Userid =	Potential SQL Injection	3	89
61	T_CONSTANT_ENCAPSED_STRING	SELECT * FROM Users WHERE Name =	Potential SQL Injection (less likely)	2	89

Figure 49. Report screen

At the top of the (Fig 49) screenshot is a progress bar that is doing background checks, this particular issue is explained later. Located at the top right is a Download button which triggers the browsers native Save to PDF functionality. The unsafe code with security vulnerabilities can be seen at the bottom of the screenshot, with a full explanation of each issue. The “severity” can be clicked to open a link to the relevant OWASP page that can suggest how this should be fixed. Further down the page are other security issues found with the PHP code. They are separated by issue type (Crypto, Sensitive data). The small black information icons can be hovered over and clicked for more information. For the hash values they show which rainbow table was used to find the original input data.

Crypto					
Line	Type	Algorithm	Text	Real Data	Severity
6	encoding	base64	aGVsbG8gZnJvbSBqYXhbg==	hello from japan	
14	hash	sha1,ripemd160	2fd4e1c67a2d28fced849ee1bb76e7391b93eb12	The quick brown fox jumps over the lazy dog ⓘ	3 ⓘ
17	encoding	base64	amFwYW4=	japan	
19	hash	md4,md5,ripemd128	c52aa11424524dcc59d3502354257dd6		2
21	hash	sha1,ripemd160	2fd4e1c67a2d28fced849ee1bb76e7391b93eb12	The quick brown fox jumps over the lazy dog ⓘ	3 ⓘ

Sensitive data		
Line	Type	Value
14	T_VARIABLE	\$pass
14	AWS_secret_key	2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
21	AWS_secret_key	2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
22	creditCard	4231321131114511
23	AWS_key_id	AKIAISY435OCLBCBRSXQ
25	AWS_key_id	AKIAISY435OCLBCBRSXQ
26	AWS_secret_key	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
37	T_CONSTANT_ENCAPSED_STRING	credentials
38	T_CONSTANT_ENCAPSED_STRING	key
38	AWS_key_id	AKIA2E0A8F3B244C9986
39	T_CONSTANT_ENCAPSED_STRING	secret
39	AWS_secret_key	5CmXeCgPPsSHuXNbjUQKyznuOzAAV16MrlQRMJqN
47	T_VARIABLE	\$credentials
47	T_STRING	Credentials
47	T_VARIABLE	\$key
47	T_VARIABLE	\$secret

Figure 50. Report screen more details

Some calls use AJAX to get more detailed information after the initial results screen. These features include checking hash values against online rainbow tables, using AWS Keys with the AWS CLI to verify they are valid. The red highlighted fields are ones which are verified to be insecure. In this case the AWS keys were verified using the AWS API, this means the keys actually work and could be used by other people who have access to the source code.

2 occurrences of valid AWS key pairs found!

Figure 51. Valid AWS keys found and verified

14	AWS_secret_key	2fd4e1c67a2d28fcd849ee1bb76e7391b93eb12
21	AWS_secret_key	2fd4e1c67a2d28fcd849ee1bb76e7391b93eb12
22	creditCard	
23	AWS_key_id	
25	AWS_key_id	AKIAISY435OCLBCBRXQ
26	AWS_secret_key	5CmXeCgPPsSHuXNbjQKyznuOzAAV16MrlQRMJqN

AWS valid key pair
 AWS_secret_key: 5CmXeCgPPsSHuXNbjQKyznuOzAAV16MrlQRMJqN
 Details: arn:aws:iam::808624997024:root

Figure 52. Report showing valid AWS key pair

E. TESTING

Software testing is used to verify that the system working in the way it is expected to. The application should be tested to ensure the needs of the requirements are being addressed. During this phase different methods of testing can be done to verify the functional and non-functional requirements, such as unit testing, integration testing and UI automation testing. This can be time-consuming, labour-intensive and is prone to human error (Ciortea, et al., 2009). Ideally, every aspect of the program would be tested, however this is not feasible due to high cyclomatic complexities that make full test coverage difficult. These same difficulties are shared by static analysers.

1. UNIT TESTS

Unit Tests have been combined with UI automation for full stack test coverage. They aim to ensure the system meets the requirements. The unit tests are split up into 3 sections.

- Backend unit tests that test the server, checking status codes and server responses.
- Backend unit tests that call the analysis functions passing source code directly. Known vulnerabilities for the code are known and checks are done to ensure the system detected them.

- Frontend UI automation activated by the unit tests. These tests do not access the source code of the system directly, but instead scrape data from the HTML and JavaScript shown in the browser. This allows file uploads and vulnerability detection to be checked along with confirmation that the reports are being displayed correctly.

Tests are executed manually by running the command “npm test” from the project directory. As the backend of the system is made using NodeJS, it seems natural to use similar tools to create the tests. Mocha and Chai are NodeJS modules that allow for asynchronous unit tests to be performed.

The tests mainly focus on the analyser function calls and the UI automation. Variable data and code behaviour are asserted to ensure analysis is consistent and correct.

Code snippets from different unit tests can be seen below.

```
/**
 * NodeJS Server related tests
 * @function
 */
function server(){
  it('Server response', function(done) {
    request('http://localhost:80' , function(error, response, body) {
      expect(response.statusCode).to.equal(200);
      done();
    });
  });
}
```

Figure 53. Simple status code response unit test

The unit test in Figure 54, checks that a correct CWE ID and name was detected from the analysis

```
assert.equal(cwe_list[x], cwe) // Does the result CWE issue match the requested code example

let cwe_name = getCweName(cwe);
if (cwe_name != null){ // Does the result CWE issue match the results Security Issue
  expect(results[i][4].includes(cwe_name)).to.equal(true);
}

for (var l=0;l<results[i].length;l++){
  expect(results[i][l]).to.not.be.null;
}
```

Figure 54. CWE ID and name unit test

The CWEs listed in the requirements are tested [80, 89, 257, 321, 454]. Then tests are done on source code with known vulnerabilities and a small sample set from the NIST PHP SARD⁵ dataset. The coding complexities listed in the requirements have been considered by the analyser. Some tests of the dataset source code prompted an iterative approach to development and small changes to the program analyser were made to reduce false negatives and fix a few simple display bugs such as escape characters being displayed incorrectly.

2. UI AUTOMATION

UI Automation takes advantage of the Selenium testing framework. Previous experience of the Python Selenium framework were the reasons behind choosing the framework over other tools. The following free tools were also considered;

- Telerik Test Studio
- Cucumber

For the project developer it was the first time using the NodeJS version of selenium, the choice of this was dictated by ease of interoperability by using the same software stack (NodeJS). The asynchronous nature of NodeJS causes some challenges with unit tests (Tayar, 2019). This meant that code had to be slightly different to what was expected and await with async functions had to be used in order to overcome these issues.

Selenium was configured to use the chrome driver which uses Google Chrome. Headless mode was used as the deployed AWS server has no GUI, and hardware acceleration was disabled as no GPU is present on the device.

```
var webdriver = require('selenium-webdriver'),
    chrome     = require('selenium-webdriver/chrome')
By            = webdriver.By,
Key           = webdriver.Key,
until         = webdriver.until,
options       = new chrome.Options();
if (os_type !== "win32"){ // only use headless mode on Linux Server
    options.addArguments('headless'); // note: without dashes
}
options.addArguments('disable-gpu');
```

Figure 55. UI automation code snippet

Selenium using the chrome driver can be seen below in Figure 56. Note the “Chrome is being controlled by automated test software.” message at the top.

⁵ <https://samate.nist.gov/SARD/>

Chrome is being controlled by automated test software.

Static Source Code Analysis

[Report](#)
[Upload File](#)

PREVIOUS REPORTS
 php_test_code.php
 xss_lod2.php
 xss_lod1.php
 xss_lod0.php

Report

Download

php_test_code.php

Potentially Unsafe Code

Line	Type	Value	Description	Severity	CWE
4	T_VARIABLE	\$_POST	Potential XSS	2	80
5	T_VARIABLE	\$_POST	Potential XSS, unless function 'customFunction' validates the input	2	80
8	T_CONSTANT_ENCAPSED_STRING	HTTP_X_FORWARDED_FOR	HTTP_ values can be modified by the client, ensure this is expected behaviour	2	454
10	T_VARIABLE	\$_POST	Potential XSS	2	80
57	T_VARIABLE	\$_POST	Potential XSS	2	80
60	T_VARIABLE	\$_GET	Potential XSS	2	80
57	T_CONSTANT_ENCAPSED_STRING	SELECT * FROM Users WHERE UserId =	Potential SQL Injection	3	89
61	T_CONSTANT_ENCAPSED_STRING	SELECT * FROM Users	Potential SQL Injection (less likely)	2	89

Figure 56. Selenium controlling chrome

```

✓ Taint Analysis
List of CWE issues found
80,89,257,321,454
✓ Server response
✓ Git File Results
✓ Results page
✓ Git Repo Page
✓ UI Testing (5668ms)

6 passing (6s)
  
```

Figure 57. Unit tests passing

The unit tests serve better for SDLC purposes. They can ensure that code that is working correctly (currently being flagged as vulnerable) and does not change when code changes later. This regression testing helps ensure existing features don't break.

See the video file attached to see a brief example of Selenium working.

F. EVALUATION

1. EVALUATING AGAINST THE REQUIREMENTS

The evaluation phase looks over the SDLC and checks whether or not the systems meets the initial requirements and objectives previously laid out. Strengths and weaknesses of the system are briefly highlighted. Then finally briefly explain how the system can be deployed for end users of the system.

All of the main requirements were met except the “acceptable low false positive rate”. More time could be spent on experimenting with techniques previously suggested such as combining static and dynamic analysis together to lower the false positive rate. This was demonstrated when detecting AWS keys as a dynamic style approach was taken once the static analyser determined issues.

Table 7. Evaluation of main functional requirements

Requirement	Status
The software must be able to accept as an input compatible source code.	Achieved. Files can be uploaded directly or scraped from a GitHub repository.
Identify software security vulnerabilities in source code listed in Table 4.	Achieved.
Report the security weaknesses that are identified, describe what kind of weaknesses they are, and finally determine the line number of the issue in the code.	Achieved.
Identify weaknesses despite the presence of coding complexities listed in Table 5.	Achieved.
Have an acceptably low false positive rate.	Debatable. The “ Evaluating against other tools ” section below proved that the false positive rate of the developed application is higher than other comparable tools, but the overall precision and sensitivity performs better than most of the existing tools.

From the optional requirements, suggesting secure code alternatives and suppression of specific issues was not met. Supporting obfuscated code was partially met. All other requirements were met. Allowing for secure code suggestions requires substantial work, as it not only needs to correctly identify the issue but also have a deep understanding of how it can be fixed in order to suggest an appropriate code alternative.

Table 8. Evaluation of optional functional requirements

Requirement	Status
Produce an easy to digest web-based report.	Achieved.
Allow specific vulnerabilities to be suppressed by the user so they do not appear in the report.	Not achieved due to time constraints. This could be done by the user selecting specific vulnerabilities (CWE ID), then have the backend simply do not perform those types of checks.
Attempt to find hash values original input using rainbow tables.	Achieved. External security knowledge online is used to check for possible matches.
Use the Common Weakness Enumeration (CWE) number beside the weakness it reports.	Achieved. Next to each identified vulnerability the related CWE ID and link to OWASP is provided.
Support obfuscated code analysis.	Deobfuscation was only achieved via the use of an API instead of developing a custom-built solution. Due to time constraints, attempting to determine if the obfuscation changes introduced new security issues was not achieved.
Suggest a secure code alternative for the security issue found	Not achieved. This would take considerably more time to develop and requires a deeper understanding of each vulnerability.

Each of the functional requirements achieved above have been explained in detail throughout the implementation stages with screenshots and example data given.

The non-functional requirements are evaluated below in the table and an in-depth performance (speed) analysis has been conducted.

Table 9. Evaluation of non-functional requirements

Requirement	Status
Usability	The system provides a simple to use UI and easy to digest web-based report of the security issues found.
Compatibility	PHP files that also have JavaScript in them are supported.
Accuracy	Achieved. See Table 11 for details.

Scalability	The system has been deployed on cloud technologies.
Performance (speed)	Achieved. See Table 11 and Figure 58 for details.
Performance (size)	Achieved. The system can analyse large files.
Maintainability	Achieved. Version control and code documentation are used.
Completeness	Achieved. Provides an application in the form of a proof of concept that works.
Portability	Achieved. Windows and Linux supported.
Reliability	Achieved. Error messages are used when exceptions are thrown.
Security	Achieved. Each instant will be deployed on an isolated environment preventing unauthorized user access.
Documentation	Achieved. JSDoc was used for automation code generation. See section “maintenance” .

The system performance (speed) was evaluated against 13 random files from different Git repositories. Although the system will likely perform better on certain code than others, such as function heavy code. It is important to evaluate against a wide variety of styles. Each test case is a single file with increasing levels of source lines of code (SLOC) and the average time of 5 runs is taken. A threshold performance value of SLOC/10 was chosen, it is important to note that this value is subjective and could be different.

Table 10. System performance evaluation

Test Case	SLOC	Threshold time to analyse (ms)	Avg time to analyse (ms)
1	362	36.2	38.8
2	394	39.4	39.4
3	395	39.5	38.8
4	611	61.1	45.8
5	827	82.7	52.2
6	851	85.1	43.0

7	931	93.1	53.8
8	1,282	128.2	60.3
9	1,551	155.1	64.5
10	2,391	239.1	76.2
Large 1	50,000	5,000	1304.4
Large 2	100,000	10,000	3032.2
Large 3	150,000	15,000	Out of memory

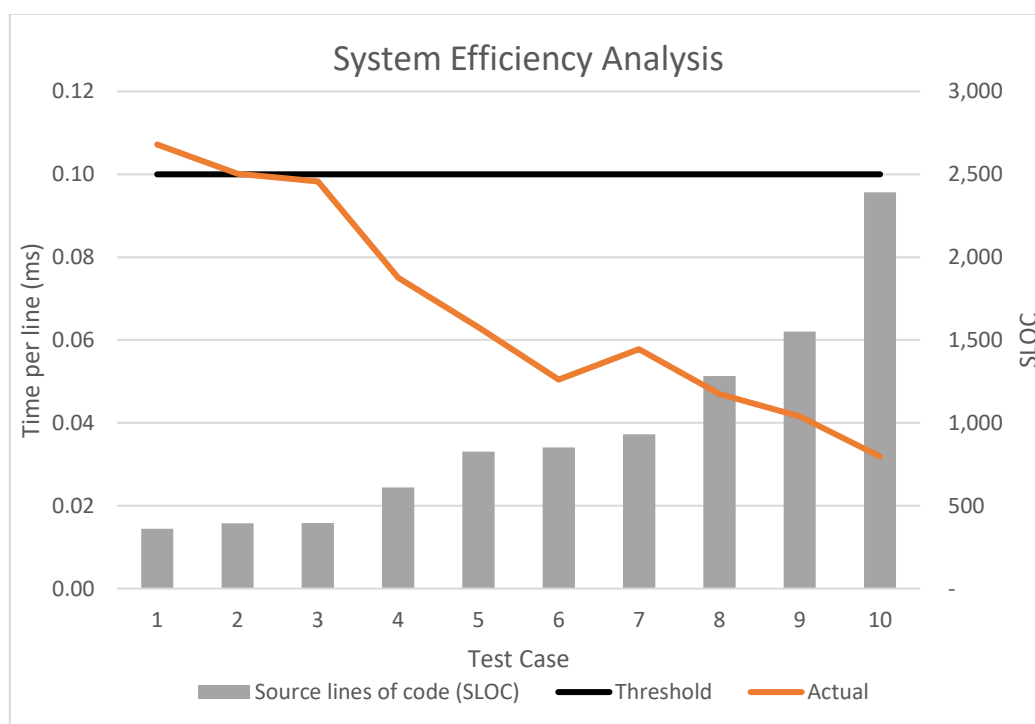


Figure 58. System efficiency analysis

Noteworthy mentions: Interestingly file 6 took less time than some other smaller files, this was due to large comments blocks in the code that the parser ignored. This resulted in around 600 lines of code that was being analysed instead of 851. A file of only 3 simple lines of took an average of 20ms to analyse. Each test case will have at least this much overhead. Files of around 150,000 lines of code resulted in an out of memory exception, this could be resolved by changing the code to offload its data into a database or a static file instead of holding everything in memory in a single run, or even running on more powerful hardware.

The speed requirement was that the system should not take more than $(\text{SLOC} / 10)$ in milliseconds to perform analysis. In almost all cases where the SLOC was greater than 400 this was achieved. The greater the SLOC the performance per line of source code actually increases. For smaller files the 20ms overhead made it not

achieve its target. In a real-world scenario most developers will be satisfied with analysis that takes less than a second.

Overcoming the coding complexities while correctly identifying security issues required significant development research in order to overcome. Difficulties included correctly establishing an architecture that uses components from compilers such as the lexical analyser. The tokenized code had to then be parsed and performing regular expressions on code also proved to be difficult. Correctly identifying the exact security issue also caused issues as sometimes an issue was caused by another. No current web-based PHP static analyser worked with obfuscated code and some shortcuts had to be taken in order for the system to work with it. The use of existing APIs saved a lot of time and proved to work quite well. This showed that it is feasible to deobfuscate code and identify vulnerabilities.

Once the parsing and program detection was established. It was simple to produce a web-based report by simply sending the information back to the frontend. Based on the type of detection that was successful the CWE ID can be understood, so this was presented next to the issue description on the report.

The main purpose of the system is to demonstrate the usefulness of static source analysers and show that there is a need for them to be adopted into software projects. One example of how the tool can be integrated into a software development project is by introducing it into the code review process.

2. EVALUATING AGAINST OTHER TOOLS

Three tools that were briefly examined previously are evaluated against the developed system. The effectiveness of each tool's detection capabilities are compared.

Precision, sensitivity (also known as recall) along with F_β scores are used to measure the tools detection accuracy. These metrics are all used in statistical analysis to measure the performance of binary classification. The scores are typically graded from 0 to 1, with 1 being perfect. Both precision and sensitive reward true positives but the difference is that precision aims at reducing false positives, while sensitivity aims at reducing false negatives. The F_β scores calculate a weighted average of precision and sensitivity, and its formula is shown below. The β value represents the weight on sensitivity, i.e. the larger the β value, the more emphasis is placed on sensitivity over precision, and vice versa. F_1 score is normally known as giving equal precedence to both precision and sensitivity. $F_{0.5}$ targets precision more than sensitivity, i.e. penalises false positives more. F_2 targets sensitivity and penalises false negatives more, and this could be a useful metric for safety critical systems.

$$Precision = \frac{TP}{TP + FP}$$

$$Sensitivity = \frac{TP}{TP + FN}$$

$$F_{\beta} \text{ Score} = \frac{(1 + \beta^2) \times Precision \times Sensitivity}{\beta^2 \times Precision + Sensitivity}$$

Five test cases were used, each of which includes a range of vulnerabilities inside, totalling 51 issues in total. Only vulnerabilities that the developed analyser can detect are used CWE ID [80, 89, 257, 321, 454]. The test cases are included with the uploaded files inside the test case directory. The test cases were taken from NIST SARD (Software Assurance Reference Dataset)⁶, OWASP and a past university assignment⁷.

Table 11. Evaluation against other tools

	Me	VCG	RIPS	ProgPilot
Precision	84%	94%	82%	88%
Sensitivity	75%	33%	53%	41%
F1 score	79%	49%	64%	56%
F0.5 score	82%	69%	74%	71%
F2 score	76%	38%	57%	46%

The results are biased towards the selected CWE issues as other tools can detect many different issues as opposed to a select few. But for the purposes of analysing the developed system which can only detect a few, this had to be done.

True negatives are not measured in the evaluation as they are difficult to quantify from source code. Rather than making every secure line a true negative, they were not evaluated. Also, zero-day attacks or missing security knowledge would have been limiting factors when highlighting all true negatives.

None of the tools tested detected commented out insecure code. This is hard to determine if it is correct behaviour. One could argue that developers may reuse that code in the future, so the static analyser should

⁶ <https://samate.nist.gov/SARD/testsuite.php>

⁷ Secure Systems and Applications (COMM047) - Assignment 1

provide a warning that commented out code is insecure. For the test it was decided these issues are true negatives, so they were ignored.

Some issues with the analysis include the difficult to categorize certain issues. Are the developers certain the issue really is a false positive in all cases? Do they know that this issue is a true negative for all supported platforms? There are some things that just have to be verified to the best of our ability.

A range of data sample sets were used for evaluating but larger test cases would have provided more accurate results. Currently GitHub repositories have to be entered manually into the system to scrape and scan the code. Changing this to automatically scrape thousands of relevant code repositories could have allowed for greater range of testing, covering a wider range of coding styles and techniques.

Although, using the tests alone, it is difficult to measure how effective the analyser is. There are many reasons for this; firstly the person implementing the test must know all of the vulnerabilities in the source code in order to create the tests, secondly due to the difference in how analysers work, they detect issues with slightly different results making it hard to compare directly. An example of this is using when using one tool, it detects a vulnerability on line 4, another detects the same vulnerability on line 5, because it's not actually been used until then.

Other problems are related to function calls from libraries or codes bases that the analyser doesn't have access to. For example, take a function from a different file, if "customFunction" sanitizes data input, then there is no security issue, if it doesn't then it's insecure. In these circumstances for the analysis the unknown function was treated as insecure, so if the tool detected one, a potentially security issue was reported. Known native sanitization functions such as "htmlspecialchars" were also checked but listed on a trusted function list to verify that checks were being done.

Additional metrics considered but were not investigated include the ease of integration into an existing project. Peter O'Hearn (2018) argues that the most important metric to measure the usefulness of a static analyser is the "Fix rate". This is where the issues are actually fixed after being reported, showing a good level of detection capabilities and importantly practical use within a project. This qualitative method is hard to measure but an organisation or future research could look at how the identified issues are reported back to the developers. Other determining factors include if the tool is runs automatically or manually on a periodic basis.

The analysis was conducted on non-obfuscated code only. The system developed for this project was the only one that could detect any issues when code was obfuscated using a tool such as PHP Obfuscator Tools⁸. This shows severe weaknesses in existing analyser tools. It is common for code to be obfuscated to protect intellectual property. Vulnerabilities in obfuscated code will not be detected, leaving code bases and ultimately organisations at risk.

⁸ tools4nerds.com/online-tools/php-obfuscator

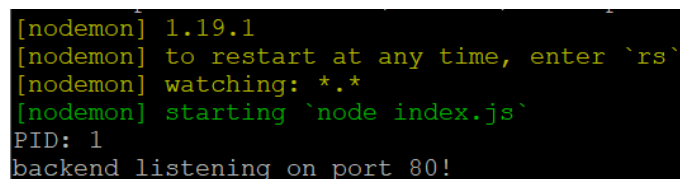
It is obvious that more testing is required to fairly compare tools. Further analysis on how each tool detects certain issues better than other tools could also be useful for organisations. The developed system has a high detection rate, but also has the highest false positives rate. However, the high detection rate helped to improve the overall metric scores. Due to time constraints, significantly less time was spent on reducing false positives. Visual Code Grepper had the lowest overall sensitivity score but had the highest precision. Additional testing is required so it is difficult to conclude useful information such as if the systems with higher false positive rate also has a higher rate of true positives. Systems with high false positive rates are difficult for developers to use as they waste lots of time.

Future work should look at techniques to reduce false positives. These techniques include combining dynamic analysis to verify issues found by static analysis. By doing this an attack can be performed against the vulnerability automatically and if successful the vulnerability can be confirmed as a true positive. Although, the application would obviously require additional work to be able to strongly determine that a detected issue is actually a false positive.

3. DEPLOYMENT

The system was built using web-based technologies and thus requires a web server with NodeJS version 8 or greater. PHP 7.2.19 or higher is required as the Zend Engine is used for lexical analysis.

Windows and Linux are both supported. Deployment was done on both a local windows machine and an AWS Ubuntu 18.04 t2.micro virtual machine. The deployment procedure has been simplified. NodeJS requires that its dependencies in the node_modules directory are based on the package.json file. Then instantiation of the server can simply be done by pointing node to the entry point of the system (index.js).

A terminal window with a black background and green text. The text shows the output of running 'nodemon' on a system. It displays the version '1.19.1', instructions to restart with 'rs', the watching pattern '*..*', the command 'starting `node index.js`', the PID '1', and the message 'backend listening on port 80!'. A green cursor is visible at the end of the last line.

```
[nodemon] 1.19.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
PID: 1
backend listening on port 80!
```

Figure 59. NodeJS deployed on AWS

It was important to not limit the application to a specific operating system or environment allowing for a greater potential adoption rate, plus based on the fact that PHP is used on a variety of servers. Once deployed, the environment was stable and worked throughout development with no major environmental issues.

G. MAINTENANCE

The application can be expanded, and new features can be added in an ad hoc fashion. To support development of new features it is important that the project uses version control, has good code documentation along with comprehensive comments in code.

GitHub was used for version control throughout, providing cloud-based backups and management of code. SourceTree by Atlassian has GitHub integration and greatly improves the usability of Git repositories. A simple but effective GUI allows code changes to be easily tracked. The software shows additions and removals of all code commits, this feature is extremely useful to spot mistakes in code changes. Using a single repository simplified tracking of code changes across all layers of the system.

JSDoc⁹ was used for the generation of code documentation. Code is tagged to give information on the parameters and return values. Code is also commented throughout, and an example of the documentation generated can be seen in figure 60 below. This allows other developers to understand how the existing features and functions work, this is crucial when extending the tool with new functionality.

Module: /tools/tools.js

Main part of the program analysis

Author: Shaun Webb

Source: [tools/tools.js, line 1](#)

Methods

(inner) analyse(data)

Analyses PHP source code for vulnerabilities

Parameters:

Name	Type	Description
data	string	PHP Source code to be analysed

Source: [tools/tools.js, line 19](#)

(inner) checkCrypto(data) → {array}

Performs Cryptographic vulnerability checks

Parameters:

Name	Type	Description
data	string	Tokenized PHP

Source: [tools/tools.js, line 269](#)

Home

Modules

- [/routes/analyse/loader.js](#)
- [/routes/fileupload.js](#)
- [/routes/scrapper.js](#)
- [/test/test-analyser.js](#)
- [/test/test-pages.js](#)
- [/test/test-ui.js](#)
- [/tools/tools.js](#)

Figure 60. JSDoc

⁹ JSDoc URL- jsdoc.app

H. SUMMARY

This section summarizes the entire software development life cycle.

An agile development methodology was chosen to allow for features to be implemented in an iterative fashion.

Functional and non-functional requirements were established initially. Then the design stage laid out the foundations for the components of the analyser. Architecture and UI designs were mocked up and an activity diagram in UML showed the standard behaviour the program should follow. PHP was targeted due to its prevalence in the online world.

The implementation stage justified the language and tools choice used to develop the system. Then system functionalities were explained in detail.

Testing included; unit tests that were conducted with Mocha and Chai. UI automation tested the frontend in Chrome using the Selenium web driver.

Evaluation showed that how the system faired against the requirements and performance. The system showed that it could keep up with, and even beat the current best web-based static analyser tools in terms of detecting vulnerabilities and analysing obfuscated code through deobfuscation. Focusing on a small subset of the most critical vulnerabilities, allowed the system to have a high level of vulnerability detection accuracy. More work needs to be done on reducing false positives, however the high accuracy rate of detecting true positives meant the precision and sensitivity scores for the system were still high. Brief deployment requirements were also highlighted. The evaluation looked at how effective the system was.

Finally, code documentation was shown and explained how this aids the maintenance of the project.

A static source code analyser was developed that targets PHP and web-based vulnerabilities. Through testing and evaluation, it proved to be useful and even better at detecting vulnerabilities than existing tools but could be improved by reducing false positives. Suggestions were made on how it can be integrated into software development projects.

IV. CONCLUSION

With the ever-growing threat of cybercrime there is a need for improving the security of software. Training developers to be aware of all security concerns is challenging and hiring specialised security teams may not be feasible for small organisations. The ubiquitous nature of web-based applications makes it a target for cyber criminals who exploit vulnerabilities in software. Static source code analysers are an underutilized tool that can be critical when identifying and removing such vulnerabilities. These tools are not perfect and due to the difficulties developing them, and false positives are still quite high. It has been proven to be theoretically

impossible to be free of false positives and false negatives in every case. However, the focus of static analysers is to highlight potential security issues or bugs in code. The fact that they are imperfect does not prevent them from having value.

The industry needs to adapt to better utilize such tools in the software development life cycle. This project has shown that static analysers have the ability to successfully identify and reduce web-based security issues. Challenges continue to exist in areas related to obfuscated code and ease of automated integration with existing software projects. Techniques and ideas of using dynamic analysis to verify identified issues has been explored. Low level components used in compilers such as lexical analysers make it possible to *understand* source code and detect vulnerabilities. The developed application has shown that the detection accuracy of current static analysers can be improved, weaknesses have been identified when obfuscated code is analysed, current reporting systems can be modernised to keep up with the fast-evolving industry and secure code suggestions instead of just a description would help to train developers on how to write secure code.

A. FUTURE WORK

We wouldn't be where we are today without standing on the shoulders of giants. Static analysis covers research going all the way back to Alan Turing through to Peter O'Hearn who is currently in charge of static analysis at Facebook. That being said, there's still large room for improvements in this area. The following section proposes ideas for future research and additional development work to be conducted on the application.

1. BROADER SCOPE

The first and more obvious direction for future work would be to expand the domain from web-based to analyse desktop, server, mobile and IoT source code. The ubiquitous nature of computers has meant that we rely on technology for everyday tasks, ranging from banking, communication, and even managing the temperature of our homes via smart thermostat devices and much more. It is critical that these devices are secure from cyber criminals and program analysis can help in that area. More research can be done to ensure static analysers are performing the best they can in each domain. Moreover, the analysers need to be easier to integrate into software development projects.

2. DEVELOPMENT WORK

The developed application can be expanded to allow for better usability, improved detection capabilities and support for older versions of PHP.

A) CONFIGURATION

The next step in expanding the developed static analyser is to allow it to be configured. This will help developers use the tool in a variety of different projects that span over a range of platforms and architectures. Allowing configuration of the tool will allow for it to be easier to adapt to different situations. Certain issue types should be suppressible before and after analysis. Reported issues should allow the user to mark an issue as a false positive. Other variations in system and platform usage should be covered, such as the support of past versions of PHP and even expansion into supporting other programming languages.

B) CODE SUGGESTIONS AND DESCRIPTIONS

Instead of just reporting back that a security issue has been found. The analyser could suggest a secure code alternative alongside a short description of the issue. This feature is similar to the features commonly seen in modern compilers and IDEs that detect syntactic errors. This would allow for developers to receive direct and immediate training about the code they write.

3. STATIC ANALYSIS AS PART OF THE BUILD PROCESS

It is important that the tool can be integrated into existing projects. When developers haven't looked at even their own code in over a year, they need to spend some time to get familiar with the code again. This process is called context switching. To solve this problem, this paper proposes future research on instant static analysis that is performed on source control code commits. The code can be committed by a developer, and a centralized server can inspect the repository and run the tool on the code change at that moment. If an issue is detected it will automatically raise an issue and proceed to show a detailed report related to that code change directly in the commit's issue description page. Integration should also be flexible so it should work with a range of version control software and tools such as Jenkins so that it can perform checks on nightly builds. This is important as organizations need a tool that not only works but is easy to integrate into their existing project.

4. ADVANCED DETECTION AND VERIFICATION

A proposal is made to combine dynamic analysis with static analysis. Issues detected by the static analyser can be tested and verified in a sandbox environment that emulates the program in a dynamic manner. Using counterexample guided abstraction refinement (CEGAR) the system can test if the issues are genuine or the result of an incomplete abstraction. The dynamic system will then report back to the static analyser based on its findings, and these steps can be repeated. This allows for false positives to be reduced dramatically, as long as the dynamic sandboxed environment is comparable enough to the real thing.

Both the static and dynamic part of the analysers can work together to maximize code coverage using concolic testing. This allows the source code to be understood and symbolic execution can be performed in conjunction with dynamic execution of the program. This overcomes the code coverage limitation of dynamic analysis.

An abductive inference-based algorithm can be used to meaningfully interact with users by generating small and relevant queries that capture exactly the information the analysis is missing to validate or refute the existence of an error in the program. This method tries to seek the simplest and most likely explanation for the observed issue.

Although Edsger Dijkstra, famously once said “program testing can be used to show the presence of bugs, but never to show their absence!” (Dijkstra, 1970). Some modern techniques attempt to prove this wrong (The MathWorks, Inc., n.d.). Further research needs to be done on abstract interpretation and how it can be used to detect or prove the absence of web-based security issues through sound approximation. This would also allow for true negatives to be provided and perform better analysis on the tool’s effectiveness.

These suggestions will use novel techniques to improve vulnerability detection coverage and accuracy, software projects will also benefit from improved integration and interoperability.

V. GLOSSARY

AJAX - Asynchronous JavaScript and XML

API - Application Programming Interface

AWS - Amazon Web Services

CLI - Command Line Interface

COCOMO - Constructive Cost Model

CPU - Central Processing Unit

CWE - Common Weakness Enumeration

EC2 - Elastic Compute Cloud

GDPR - General Data Protection Regulation

GUI - Graphical User Interface

HTTP - Hypertext Transfer Protocol

IDE - Integrated Development Environment

MD - Message Digest

MVC - Model–View–Controller

NIST - National Institute of Standards and Technology

OGNL - Object-Graph Navigation Language

OS - Operating System

OWASP - Open Web Application Security Project

REST - Representational State Transfer

RSA - Rivest–Shamir–Adleman

SARD - Software Assurance Reference Dataset

SDK - Software Development Kit

SDLC - Software Development Life Cycle

SHA - Secure Hash Algorithm

SLOC - Source Lines of Code

SQL - Structured Query Language

SSH - Secure Shell

UI - User Interface

URL - Uniform Resource Locator

XML - Extensible Markup Language

XSS - Cross-Site Scripting

XXE - XML External Entity

VI. REFERENCES

Aho, A. V., Lam, M. S., Sethi, R. & Ullman, J. D., 2006. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Essex: Pearson.

AV-TEST, 2019. *Malware Statistics & Trends Report*. [Online]
Available at: <https://www.av-test.org/en/statistics/malware/>
[Accessed 4 August 2019].

Bakken, S. S. et al., 2004. *Tokenizer functions*. [Online]
Available at: <https://www.macs.hw.ac.uk/~hwloidl/docs/PHP/ref.tokenizer.html>
[Accessed 8 August 2019].

Barbosa, E., 2009. *Taint Analysis*. São Paulo, s.n.

Beck, K. et al., 2001. *Manifesto for Agile Software Development*. [Online]
Available at: <http://agilemanifesto.org/>
[Accessed 8 August 2019].

Chess, B. & West, J., 2007. *Secure Programming with Static Analysis*. s.l.:Addison-Wesley Professional.

Ciortea, L. et al., 2009. *Cloud9: A Software Testing Service*, s.l.: s.n.

Cloudflare, 2019. *What Is OWASP? What Are The OWASP Top 10?*. [Online]
Available at: <https://www.cloudflare.com/learning/security/threats/owasp-top-10/>
[Accessed 9 July 2019].

Cybersecurity Ventures, 2018. *Application Security Report 2017*. [Online]
Available at: <https://cybersecurityventures.com/application-security-report-2017/>
[Accessed 4 August 2019].

Department for Digital, Culture, Media and Sport, 2019. *Cyber Security Breaches Survey 2019: Statistical Release*, s.l.: s.n.

Dijkstra, E. W., 1970. *Notes On Structured Programming*, s.l.: Techn. Hogeschool.

Existek, 2017. *SDLC Models Explained: Agile, Waterfall, V-Shaped, Iterative, Spiral*. [Online]
Available at: <https://existek.com/blog/sdlc-models/>
[Accessed 7 August 2019].

Ghahrai, A., 2018. *Static Analysis vs Dynamic Analysis in Software Testing*. [Online]
Available at: <https://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/>
[Accessed 5 July 2019].

GitHub, 2019. *Code Search · GitHub*. [Online]
Available at: <https://github.com/search>
[Accessed 7 July 2019].

Gleirscher, M., Golubitskiy, D., Irlbeck, M. & Wagner, S., 2014. Introduction of Static Quality Analysis in Small and Medium-Sized Software Enterprises: Experiences from Technology Transfer. *Software Quality Journal*, 22(3), pp. 499-542.

Goldstein, A., 2019. *Is One Programming Language More Secure Than The Rest?*. [Online]
Available at: <https://resources.whitesourcesoftware.com/blog-whitesource/is-one-language-more-secure>
[Accessed 7 August 2019].

Gousios, G., 2013. The GHTorrent Dataset and Tool Suite. *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 233-236.

Hex-Rays SA, 2015. *IDA: About*. [Online]
Available at: <https://www.hex-rays.com/products/ida/>
[Accessed 21 July 2019].

Jones, N., 1997. *Computability and Complexity: From a Programming Perspective*. Massachusetts, USA: MIT Press.

Kumar, R. & Garg, . K., 1994. *Modeling and Control of Logical Discrete Event Systems*. New York, US: Springer.

Lent, J., 2014. *Experts flunk out on secure coding practices*. [Online]
Available at: <https://searchsoftwarequality.techtarget.com/opinion/Experts-flunk-out-on-secure-coding-practices>
[Accessed 9 August 2019].

Mitchell, R., 2015. *Web Scraping with Python*. 1st ed. Sebastopol, CA: O'Reilly Media, Inc..

Modern CISO, 2018. *Securing Industrial Control Systems: A Holistic Defense-In-Depth Approach*. [Online]
Available at: <https://modernciso.com/2018/05/15/securing-industrial-control-systems-a-holistic-defense-in-depth-approach/>

[Accessed 5 August 2019].

Møller, A. & Schwartzbach, M. I., 2018. *Static Program Analysis Part 1 – the TIP language*. [Online]

Available at: <https://cs.au.dk/~amoeller/spa/1%20-%20TIP.pdf>

[Accessed 21 July 2019].

Moser, A., Kruegel, C. & Kirda, E., 2007. Limits of Static Analysis for Malware Detection. *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 421-430.

National Health Executive, 2018. *WannaCry cyber-attack cost the NHS £92m after 19,000 appointments were cancelled*. [Online]

Available at: <http://www.nationalhealthexecutive.com/Health-Care-News/wannacry-cyber-attack-cost-the-nhs-92m-after-19000-appointments-were-cancelled>

[Accessed 4 August 2019].

nodejs.org, n.d. *Cluster | Node.js v12.8.0 Documentation*. [Online]

Available at: <https://nodejs.org/api/cluster.html>

[Accessed 8 August 2019].

O’Hearn, P., 2018. Continuous reasoning: Scaling up the impact of formal. *33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, p. 13.

OWASP, 2017. *OWASP Top 10 - 2017*. [Online]

Available at: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

[Accessed 8 August 2019].

OWASP, 2019. *Static Code Analysis - OWASP*. [Online]

Available at: https://www.owasp.org/index.php/Static_Code_Analysis

[Accessed 7 July 2019].

php.net, 2019. *PHP: List of Parser Tokens - Manual*. [Online]

Available at: <https://www.php.net/manual/en/tokens.php>

[Accessed 8 August 2019].

php.net, 2019. *PHP: token_get_all - Manual*. [Online]

Available at: <https://www.php.net/manual/en/function.token-get-all.php>

[Accessed 7 July 2019].

Rao, M., 2019. *Are Static Application Security Testing (SAST) Tools Glorified Grep?*, s.l.: Synopsys.

Schrittwieser, S. & Katzenbeisser, S., 2011. Code Obfuscation against Static and Dynamic Reverse Engineering. *Information Hiding*, pp. 270-284.

Scrapy, 2018. *Downloader Middleware - Scrapy 1.7.3 documentation*. [Online]

Available at: <http://doc.scrapy.org/en/latest/topics/downloader-middleware.html>

[Accessed 7 August 2019].

Sikorski, M. & Honig, A., 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. s.l.:No Starch Press.

- Singh, J. & Singh, J., 2018. Challenges of Malware Analysis: Obfuscation Techniques. *International Journal of Information Security Science*, 7(3), pp. 100-110.
- Sinha, V. S. et al., 2015. Detecting and Mitigating Secret-Key Leaks in Source Code Repositories. *Proceedings of the 12th Working Conference on Mining Software Repositories*, pp. 396-400.
- Sipser, M., 2012. *Introduction to the Theory of Computation*. 3rd ed. s.l.:Course Technology.
- Tayar, G., 2019. *JavaScript Asynchrony and async/await in Selenium WebDriver Tests*. [Online] Available at: <https://applitools.com/blog/javascript-asynchrony-and-asyncawait-in-selenium> [Accessed 8 August 2019].
- The MathWorks, Inc., n.d. *Proving Absence of Run-Time Errors in Software*. [Online] Available at: <https://uk.mathworks.com/company/events/webinars/upcoming/proving-absence-of-run-time-errors-in-software-2590142.html> [Accessed 28 July 2019].
- The MITRE Corporation, 2015. *Disrupting the Attack Surface: Making Life Hard for the Adversary*, s.l.: s.n.
- Thor, W.-M., 2018. *5 top programming languages to learn server-side web development*. [Online] Available at: <https://twm.me/best-programming-languages-and-frameworks-for-server-side-web-development/> [Accessed 7 August 2019].
- Truher, J., 2019. *Invoke-WebRequest*. [Online] Available at: <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest?view=powershell-6> [Accessed 7 August 2019].
- Turing, A. M., 1937. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2), pp. 230-265.
- Viennot, N., Garcia, E. & Nieh, J., 2014. A Measurement Study of Google Play. *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, pp. 221-233.
- W3Techs, 2019. *Usage Statistics and Market Share of Server-side Programming Languages for Websites*. [Online] Available at: https://w3techs.com/technologies/overview/programming_language/all [Accessed 7 August 2019].
- Whitman, M. & Mattord, H., 2014. *Principles of Information Security*. 4th ed. Massachusetts, USA: CENGAGE Learning Custom Publishing.
- Wögerer, W., 2005. *A Survey of Static Program Analysis Techniques - Technische Universität Wien*, s.l.: s.n.
- Woschek, M., 2015. *OWASP Cheat Sheets*, s.l.: s.n.
- Yuschuk, O., 2014. *OlllyDbg*. [Online] Available at: <http://www.ollydbg.de/> [Accessed 21 July 2019].
- Zaharia, A., 2019. *300+ Terrifying Cybercrime and Cybersecurity Statistics & Trends [2019 EDITION]*. [Online] Available at: <https://www.comparitech.com/vpn/cybersecurity-cyber-crime-statistics-facts-trends/> [Accessed 4 August 2019].

Zahger, D., 2017. *Static Code Analysis: Binary vs. Source*. [Online]
Available at: <https://www.checkmarx.com/2017/11/21/static-code-analysis-binary-vs-source/>
[Accessed 21 July 2019].

Zorabedian, J., 2017. *What Developers Need to Know About the State of Software Security Today*. [Online]
Available at: <https://www.veracode.com/blog/secure-development/what-developers-need-know-about-state-software-security-today>
[Accessed 4 August 2019].

Zorabedian, J., 2017. *What Developers Need to Know About the State of Software Security Today*. [Online]
Available at: <https://www.veracode.com/blog/secure-development/what-developers-need-know-about-state-software-security-today>
[Accessed 4 August 2019].

Zorz, Z., 2019. *How are businesses facing the cybersecurity challenges of increasing cloud adoption?*. [Online]
Available at: <https://www.helpnetsecurity.com/2019/02/21/enterprise-cloud-adoption-security/>
[Accessed 20 July 2019].