A Report submitted in partial fulfilment of the regulations governing the award of the Degree of BSc (Honours) Ethical Hacking for Computer Security at the University of Northumbria at Newcastle

Project Report

Implementing Basic Static Code Analysis into Integrated Development Environments (IDEs) to Reduce Software Vulnerabilities.

Ryan Dewhurst - 08026925

2011/2012

General Computing Project

Authorship Declaration

I declare the following:

(1) that the material contained in this dissertation is the end result of my own work and that due acknowledgement has been given in the bibliography and references to **ALL** sources be they printed, electronic or personal.

(2) the Word Count of this Dissertation is ~20,000

(3) that unless this dissertation has been confirmed as confidential, I agree to an entire electronic copy or sections of the dissertation to being placed on the eLearning Portal (Blackboard), if deemed appropriate, to allow future students the opportunity to see examples of past dissertations. I understand that if displayed on eLearning Portal it would be made available for no longer than five years and that students would be able to print off copies or download.

(4) I agree to my dissertation being submitted to a plagiarism detection service, where it will be stored in a database and compared against work submitted from this or any other School or from other institutions using the service.

In the event of the service detecting a high degree of similarity between content within the service this will be reported back to my supervisor and second marker, who may decide to undertake further investigation that may ultimately lead to disciplinary actions, should instances of plagiarism be detected.

(5) I have read the Northumbria University/CEIS Policy Statement on Ethics in Research and Consultancy and I confirm that ethical issues have been considered, evaluated and appropriately addressed in this research.

SIGNED:

Ryan Dewhurst

Date:

Acknowledgements

Marie for her love and support.

Dr. Christopher Laing for his continued support and guidance.

My university colleagues whom have been a source of inspiration, motivation and friendship.

Marisa Fagan for her support and guidance.

Johannes Dahse of the RIPS project for giving me permission to use their Taint Analysis data.

The late Alan Turing for his contributions to modern computing.

Finally, to the many giants within the security industry that has contributed software, research, knowledge, opinion and time to enhance software security.

Abstract

Software security is a problem that has no easy solution. Almost daily there are reports of companies being breached and personal data being compromised. This report aims to research the problem of software security; this includes the software development process and how security plays a role in that process.

This report discusses the design, implementation and testing of a product that aims to help in developing secure software by integrating Static Code Analysis techniques into Integrated Development Environments (IDEs).

This report will conclude that all aims and objectives set in the initial Terms of Reference document were met and recommends that the PHP Interpreter should implement Taint Analysis as well as recommending a debate on a UK law that focuses on software security.

List of Contents

1. Introduction

Traditional methods of reducing software security vulnerabilities are normally employed after the software development process. Instead, this can be done in the early stages of development through the following of a methodology such as a Security Development Lifecycle (SDL). It is evident by the amount of vulnerabilities being discovered within software that SDLs are not widely used or that SDLs are not as efficient as they possibly could be.

According to Microsoft, the later vulnerabilities are found within the development of software, the more time it takes the developer to fix, thus, increasing the cost of the overall development process (Microsoft, 2011).

During the coding stage of a Software Development Life Cycle (SDLC), Microsoft's Secure Development Lifecycle (SDL) recommends three practices within their Implementation phase.

- SDL Practice 8: Use Approved Tools

  The development team should choose and agree on a list of tools and their versions that should be used within the development process.

- SDL Practice 9: Deprecate Unsafe Functions

  The development team should not use unsafe functions within their software. This includes functions that have been deprecated by the programming language.

- SDL Practice 10: Perform Static Analysis

  The development team should perform Static Code Analysis on their software in order to detect potential vulnerabilities.

(Microsoft, 2011)

The author will research modern software development, how security is implemented into the software development process and current static code analysis techniques. A software product will be designed and implemented in the area of secure software development.

The product produced implements deprecated function matching and Static Code Analysis into an Integrated Development Environment (IDE). The purpose of the work is to demonstrate the possibility of combining IDEs with Static Code Analysis to make security more accessible to software developers.

The report concludes that current software development techniques have their limitations. For smaller software projects a Security Development Lifecycle may be too resource intensive.

The report goes on to recommend the wide adoption of Static Code Analysis combined with Integrated Development Environments (IDE), the implementation of Taint Analysis within the PHP Interpreter and a debate on the introduction of a UK law on software security.

1.1. Structure of Report

The main content of the report concentrates on software development, software security and the implementation of a software product. The start of each of the following sections is clearly stated and underlined within the report.

- **Analysis**

  This section describes the problem of software security, giving examples and evidence of the problem. A literature survey of the software development process is carried out to identify how security plays a role within software development. A discussion of possible solutions, justification and specification for a product is given.

- **Synthesis**

  This section describes the work done to develop the product produced. The design of the product is discussed along with design documentation. Particular interesting points of the product's implementation is discussed a long with code examples and figures where appropriate. Finally, the product's testing is discussed.

- **Evaluation and Conclusions**

  This section gives an evaluation on the product and project process. Discussion of problems faced and overcome. The project is concluded and recommendation's for further work is given.

Analysis

2. Software Security

Human beings often make mistakes when writing code for computer applications. These mistakes are known as ‘bugs’ and these ‘bugs’ may have unforeseen consequences, some of them minor, some of them quite serious. For example, a software bug in a Point of Sale (PoS) device may calculate decimal places incorrectly leading to over or under charging the customer. The same bug miss-calculating decimal places within a missile launch system could send a missile on the wrong trajectory.

A particular serious and possibly fatal consequence of a software error occurred between 1985 and 1987. The Therac-25 radiation therapy machine massively overdosed its patients by 100 times the intended dose (Baase, 2002). The reason for the machine overdosing patients was a software bug attributed to bad software design and development (Leveson, 1995).

Some software bugs have security implications such as affecting confidentiality, availability and/or integrity of itself or the wider system it runs on. According to Allen (Allen, et al. 2008) the core properties of secure software are:

- Confidentiality: Only authorised users are able to access the information or resources allocated to them.

- Integrity: Unauthorised users are unable to make modifications to the system or its assets.

- Availability: The system is accessible and operational to authorised users.

Security bugs, more commonly known as ‘vulnerabilities’, that inhibit certain characteristics have over the years been given names, such as Buffer Overflow, Cross-Site Scripting (XSS), Integer Overflow, Information Leakage, Race Conditions and many more.

Buffer Overflow/Overrun

Buffer Overflows are known to affect low-level programming languages such as C and C++ that have direct access to the systems Random Access Memory (RAM) data allocation. A Buffer Overflow occurs when a part of memory has been allocated to the software and user controlled data is put into that part of memory that is too large for the buffer, causing an ‘overflow’ into other parts of the system’s memory. The effect of such a vulnerability being exploited could lead to a system crash causing a Denial of Service (DoS) or allowing a remote attacker to execute commands on the Operating System (OS) known as Remote Command Execution (RCE) (Howard, M & LeBlanc, D et al, 2005).

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is a security vulnerability that generally affects web applications. Cross-Site Scripting (XSS) occurs when un-sanitised user supplied input is used by the application in its output. The effect of XSS being exploited can lead to the defacement of a web application, the theft of users authentication (session) cookies or it can be used as a communication channel to exploit a browser vulnerability such as a Buffer Overflow (Howard, M & LeBlanc, D et al, 2005).

Information Leakage

Information Leakage is a security vulnerability that occurs when the application intentionally or unintentionally discloses certain information about itself or its environment that could be useful to an attacker to carry out further attacks. Information Leakage can occur within application error messages, within client side source code, within TCP packets and many other places. For example, Information Leakage could lead to an attacker knowing internal network Internet Protocol (IP) addresses, the internal network topology and valid system usernames (Howard, M & LeBlanc, D et al, 2005).

An example of a serious consequence of security breach occurred between 2006 and 2008. Alberto Gonzales and two accomplices compromised a US company called Heartland Payment Systems, at the time described by the United States Department of Justice (DoJ) as 'the single largest data breach in US history' (Department of Justice, 2009). According to the Indictment the vulnerability exploited was SQL Injection via one of the companies web applications (Department of Justice, 2009).

It is evident by looking at public online vulnerability tracking databases that in the short term the problem of insecure software is not going to go away. By conducting a search on the Open Source Vulnerability Database (OSVDB) for all vulnerabilities listed from the 1st of January 2011 until the 21st of December 2011 shows that a total of 6,652 separate vulnerabilities were submitted to its database during the time frame (OSVDB, 2011). The same search criteria carried out on the National Institute of Standards and Technology (NIST) National Vulnerability Database shows a total of 4,053 reported vulnerabilities during 2011 (NIST, 2011). These databases rely on vulnerabilities being submitted to them by various organisations and volunteers; the differences in the amount of vulnerabilities between them both could be explained by the popularity of one over the other.

During 2005 a paper titled 'Cyber Security: A Crisis of Prioritization' was presented to the then President of the United States, George W. Bush (PICTA, 2005). PICTA believed that "software development is not yet a science or a rigorous discipline, and the development process by and large is not controlled to minimize the vulnerabilities that attackers exploit." (PICTA, 2005).

In a similar vein the US Department of Homeland Security describes the major difference between secure and insecure software as being the software development process itself, in particular those processes that are used in the requirements specification, design, implementation, deployment, support, and update of the software (Department of Homeland Security, 2006).

This project intends to address some of those concerns by creating a product that can be used within the development of software to further reduce vulnerabilities in the software when implemented.

3. Software Development Life Cycles (SDLCs)

3.1 Introduction

In this section an overview of the Software Development Life Cycle (SDLC) is presented, in doing so different SDLCs will be introduced and their involvement in the development of modern software will be presented.

3.2 Software Development Life Cycle (SDLC)

A Software Development Life Cycle (SDLC), also known as a Software Development Process, is a set of guidelines (a methodology) followed by an individual software developer or by a team of developers to develop and/or maintain new software products. The aim of a SDLC is to avoid past mistakes and improve the development process that in turn should reduce development cost and increase software quality.

The International Standards Organisation (ISO) produces several standards in relation to SDLCs; a brief selection of these is:

ISO 12207: Systems and software engineering - Software life cycle processes. Establishes a common framework for software life cycle processes (ISO, 2008).

ISO 15288: Systems and software engineering - System life cycle processes. Establishes a common framework for the life cycle of for man-made systems (ISO, 2008).

ISO 15026: Systems and software engineering - Systems and software assurance. Establishes a common framework for the measurement of software assurance (IS0, 2008).

Access to these ISO documents is limited, however each standard is concerned with the design and development of software.

The U.S. Department of Health & Human Services (HSS) list five SDLCs they deem as ‘acceptable’: Waterfall, Prototyping, Incremental, Spiral and Rapid Application Development (RAD) (The U.S. Department of Health & Human Services, 2008). These SDLCs will be used to give a brief understanding of the differences between them.

3.3 Waterfall

The Waterfall is a SDLC that ensures software development continually progresses in a linear fashion (like a waterfall), progressing through each of the, typically, six stages. The six stages are: Initial Investigation, Requirements, Design, Coding, Implementation and Operation & Support (maintenance). These six stages or variants of them are seen throughout most other SDLCs.

The Initial Investigation: involves the investigation of any specific knowledge the development team may require to develop the software.

The Requirements: consists of problem analysis and product descriptions. Typically this involves a number of Software Requirements Specification (SRS) documents being completed. A SRS document contains a complete and concise description of the external interface of the software system with its environment, including other software, communication channels, hardware and users (Davis, 1993).

The Design: issues to be considered such as future proofing, usability and security. Detailed plans may be drawn up. The more detailed the design, the less likely it is for problems to arise during later stages.

The Coding: the act of writing the software itself, using the knowledge gained within the initial investigation, requirements and design stages, to construct the software's algorithms. Testing should be carried out on the software at milestones previously set during the design stage.

The Implementation: deployment into the environment in which it is to be used. Further testing should be carried out at this stage for Quality Assurance (QA).

The Support: at this stage changes may be made to the code due to user feedback, bugs, environment changes or any other code changes made after the software has been released.

3.4 Prototyping

Prototyping, instead of being a complete SDLC, focuses on incomplete prototypes of the software. Proof of Concept (PoC) code is produced and re-produced until the software matures and develops into the specification needed.

3.5 Incremental

Within Incremental SDLC many mini waterfalls are performed, all phases of the actual waterfall model are completed for only small parts of the software system, before proceeding to the next increment (The U.S. Department of Health & Human Services, 2008).

3.6 Spiral

The Spiral SDLC is a meta-process that can be implemented into other SDLC processes. There are typically four different stages: Analysis, Evaluation, Development and Planning. As the spiral shape suggests, this is a 'looped' process, once an iteration of the SDLC is complete, the SDLC process starts again.

3.7 Summary

In this section various SDLCs have been briefly introduced. The actual production of code represents a small element of the overall Software Development Life Cycle (SDLC), however it is this element that has the potential to introduce the greatest number of human errors.

4. Security Development Lifecycle (SDL)

4.1 Introduction

In this section the reader will be introduced to the concept of Security Development Lifecycles (SDLs) and see how they fit into a traditional SDLC.

4.2 Security Development Lifecycle (SDL)

A Security Development Lifecycle (SDL) is an extra layer on top of a traditional Software Development Life Cycle (SDLC). In a traditional SDLC, security testing is normally left until the coding and implementation phases. An SDL ensures security plays a role within every stage of the SDLC, from the initial investigation, right through to the support/maintenance phase.

Once such SDL is the Microsoft SDL and according to Microsoft, "[…] the Microsoft SDL is a collection of mandatory security activities, presented in the order they should occur and grouped by the phases of the traditional software development life cycle (SDLC)" (Microsoft, 2010).

There are other SDLs available such as the Adobe Secure Product Lifecycle (Adobe, 2011) and the Cisco Secure Development Lifecycle (CISCO, 2011). This paper will concentrate on one SDL, the Microsoft SDL, as it is the SDL the author is most familiar with and this section's aim is to only introduce the basic concepts of a SDL and not to compare them.

The main reasons to implement a SDL are:

Increase software security

Microsoft has had a mandatory SDL process implementation policy on all software products developed since 2004 (Microsoft, SDL Process Guidance Version 5.1). We can safely assume that all software developed pre 2004 did not undergo an SDL process and that software developed post 2004 did undergo a SDL process within Microsoft.

The Microsoft Windows XP Operating System (OS) was released in 2001, three years before SDLs were mandatory. The Microsoft Windows Vista Operating System (OS) was released in 2007, three years after SDLs were mandatory and according to Microsoft, 'the first Microsoft operating system to benefit from the SDL' (Microsoft, 2011).

According to Microsoft, 'After the first year, Windows Vista had 45% fewer vulnerabilities than Windows XP' (Microsoft, 2011).

Compliance

There are a number of different industry compliance standards depending on the type of software being developed and there are a number of different government standards depending on what country the software is developed or deployed in.

One example of industry compliance standards would be the Payment Card Industry Data Security Standard (PCI-DSS). The standard outlines 'a comprehensive list of requirements that a payment processor must comply with in order to process credit card payments' (PCI-DSS, 2010).

The most applicable requirement to software security and secure software development is PCI Requirement 6, 'Develop and maintain secure systems and applications'. In particular, requirement 6.5, "Develop applications based on secure coding guidelines." (PCI-DSS, 2010).

The PCI-DSS version 2 does not state what 'secure coding guidelines' the payment card processor must use, leaving it open to interpretation.

Other security compliance measures include the US Government Health Insurance Portability and Accountability Act (HIPPA).

Reduce development cost

According to Microsoft, the later vulnerabilities are found within the development of software, the more time it takes the developer to fix, thus, increasing the cost of the development process (Microsoft, 2011).

This is confirmed by a survey produced by The National Institute of Standards and Technology (NIST), published within a report called 'The Economic Impacts of Inadequate Infrastructure for Software Testing' (NIST, 2002). The below table is the direct result of the survey. It illustrates the amount of hours it takes to fix bugs at specific stages of a Software Development Life Cycle (SDLC).

Stage Introduced	**Requirements**	**Coding/unit testing**	**Integration**	**Beta Testing**	**Post-product Release**
Requirements	1.2hrs	8.8hrs	14.8hrs	15.0hrs	18.7hrs
Coding/unit testing	NA	3.2hrs	9.7hrs	12.2hrs	14.8hrs
Integration	NA	NA	6.7hrs	12.0hrs	17.3hrs

Table 4-1 –The time it takes to fix bugs at specific stages of the SDLC.

Although the cost of fixing bugs may reduce with the implementation of a SDL, there are other additional costs to take into consideration; the additional man-hours through planning and paper work, there may also be costs associated with the purchase of additional, possibly outsourced, security products and/or services.

4.3 Microsoft Security Development Lifecycle (SDL)

The Microsoft Security Development Lifecycle (SDL) has five phases; each one designed to be implemented into an existing traditional Software Development Life Cycle (SDLC) phase. The five phases are: Requirements, Design, Implementation, Verification and Release. There is also a Pre-SDL requirement, which is Training and a Post-SDL requirement, which is Response. Within the seven phases there are a total of seventeen different ‘practices’.

The table below shows the relationships between the five Microsoft SDL phases, including the Pre and Post SDL requirements and the five Waterfall SDLC phases.

Microsoft SDL	Training	Requirements	Design	Implementation	Verification	Release	Response
Waterfall SDLC	N/A	Requirements	Design	Coding		Implementation	Support

Table 4-2 – Tables comparing the phases of the Microsoft SDL and the Waterfall SDLC.

Pre-SDL - Training

Microsoft describes security training (practice 1) as a ‘Pre-SDL requirement’ as it is not directly related to the actual software being developed. The security training is designed to give all of the software development team, including developers, testers and managers a basic understanding of security concepts and to inform them of the latest trends in security and privacy. Microsoft advises that one unique training class should be taken at least once a year (Microsoft, 2011).

Phase 1 - Requirements

SDL-Practice 2: Security Requirements: This practice is designed to lay the groundwork for the SDL. Assigning security professionals to the project, assigning minimum security and privacy standards and creating a security vulnerability tracking system (Microsoft, 2011).

SDL-Practice 3: Define Quality Gates/Bugs Bars: A bug bar is an agreed level of severity that the application is required to meet before being released. The bug bar defines four levels of severity: Critical, Important, Moderate and Low. An example would be that a development manager would agree with his team that the software could not be released with any bugs classified as Moderate or above (Microsoft, 2011).

SDL-Practice 4: Security and Privacy Risk Assessment: This practice requires that a Security Risk Assessment (SRA) and a Privacy Risk Assessment (PRA) be completed (Microsoft, 2011).

Phase 2 – Design

SDL-Practice 5: Establish Design Requirements: Within this practice, security and privacy should be designed to fit in with the overall software design specification.

This prevents secure and security features being 'bolted on' at a later stage of the SDL (Microsoft, 2011).

SDL-Practice 6: Attack Surface Analysis/Reduction: Attack surface reduction reduces the risk of an attack from a malicious user by reducing the potential weak points in the application. Disabling or limiting features within the software can reduce the attack surface (Microsoft, 2011).

SDL-Practice 7: Threat Modelling

A threat model is an analysis that helps determine the level of risk to the application and how attacks can occur. A threat model's objective is to highlight areas of the application of most risk to vulnerabilities or attack, allowing time and effort to be concentrated in those areas (Microsoft, 2011).

Phase 3 - Implementation

SDL-Practice 8: Use Approved Tools: This practice requires the development team to define and keep updated a list of approved tools to be used during the coding phase of a SDLC. The list of tools should contain the version of the tool to be used (this would normally be the latest) and its required configuration (Microsoft, 2011).

SDL-Practice 9: Deprecate Unsafe Functions: This practice requires the development team to define a list of unsafe functions. Some programming language functions and APIs may be deemed as insecure or less secure than others, these may include deprecated functions (Microsoft, 2011).

SDL-Practice 10: Perform Static Analysis: This practice requires the development team to perform static code analysis on the software. Static code analysis as the name suggests is an analysis of the source code while it is in a 'static' state, static, meaning pre-compilation or interpreted source code. Static code analysis may include automated solutions such as static code analysis software or human manual code reviews on the source code. Static code analysis may be referred to as a 'White Box' testing technique, where the tester has all of the information at hand unlike a 'Black Box' testing technique where the tester has no prior or internal knowledge about the test subject. Static Code Analysis will be discussed in more detail later on in the paper in section 4.

Phase 4 – Verification

SDL-Practice 11: Perform Dynamic Analysis: Dynamic Analysis is when software is compiled/interpreted and then analysed in its running state to assess that it behaves as expected. Some of the behaviours that will be observed are memory leaks, memory allocation and/or software crashes (Microsoft, 2011).

SDL-Practice 12: Fuzz Testing: The act of 'fuzzing' is to send random data to all of a software applications input points. This is done while the software is in a dynamic state and in an automated fashion. While the software is processing the random data it is being monitored with the intent to find any potential bugs (Doyle & Fly. et al. 2007).

SDL-Practice 13: Attack Surface Review: This practice is to ensure the original attack surface review completed during the design stage and compare it against a new one completed after the source code has been written (Microsoft, 2011).

Phase 5 – Release

SDL-Practice 14: Incident Response Plan: An Incident Response Plan (IRP) should be implemented even if no software vulnerabilities were found to affect the software at the time of release. This is due to the possibility of new threats emerging over time (Microsoft, 2011).

SDL-Practice 15: Final Security Review: The Final Security Review (FSR) is an examination of the entire Security Development Lifecycle (SDL) process on the software. The FSR would include a review of the threat model, tools output and the software's performance against the quality/bug bars (Microsoft, 2011).

SDL-Practice 16: Release/Archive: Before the software is released the development team must certify that all security and privacy criteria have been met. All documentation and source code must be archived in a secure place and backed-up in case it needs to be referred to in future (Microsoft, 2011).

Post-SDL - Response

SDL-Practice 17: Execute Incident Response Plan: Execute the Incident Response Plan (IRP) if a security incident occurs. This requirement is considered to be 'Post-SDL', a requirement that exists after the SDL has been completed. Although not part of the original Microsoft SDL, it would be recommended to monitor system and Intrusion Detection Log files at this point (Microsoft, 2011).

4.4 SDL Effectiveness

Michael Howard is the Principal Security Program Manager at Microsoft and the author of several software security books such as 'Writing Secure Code', 'The Security Development Lifecycle' and '19 Deadly Sins of Software Security'. Speaking of his own experiences he goes on to conclude that there is no one solution to software security. A Security Development Lifecycle (SDL) will help to reduce software vulnerabilities, however, achieving zero vulnerabilities within software is unachievable (Howard, 2007).

New types of vulnerabilities are constantly being discovered; some recent examples include Cross-Site Tracing (XST) (Grossman, 2003) or Clickjacking (Hansen & Grossman, 2008). It would be hard if not impossible to predict these vulnerabilities during the SDL, however, it would be possible to reduce the risk to exposure of new vulnerabilities by reducing the attack surface as discussed in Microsoft SDL practices 6 and 13.

Another possibility to further improve the SDL's effectiveness could be to combine the three practices within 'Phase 3 – Implementation' of the SDL. By combining 'SDL-Practice 8: Use Approved Tools', 'SDL-Practice 9: Deprecate Unsafe Functions' and 'SDL-Practice 10: Perform Static Analysis' into the development environment itself, it may be possible to lower the time needed to carry out these practices as well as reduce vulnerabilities in the final product.

5. Static Code Analysis

In this section of the paper Static Code Analysis tools and techniques will be researched in order for the author to have a better understanding of the topic. This will benefit the author when implementing the final product.

5.1 Introduction

Static Code Analysis as briefly described in Microsoft SDL-Practice 10 is when source code is analysed in an automated fashion in a 'static' (non-running) state. Modern compilers use Static Code Analysis to translate source code into a machine-readable language. Specialist software tools use the same techniques to discover potential vulnerabilities. There are both Open Source/Free and commercial Static Code Analysis tools available. As well as the specialist tools, basic Operating System commands such as the Linux 'grep' command can be used to locate potential vulnerabilities.

Example recursive grep command used to find MySQL queries within PHP:

```
grep -ir "mysql_query" *
```

A small selection of Static Code Analysis tools and the programming languages they support are listed below.

Open Source / Free tools:

Name	Analyses
RIPS	PHP
Agnitio	Objective-C, C#, Java & Android
Microsoft FxcCop	.NET
FlawFinder	C/C++
Pixy	PHP

Table 5-1 – Table showing the different free SCA tools available.

Commercial tools:

Name	Analyses
HP Fortify	Multiple
Veracode	Multiple

CodeSecure	Multiple
Rational AppScan	COBOL & SAP

Table 5-2 – Table showing the different commercial SCA tools available.

5.2 Static Code Analysis Techniques

5.2.1 Data Flow Analysis

Data flow analysis is used to collect run-time (dynamic) information about data in software while it is in a static state (Wögerer, 2005).

There are three common terms used in data flow analysis, basic block (the code), Control Flow Analysis (the flow of data) and Control Flow Path (the path the data takes):

Basic block: A sequence of consecutive instructions where control enters at the beginning of a block, control leaves at the end of a block and the block cannot halt or branch out except at its end (Wögerer, 2005).

Example PHP basic block:

```
$a = 0;
$b = 1;

if ($a == $b)
{ # start of block
  echo "a and b are the same";
} # end of block
else
{ # start of block
echo "a and b are different";
} # end of block
```

Control Flow Graph (CFG): An abstract graph representation of software by use of nodes that represent basic blocks. A node in a graph represents a block; directed edges are used to represent jumps (paths) from one block to another. If a node only has an exit edge, this is known as an 'entry' block, if a node only has a entry edge, this is know as an 'exit' block (Wögerer, 2005).

Example Control Flow Graph; 'node 1' represents the entry block and 'node 6' represents the exit block.

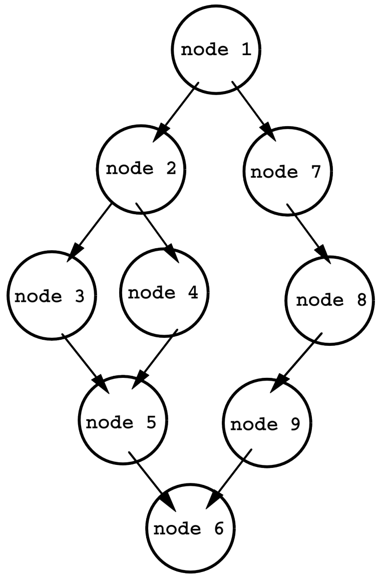


Figure 5-1 – Example Control Flow Graph.

Control Flow Path (CFP): The logical path flow of the source code represented with nodes. In the example Control Flow Graph above the 'path' is the ordering of the nodes and the arrows that link them, beginning with an entry node and ending with an exit node.

5.2.2 Taint Analysis

Potentially Vulnerable Functions (PVFs) also known as 'sinks' are identified within source code via pattern string matching. Parameters passed to the PVFs are traced back to their inception via use of Data Flow Analysis. If the PVFs parameters were 'tainted' by user controlled input, the source code being examined is marked as being vulnerable. If, however, the user-tainted variables have been secured by the use of a 'securing' (sanitizing) function the source code being examined will not be marked as vulnerable (Dahse, 2010). Taint (verb), "contaminate or pollute (something)" (Oxford, 2010).

Some programming languages such as Perl (Patwardhan. et al, 2002) and Ruby (Thomas. et al, 2009) have Taint Checking built into them and enabled in certain situations such as accepting data via CGI.

Using PHP as an example to show basic taint analysis.

```
1. $user_tainted = $_GET['tainted'];
2. echo($user_tainted);
```

In the example above the variable $user_tainted is tainted by user supplied input via the super global $_GET variable. The $user_tainted variable is then passed to a sink 'echo()' that outputs the user supplied input. This is a simplified example of Cross-Site Scripting (XSS) vulnerability.

```
1. $user_tainted = $_GET['tainted'];
2. $sanitized = htmlspecialchars($user_tainted);
3. echo($sanitized);
```

The example above is the same as the previous example except that the $user_tainted variable is now passed through a ‘sanitizing’ function, htmlspecialchars, rendering the piece of code not vulnerable to Cross-Site Scripting (XSS).

The htmlspecialchars PHP function converts the following special characters into their HTML entities equivalent (PHP, 2012).

Special Character	HTML
&	&
"	"
‘	'
<	<
>	>

Table 5-3 – Table showing the HTML equivalents to special chatacters.

Example JavaScript passed through the htmlspecialchars function:

Before	<script>alert(1)</script>
After	<script>alert(1)</script>

Table 5-4 – Table showing the before and after of a string passed through the htmlspecialchars function.

Taint analysis uses three arrays of data to identify potential vulnerabilities; sources, sinks and sanitizing functions. Using PHP to show examples of sources, sinks and sanitizing functions below. This is not a comprehensive list, however it should give an idea of the various functions that could be used in PHP taint analysis.

Sources

Sources are possible tainted data from users, files, databases or any other user controllable input.

- User tainted: $GLOBALS, $_SERVER, $_GET, $_POST, $_FILES, $_COOKIE, $_SESSION, $_REQUEST, $_ENV.

- File tainted: file(), fopen(), popen(), file_get_contents(), fread(), fscanf().

- Database tainted: mysql_fetch_array(), mysql_fetch_assoc(), mysql_fetch_field(), mysql_fetch_object(), mysql_fetch_row().

Sinks

Sinks are potential vulnerable functions where sources may end up, such as outputting data or running SQL queries.

- Cross-Site Scripting (XSS): echo(), print(), printf().

- SQL Injection: mysql_query(), pg_query().
- Command Execution: exec(), shell_ exec(), system(), proc_open().
- File Inclusion: include(), require(), require_once(), include_once().

Sanitizing Functions

Sanitizing functions are functions that make sources of tainted variables 'un-tainted' before reaching a sink.

- Cross-Site Scripting (XSS): htmlspecialchars(),htmlentities().
- SQL Injection: mysql_real_escape_string(),sqlite_escape_string().
- Command Execution: escapeshellcmd(),escapeshellarg().
- File Inclusion: None found within PHP.

5.2.3 String Matching

String matching is simple string comparison that can be used to match function names within source code to a list of undesirable functions. This technique can be used to identify deprecated and/or unsafe functions as described in Microsoft SDL-Practice 9.

PHP started to deprecate functions and show warnings since the release of version 5.3.x. The following functions were deprecated within PHP 5.3x (PHP, 2012):

call_user_method
call_user_method_array
define_syslog_variables
dl
ereg
ereg_replace
eregi
eregi_replace
set_magic_quotes_runtime
session_register
session_unregister
session_is_register
set_socket_blocking
split
spliti
sql_regcase
mysql_db_query

mysql_escape_string

Tabke 5-6 – Table showing a list of PHP deprecated functions.

5.2.4 Lexical Analysis

Lexical Analysis converts source code syntax into 'tokens' of information in an attempt to abstract the source code and make it easier to manipulate (Sotirov, 2005).

The PHP tokeniser allows users to use the PHP Lexical Analysis engine (PHP, 2012). PHP has a built in API function that converts PHP source code into an array of PHP tokens. token_get_all() parses the given source string into PHP language tokens using the Zend engine's lexical scanner" (PHP, 2012).

The tokenisation source code is in the 'ext/tokenizer/tokenizer.c' file in PHP version 5.

Pre tokenised source code:

```
<?php $name = "Ryan"; ?>
```

Post tokenised source code:

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

Sample PHP tokens (see Appendix C for a complete list):

<table>
<tr><th>Token</th><th>Syntax</th></tr>
<tr><td>T_VARIABLE</td><td>$foo</td></tr>
<tr><td>T_FOR</td><td>for</td></tr>
<tr><td>T_IF</td><td>if</td></tr>
<tr><td>T_COMMENT</td><td>// or #, and /* */ in PHP 5</td></tr>
<tr><td>T_BOOLEAN_AND</td><td>&&</td></tr>
<tr><td>T_BOOLEAN_OR</td><td>||</td></tr>
</table>

Table 5-7 – Table showing a small sample of PHP tokens.

5.3 Limitations

Static Code Analysis will not detect business logic flaws. Static Code Analysis tools have no awareness of context; by analysing the source code they are unaware of what functionality should be available to individual users. For example static code analysis tools would not know whether an individual user should be able to make database amendments or not.

Another limitation to Static Code Analysis tools is False Positives and False Negatives. Static Code Analysis tools are dependant on the rules their developers have written for them; these rules may be too limited or too greedy in their pattern matching.

5.3.1 False Positives

Pixy, a PHP Static Code Analysis tool was found to produce a false positive rate of around 50% (Jovanovic, et al. 2007). Every false positive has to be manually reviewed; the more there are the more time it takes to interpret the results in a meaningful way. If the false positive rate of Pixy is around 50% then maybe it is no better than a flip of a coin in deciding if an algorithm is vulnerable or not.

In the following example the RIPS version 0.51 PHP Static Code Analysis tool was used to scan the source code of a custom dynamic PHP web application. The screenshot shows that RIPS incorrectly identified a piece of code as being vulnerable to Cross-Site Scripting (XSS).

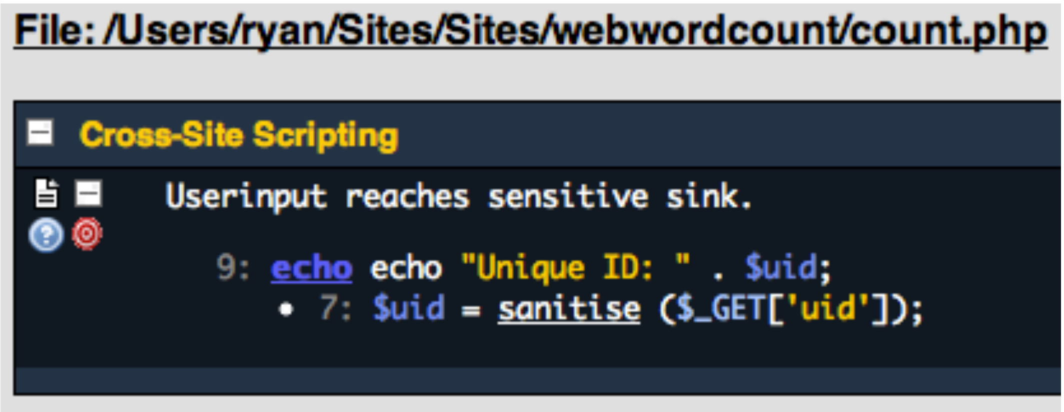


Figure 5-2 – RIPS.

The reason RIPS thinks it is vulnerable because it identifies the source of user tainted variable (uid) being passed to a sensitive sink (echo). RIPS does not take into account that the 'uid' variable is fist passed through a custom function called 'sanitise' that contains the PHP htmlspecialchars sanitising function. As discussed in the 'Taint Analysis' section of the paper, htmlspecialchars converts special characters to their HTML equivalent, rending the source code not vulnerable to Cross-Site Scripting (XSS).

To fix the aforementioned False Positive within RIPS, the 'sanitise' function could be added to RIPS's 'sanitizing functions' array. A better solution would be for RIPS to analyse the contents of the 'sanitise' function to discover the htmlspecialchars function within it.

5.3.2 False Negatives

A false negative, in the context of Static Code Analysis, is when vulnerabilities exist within the software being analysed, however the Static Code Analyser does not identify them. False negatives could be seen as being more serious than a false positive, at least with a false positive you can deduce if it is vulnerable or not. A False Negative may give a false sense of security.

False Negatives can occur when Static Code Analysis tools do not check for certain types of vulnerabilities, check for the vulnerabilities in the wrong way or there is an underlying logic problem with the Static Code Analysis tools themselves.

6. Integrated Development Environments (IDEs)

Integrated Development Environments (IDEs) are programs that aid in the development of software, specifically in the writing of source code. IDEs are commonly desktop applications with Graphical User Interfaces (GUIs), however, there are web based IDEs available and GUIs are not always necessary.

Textmate

Textmate is a commercial Mac OS X only IDE with support for 'over 50 programming languages' (Textmate, 2012).

NetBeans

NetBeans is Open Source/Free cross-platform IDE with support for 'Java, XML, DTD, CSS, HTML, ERB, RHTML, JSP, Javadoc, JavaScript, PHP, Groovy, C and C++, and more' (NetBeans, 2012).

Eclipse

Eclipse is a Java OpenSource/Free cross-platform IDE with support for 'Ada, C, C++, COBOL, Java, Perl, PHP, Python, R, Ruby (including Ruby on Rails framework), Scala, Clojure, Groovy and Scheme' (Eclipse, 2012).

Cloud9

Cloud9 is an OpenSource/Free/Commercial online AJAX 'Development as a Service' (DaaS) IDE with support for 'Javascript and HTML/CSS' as well as limited support for 'Coffeescript, Ruby, PHP and many others' (Cloud9, 2012).

The above IDEs share common functionality, such as:

- Graphical User Interface (GUI).
- The ability to open/edit/save source code.
- Syntax highlighting.
- Debugging features.
- Compiler/interpreter.
- File management.

7. Product Development

The problem discussed within this paper is that of insecure software and its consequences. The author has researched the software development process and how security can be implemented into a development process. From the many examples given throughout the paper it is clear to the author that there is room for improvement in the development process.

The product outlined within the following sections will attempt to make it easier to detect potential vulnerabilities at the earliest stage possible within the coding stage of the development process. This will be accomplished by the use of Static Code Analysis techniques within the IDE itself.

7.1 Product Requirements

Based on the research, literary review and the project's main aims and objectives set out in the Terms of Reference (TOR) document. There are three product requirements as stated in the list below.

The product's three requirements are:

- To combine the three stages of the Implementation phase of the Microsoft SDL.

  This requirement was one of the main objectives set out in the Terms of Reference (TOR) document and will combine the three Implementation phases of the Microsoft SDL into one product. The three practices are practice 8 (Use Approved Tools), practice 9 (Deprecate Unsafe Functions) and practice 10 (Perform Static Analysis).

  By combining the three phases it is hoped that the Implementation phase of an SDL is easier to implement and thus the use of SDLs more widely adopted.

- To reduce the overall amount of vulnerabilities within software.

  This requirement will aim to detect vulnerabilities within software thus increasing its security. The whole point of the product and this project is to increase software security by combining an IDE with Static Code Analysis.

- To create a basic but usable IDE with basic Static Code Analysis built in.

  This requirement was one of the main objectives set out in the Terms of Reference (TOR) document. The concept of combining an IDE with Static Code Analysis to detect vulnerabilities at the earliest stage, minimising development/support cost and increasing software security.

7.2 Product Specification

The intention of the solution is a product that improves the SDL by helping decrease the amount of vulnerabilities within software. The product will be a basic Integrated Development Environment (IDE) that combines Static Code Analysis techniques within it. This will make the developer aware of vulnerabilities within their software at the earliest stage, allowing them to fix the problem and produce code that is more secure.

From the research carried out in the Analysis section of this report, it is clear that there is a general consensus that the problem of insecure software is down to the development process. For that reason, the development process, specifically the coding stage, will be combined with Static Code Analysis.

The product will consist of:

- A basic IDE with a GUI.
- Identify deprecated and/or unsafe functions within source code.
- Identify possible vulnerabilities within source code.

7.3 Product Function

The Static Code Analysis engine needs to take source code as input from the GUI and return its findings as output to the GUI. The Static Code Analysis engine and IDE GUI source code should be kept separate and not rely on each other. The Static Code Analysis engine should be developed to initially support one programming language due to time and scope of the project; however, consideration for future expansion should be kept in mind.

The Static Code Analysis engine must ensure all code passed to it is properly sanitised as to avoid vulnerabilities like command or code execution.

The following steps could be used:

1. **Clean/Normalise the code.**
   Remove code comments.
   Remove unwanted formatting characters.
   Transform the code into a dataset.

2. **Analyse the code.**
   String Matching.
   Lexical Analysis.
   Data Flow Analysis.
   Taint Analysis.

3. **Output Findings.**
   Format the findings.
   Output findings.

7.3.3 Technologies

The Static Code Analysis engine will initially only analyse PHP source code due to time and scope. PHP allows the use of its own Lexical Analysis; this could be used within the Static Code Analysis engine. PHP is also one of the programming languages the author is most familiar with. PHP is free and Open Source.

According to TIOBE which uses search engines to create statistical data on which programming language has the most lines of code written, PHP comes 6th, making it the number one web programming language on the list (TIOBE, 2012).

Because the source code being analysed will be PHP, to allow for as much integration as possible and minimise the code base, the IDE and Static Code Analysis engine could be developed in PHP.

Because of the choice of development language is a web programming language, the IDE could be web based. This would allow for a cloud9 type of IDE, a 'cloud' IDE.

- **Advantages to the user:**
  Available anywhere with an Internet connection.
  Data backed up remotely.
  Minimised local system resources used.

- **Disadvantages to the user:**
  Internet connection needed.

The IDE GUI could use client side scripting such as JavaScript to create a rich user environment. JavaScript derivatives such as AJAX and jQuery can also be used for asynchronous client-server communications.

Apache will be used as the web server as it is both free and Open Source. The author is also most familiar with Apache over any other web server.

A backend SQL database could be used such as MySQL or SQLite to store user and/or system data. If a backend database is needed, paramatised queries must be used on all queries to prevent SQL Injection.

An alternative method to the solution could be to develop the whole Static Code Analysis engine in JavaScript/AJAX. This would allow for the whole process to take place on the users browser and analysis could take place instantaneously as the users write their code. This would also suppress the need for having to pass the users code to the back-end server for processing and then the results passed forward to the client when complete. If database storage is needed, new features present in HTML5 could be used such as client-side storage, although, this would require users to use modern updated browsers in order for the product to function properly. Another disadvantage to using JavaScript/AJAX would be that my skill level in these languages is low compared to my skill level in PHP and the Lexical Analysis engine built into PHP could not be used.

7.3.4 Product Development

A Waterfall type Software Development Life Cycle (SDLC) will be used and it will incorporate some aspects of the Microsoft Security Development Lifecycle (SDL) where appropriate and as time permits. The benefit of using the Waterfall SDLC is that the development stages are clearly defined. The Microsoft SDL will ensure the product developed will be security focused and contain minimal vulnerable code.

7.4 Testing strategy

The product will have two parts, the IDE and the Static Code Analysis engine, although it should be noted that the IDE would have less importance as compared to the Static Code Analysis engine.

The IDE's functionality can be tested individually. For example, does the menu work as indented? Does the Code Editor work as intended? The GUI should be intuitive and easy to use. These are all based on the users perspective; a survey could be produced and distributed amongst software developers to gather their opinions. Once the IDE is in a working state, the IDE can be used to develop itself, making it easier to spot any potential usability issues.

The Static Code Analysis engine can be tested against known vulnerable source code to see how many False Positives and/or False Negatives are produced. This will be a good indicator of accuracy/effectiveness. The Static Code Analysis engine can also be compared against other Static Code Analysers by analysing the same piece of code and comparing the results.

Both the IDE and Static Code Analysis engine should undertake a white box and black box tests for security vulnerabilities to ensure vulnerabilities are kept to a minimum. This will be part of the Security Development Lifecycle.

8. Summary

8.1 Software Security

Although there exists many products and techniques to reduce vulnerabilities in software, it is apparent that at this moment in time there is no one solution and there may never be a 'silver bullet'. Even with all of the products and techniques available there is no law against producing or selling insecure software to the consumer. There is software compliance such as the PCI DSS, however this only applies to organisations that take or process payments from certain credit cards. Even then, it only provides "a *baseline* of technical and operational requirements" (PCI-DSS, 2010).

8.1.2 Consequences

Campbell et al researched the stock market share price of various companies that had been compromised, they found that there was "some support for the argument that information security breaches adversely affect the future economic performance of affected firms" (Campbell et al, 2003).

Not only does software security possibly impact an organisation's share price, vulnerabilities within software can be used to infect and spread malware. Stuxnet is a piece of 'sophisticated' malware that specifically targets industrial control systems by exploiting vulnerabilities in the Microsoft Windows Operating System and in Siemens control software (Falliere. et al, 2011).

From the examples given throughout this paper, it is clear that software in-security can have a negative financial effect on organisations and can also be used to attack critical infrastructure.

There may also be reputational damage caused, however, this is hard to quantify in terms of concrete financial figures. One example of reputational loss is Ratners jewellers: the then CEO Gerald Ratner publicly described his products as 'crap' which lead to losses in sales, the resignation of Gerald Ratner and a change of the company's name (McKeone, 1995).

8.2 What are SDLCs/SDLs?

Software Development Life Cycles (SDLCs) are a way for development teams to organise their software development into stages, with the intent on lowering development time and increasing code quality. Additionally a Security Development Lifecycle (SDL) can be used to implement security into the development process.

Although SDLs are intended to add security to the development process they are not the 'silver bullet' and as seen previously in this paper zero vulnerabilities within software is unachievable. SDLs may also be costly and time consuming to implement, however, there were neither studies nor data found to support this.

Synthesis

The previous chapters consisted of research and literary reviews into software security, modern software development through the use of Software Development Life Cycles (SDLC), Security Development Lifecycles (SDLs) and a brief look at the different Integrated Development Environments (IDEs) available.

9. Design

One of the main objectives of the project, as outlined in the Terms of Reference (TOR) document available in Appendix A, was to combine the functionality of an Integrated Development Environment (IDE) with the three Implementation phase practices of the Microsoft Security Development Lifecycle (SDL). The three Microsoft SDL practices of the Implementation phase are:

- Microsoft SDL practice 8; Use Approved Tools.
- Microsoft SDL practice 9; Deprecate Unsafe Functions.
- Microsoft SDL practice 10; Perform Static Analysis.

Practice 8 of the Microsoft SDL, the use of approved tools, cannot be implemented into the product. As briefly discussed in the previous Analysis section, this practice requires that the development team define a list of tools that should be used within their software development. The list of tools each development team defines will differ depending on the software they are developing and the technologies they use. Practice 8 is there for not a practice that can be implemented into the author's product.

The product will consist of a Static Code Analysis engine and deprecated function string matching built into an Integrated Development Environment (IDE). This will mean that practice 9 and practice 10 of the Microsoft SDL will be implemented into an IDE.

9.1. Software Development Life Cycle (SDLC)

The Software Development Life Cycle (SDLC) that will be used is the Waterfall SDLC as described earlier in the paper. Development will be split into phases such as requirements, design, implementation, verification and finally maintenance. The splitting of phases into clearly defined sections will help organise the development of the product.

9.2. Integrated Development Environment (IDE)

The IDE GUI will be designed to conform to the design of traditional IDEs to ensure developer familiarity. Figure 9-1 shows the GUI of the Cloud9 web based IDE and Figure 9-2 shows the GUI of the Eclipse desktop based IDE, both representing traditional IDE GUI design. Taking into account the design of the Cloud9 and Eclipse IDEs, the product will have a source code editing area where the user can write, edit and/or paste their source code, line numbering, syntax highlighting, a menu and an information panel.

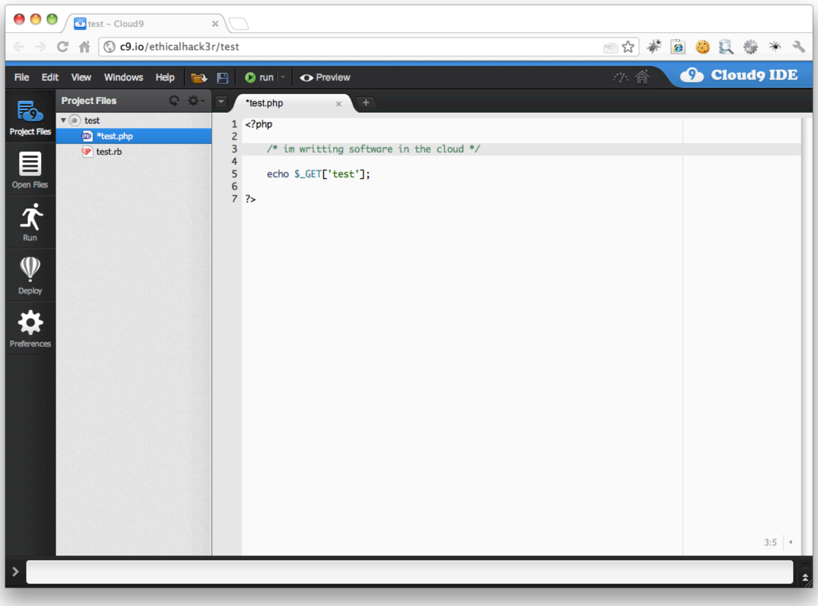


Figure 9-1 – Cloud9 Web IDE GUI.

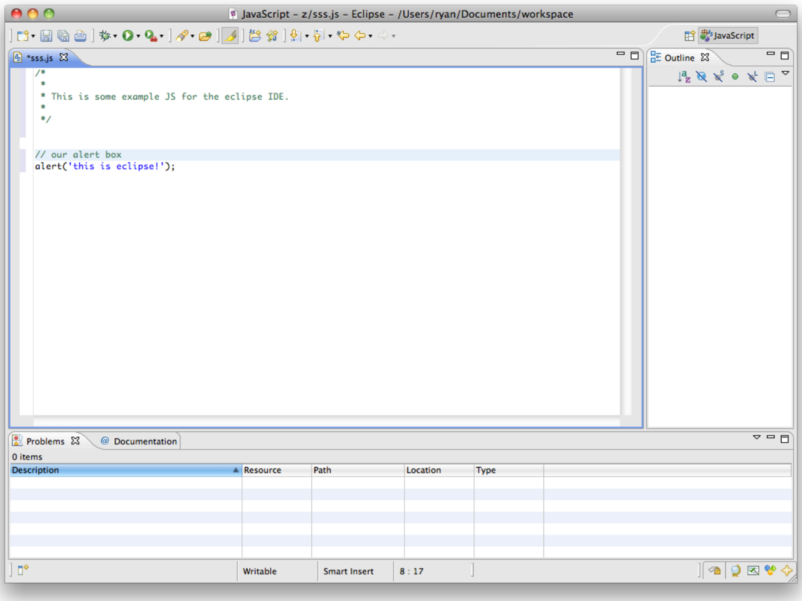


Figure 9-2 – Eclipse Desktop IDE GUI.

The product could be implemented as a desktop application or a web based application. The author has some experience in creating web applications. Web based applications also offer some advantages over desktop applications.

These advantages include but are not limited to:

- No user installation: Web based applications do not require installation.

- No need to support varied user environments: Web based applications do not rely on the Operating System (OS) type, any OS can be used to use web applications.

- Minimal user system resources used: No files will be needed to be on the users file system. No additional software packages to support the application will be run.

- Easier to push updates: Updates can be applied to the central web application, making the updates instant and no need for users to downloaded updated versions.

- Accessible anywhere with an Internet connection: As a minimum, web based applications only require a web browser and an Internet connection to be used.

The disadvantages of having a web based application:

- The user may need a modern browser: Some older web browsers may not have the required functionality that is used within the web application.

- Not all browsers may be supported: Not all web browsers behave in the same way, this may affect support for some browsers due to the time needed to accommodate for them.

- An Internet connection is needed: Web based applications require an Internet connection if the application relies on the server for some of its processing.

The disadvantages may be less of a problem when taking the target audience, software developers, into account. It would be expected that a software developer has an Internet connection and use a modern web browser.

The whole user experience will take place on one page without the need for any reloading of the page. This will be achieved with JavaScript, AJAX and HTML5 techniques. The code editor will load with some sample PHP source code, with a brief explanation of what the product does; this will ensure the purpose of the application is made clear and the user can test how the application works with little effort.

There will be three main components to the IDE GUI, the main menu, the code editor and the information panel.

1. **Main Menu**

   The main menu will sit at the very top of the GUI window and will consist of a number of different clickable buttons. The menu will consist of the following buttons:

   - Run – start static analysis
   - Clear – clear the code editor
   - Help – help/usage information

- About – concept/contact information

2. **Code Editor**

   The Code Editor will sit underneath the menu, expanding all the way to the right and left hand side window edges. The Code Editor should be as large as possible as this is where the developer will be writing their code. The Code Editor should have line numbers and possibly syntax highlighting. This design will replicate the code editing areas of the previously seen IDEs.

   The Code Editor is a source of user input and all data will need to be sanitised without interfering with the users raw code.

3. **Information Panel**

   The Information Panel will be where the Static Code Analysis information and other general IDE information will be displayed to the user. The Static Code Analysis results will have the line number of the vulnerability identified, the type of vulnerability, some basic remediation advice and a link to where the user can obtain further information. The Information Panel will be located underneath the Code Editor.

   The Information panel is a source of output; all data output is needed to be properly sanitised.

9.3. Static Code Analysis (SCA)

The Static Code Analysis engine will consist of some of the Static Code Analysis techniques discussed earlier in the paper. The following three techniques will be used:

- Lexical Analysis: will be used to turn the users raw source code into tokens.

- Taint Analysis: will be used to analyse tokens for tainted variables.

- String Matching: will be used to identify deprecated functions in the users source code.

The raw source code will first go through a Lexical Analysis engine and then through Taint Analysis to detect any potential vulnerabilities. Sting matching will then used on the raw source code to detect the use of any deprecated functions.

The following diagram is a visual representation of the Static Code Analysis data flow and software architecture. Each rectangular box represents a piece of logical functionality with each arrow representing the flow of data.

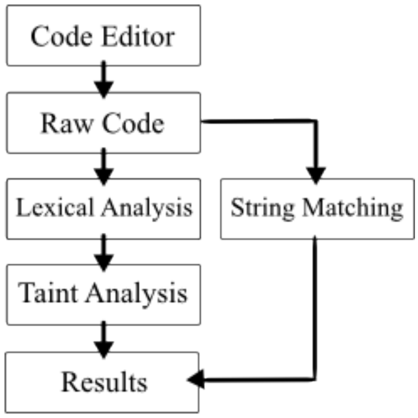


Figure 9-3 - Static Code Analysis data and logic flow diagram.

Each logical operation depicted in the above diagram is explained below:

- **The Code Editor:** will consist of an area on the screen where the user can write, edit, copy and paste their source code. Other code viewing features such as line numbering and syntax highlighting will be implemented to aid the user.

- **The Raw Code:** this is the source code extracted from the Code Editor without any changes made to it, the users source code will be sent to the server for tokenisation by the Lexical Analysis engine.

- **Lexical Analysis:** here the raw source code will be passed through a Lexical Analysis algorithm, turning raw code into token/value pairs for later analysis.

- **Taint Analysis:** once the raw source code is tokenised, Taint Analysis will take place. Here, vulnerabilities are identified from sources of input that are followed through the code into potentially vulnerable functions (sinks).

- **String Matching:** taking the raw source code as input, string matching will attempt to match deprecated function names that are listed in Appendix B.

- **Results:** here the results from the String Matching and the Taint Analysis will be correlated and displayed to the user in the IDE Information Panel.

9.4. UML Use Case Diagram

The Unified Modelling Language (UML) is a modelling language standard that helps in the design and helps to visualise different parts of a software application or business process. A Use Case Diagram is one of the five UML diagrams that model the behaviour of a system (Booch, 1998). The Use Case Diagram below represents the product to be implemented.

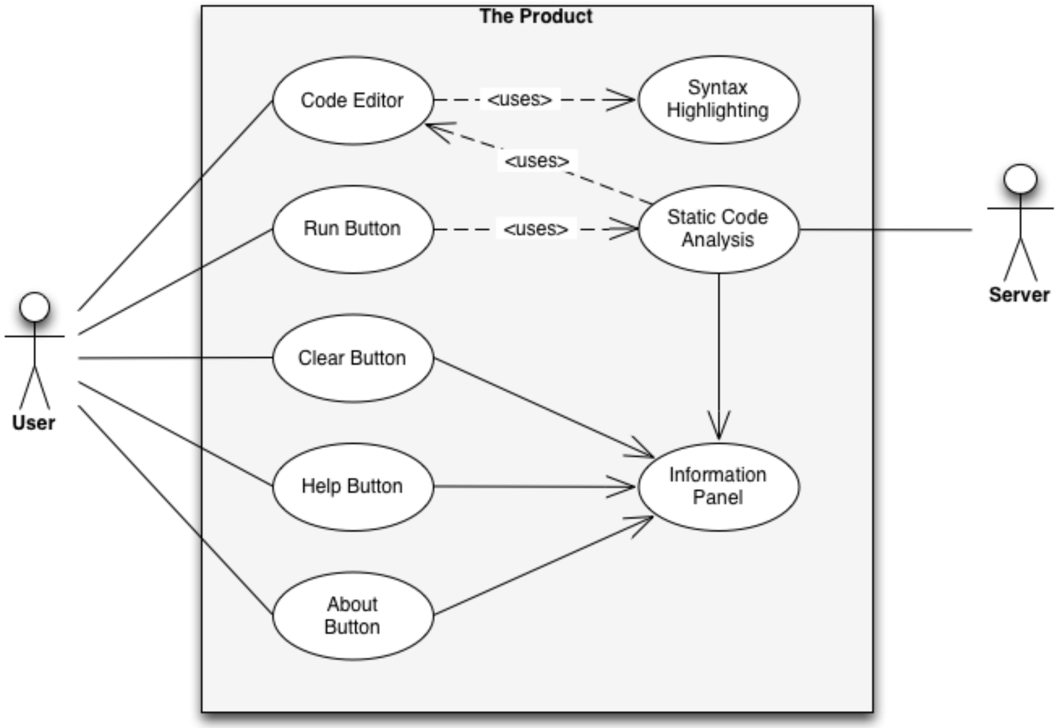


Figure 9-4 – UML Use Case Diagram.

9.5. Use Case Description

The product's main functionality, the running of the Static Code Analysis, is triggered when the user clicks on the Run button. Below is a Use Case Description of the codeAnalysis() class that contains the main Static Code Analysis logic, please refer to Figure 9-5 for the product's full class diagram.

Use Case	codeAnalysis()
Summary	This class contains the main Static Code Analysis logic. It iterates over tokens, assigning taint markers and propagating them.
Actor	The user
Trigger	This is triggered when the user clicks on the Run button, after Lexical Analysis has taken place.
Primary Scenario	1. User loads application. 2. User presses Run button. 3. Raw source code is sent to server for Lexical Analysis. 4. Taint Analysis takes place. 5. Results are displayed to the user.

Alternative Scenario	1. User loads application. 2. User presses Run button. 3. No source code to be analysed. 4. No Taint Analysis takes place. 5. No results are displayed to the user.
Exceptional Scenario	None.
Pre-Conditions	The product fully loaded in the user's web browser.
Post-Conditions	The Information Panel is populated by results, if any.
Assumptions	There is source code to analyse.

Table 9-1 – codeAnalysis() Use Case Description.

9.6. Pseudo Code

Pseudo code is the simplification of software programs and algorithms that allows the programmer to concentrate on the logical aspects and not worry about the source code syntax.

1. User loads product in their browser.
    a. If browser supports localstorage.
        i. Check if anything is stored.
        ii. Load anything that is stored into the Code Editor.
2. User clicks the Run button.
    a. Send Code Editor source code to server.
        i. Server tokenises code and returns it.
    b. Returned tokens are put through Taint Analysis.
        i. For every token:
        ii. If the token is an assignment variable:
            1. Check if the token is a previously tainted variable and check if it has been re-assigned.
                a. If a tainted variable has been re-assigned, remove the tainted variable from the ‘tainted’ array.
            2. Check if the token is a previously tainted variable and check if it is being copied or concatenated onto another variable.
                a. If the tainted variable has been copied or concatenated, place the variable it was copied into in to the ‘tainted’ array.
            3. Check if the variables value contains any sources of user input.
                a. If it does, put the variable into the ‘tainted’ array.

iii. For every other token:
    1. Check if token is a sink.
        a. Does the sink's parameters contain sources?
            i. If yes, vulnerability found.
        b. Does the sink's parameters contain any tainted variables?
            i. If yes, vulnerability found.

c. Display the results to the user.

3. User clicks Clear button.
    a. Set the Code Editor value to blank.
    b. Set the Information Panel value to blank.
4. User clicks Help button.
    a. Displays help information in Information Panel.
5. User clicks About button.
    a. Displays about information in Information Panel.
6. User closes window.
    a. Save Code Editor contents to localstorage if the browser supports it.

9.7. Class Diagram

Having designed the logic flow diagram, the use case diagram and the pseudo code a class diagram has been designed, although classes may change throughout the implementation of the product.

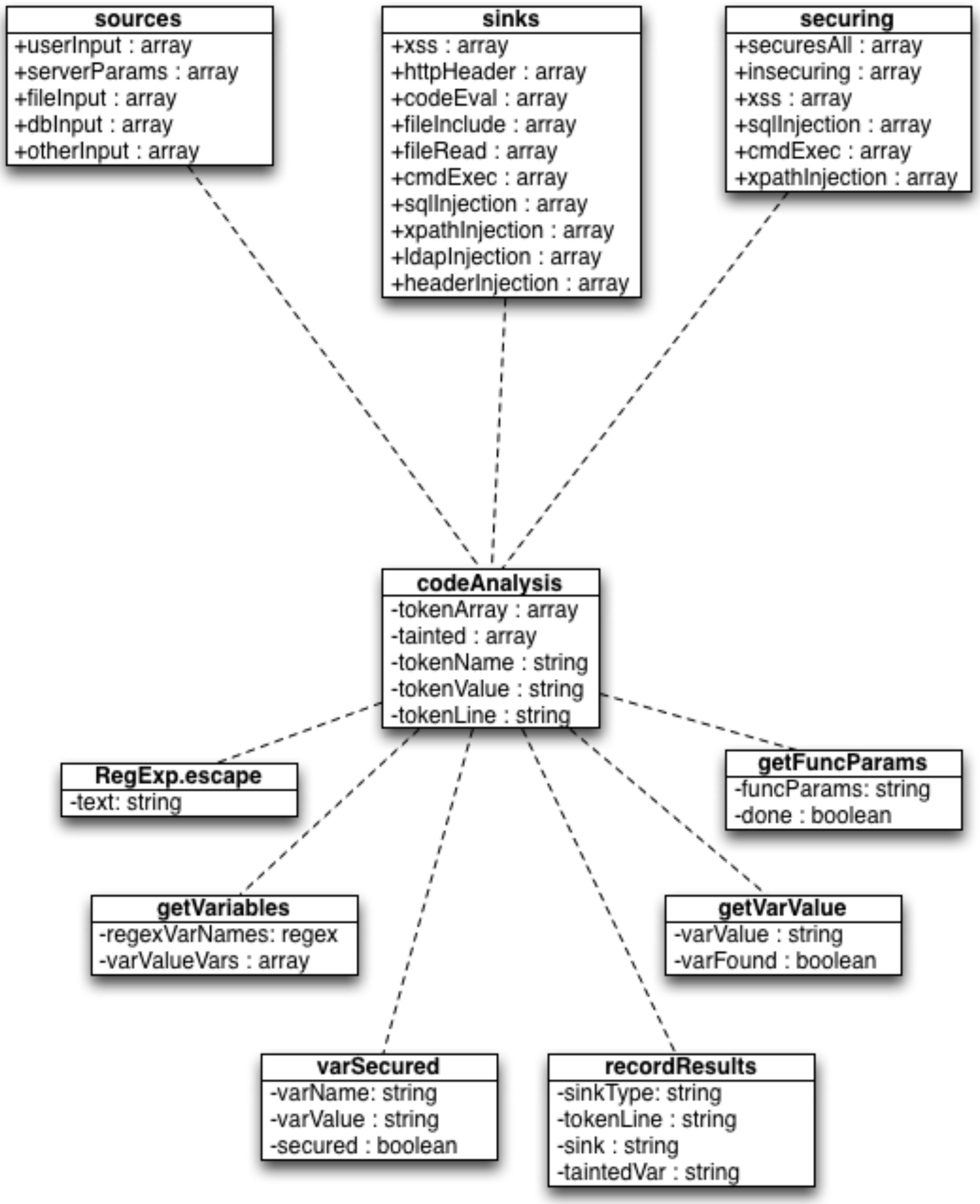


Figure 9-5 – Class Diagram design.

10. Implementation

This section of the paper will describe the implementation of the software product, the writing of the source code, any problems faced and any problems overcome.

Within the implementation section there will be three specific problems discussed:

- The Code Editor
- Deprecated Functions
- Static Code Analysis

10.1. The Code Editor

The code editor would consist of a rectangular box where the user could copy, paste and edit their source code. The code editor would have to be easily manipulated in order to be able to implement features such as syntax highlighting, source code parsing and source code indentation.

HTML TextArea

The first idea was to use the textarea HTML tag to create the code editor. The textarea HTML tag would allow the author to easily create a code editor that would allow the user to edit their source code. The HTML textarea tag represents a multi line text field (Berners-Lee & Connolly, 1995).

While attempting to manipulate the HTML textarea with JavaScript, it proved increasingly difficult to do effectively. Extracting the content and counting the number of newline characters (\n) would allow the calculation of the total number of lines within the textarea.

Example:

```
<textarea rows="2" cols="20>
This is the content of the textarea!\n
This is the second line.\n
This is the third line.\n
</textarea>
```

We can see from the example above that each line is separated by a newline character (\n), by counting these we can figure out the total amount of lines within the textarea.

The next challenge was to find out what line number the user was currently editing. This information could be used to only parse that particular line every time the user made any changes, rather than parsing the whole source code every time.

The first attempt at working out the current line the user was editing seemed to work as expected. The basic principle was to detect the user's keyboard presses and keep track of where the user was moving the cursor within the textarea.

For example, if the user's cursor started on line 1, if the user pressed the down key on their keyboard we could guess that the cursor was now on line 2. This worked fine when the user did not leave the code editor area, for example, not clicking outside of the code editor window. When the user would do this, however, there was no way to know where the user inserted the cursor when they clicked back in the code editor area and so we would lose track of the current line number the user was editing.

Pseudo code example of the first implementation of cursor tracking:

```
1. If the user presses 'Enter':
      a. Add 1 to the current line count.
2. If the user presses 'Backspace':
      a. If the line count is more than the total line
         count:
           i. Minus 1 from the current line count.
3. If the user presses 'Up':
      a. If the total line count is more than 1 and the
         current line count is more than 1:
           i. Minus 1 from the current line count.
4. If the user presses 'Down':
      a. If the total line count is more than 1 and the
         current line count is less than the total line
         count:
           i. Add 1 to the current line count.
5. If the user presses any other key:
      a. Do nothing.
```

The textarea HTML tag has a 'selectionStart' attribute within the browser Document Object Model (DOM). This could have been used to keep track of the cursor's position within the textarea. However, at this point the author decided to see if there might be a more developer friendly way to create an editable code area within a browser as using the textarea HTML tag was becoming increasingly complicated to do simple tasks.

HTML DIV

The div HTML tag allows developers to define sections of a web page; it was the third most common HTML tag on the web in a 2003 study and is supported by all major browsers (Craven, 2003). Developers use div tags to split their web applications into sections to allow them to assign different attributes to different parts of a page, such as colours, borders or text size.

The problem with using div HTML tags is that they are not normally editable in HTML version 4 like a textarea is. A user cannot normally manipulate the contents of a div, however, after some research it was found that there was a new HTML version 5 attribute that could be assigned to a div to make it editable, the contenteditable attribute allows users to edit the content of div tags (Pilgrim, 2009).

After some time testing the suitability of using contenteditable HTML div tags for the code editor it was decided that too much time was being spent trying to create a code editor and it may be best to use an existing solution.

Existing Solutions

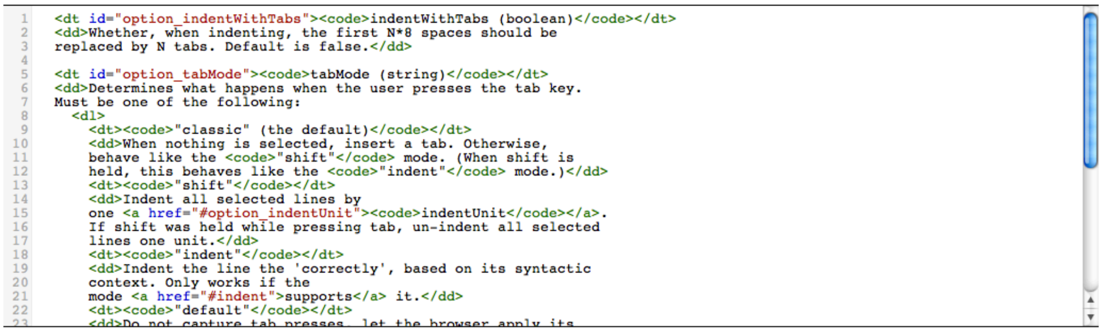
There are various open source web based code editors available. A popular solution is a JavaScript solution called TinyMCE that is used by the popular WordPress blogging platform (TinyMCE, 2012).

Another option could have been to use the Ace open source web based IDE originally developed by the Mozilla Foundation. The commercial cloud9 web based IDE uses Ace as part of their offering (Ace, 2012).

Both of the above solutions and others seemed to be too feature rich and complex to manipulate. They would have needed some time investment in researching how to properly use and edit them.

Another solution was an open source IDE called the Web Installed Open Development Environment (WIODE). After conducting a brief security audit on WIODE it was found to contain many security vulnerabilities. This would not have been acceptable for the software product. The authors of WIODE were informed of these security vulnerabilities and they have since been fixed (WIODE, 2011).

Finally, the author came across a project called CoreMirror. CodeMirror offered just a code editing area with syntax highlighting and code indentation. CodeMirror had great documentation and an active user base. The CodeMirror source code seemed to be very developer friendly that offered an API that allowed developers to easily work with it (CodeMirror, 2012).



```
<dt id="option_indentWithTabs"><code>indentWithTabs (boolean)</code></dt>
<dd>Whether, when indenting, the first N*8 spaces should be
replaced by N tabs. Default is false.</dd>

<dt id="option_tabMode"><code>tabMode (string)</code></dt>
<dd>Determines what happens when the user presses the tab key.
Must be one of the following:
  <dl>
    <dt><code>"classic" (the default)</code></dt>
    <dd>When nothing is selected, insert a tab. Otherwise,
    behave like the <code>"shift"</code> mode. (When shift is
    held, this behaves like the <code>"indent"</code> mode.)</dd>
    <dt><code>"shift"</code></dt>
    <dd>Indent all selected lines by
    one <a href="#option_indentUnit"><code>indentUnit</code></a>.
    If shift was held while pressing tab, un-indent all selected
    lines one unit.</dd>
    <dt><code>"indent"</code></dt>
    <dd>Indent the line the 'correctly', based on its syntactic
    context. Only works if the
    mode <a href="#indent">supports</a> it.</dd>
    <dt><code>"default"</code></dt>
```

Figure 10-1 – CodeMirror JavaScript code editor.

Final Solution

CodeMirror was installed and with help from its documentation the author was able to quickly start using and manipulating it through its API. CodeMirror is a JavaScript application, which meant that no server resources would be used to run it.

10.2. Deprecated Functions

One of the goals of this project was to identify and warn the user about the use of deprecated and/or unsafe functions (Microsoft SDL practice 9; Deprecate Unsafe Functions). As CodeMirror already does syntax highlighting, the idea was to use this existing functionality in order to highlight the deprecated functions within the code editor. CodeMirror parses the code editor anytime the user changes the code editor, for example, when the user presses a button or clicks the mouse.

CodeMirror supports syntax highlighting for many different programming languages. In CodeMirror each programming language is called a ‘mode’, every mode has its own folder and a file that contains the functionality in order to carry out the syntax parsing, as each programming language syntax is different.

The CodeMirror PHP mode relies on functionality from other modes, such as the XML, CSS and HTML modes. This is so that the code editor can highlight other common programming language syntax that is often mixed within PHP. The C mode is used within the PHP mode to borrow functionality from the C syntax parsing functionality as PHP uses C like syntax, this looks to be a design decision to maximise code reuse.

CodeMirror parses the code editor’s contents character by character, looking for particular patterns. If for example a dollar ($) character is seen, this is an indication that the following characters up until a white space or semi-colon (;) is a variable name. If a variable name is found, the variable name is wrapped in a HTML span tag with a class name of ‘cm-variable-2’. The parsed code editor content that now contains span HTML tags with class names around certain syntax is then replaced with the original code editor contents. This then allows for CSS styling to be applied to the HTML span tags via its class name. One example would be to colour all span HTML tags contents that have the ‘variable’ class to purple.

Pre-highlighted source code:

```
$name = “Ryan”;
```

Post-highlighted source code:

```
▼<pre>
   <span class="cm-variable-2">$name</span>
   <span class="cm-operator">=</span>
   <span class="cm-string">"Ryan</span>
   <span id="CodeMirror-temp-40f62b">"</span>
 </pre>
```

Figure 10-2 – CodeMirror post highlighted source code.

CodeMirror already highlights a selection of function names within the PHP mode. These functions are kept in an array within the PHP mode file.

```
keywords: keywords("abstract and array as break case
catch cfunction class clone const continue declare
default do else elseif enddeclare endfor endforeach
endif endswitch endwhile extends final for foreach
function global goto if implements interface instanceof
namespace new or private protected public static switch
throw try use var while xor return die echo empty exit
eval include include_once isset list require
require_once print unset")
```

The above keywords are highlighted in purple by a CSS attribute. To highlight deprecated functions, a new array was added to the CodeMirror PHP mode file that contained the PHP deprecated function names.

```
deprecated: keywords("call_user_method
call_user_method_array define_syslog_variables dl ereg
ereg_replace eregi eregi_replace
set_magic_quotes_runtime session_register
session_unregister session_is_register
set_socket_blocking split spliti sql_regcase
mysql_db_query mysql_escape_string")
```

The CodeMirror C mode then had to be slightly modified to account for the new 'deprecated' array.

The author added the following to the mode/clike.js file on line 4:

```
deprecated = parserConfig.deprecated || {},
```

And in the same file the author added the following to line 50 (line 51 after adding the line above):

```
if (deprecated.propertyIsEnumerable(cur)) {
  if (blockKeywords.propertyIsEnumerable(cur))
    curPunc = "newstatement";
    return "deprecated"; // Class to apply
}
```

The above would return any PHP function listed in the deprecated array within span HTML tags with a class of "deprecated". All that was left to do was to create a CSS rule to change the colour of all HTML tags with the "deprecated" class to red.

Any deprecated PHP function typed into our code editor will now be coloured red as seen in 10-3, an indication that it is a deprecated function. To detect that any deprecated functions has been found, so that this information could be used later to further warn the user, it is just a simple case of using JavaScript to search the DOM for HTML tags with the "deprecated" class.

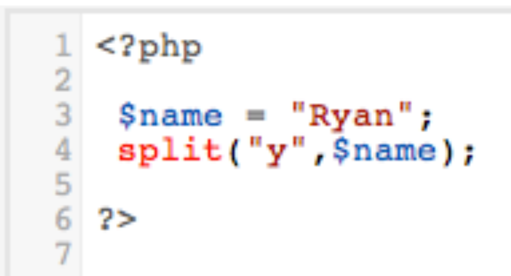


Figure 10-3 – Split deprecated function shown highlighted in red.

The following code is an example of using the jQuery JavaScript library to detect any deprecated functions within the DOM.

```
if ($(".CodeMirror-lines .cm-deprecated").length > 0) {
  // deprecated function detected
}
```

If any deprecated functions are detected, the user is warned within the Information Panel as seen in Figure 10-4.

Warning: split() function deprecated in PHP => 5.3.0. Relying on this feature is highly discourage

Figure 10-4 – Information Panel showing deprecated function warning.

10.3. Static Code Analysis

Source Code Parsing

There were two ways in which Static Code Analysis could have been implemented into the product. The first would have been to use CodeMirror's own parser to parse the code editor contents. Potential vulnerabilities could then be coloured much like the deprecated functions. To do this a new CodeMirror mode would have had to be created, although CodeMirror provided excellent documentation, it was decided that the Static Code Analysis engine should be done separately and the CodeMirror parser would not be used.

This decision was made because the author did not want to rely too heavily on third party software. Using third party software could limit flexibility in implementing new functionality in future.

Taint Analysis Data

Once it was decided to do the Static Code Analysis separate from CodeMirror, Taint Analysis data was needed. The data needed to carry out Taint Analysis was discussed within the Analysis section of this report.

The Taint Analysis data needed consists of three arrays:

- Sources – sources of tainted data.
- Securing – functions that secure or insecure variables.
- Sinks – functions where tainted data can cause vulnerabilities.

Gathering the PHP sources, securing functions and sinks would entail a lot of time investment in testing and sourcing this information from the PHP documentation.

An email was sent to the author of another PHP Static Code Analysis tool called RIPS, asking for the permission to use the three arrays of data (sources, securing and sinks) within the software product. The author of RIPS, Johannes Dahse, replied with his agreement.

If the Taint Analysis data was sourced independently, it would be ignoring an already established rich resource of information and possibly introduce false positives and false negatives by accidently missing out useful data.

The Taint Analysis data was transferred over from the RIPS PHP Static Code Analysis tool to create three JavaScript objects called ‘sources’, ‘securing’ and ‘sinks’. These three objects act like databases and are kept in their own file separate from the rest of the code base so that accidental changes are kept to a minimum.

The following table outlines and describes each object’s properties:

Object Properties	Property Description
sources.userInput	sources of user supplied input
sources.serverParams	PHP $_SERVER variable parameters
sources.fileInput	sources of file input
sources.dbInput	sources of database input
sources.otherInput	sources of other input
securing.securesAll	secures against all vulnerabilities
securing.insecuring	insecures a previously secured variable
securing.xss	secures against XSS vulnerabilities
securing.sqlInjection	secures against SQL Injection vulnerabilities
securing.cmdExec	secures against Command Execution vulnerabilities
securing.xpathInjection	secures against XPath Injection vulnerabilities
sinks.xss	XSS sinks
sinks.httpHeader	HTTP Header sinks
sinks.codeEval	Code Evaluation sinks
sinks.fileInclude	File Inclusion sinks
sinks.fileRead	File Read sinks
sinks.cmdExec	Command Execution sinks
sinks.sqlInjection	SQL Injection sinks
sinks.xpathInjection	Xpath Injection sinks
sinks.ldapInjection	LDAP Injection sinks
sinks.headerInjection	Header Injection sinks

Table 10-1 – The product’s source, securing & sinks object’s properties.

It was decided to trigger the Static Code Analysis when a button was pressed rather than attempting to analyse the code ‘on the fly’ as the user typed in their code. This would minimise the complexity of parsing the user’s source code and allow for more time to be spent implementing the Static Code Analysis engine itself.

Lexical Analysis

Lexical Analysis is used to tokenise the raw source code, this allows for greater accuracy when conducting the Taint Analysis later on as the user’s source code is split into separate distinct tokens. It would have been desirable to do the Lexical Analysis in the user's browser with JavaScript. This would mean that no source code would have to be sent to the server, minimising server resources and allowing for the application to be used offline.

Another option was to use PHP's built in Lexical Analysis engine; this would bring many advantages over implementing a new solution. The PHP Interpreter uses its own Lexical Analysis engine to interpret PHP source code. Although using PHP's own Lexical Analysis engine would mean sending the user's source code to the server, this method would create better Lexical Analysis results.

The user's raw PHP source code is sent to the server via the use of AJAX for tokenisation by the PHP Lexical Analysis engine. Before the tokenised code is returned to the client, each token name/value pair is concatenated to a string.

PHP's Lexical Analysis engine does not assign tokens to some characters, for easier parsing on the client, any characters without tokens are given the 'T_NOTOKEN' token. As well as adding tokens, all PHP comment tokens are removed, as this data is useless for Taint Analysis because source code comments cannot contain vulnerabilities.

An example token after being parsed by the server looks like the following string:

```
"T_OPEN_TAG<:::><?php<:::>1"
```

The first element, T_OPEN_TAG, is our token name. The second element, <?php, is our token value. The third element, 1, is the line number the token originated from. The '<:::>' characters are used as unique separators for the different token elements. These characters were unique enough to be used as separators as this sequence of characters is not valid PHP syntax. As all PHP comments are first removed this ensures they do not interfere with our unique separator. If in future the chosen separator shows to be a problem, it can be easily changed to something else.

Using JSON to return the tokenised code was considered and some experiments using JSON were carried out. Returning the tokenised data in a format the author expected was hard to control, mainly due to the author's lack of experience in using the technology, and so the custom string method was adopted. If the author had more experience in using JSON, it may have been a viable solution and would negate the need for using custom token element separators.

Taint Analysis

The basic steps implemented to carry out the Taint Analysis are as follows:

1. Assign taint markers to sources of user input.
2. Propagate markers when string is copied or concatenated.
3. Check if string has been sanitised or un-sanitised.
4. Report vulnerability when tainted string passed to sink un-sanitised.

Taint Analysis is carried out within the codeAnalysis() function in the 'js/devbug.js' file. The codeAnalysis function takes the tokenised code as string input. The token string is split by the newline character (\n) to create an array of individual tokens. Each element (token) of the array is then iterated over in a loop.

To find sources of direct user input the token name is checked to see if it is equal to 'T_VARIABLE'. Variables are the only way that tokens can contain direct user supplied input; all other tokens are ignored at this stage for the above reason.

If the variable value contains any elements from any of the 'sources' JavaScript object the variable name is added to a 'tainted' array.

During the token loop, if a sensitive sink is come across, the functions parameters are checked to see if they match any variables in our 'tainted' array. If they do, the variables name, the sinks name, the type of sink and the line number is added to an output variable that is later output to the Information Panel along with a hyperlink to further information about the specific vulnerability. The following screenshot shows the output in the Information Panel:

Warning: split() function deprecated in PHP => 5.3.0. Relying on this feature is highly discouraged.
Line 7: Cross-Site Scripting (XSS) in 'echo' via '$name'
Line 11: SQL Injection in 'mysql_query' via '$id'
Line 15: Command Injection in 'exec' via '$cmd'

Figure 10-5 – Information Panel output.

During the above process the tainted variables are traced to see if they assigned to other variables. For example:

```
1. $tainted = $_GET['tainted'];
2. $another_variable = "string" . $tainted;
```

In the example above, both the $tainted variable and the $another_variable are sources of user supplied input because the $tainted variable is used in the value of $another_variable and thus both variables are marked as being tainted.

In another scenario, a variable may get re-assigned later on in the code:

```
1. $tainted = $_GET['tainted'];
2. $tainted = "a string";
```

In the example above, $tainted is tainted by user supplied input on line 1, however, $tainted is then re-assigned to a string on line 2, making $tainted no longer tainted by user supplied input past line 2.

The Pseudo Code for the above explanation can be found within the design documentation.

11. Testing

In this section the product produced will be tested to prove that it is a viable and robust piece of software. In total four distinct top-level tests were chosen.

1. Static Code Analysis: Does the Static Code Analysis engine detect vulnerabilities? Are there any false positives or false negatives?

2. Web Accessibility: Does the product comply with web accessibility standards? Making it accessible to as many users as possible.

3. Security: Does the product contain any vulnerabilities?

4. Usability & Compatibility: Does the application's functionality work as expected? Does the application's functionality work in the most popular web browsers?

11.1 Test 1: Static Code Analysis

In order to test the software product's ability at detecting potential PHP security vulnerabilities via Static Code Analysis, a web application developed by the final year BSc (Hons) Web Design & Development students from Northumbria University was used. This web application was provided to the author as part of another university module's assignment.

The RIPS version 0.51 PHP Static Code Analysis tool that the product used the sources, sinks and securing functions data from was run against the student's entire web application. In total RIPS detected 206 vulnerabilities.

The following table is a summary of RIPS's results:

Vulnerability Type	Amount Detected
Command Execution:	4
File Disclosure:	5
File Manipulation:	16
SQL Injection:	42
Cross-Site Scripting:	3
HTTP Response Splitting:	135
Possible Flow Control:	1
Sum:	206

Table 11-1 – Table showing RIPS's results.

Test 1.1

The first PHP file scanned by RIPS was 'admin/categories.php', RIPS reported four 'HTTP Response Splitting' vulnerabilities within this file. The author's product reported one 'HTTP Response Splitting' vulnerability and one 'PHP File Inclusion' vulnerability.

One thing that RIPS does do and that the author's product does not do is to include other PHP source code within a PHP file when called by using the 'include' or 'require' PHP functions.

For example, if we take the following piece of code:

```
1. <?php
2.
3.   include('file.php');
4.
5. ?>
```

RIPS will analyse line 3 as well as all of the PHP source code contained within the included file, file.php. The author's product only has the capability to analyse one file or code snippet at a time by its very design.

Because of the above functionality within RIPS, it has included another file 'functions/functions_categories.php' within the scanned file of 'admin/categories.php'. The actual sinks that RIPS detected are not in the originally scanned file, however, are instead located within the included file of 'functions/functions_categories.php'.

RIPS in fact detected no vulnerabilities within the source code of the originally scanned file whereas the author's product detected two.

The two vulnerabilities detected by the author's product all occurred on one line; a source of user input directly inserted into the parameter of a sensitive sink.

Vulnerability 1: PHP File Inclusion.

```
4.include($_SESSION['link2'].'functions/functions_categ
ories.php');
```

Vulnerability 2: HTTP Response Splitting.

```
47.header('Location:'.$_SESSION['link'].'error.php?msg=
'.$value);
```

In both of the above lines of code, the $_SESSION['link'] variable is placed within sensitive sinks, include() and header(), causing the author's product to report the vulnerabilities. When migrating the 'sources' data during the implementation over from RIPS, the author noticed that the $_SESSION global variable was missing from RIPS's 'sources' data and so it was added to the author's product.

Adding the $_SESSION PHP global variable to the sources list in the author's product was a mistake. After some testing it is apparent that PHP $_SESSION variables can be set by a user via editing their cookies, however, instead of being treated as a 'source' it should be treated as any other variable would be treated. It should only become a source when it is assigned a source. The following is an example of when a $_SESSION variable can become tainted:

```
$_SESSION['tainted'] = $_GET['tainted'];
```

The author's product produced two false positives because the author added a source that was in fact not a source of user-supplied input.

The $_SESSION PHP global variable has since been removed from the author's product's source list, fixing the above False Positives. Running the product now shows zero vulnerabilities in the 'admin/categories.php' file as did RIPS.

Test 1.2

The author's product was run against the file that RIPS included, 'functions/functions_categories.php'. The author's product returned nine 'Header Injection' vulnerabilities, whereas RIPS returned four 'Header Injection' Vulnerabilities. The author's product claimed to have identified five more 'Header Injection' vulnerabilities than RIPS had.

This indicates that either RIPS or the product is causing either False Negative or False Positive results.

RIPS's four 'Header Injection' vulnerabilities were found on the following lines of code, 93, 259, 386 and 391. As well as identifying the vulnerabilities on those lines, the product identified vulnerabilities on lines 90, 116, 256, 282 and 383.

A simplified version of the source code between lines 82 and 98 of the original file looks like the following example. According to RIPS one vulnerability is present between those lines, on line 90 of the original file, according to the author's product there are two, on lines 90 and 93 of the original file.

```
1. if (isset($_REQUEST['id']) &&isset($_REQUEST['name'])) {
2.
3.  $id = $_REQUEST['id']; // source of user input
4.  $name = $_REQUEST['name']; // source of user input
5.
6.  if (strlen($name) < 40) { // $name less than 40 chars?
7.
8.    $value = $name;
9.    header('Location: somewhere/' . $value); // sink
10.
11.  } else {
12.
13.    header('Location: somewhereelse/' . $name); // sink
14.
15.  }
16.
17. } // end if statement
```

On line 1, the code looks to see if the $_REQUEST['id'] and $_REQUEST['name'] variables have been set. If they have been previously set, the code re-assigns both variables to new variables, $id and $name on lines 3 and 4. These two variables have now also become the source of user-supplied input. The code then checks the length of the $name variable via the PHP strlen() function on line 6, if the $name variable contains less than 40 characters, $name is re-assigned to $value on line 8 and then $value is placed in a sensitive sink, header(), on line 9. If, however, $name is equal to or more than 40 characters $name is placed directly into a sink, header(), on line 13.

The above example highlights another difference between RIPS and the product. RIPS carries out Data Flow Analysis whereas the author's product does not.

On this occasion the author's product correctly detected five more vulnerabilities within the PHP file than RIPS did. RIPS version 0.51 wrongly assumes that checking if a string has less than 40 characters is some kind of securing mechanism, when in fact it is not.

11.2. Test 2: Web Accessibility

The product produced needs to be accessible to as many people as possible. People with impaired vision and/or other disabilities need to be accounted for.

The World Wide Web Consortium (W3C) is an international standards organisation that creates web content accessibility guidelines.

In total there are 14 guides the W3C recommend (W3C, 1999).

1. Provide equivalent alternatives to auditory and visual content.
2. Don't rely on color alone.
3. Use markup and style sheets and do so properly.
4. Clarify natural language usage.
5. Create tables that transform gracefully.
6. Ensure that pages featuring new technologies transform gracefully.
7. Ensure user control of time-sensitive content changes.
8. Ensure direct accessibility of embedded user interfaces.
9. Design for device-independence.
10. Use interim solutions.
11. Use W3C technologies and guidelines.
12. Provide context and orientation information.
13. Provide clear navigation mechanisms.
14. Ensure that documents are clear and simple.

(W3C, 1999)

To test the author's product for accessibility the author used the above guidelines in a checklist format to ensure the author conformed as much as possible and where applicable. Where the product was not conforming to the W3C accessibility guidelines changes were made where applicable. The checklist used can be found in Appendix D.

Issue 1

Whilst going through the checklist it was apparent that the HTML markup was not adhering to accessibility guidelines. To remedy this, 'alt' and 'title' HTML attributes were added to HTML tags where appropriate. These attributes show alternative text descriptions when needed, for example, for automated screen readers or when images fail to load or are difficult to see visually.

Issue 2

The default colouring of the author's product was a light grey background with white boxes for the Code Editor and the Information Panel. It became apparent that these colours might not be visually distinct enough for visually impaired users. It could have been possible to change the default colours however a better option was to make the applications theme selectable. Using CSS and JavaScript a selection box was implemented that gives the user an additional theme option. The additional theme was named 'Black and White' and turned the applications colours to black and white as well as making fonts larger and disabling syntax highlighting.

Issue 3

If the HTML markup used in the application had any mistakes it could make the application difficult for screen reading software or browsers to render properly. The W3C Markup Validation Service is an online service that allows you to scan web applications for any HTML errors (W3C, 2012). When the W3C Markup Validator was fist run against the author's product, it returned 28 errors and 50 warnings within the HTML markup. The main reasons for this were omitting the 'doctype' HTML tag, not specifying a language for the application and using 'title' attributes where 'alt' attributes should have been used. After implementing and fixing the errors returned by the W3C Markup validator, no further errors or warnings were returned.

Summary

As seen in the checklist in Appendix D, before testing, the application was only compliant with 6 of the W3C's accessibility guidelines. After testing and implementation of changes, the application is now compliant with 11 of the W3C's 14 guidelines. Two of the guidelines, five and seven were not applicable.

Another online service called the Web Accessibility Checker was used to check the applications compliance to the W3C's accessibility guidelines (AChecker, 2012). The tool responded with no errors or warnings as to the accessibility of the application. The full report from this service can be found in Appendix E.

11.3. Test 3: Application Security

The product produced is a web application that relies on the JavaScript programming language. By using JavaScript it may be possible that user controllable input reaches a sensitive sink within the application itself causing DOM based Cross-Site Scripting (XSS) vulnerabilities.

Apart from the client side JavaScript, the users raw source code is sent to the server for tokenisation. The user's raw code is a source of tainted data and should not be trusted.

A number of different tools were used to assess the products own security after implementation. It should be noted that the attack surface of the application is low, with only one source of user input.

DOMinator - The DOMinator tool designed to detect DOM based XSS (Paola, 2011) gave no warnings meaning that it detected no DOM based XSS.

Arachni – The dynamic web application security scanner that checks for various web application security vulnerabilities (Laskos, 2012) returned no issues related to the application's security.

RIPS – PHP Static Code Analysis tool that attempts to detect various PHP related vulnerabilities using a white box approach detected no server side PHP vulnerabilities within the product.

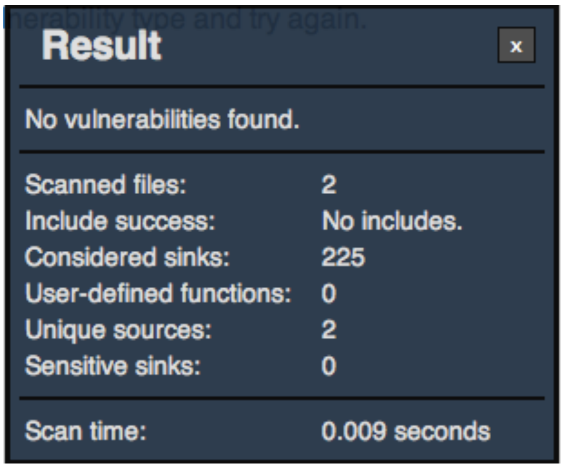


Figure 11-1 – RIPS scan output.

11.4. Test 4: Usability & Compatibility Inspection

During this test a checklist was produced to inspect the usability and browser compatibility of the author's product. This inspection will ensure that all the components of the user interface work as expected and work with some of the most common web browsers.

The browsers used to test the application were:

- Mozilla Firefox version 8.0.1
- Google Chrome version 17.0.963.65
- Apple Safari version 5.1.2
- Microsoft Internet Explorer version 8

The complete checklist containing all of the results of the test can be found in Appendix F. The following actions were tested: page load, run button, clear button, help button, about button, theme selection and unload of the page.

The following checklist was produced for the Mozilla Firefox version 8.0.1 browser, checklists for the other browsers can be found in Appendix F:

Action	Expected Re-action	Actual Re-action	Pass
Load Page	Page loads as expected, user interface as expected, loads in a timely manner.	Page loads as expected, user interface as expected, loads in a timely manner.	Yes
Run button pressed	Source code in the Code Editor gets analysed and results displayed in the Information Panel properly.	Source code in the Code Editor gets analysed and results displayed in the Information Panel properly.	Yes
Clear button pressed	Code Editor, Information Panel and the browsers localstorage is cleared.	Code Editor and the browsers localstorage is cleared. Information Panel is not cleared.	No
Help button pressed	Help information displayed in the Information Panel in correct font, size, colour.	Help information displayed in the Information Panel in correct font, size, colour.	Yes
About button pressed	About information displayed in the Information Panel in correct font, size, colour.	About information displayed in the Information Panel in correct font, size, colour.	Yes
Theme selection	Theme is able to be selected. The chosen theme displays as intended.	Theme is able to be selected. The chosen theme displays as intended.	Yes
Unload Page	Contents of the Code Editor should be saved to localstorage and then retrieved on page load.	Contents of the Code Editor should be saved to localstorage and then retrieved on page load.	Yes

Table 11-2 – Table showing the usability test results from the Firefox browser.

A common occurrence across all browsers was that the Information Panel was not cleared of its contents when the Clear button was pressed.

Apart from the Information Panel not clearing, Firefox, Chrome and Safari passed all of the tests successfully. Internet Explorer however failed during many of the tests. The theme selection box was out of place making the whole user interface distorted and hardly usable. When the run button was pressed, the expected action is that the contents of the Code Editor are analysed and then the results are displayed in the Information Panel, this did not happen. When the run button was pressed, nothing happened. The rest of the buttons seemed to be working as expected.

Because of the faults when using the Internet Explorer browser, the browser will be deemed as unsupported until more time is available to attempt to fix the problems encountered. A short-term solution will be to display a warning when the user is running Internet Explorer informing them that it is not supported.

11.5. Testing summary

Before testing was carried out it was assumed that the product was robust and fit for its purpose. What testing allowed was to test that assumption. There were some defects found, however, fixes were implemented where appropriate.

Overall there were 4 distinct tests carried out.

1. Static Code Analysis: Testing the code analysis engine and its ability to detect potential vulnerabilities as compared to a similar product.

2. Web Accessibility: Testing the applications accessibility, ensuring that all types of users are catered for.

3. Application Security: Ensuring the application it self was secure from any critical vulnerabilities.

4. Usability & Compatibility Inspection: Testing that the application behaved as intended and testing how the application behaved under different web browsers.

The above tests were selected because they offer a large coverage of the applications functionality; the Static Code Analysis, accessibility, security and compatibility.

Evaluation and Conclusions

12. Evaluation

12.1. Product Evaluation

The product's three initial aims were to develop or use an existing Integrated Development Environment (IDE), implement unsafe/deprecated function matching and implement Static Code Analysis.

The following points will evaluate each of these aims in further detail:

- Integrated Development Environment (IDE)

  The objective was to either use an existing IDE or to develop a new one. After reviewing the existing IDEs available it was decided that a new IDE should be developed as the current products available seemed bloated with functionality that added unnecessary complexity. The most significant part of an IDE is the code editing area where users can edit, save and paste their source code, as the decision was made to create a web based product, a web based solution for a code editing area was needed. After some research it was found that an existing solution called CodeMirror would be best suited as it offered good documentation, developer APIs and the flexibility to turn off unneeded features. After the implementation and testing of the product, this decision still stands to be the best solution.

  As well as the code editing area of an IDE, other important aspects such as buttons for triggering different functionality and a way to display important information to the user were needed. The buttons chosen were created using by using CSS. Not using images for the buttons does add some unnecessary complexity, for example, when the buttons aesthetics needed to be changed. Another downside to using CSS buttons is that older browsers may have difficulty rendering them.

  Displaying important information to the user is done via an Information Panel, this is a designated box at the bottom of the page used to display important information such as vulnerabilities identified. The Information Panel takes up a lot of the screens space and is only currently suitable for displaying text-based information.

- Unsafe/Deprecated function matching

  One of the objectives was to warn the user if any unsafe or deprecated functions were being used within their source code. This was done by using CodeMirror's own functionality by amending one of its 'modes'. Although simple to implement it did require some investment in understanding the way CodeMirror worked. One disadvantage to this method is that when CodeMirror release new updates that fix bugs or implement new features, if those updates are to be implemented into the product then CodeMirror would have to be amended to account for the unsafe/deprecated function matching every time. This adds unnecessary work when CodeMirror updates are applied and may delay new CodeMirror updates being implemented due to the extra work involved.

- Static Code Analysis

  The most important objective was to implement Static Code Analysis into an Integrated Development Environment (IDE). The Static Code Analysis takes part in two stages, the Lexical Analysis and then the Taint Analysis.

  The Lexical Analysis makes use of PHP's own Lexical Analysis engine. The raw source code from the Code Editor is sent to the server where the Lexical Analysis takes place, removing any unnecessary tokens such as comments, adding new tokens where necessary and then returning the tokenised source code as a string. Sending the raw source code over the Internet to the server is almost instantaneous, although when a lot of source code is sent, around 5000 lines, the Lexical Analysis engine may take 3-5 seconds to return the tokenised string. Currently the users source code comments are sent to the server where the server removes them, a better solution may have been to remove these first, minimising the amount of code being sent over the Internet and thus minimising the time taken. The server currently returns a string of tokens separated by a custom separator and newline characters. Although this has caused no problems there may be better ways to return the tokenised code, such as in JSON or XML format.

  The Taint Analysis takes place on the users browser by using JavaScript, this minimises server resources and speeds up the process. The Taint Analysis engine takes the tokenised code from the servers Lexical Analysis output as input. The Taint Analysis engine iterates over every token looking for certain tokens, such as variables that could contain sources of user input or sinks where those variables may end up. One Static Code Analysis technique that the product does not make use of is Data Flow Analysis. This technique would lower the amount of potential False Positives within the product as it would have a much more detailed understanding of the flow of data through the code. However, as seen during testing in the RIPS Static Code Analysis tool, Data Flow Analysis adds further complexity that could cause potential False Negatives.

  One issue that may arise due to the nature of the product is that people blindly believe its results. For example, if the product returns that no vulnerabilities were found the user might think that this means their code can be labelled as 'secure'. As we have seen during the Analysis section, software can never be 100% vulnerability free. Appropriate terms and conditions as well as clearly stating that their code is not 'secure' could be a viable option to avoid this.

Understanding the strengths and weaknesses of the product is important for its future development and will help the author understand where improvements could be made.

Strengths

Having Static Code Analysis built into an IDE allows for the identification of potential vulnerabilities as the earliest stage of development. Rather than programmers running separate Static Code Analysis tools after they have written their code, they can run Static Code Analysis within their development environment.

The product being a web-based product is a great advantage over existing Static Code Analysis tools. There is no need for installation; users only need a modern browser and an Internet connection to use the product. Novice programmers or users can use the product without being put off by complicated installation and usage instructions. The product can be used to quickly test small snippets of untrusted PHP code found on the Internet or to directly write PHP source code within. Users do not need to worry about whether or not they are running the latest version of the product, as the online version will always be the latest one.

The product has been tested and works in all major browsers except for one. The Internet Explorer browser renders the HTML markup differently to other browsers tested. The HTML markup, CSS and accessibility are all W3C compliant. The theme of the user interface can be changed to be more suitable for partially sited users.

The Static Code Analysis engine does not use Data Flow Analysis; this brings the advantage of lowering complexity and lowering the chances of any false negative results. However this does cause some weaknesses, as discussed below.

Weaknesses

The product was designed to analyse snippets of PHP code or individual pages. One weakness due to its design is that a user cannot analyse their full application in one go. Another weakness due to this design approach is that when other functions are included within PHP from separate files, these functions will be ignored, possibly causing false negatives or false positives.

The product does not carry out any Data Flow Analysis, this could cause false negatives however should decrease the potential amount of false positives within the results.

The one page design limits the amount of information that can be displayed; this could be a limiting factor if any future functionality is to be implemented. A better arrangement of the displayed information is something that could be looked at in future development.

Alternatives

Software security is a big problem, a problem that has no easy solution. There is never going to be one product or guideline that can 100% guarantee the security of software.

With the combined efforts of vendors, developers, educational institutions and security professionals the problem of software security could become less of an issue than it is today.

By raising awareness of the issue it maybe possible to influence the consumers of software to start demanding a certain level of security within the software they are purchasing. Only when consumers start to demand software security from their vendors the author believes that the problem of software security can be significantly reduced. This is of course assuming that one day the majority of consumers will care about the security of their software enough to make purchasing decisions. It is unlikely that the average consumer of software will ever care as much about the security of software as the author does, the average consumer does not have the time to worry about such things in their daily lives.

If there is no demand for secure software from the consumer then it could be possible to enforce software security.

Compilers and Interpreters could be more security conscious. As mentioned in previous chapters, the Perl and Ruby programming languages have Taint Analysis engines built into their Interpreters that do not allow the code to be run if a potential vulnerability is detected. The PHP Interpreter does not have a Taint Analysis engine, there have been patches written by third parties to implement such functionality (Core Security Technologies, 2012), however, these have not been implemented into the PHP core Interpreter.

There are already laws and compliance standards that enforce software vendors and consumers to create and maintain secure software. The Payment Card Industry (PCI) Data Security Standard (DSS) is one such compliance standard required by credit card merchants. The Data Protection Act (DPA) requires a certain level of security to ensure the safety of data. As far as the author is aware there is no UK law that holds the creators of software liable for software in-security. It is possible that if such a law was introduced that it would discourage small companies or freelance software engineers from creating new software products. This would be bad for innovation, the economy and Computer Science. A better solution may be to have a UK wide law that holds companies liable that sell software and/or software services that do not meet a specified security standard to a large proportion of the population. Such a law that only targeted software that is widely consumed and had a commercial interest could significantly improve the state of software security in the UK. If such as law should be considered, it should be first debated and include a wide range of organisations and people from the software industry to voice their concerns.

12.2. Process Evaluation

At the beginning of the project the author struggled to get to grips with the core concepts behind Static Code Analysis. A lot of research was carried out that monopolised a lot of the author's time due to the steep learning curb needed. The initial research could have been made easier if the author had chosen a more familiar topic. However the author would not have gained as much value out of the project process.

Not only is the author now familiar with the core concepts of Static Code Analysis, there were many new technologies used by the author that he had not used before; technologies such as jQuery and JavaScript. The author took a significant risk in choosing an unfamiliar topic and to work with new technologies, however, the benefits from doing this are much higher than if a familiar topic and technologies were used.

Within the Terms of Reference (TOR) document attached in the Appendices, there were twelve objectives defined.

The first five objectives were to research modern Software Development Life Cycles (SDLC) and to research the Microsoft Security Development Lifecycle (SDL) with particular emphasis on the Implementation phase. These five objectives were achieved and their outcome can be seen from the production of the Analysis section.

Other objectives outlined in the TOR include the design, development, implementation and testing of a software product. These objectives were achieved. The design documentation, discussion on the product's implementation and the discussion of the product's testing can be found within the Synthesis section of this report.

A Software Development Life Cycle (SDLC) was used during the development of the product. Although using an SDLC did help in organising the development of the product to an extent, the author found that it was hard to know when each section of an SDLC was complete. The author also found that sometimes previously completed sections of the SDLC had to be revisited as changes were implemented. The author believes that an SDLC may be more appropriate to larger teams of developers and/or developers that are constantly developing new software products.

The author started the project process by following a Security Development Lifecycle (SDL), specifically the Microsoft SDL. The author found that the work needed to implement and carry out an SDL on a small one off project was unrealistic. The amount of additional work would not have been possible to take on when on a tight deadline. The author believes that an SDL would be most beneficial when working on large projects and/or with large development teams. A possibility may be to develop a lightweight SDL for smaller projects.

The author found that some tasks took longer than planned where as others took less time than planned. Due to the steep learning curve needed, the Analysis section took longer than planned to produce. Due to the amount of work carried out during the Analysis section the Synthesis section seemed to be a lot quicker to produce. Overall, even though some sections took longer time and others a shorter time than planned, the project will be completed on time.

The author has improved in time management, research skills, programming ability, secure coding and writing skills. These are all skills that will benefit the author later on in his career. The deeper understanding of software security and secure coding will be of great advantage.

The confidence of the author in these areas has also improved. Taking on and completing such a big project has given the author the confidence that great things can be achieved with time and effort.

13. Conclusions

The project has achieved research into modern Software Development, Security Development Lifecycles (SDL) and Static Code Analysis. As well as the implementation of a Static Code Analysis engine into an Integrated Development Environment (IDE).

The two main aims outlined within the Terms of Reference (TOR) document were:

- To investigate and analyse the Microsoft Secure Development Lifecycle (SDL).

  This aim was achieved with the creation of this report, specifically the creation of the Analysis section and the software product.

- To amend a current Integrated Development Environment (IDE) to implement as a Proof of Concept (PoC) the three practices of the Microsoft Secure Development Lifecycle (SDL).

  This aim was achieved with the design, implementation and testing of the software product produced. This aim states 'to amend' an Integrated Development Environment (IDE), the author did not amend an IDE and instead created his own IDE via the use of a web interface.

It was found that Software Development Life Cycles (SDLCs) have their specific problems. For example, the Waterfall SDLC splits the development process into sections. Once one section is complete the developer must move onto the next section. The author found it difficult to know when a section should be labelled as 'complete' and when to move onto the next section. In reality the Waterfall SDLC did provide a good guideline in the development of the product, however, it was not a perfect framework to use. It may be possible that no one methodology or process is ever going to be an exact match for every possible development project, however, what they do is provide an abstract roadmap of how best to develop software.

For a small one off project the Microsoft Security Development Lifecycle (SDL) seemed to be an excessive guideline to follow if followed word for word. By keeping security and privacy into consideration during the development process, this seemed to achieve the same goals as an SDL when used on a small project. The Microsoft Security Development Lifecycle (SDL) may have been more beneficial if implemented on a large team of developers or onto a long-term, complex, piece of software.

With the use of the product, PHP Static Code Analysis will be easily accessible to developers whom wish to check their code for any potential vulnerabilities. The product will be released online for users to use freely, over time this should have an impact, no matter how small, on the overall security of software being developed. By releasing the product online it will not only actively mitigate potential vulnerabilities, however, raise awareness to the problem of software security. Another advantage of releasing the product will be the feedback from users will be invaluable in shaping the future path the product takes.

13.1. Recommendations

This section will give recommendations for any future work within the area of the topics covered and points discussed throughout the project.

- Integrated Development Environment (IDE) Static Code Analysis integration: It would be good to see the wide adoption of Static Code Analysis within the IDE. The detection of potential vulnerabilities at the earliest stage brings lots of advantages to software development as discussed within the project.

- Availability of the product: The product created for this project will be released online as a free service for anyone to use. This will generate interest in Static Code Analysis and produce valuable feedback from users, which can be implemented into the product.

- PHP Interpreter Taint Analysis: The native adoption of Taint Analysis within the PHP Interpreter would see a significant increase in the security of web applications on the Internet. The core PHP developers should revisit the benefits of doing so. With improvements to PHP's performance over its release cycles, these performance benefits may be enough to offset the performance decline of the Taint Analysis.

- Debate on UK law enforcing software security: There is no overall law in the UK that governs software security. Other industries such as the car manufacturing industry have safety laws. If done correctly a UK software security law could improve the state of software security.

References

Ace. (2012) *ajax.org Cloud9 Editor* [Online]. Available at: http://ace.ajax.org/ (Accessed: 12th January 2012).

AChecker. (2012) *Web Accessibility Checker* [Online]. Available at: http://achecker.ca/checker/index.php (Accessed: 2nd February 2012).

Adobe. (2011) *Adobe Secure Product Lifecycle (SPLC)* [Online]. Available at: https://www.adobe.com/security/splc/ (Accessed: 12 November 2011).

Allen, J. et al. (2008) *Software security engineering: a guide for project managers.* Addison-Wesley Professional.

Baase, S. (2002) *Gift of Fire: Social, Legal, and Ethical Issues for Computers and the Internet.* Prentice Hall.

Berners-Lee, T. & Connolly, D. (1995) *RFC 1866 Hypertext Markup Language - 2.0*. MIT/W3C.

Booch, G. et al. (1998) The Unified Modeling Language User Guide. Addison Wesley.

Campbell. et al. (2003) *The economic cost of publicly announced information security breaches: empirical evidence from the stock market.* Journal of Computer Security 11.

CISCO. (2011) *CISCO Secure Development Lifecycle (CSDLC)* [Online]. Available at: http://www.cisco.com/web/about/security/cspo/csdl/docs/External_CSDL_Whitepaper_Final.pdf (Accessed: 21 November 2011).

Cloud9. (2012) *Cloud9 IDE* [Online]. Available at: http://c9.io/ (Accessed: 5 February 2012).

CodeMirror. (2012) *In-browser code editing made bearable* [Online]. Available online: http://codemirror.net/ (Accessed: 22nd January 2012).

Core Security Technologies. (2012) *CORE GRASP* [Online]. Available online: http://grasp.coresecurity.com/ (Accessed: 28th February 2012).

Craven, T. (2003) *HTML Tags as Extraction Cues forWeb Page Description Construction.* The University of Western Ontario, London, Ontario, Canada.

Dahse, J. (2010) *RIPS – A static code analyser for vulnerabilities in PHP scripts* [Online]. Available at: http://www.php-security.org/downloads/rips.pdf (Accessed: 5 January 2012).

Davis, A.M. (1993) *Software Requirements; Objects, Functions & States.* Colorado: Prentice-Hall, Inc.

Department of Homeland Security. (2006) *Security in the Software Lifecycle. DRAFT Version 1.2.* Department of Homeland Security.

Department of Justice (DoJ). (2009) *Three Men Indicted for Hacking into Five Corporate Entities, including Heartland, 7-Eleven, and Hannaford, With Over 130 Million Credit and Debit Card Numbers Stole*. New Jersey: United States Department of Justice.

Doyle, F. & Fly, R. et al. (2007) *Open Source Fuzzing Tools*. Burlington, U.S.A:Syngress.

Eclipse. (2012) *Language IDE* [Online]. Available at: http://www.eclipse.org/home/categories/index.php?category=ide&tab=learn (Accessed: 5 February 2012).

Falliere, N. et al. (2011) *W32.Stuxnet Dossier.* Symantec: Security Response.

Grossman, J. (2003) *CROSS-SITE TRACING (XST)* [Online]. Available at: http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf (Accessed: 29 December 2011).

Hansen, R. & Grossman, J. (2008) *Clickjacking* [Online]. Available at: http://www.sectheory.com/clickjacking.htm (Accessed: 3 January 2012).

Howard, M. (2007) *Lessons Learned from Five Years of Building More Secure Software* [Online]. Available at: http://download.microsoft.com/download/A/E/1/AE131728-943B-42B4-B130-C1DEBE68F503/Trustworthy%20Computing.pdf (Accessed: 3 January 2012).

Howard, M & LeBlanc, D et al. (2005) *19 Deadly Sins of Software Security.* California, U.S.A: McGraw-Hill Companies.

Howard, M & LeBlanc, D. (2003) *Writing Secure Code.* 2nd edition. Washington, U.S.A: Microsoft Press.

ISO. (2008) Systems and software engineering - Software life cycle processes. IEEE.

ISO. (2008) Systems and software engineering - System life cycle processes. IEEE.

ISO. (2008) Systems and software engineering - Systems and software assurance. IEEE.

Jovanovic, N. Kruegel, C & Kirda, E. *(2007) Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)*. Vienna. Technical University of Vienna.

Laskos, T. (2012) *Arachni –Web Application Security Scanner Framework* [Online]. Available at: http://arachni-scanner.com/ (Accessed: 9th February 2012).

Leveson, N. (1995) *Medical Devices: The Therac-25.* Washington, U.S.A: University of Washington.

McKeone, D. (1995) *Measuring your media profile.* Gower Publishing Limited.

Microsoft. (2011) *SDL Helps Reduce the Total Cost of Development* [Online]. Available at: http://www.microsoft.com/security/sdl/learn/costeffective.aspx (Accessed: 21 October 2011).

Microsoft. (2011) *SDL Helps Build More Secure Software* [Online]. Available at: http://www.microsoft.com/security/sdl/learn/measurable.aspx (Accessed: 9 December 2011).

Microsoft. (2011) *SDL Process Guidance Version 5.1* [Online]. Available at: http://www.microsoft.com/download/en/confirmation.aspx?id=9295 (Accessed: 6 December 2011).

Microsoft. (2010) *Simplified Implementation of the Microsoft SDL* [Online]. Available at: http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=12379 (Accessed: 14 November 2011).

Munassar, N & Govardhan, A. (2010) *A Comparison Between Five Models Of Software Engineering* [Online]. Available at: http:// www.ijcsi.org/papers/7-5-94-101.pdf (Accessed: 5 December 2011).

NetBeans. (2012) *All Features and Supported Technologies* [Online]. Available at: http://netbeans.org/features/all.html (Accessed: 5 February 2012).

NIST. (2011) *National Vulnerability Database* [Online]. Available at: http://web.nvd.nist.gov/view/vuln/search-results?adv_search=true&cves=on&cve_id=&query=&cwe_id=&pub_date_start_month=0&pub_date_start_year=2011&pub_date_end_month=11&pub_date_end_year=2011&mod_date_start_month=-1&mod_date_start_year=-1&mod_date_end_month=-1&mod_date_end_year=-1&cvss_sev_base=&cvss_av=&cvss_ac=&cvss_au=&cvss_c=&cvss_i=&cvss_a= (Accessed: 21 December 2011)

NIST. (2002) *The Economic Impacts of Inadequate Infrastructure for Software Testing* [Online]. Available at: http://www.nist.gov/director/planning/upload/report02-3.pdf (Accessed: 21 October 2011).

OSVDB. (2011) *Search Query: text_type: titles s_date: January 1, 2011 e_date: December 21, 2011* [Online]. Available at: http://osvdb.org/search/search?search%5Bvuln_title%5D=&search%5Btext_type

%5D=titles&search%5Bs_date%5D=January+1%2C+2011&search%5Be_date%5D=December+21%2C+2011&search%5Brefid%5D=&search%5Breferencetypes%5D=&search%5Bvendors%5D=&search%5Bcvss_score_from%5D=&search%5Bcvss_score_to%5D=&search%5Bcvss_av%5D=*&search%5Bcvss_ac%5D=*&search%5Bcvss_a%5D=*&search%5Bcvss_ci%5D=*&search%5Bcvss_ii%5D=*&search%5Bcvss_ai%5D=*&kthx=search (Accessed: 21 December 2011).

Oxford. (2010) *Oxford Dictionary of English.* 3rd edn. OUP Oxford.

Paola, S. (2011) *The DOMinator Project* [Online]. Available at: http://blog.mindedsecurity.com/2011/05/dominator-project.html (Accessed: 9th February 2012).

Patwardhan, N. et al. (2002) *Perl in a Nutshell.* O'Reilly Media.

PCI-DSS. (2010) *Requirements and Security Assessment Procedures Version 2.0* [Online]. Available at: https://www.pcisecuritystandards.org/documents/pci_dss_v2.pdf (Accessed: 15 November 2011).

PICTA. (2005) *Cyber Security:A Crisis of Prioritization.* National Coordination Office for InformationTechnology Research and Development.

Pilgrim, M. (2009) *The Road to HTML 5: contentEditable.* Web Hypertext Application Technology Working Group.

PHP. (2012) *Deprecated features in PHP 5.3.x* [Online]. Available at: http://www.php.net/manual/en/migration53.deprecated.php (Accessed: 4 February 2012).

PHP. (2012) *List of Parser Tokens* [Online]. Available at: http://www.php.net/manual/en/tokens.php (Accessed: 5 February 2012).

PHP. (2011) *Safe Mode* [Online]. Available at: http://php.net/manual/en/features.safe-mode.php (Accessed: 6 December 2011).

PHP. (2012) *htmlspecialchars* [Online]. Available at: http://php.net/manual/en/function.htmlspecialchars.php (Accessed: 3 February 2012).

Sotirov, A. (2005) *Automatic Vulnerability Detection Using Static Source Code Aanalysis.* The University of Alabama.

Textmate. (2012) *textmate - the missing editor* [Online]. Available at: http://macromates.com/ (Accessed: 5 February 2012).

TinyMCE. (2012) *TinyMCE - Javascript WYSIWYG Editor.* Available at: http://www.tinymce.com/ (Accessed: 17th March 2012).

The U.S. Department of Health & Human Services. (2008) *Selecting a Development Approach* [Online]. Available at: http://www.cms.hhs.gov/SystemLifecycleFramework/Downloads/SelectingDevelopmentApproach.pdf (Accessed: 11 November 2011).

Thomas, D. et al. (2009) *Programming Ruby 1.9: The Pragmatic Programmers' Guide.* Pragmatic Bookshelf.

TIOBE. (2012) *TIOBE Programming Community Index for February 2012* [Online]. Available at: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (Accessed: 6 February 2012).

W3C. (2012) *W3C Markup Validation Service.* Available at: http://validator.w3.org/ (Accessed: 23rd January 2012).

W3C. (1999) *Web Content Accessibility Guidelines 1.0.* Available at: http://www.w3.org/TR/WAI-WEBCONTENT/#Guidelines (Accessed: March 1 2012).

WIODE. (2011) *Improved Multi-User, Performance & Security in 2.5.5.* Available at: http://www.wiode.org/improved-multi-user-performance--security-in-255 (Accessed: 15th February 2012).

Wögerer, W. (2005) *A survey of Static Program Analysis Techniques.* Technische Universität Wien.

Bibliography

Howard, M & LeBlanc, D. (2003) *Writing Secure Code.* 2nd edition. Washington, U.S.A: Microsoft Press.

OWASP Foundation. (2009) *Software Assurance Maturity Model (SAMM) v1.0.* OWASP Foundation.

OWASP Foundation. (2008) *OWASP Testing Guide v3.0.* OWASP Foundation.

OWASP Foundation. (2008) *OWASP Code Review Guide v1.1.* OWASP Foundation.

OWASP Foundation. (2006) *OWASP CLASP v1.2*. OWASP Foundation.

Potter, B. & McGraw, G. (2004) *Software security testing*. IEEE Security & Privacy Magazine.

Stuttard, D & Pinto, M. (2011) *The Web Application Hacker's Handbook.* 2nd edn. Indianapolis, U.S.A: Wiley Publishing, Inc.

Appendices

Appendix A – Terms of Reference (TOR)

CM0645: Individual Project

Project Terms of Reference

Reducing software vulnerabilities within the coding stage of a Software Development Lifecycle (SDLC).

Name: Ryan Luke Dewhurst
Student ID: 08026925
Course: Ethical Hacking for Computer Security
Supervisor: Dr. Christopher Laing
Second Marker: Dr. Emil Petkov

General Computing Project

a) Project Title

Reducing software vulnerabilities within the coding stage of a Software Development Lifecycle (SDLC).

b) Background to Project

Traditional methods of reducing software security vulnerabilities are normally employed after the source code has been written. This can be done in the early stages of development through the following of a methodology such as a Secure Development Lifecycle (SDL). It is evident by the amount of vulnerabilities being found within software that SDLs are not widely used or that SDLs are not as efficient as they possibly could be.

According to Microsoft, the later vulnerabilities are found within the development of software, the more time it takes the developer to fix, thus, increasing the cost of the development process. (Microsoft, 2011)

This is confirmed by a survey produced by The National Institute of Standards and Technology (NIST), published within a report called *'The Economic Impacts of Inadequate Infrastructure for Software Testing'* (NIST, 2002). The below table is the result of that survey.

Stage Introduced	Require ments	Coding/unit testing	Integratio n	Beta Testing	Post-product Release
Requirements	1.2	8.8	14.8	15.0	18.7
Coding/unit testing	NA	3.2	9.7	12.2	14.8
Integration	NA	NA	6.7	12.0	17.3

The above table shows that it takes the least hours (1.2) to fix a bug during the requirements stage that was found during the requirements stage. It takes the second least amount of time, 3.2 hours, to fix a bug during the coding/unit testing stage when found during that stage. The above table also indicates that it takes the most amount of time to fix a bug during post-product release no matter what stage the bug was introduced at. It takes an extra 11.6 hours to fix a bug introduced during the coding stage at post-product release.

The Microsoft Secure Development Lifecycle (SDL) process fits into a traditional Software Development Lifecycle (SDLC) consisting of seven phases; training, requirements, design, implementation, verification, release, and response.

During the coding stage of a SDLC, Microsoft's SDL recommends three practices within their Implementation phase. These are, #8 use approved tools, #9 deprecate unsafe functions and #10 perform static code analysis.

- SDL Practice #8: Use Approved Tools

"Define and publish a list of approved tools and associated security checks, such as compiler/linker options and warnings. The list should be regularly updated with the latest versions of the tools."

- SDL Practice #9: Deprecate Unsafe Functions
  "Determine the list of banned functions, use header files, newer compliers, or code scanning tools to check code for the existence of banned functions, and then replace those banned functions with safer alternatives."

- SDL Practice #10: Perform Static Analysis
  "Static analysis consists of analyzing the source code prior to compile."

(Microsoft, 2011)

Implementing and maintaining the above three rules within the coding stage of a SDLC is both time consuming and resource intensive. The individual project hopes to solve most of the above three SDL practices within one product.

The product will be the approved tool and not many other tools will be needed during the coding stage. The product will do basic string matching to identify unsafe functions 'on the fly'. The product will also perform basic static code analysis, either 'on the fly' or before the source code is saved to the hard disk.

The product and research will be useful to companies that have implemented a SDL within their SDLC and want to improve efficiency during the Implementation stage of the SDL. I will need to research and learn how current static code analysis technologies work and which method is best to integrate into the product.

The solution could be in the form of an Integrated Development Environment. (IDE) which does basic string matching for unsafe functions and has a static code analysis engine built in. Research into the different IDEs available and what functionality they currently offer to the developer will need to be carried out.

The product should initially only support one programming language, however, it will be designed in a way that is easily extended to support many other programming languages. As PHP is one of the programming languages I am familiar with and according to the *'TIOBE Programming Community Index for October 2011'*, it is the 4th most popular programming language used (TIOBE, 2011). I believe PHP to be a good programming language to start with.

c) Aims of the project

To investigate and analyse the Microsoft Secure Development Lifecycle (SDL).

To amend a current Integrated Development Environment (IDE) to implement as a Proof of Concept (PoC) the three practices of the Microsoft Secure Development Lifecycle (SDL).

d) Objectives

- Research Software Development Life Cycles (SDLC).
- Research the Microsoft Secure Development Lifecycle (SDL).
- Research the Microsoft SDL practice 8; Use Approved Tools.
- Research the Microsoft SDL practice 9; Deprecate Unsafe Functions.
- Research the Microsoft SDL practice 10; Perform Static Analysis.
- Product design.
- Product development.
- Product implementation.
- Product testing.
- Production of chapters of the project report.
- Product evaluation.
- Process evaluation.

e) Ethical issues

None.

f) Relationship to the course

- CM0429 - Relational Databases will contribute;
  The knowledge of how databases work and how applications interact with them.

- EN0402 - Programming Fundamentals with Robots will contribute;
  Basic programming skills.

- EN0403 - Introduction to Ethical Hacking will contribute;
  Basic security concepts.

- EN0156 - Network Technology 1 will contribute;
  Basic server/client architecture and how applications communicate over a network.

- EN0273 - Programming in C will contribute;
  Basic programming in C.

- IS0503 - Integrative Consultancy Project will contribute;
  Communication and presentation skills.

- CM0567 - CEIS Professional Placement will contribute;
  Practical skills in securing applications and networks.

The course has not addressed advanced application security vulnerabilities, prevention, exploitation or remediation.

g) Sources of information / Bibliography

Bibliography:

Howard, M & LeBlanc, D et al. (2005) *19 Deadly Sins of Software Security.* California, U.S.A: McGraw-Hill Companies.

Howard, M & LeBlanc, D. (2003) *Writing Secure Code.* 2nd edition. Washington, U.S.A: Microsoft Press.

OWASP Foundation. (2009) *Software Assurance Maturity Model (SAMM) v1.0* [Online]. Available at: http://www.opensamm.org/downloads/SAMM-1.0-en_US.pdf (Accessed: 21 October 2011).

OWASP Foundation. (2008) *OWASP Testing Guide v3.0* [Online]. Available at: http://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf (Accessed: 21 October 2011).

OWASP Foundation. (2008) *OWASP Code Review Guide v1.1* [Online]. Available at: https://www.owasp.org/images/2/2e/OWASP_Code_Review_Guide-V1_1.pdf (Accessed: 21 October 2011).

OWASP Foundation. (2006) *OWASP CLASP v1.2* [Online]. Available at: http://www.lulu.com/content/content_download_redirect.php?contentId=1401307&version=3 (Accessed: 21 October 2011).

Stuttard, D & Pinto, M. (2011) *The Web Application Hacker's Handbook.* 2nd edn. Indianapolis, U.S.A: Wiley Publishing, Inc.

References:

Microsoft. (2011) *SDL Helps Reduce the Total Cost of Development* [Online]. Available at: http://www.microsoft.com/security/sdl/learn/costeffective.aspx (Accessed: 21 October 2011).

NIST. (2002) *The Economic Impacts of Inadequate Infrastructure for Software Testing* [Online]. Available at: http://www.nist.gov/director/planning/upload/report02-3.pdf (Accessed: 21 October 2011).

TIOBE. (2011) *TIOBE Programming Community Index for October 2011* [Online]. Available at:

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
(Accessed: 21 October 2011).

h) Resources

I will need to purchase books from the OWASP Foundation, as the university library does not supply any.

I will require a development computer to develop the product; this will be achieved with a virtual machine on a host computer that I already own. All software used will be open source or free to use. For example; Linux, Apache, PHP and MySQL.

For the final demo I will require a projector and a computer with Internet access.

i) Structure and contents of project report

Abstract
Project abstract.

Chapter I: Introduction
An introduction to the project, its aims and objectives.

- Problem description.
- The purpose, motivation or relevance.
- The methods.
- The results.
- Conclusion.

Chapter II: Secure Development Lifecycles (SDL)
An introduction and overview of current SLC practices and methodologies.

- Introduction.
- Training.
- Requirements.
- Design.
- Implementation.
- Verification.
- Release.
- Response.

Chapter II: Static Code Analysis
An introduction and overview of Static Code Analysis practices and methodologies.

- Introduction.
- Manual Vs Automated.

- Model checking.
- Data-flow analysis.
- Abstract interpretation.
- Assertions.

Chapter IV: Integrated Development Environments (IDE)
An introduction and overview of IDEs and their features.

- Introduction.
- Features.
- Design.

Chapter V: The product
Design, implementation, introduction and overview of the product, how it works and how it can be used.

- Design
- Implementation
- Testing

Chapter VI: Evaluation & Conclusions
The projects conclusions, product effectiveness and identification of possible further work.

- Product evaluation.
- Process evaluation.
- Conclusions.

j) Marking scheme

i. Project Type

General Computing Project

ii. Project Report

- Introduction
  Abstract & Introduction
- Analysis
  Secure Development Lifecycle (SDL)
  Static Code Analysis
  Integrated Development Environment (IDE)
- Synthesis
  Design overview
  Implementation
  Testing
- Evaluation and Conclusions

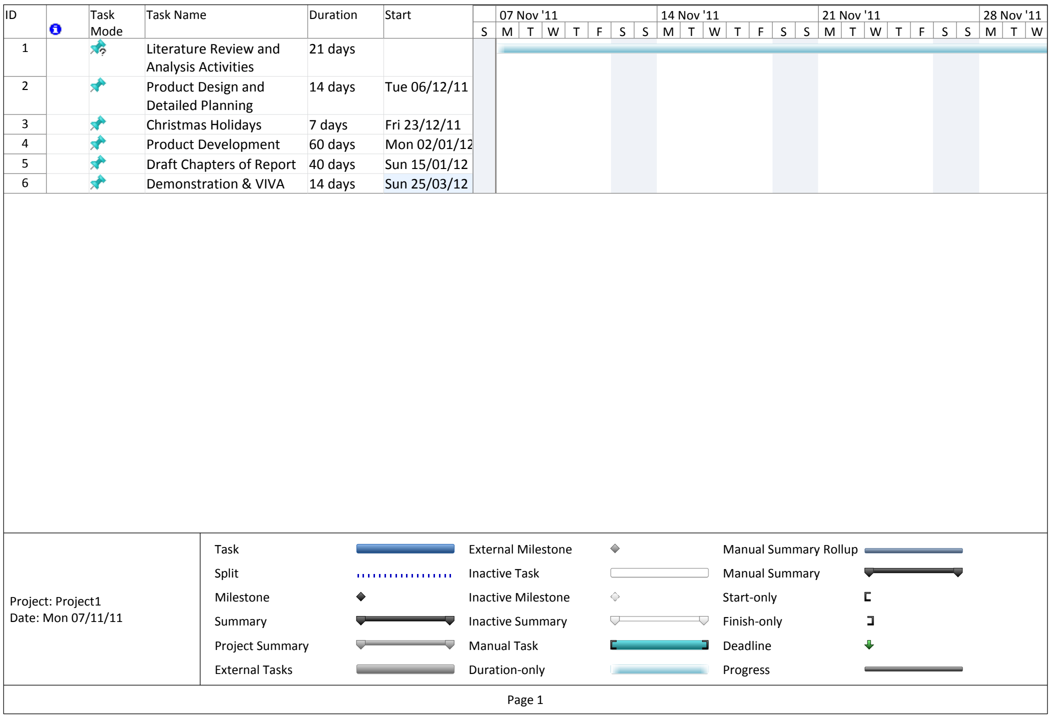
Product
Process
Conclusion

iii. Product

- An Integrated Development Environment (IDE).
- Unsafe function string matching.
- Static Code Analysis integration.

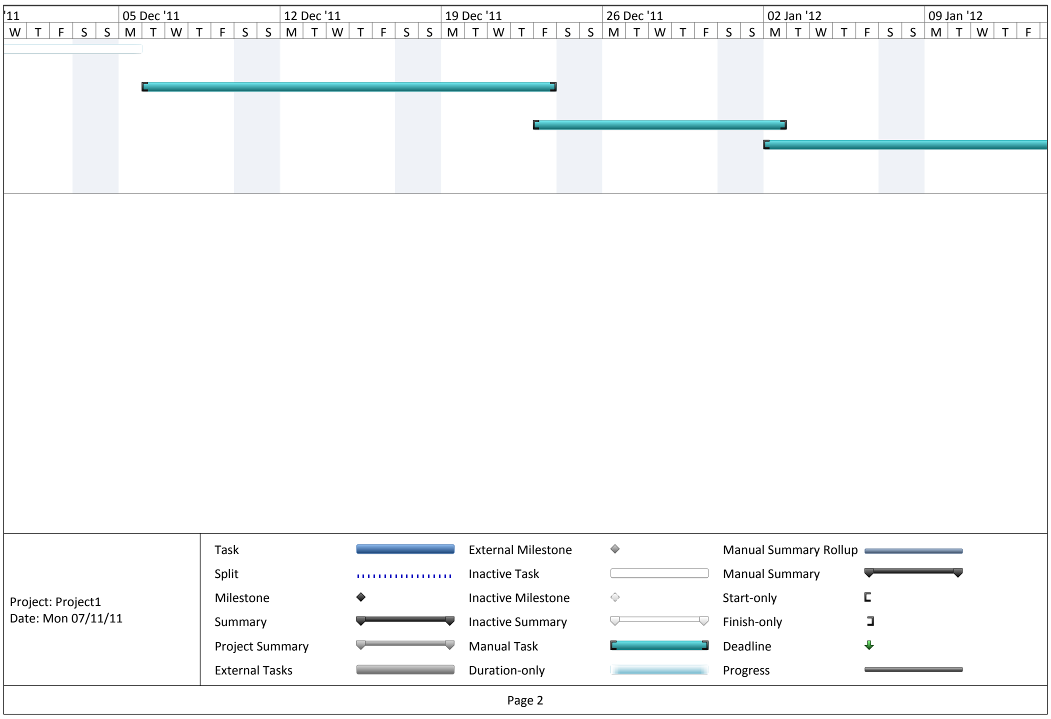
l) Project plan

Page 1

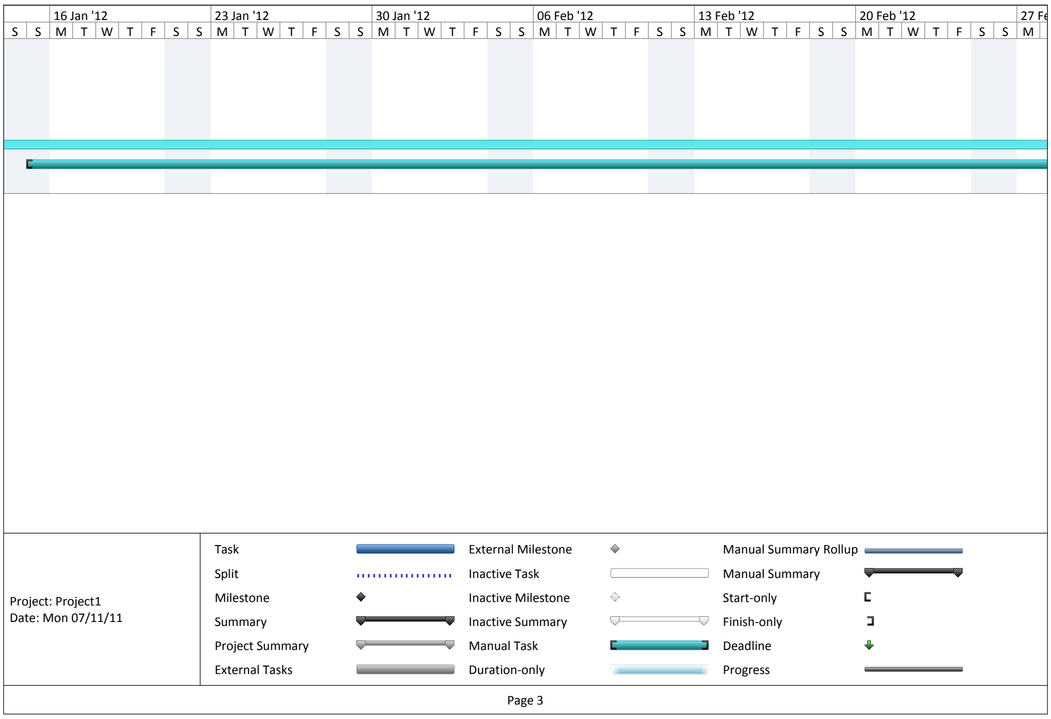


ID		Task Mode	Task Name	Duration	Start
1			Literature Review and Analysis Activities	21 days	
2			Product Design and Detailed Planning	14 days	Tue 06/12/11
3			Christmas Holidays	7 days	Fri 23/12/11
4			Product Development	60 days	Mon 02/01/12
5			Draft Chapters of Report	40 days	Sun 15/01/12
6			Demonstration & VIVA	14 days	Sun 25/03/12

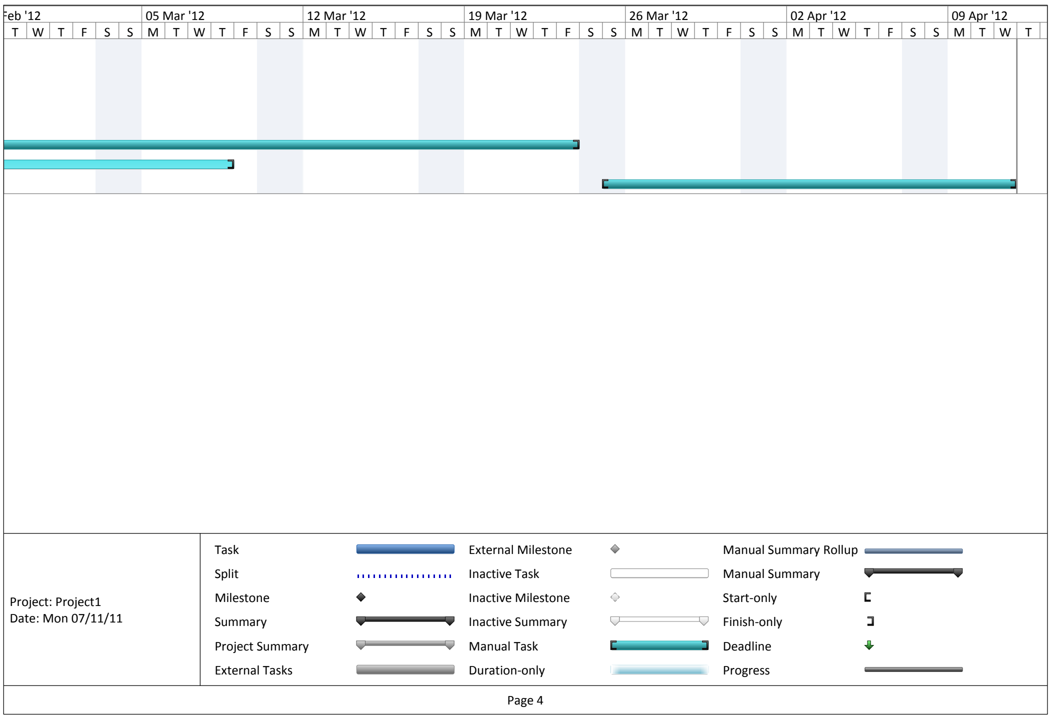
Page 2


'11
05 Dec '11
12 Dec '11
19 Dec '11
26 Dec '11
02 Jan '12
09 Jan '12
Project: Project1
Date: Mon 07/11/11
Task
Split
Milestone
Summary
Project Summary
External Tasks
External Milestone
Inactive Task
Inactive Milestone
Inactive Summary
Manual Task
Duration-only
Manual Summary Rollup
Manual Summary
Start-only
Finish-only
Deadline
Progress
Page 2

Page 3


16 Jan '12
23 Jan '12
30 Jan '12
06 Feb '12
13 Feb '12
20 Feb '12
Project: Project1
Date: Mon 07/11/11
Task
Split
Milestone
Summary
Project Summary
External Tasks
External Milestone
Inactive Task
Inactive Milestone
Inactive Summary
Manual Task
Duration-only
Manual Summary Rollup
Manual Summary
Start-only
Finish-only
Deadline
Progress
Page 3

Page 4


Feb '12
05 Mar '12
12 Mar '12
19 Mar '12
26 Mar '12
02 Apr '12
09 Apr '12
Project: Project1
Date: Mon 07/11/11
Task
Split
Milestone
Summary
Project Summary
External Tasks
External Milestone
Inactive Task
Inactive Milestone
Inactive Summary
Manual Task
Duration-only
Manual Summary Rollup
Manual Summary
Start-only
Finish-only
Deadline
Progress
Page 4

Appendix B – PHP Deprecated Functions

call_user_method
call_user_method_array
define_syslog_variables
dl
ereg
ereg_replace
eregi
eregi_replace
set_magic_quotes_runtime
session_register
session_unregister
session_is_register
set_socket_blocking
split
spliti
sql_regcase
mysql_db_query
mysql_escape_string

Appendix C – PHP Lexical Analysis Tokens

Token	Syntax
T_ABSTRACT	abstract
T_AND_EQUAL	&=
T_ARRAY	array()
T_ARRAY_CAST	(array)
T_AS	as
T_BAD_CHARACTER	
T_BOOLEAN_AND	&&
T_BOOLEAN_OR	||
T_BOOL_CAST	(bool) or (boolean)
T_BREAK	break
T_CASE	case
T_CATCH	catch
T_CHARACTER	
T_CLASS	class
T_CLASS_C	__CLASS__
T_CLONE	clone
T_CLOSE_TAG	?> or %>
T_COMMENT	// or #, and /* */ in PHP 5
T_CONCAT_EQUAL	.=
T_CONST	const

T_CONSTANT_ENCAPSED_STRING	"foo" or 'bar'
T_CONTINUE	continue
T_CURLY_OPEN	{$
T_DEC	--
T_DECLARE	declare
T_DEFAULT	default
T_DIR	__DIR__
T_DIV_EQUAL	/=
T_DNUMBER	0.12, etc
T_DOC_COMMENT	/** */
T_DO	do
T_DOLLAR_OPEN_CURLY_BRACES	${
T_DOUBLE_ARROW	=>
T_DOUBLE_CAST	(real), (double) or (float)
T_DOUBLE_COLON	::
T_ECHO	echo
T_ELSE	else
T_ELSEIF	elseif
T_EMPTY	empty
T_ENCAPSED_AND_WHITESPACE	" $a"
T_ENDDECLARE	enddeclare
T_ENDFOR	endfor
T_ENDFOREACH	endforeach
T_ENDIF	endif
T_ENDSWITCH	endswitch
T_ENDWHILE	endwhile
T_END_HEREDOC	
T_EVAL	eval()
T_EXIT	exit or die
T_EXTENDS	extends
T_FILE	__FILE__
T_FINAL	final
T_FOR	for
T_FOREACH	foreach
T_FUNCTION	function or cfunction
T_FUNC_C	__FUNCTION__
T_GLOBAL	global
T_GOTO	goto
T_HALT_COMPILER	__halt_compiler()
T_IF	if
T_IMPLEMENTS	implements
T_INC	++
T_INCLUDE	include()
T_INCLUDE_ONCE	include_once()

T_INLINE_HTML	
T_INSTANCEOF	instanceof
T_INT_CAST	(int) or (integer)
T_INTERFACE	interface
T_ISSET	isset()
T_IS_EQUAL	==
T_IS_GREATER_OR_EQUAL	>=
T_IS_IDENTICAL	===
T_IS_NOT_EQUAL	!= or <>
T_IS_NOT_IDENTICAL	!==
T_IS_SMALLER_OR_EQUAL	<=
T_LINE	__LINE__
T_LIST	list()
T_LNUMBER	123, 012, 0x1ac, etc
T_LOGICAL_AND	and
T_LOGICAL_OR	or
T_LOGICAL_XOR	xor
T_METHOD_C	__METHOD__
T_MINUS_EQUAL	-=
T_ML_COMMENT	/* and */
T_MOD_EQUAL	%=
T_MUL_EQUAL	*=
T_NAMESPACE	namespace
T_NS_C	__NAMESPACE__
T_NS_SEPARATOR	\
T_NEW	new
T_NUM_STRING	"$a[0]"
T_OBJECT_CAST	(object)
T_OBJECT_OPERATOR	->
T_OLD_FUNCTION	old_function
T_OPEN_TAG	<?php, <? or <%
T_OPEN_TAG_WITH_ECHO	<?= or <%=
T_OR_EQUAL	|=
T_PAAMAYIM_NEKUDOTAYIM	::
T_PLUS_EQUAL	+=
T_PRINT	print()
T_PRIVATE	private
T_PUBLIC	public
T_PROTECTED	protected
T_REQUIRE	require()
T_REQUIRE_ONCE	require_once()
T_RETURN	return
T_SL	<<
T_SL_EQUAL	<<=

T_SR	>>
T_SR_EQUAL	>>=
T_START_HEREDOC	<<<
T_STATIC	static
T_STRING	"parent"
T_STRING_CAST	(string)
T_STRING_VARNAME	"${a
T_SWITCH	switch
T_THROW	throw
T_TRY	try
T_UNSET	unset()
T_UNSET_CAST	(unset)
T_USE	use
T_VAR	var
T_VARIABLE	$foo
T_WHILE	while
T_WHITESPACE	\t \r\n
T_XOR_EQUAL	^=

Appendix D – Usability and Compatibility Test

Appendix E – AChecker Accessibility Report

ACHECKER®

Web Accessibility Checker
atutor.ca/achecker

Thursday March 1, 2012 12:59:37

Source URL: http://46.64.8.240/~ryan/Sites/devbug/#
Source Title: DevBug - PHP Static Code Analysis

Accessibility Review (Guidelines: WCAG 2.0 (Level AA))

Report on known problems (0 found):

Congratulations! No known problems.

Page 1/1

Appendix F – Usability & Compatibility Inspection

Mozilla Firefox version 8.0.1

Action	Expected Re-action	Actual Re-action	Pass
Load Page	Page loads as expected, user interface as expected, loads in a timely manner.	Page loads as expected, user interface as expected, loads in a timely manner.	Yes
Run button pressed	Source code in the Code Editor gets analysed and results displayed in the Information Panel properly.	Source code in the Code Editor gets analysed and results displayed in the Information Panel properly.	Yes
Clear button pressed	Code Editor, Information Panel and the browsers localstorage is cleared.	Code Editor and the browsers localstorage is cleared. Information Panel is not cleared.	No
Help button pressed	Help information displayed in the Information Panel in correct font, size, colour.	Help information displayed in the Information Panel in correct font, size, colour.	Yes
About button pressed	About information displayed in the Information Panel in correct font, size, colour.	About information displayed in the Information Panel in correct font, size, colour.	Yes
Theme selection	Theme is able to be selected. The chosen theme displays as intended.	Theme is able to be selected. The chosen theme displays as intended.	Yes
Unload Page	Contents of the Code Editor should be saved to localstorage and then retrieved on page load.	Contents of the Code Editor should be saved to localstorage and then retrieved on page load.	Yes

Google Chrome version 17.0.963.65

Action	Expected Re-action	Actual Re-action	Pass
Load Page	Page loads as expected, user interface as expected, loads in a timely manner.	Page loads as expected, user interface as expected, loads in a timely manner.	Yes

Run button pressed	Source code in the Code Editor gets analysed and results displayed in the Information Panel properly.	Source code in the Code Editor gets analysed and results displayed in the Information Panel properly.	Yes
Clear button pressed	Code Editor, Information Panel and the browsers localstorage is cleared.	Code Editor and the browsers localstorage is cleared. Information Panel is not cleared.	No
Help button pressed	Help information displayed in the Information Panel in correct font, size, colour.	Help information displayed in the Information Panel in correct font, size, colour.	Yes
About button pressed	About information displayed in the Information Panel in correct font, size, colour.	About information displayed in the Information Panel in correct font, size, colour.	Yes
Theme selection	Theme is able to be selected. The chosen theme displays as intended.	Theme is able to be selected. The chosen theme displays as intended.	Yes
Unload Page	Contents of the Code Editor should be saved to localstorage and then retrieved on page load.	Contents of the Code Editor should be saved to localstorage and then retrieved on page load.	Yes

Apple Safari version 5.1.2

Action	Expected Re-action	Actual Re-action	Pass
Load Page	Page loads as expected, user interface as expected, loads in a timely manner.	Page loads as expected, user interface as expected, loads in a timely manner.	Yes
Run button pressed	Source code in the Code Editor gets analysed and results displayed in the Information Panel properly.	Source code in the Code Editor gets analysed and results displayed in the Information Panel properly.	Yes
Clear button pressed	Code Editor, Information Panel and the browsers localstorage is cleared.	Code Editor and the browsers localstorage is cleared. Information Panel is not cleared.	No

Help button pressed	Help information displayed in the Information Panel in correct font, size, colour.	Help information displayed in the Information Panel in correct font, size, colour.	Yes
About button pressed	About information displayed in the Information Panel in correct font, size, colour.	About information displayed in the Information Panel in correct font, size, colour.	Yes
Theme selection	Theme is able to be selected. The chosen theme displays as intended.	Theme is able to be selected. The chosen theme displays as intended.	Yes
Unload Page	Contents of the Code Editor should be saved to localstorage and then retrieved on page load.	Contents of the Code Editor should be saved to localstorage and then retrieved on page load.	Yes

Microsoft Internet Explorer version 8

Action	Expected Re-action	Actual Re-action	Pass
Load Page	Page loads as expected, user interface as expected, loads in a timely manner.	The page loads in a timely manner, however, the user interface is distorted. The theme selection box is not parallel with the buttons. And the buttons are square instead of round.	No
Run button pressed	Source code in the Code Editor gets analysed and results displayed in the Information Panel properly.	Seems as though the source code is not analysed as no results are displayed within the Information Panel.	No
Clear button pressed	Code Editor, Information Panel and the browsers localstorage is cleared.	Code Editor and the browsers localstorage is cleared. Information Panel is not cleared.	No
Help button pressed	Help information displayed in the Information Panel in correct font, size, colour.	Help information displayed in the Information Panel in correct font, size, colour.	Yes

About button pressed	About information displayed in the Information Panel in correct font, size, colour.	About information displayed in the Information Panel in correct font, size, colour.	Yes
Theme selection	Theme is able to be selected. The chosen theme displays as intended.	Theme is able to be selected. The chosen theme displays as intended.	Yes
Unload Page	Contents of the Code Editor should be saved to localstorage and then retrieved on page load.	Contents of the Code Editor should be saved to localstorage and then retrieved on page load.	Yes