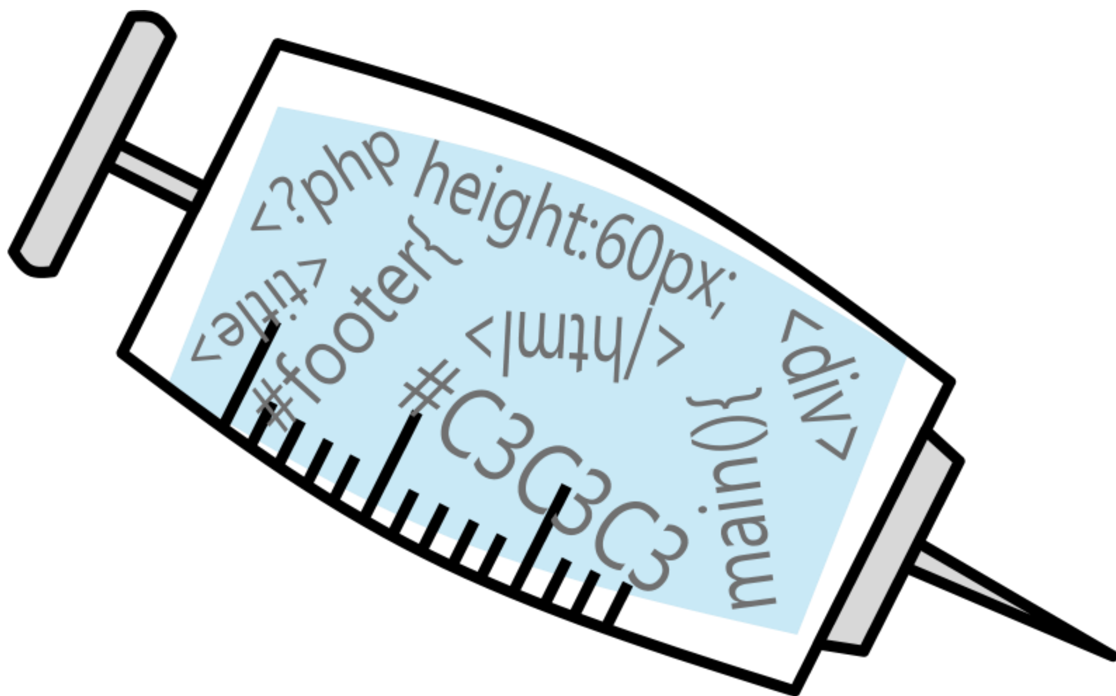


Countering Web Injection Attacks: A Proof of Concept

MSc Project Background Report

Benjamin Hall

Table of Contents

Abstract iii
Introduction 1
Why is the problem worth investigating? 1
Aims of the project 1
Project Background 2
Web Injections 2
Cross-Site Scripting (XSS) 3
Non-Persistent 3
Persistent 3
DOM 4
Cross-Site Request Forgery 4
SQL Injection 5
Tautology 5
Union queries 5
Piggy-back queries 5
Stored procedures 5
Illegal or logically incorrect queries 5
Inference 6
Alternate encodings 6
HTTP Header Injection 6
Cache Poisoning 6
Carriage Return and Line Feed (CRLF) 6
HTTP Request/Response Splitting 6
HTTP Request/Response Smuggling 7
Session Fixation 7
Cross-Site Cooking 8
Cross-Sub domain Cooking 8
Detection and Prevention Techniques 8
Validation 8
Filtering 9
Escaping 9
Encoding 9
Encryption 9
Tokenisation 10
Issues with filtering methods 10
Existing Methods and Tools 10

Development languages 10
Development techniques and other methods 12
Conclusions 13
Research Methods 14
Objectives of the project 14
Technical Requirement Specification 14
Injection tool 15
Deliverables 16
Development methodologies 16
Design & Implementation 16
Testing 16
Evaluation & appraisal 17
Project plan 17
Bibliography .... A
Appendix .... B
Glossary of Terms .... B
Cross-Site Scripting C
Non-Persistent/Reflected Example: C
Persistent Example: C
Cross-Site request forgery .... D
Cross-Site request forgery Example: .... D
PHP Sanitation methods .... E
Filter .... E
Sanitisation .... E
Flags .... E
Flag – ASCII filter .... F
Project Plan .... G

Abstract

Web injection attacks are a set of web vulnerabilities intended to exploit the web application's security, by inserting a malicious payload into the web application. Their purpose is typically to gain elevated right of entry in order to access application data or cause malicious damage such as data manipulation or destruction.

This report looks at some of the most common injection attacks containing the most predominant exploitation outcomes. Subsequently injection detection and prevention techniques are outlined as a proof of concept, from both an application environment perspective along with available development practice and procedures for language specific developments.

The project aims to produce a Technical Requirement Specification, providing guidance to web application developers regarding the importance of injection attacks and ways of detecting and preventing injection attacks. Furthermore, the project will develop a scripting library in order to demonstrate these techniques within the web application.

Finally, this report includes a plan of how the project will be accomplished, the development methodologies, along with the required resources and a rough project timescale.

Introduction

Why is the problem worth investigating?

Today's web is used for international commerce and online banking, which has led to a multi-million dollar criminal industry. With more than 10,000 new threats detected every day (McAfee Labs Presentation,12/2010), threat intelligence has become one of the most critical aspects in protecting web based businesses to ensure high levels of business continuity. There are many dedicated companies whose aim is to try and protect organisations including banks, governments, small businesses and everyday users in order to make sure their online experiences are safe and secure.

Web technologies include large numbers of different web programming languages; frameworks, libraries and development paradigms; communication standards including HTTP, FTP, SMTP; and also server-side platforms such as Apache. When developing new technologies, web security is often an afterthought in sacrifice to innovation. Constant development of new technologies also increases the complexity of securing a web application. In recent years we have seen the growth of web security attacks, most commonly these attacks are performed using web application injections.

Web injection attacks are a set of web application vulnerabilities whereby an attacker inserts a malicious payload into the web application through a variety of entry points. Their purpose is to exploit the application, providing the attacker with some gain through either elevated access or malicious sabotage. The most common and severe web injection attacks are discussed within the Project Background section of this report.

Aims of the project

This project will focus around securing a web application against web injection attacks. In particular this will include performing research into the various types of injection attacks, their sub-variants and how they are successfully executed. The aim is to develop an understanding of how web injection attacks may be detected and ultimately prevented. Based upon these findings the project will aim to develop an appropriate web injection solution through the production of a programming scripting library or plug-in.

The development will focus on providing a solution that is easy to implement and use, thus it must try to eliminate the need for training and reprogramming of the web application. This also includes the avoidance of programming conflict with the existing system, possibly through the use of a separate coding namespace. The package must allow widespread implementation over a number of development platforms and must not be constricted to a single development programming language. Finally, due to the fast evolving nature of these attacks, the package must be extensible allowing fast inclusion of new injection measures and definitions.

The second development phase will be the production of a Technical Guidelines Specification. The guidelines are aimed at educating technical specialists with respect to the known types of web injection attacks, along with some methods for their detection and prevention. The guidelines will focus more towards other security measures available to the developer along with advised development practices and procedures to help prevent injection attacks.

Project Background

Large areas of web security often referred to as 'black areas' or 'zero day attacks' exist for which we cannot estimate or begin to defend against. However there are a large number of known web injection attacks which we can try to understand, alleviating the risk from these attacks occurring by and implementing measures to try detect and possible prevent these attacks.. Furthermore, the amount of web injections we know are in existence is exhaustive. Some of these vulnerabilities are trivial flaws within the web application, however others consist of sophistically complex actions designed to manipulate the web application's security and data.

This section outlines some of the most serious and common injection attacks, how they operate and examples of successful exploits. Afterwards, this proof of concept outlines the previous work carried out to try detecting and preventing these injection attacks. Finally, this report covers the current solutions along with other methods of particular interest to a technical specialist when attempting to develop this solution.

Most of the information within this section has been taken from academic papers, industry white papers, news articles and books. Due to the nature of this project a large number of resources have been taken from the Internet including web application security bodies, organisations and standards associations. Appropriate referencing has been included where these Internet sources have been discussed, including the date of access at time of writing. Please note some of these links may have been revised since this initial access.

Web Injections

Input validation exploits are the most prominent form of application vulnerabilities, consisting of maliciously constructed input strings to exploit browser behaviour. Specific malicious intentions vary depending on the form of injection; nevertheless all malicious exploits endeavour to compromise the three areas of information security: confidentiality, integrity and availability. Some examples of successful application compromise include privilege escalation, information leakage or destruction of data. (1)

Injection techniques include the use of HTTP headers to pass input data to the server-side web application, more specifically the GET and POST methods may include malicious parameters processed by the web application. The GET and POST methods are commonly sent within the HTTO header by the use of html forms or, for GET requests, the use of specially formatted URL addresses. McAfee's threat report (2) contains an alarming increase in malware with the detection of more than 20 million new threats last year alone. Consequently many of these malware variants contain the ability to inject arbitrary code into the client side browser, normally undetected and without the knowledge of the server-side application.

Injected scripts often utilise the functionality of client-side 'Turing complete' web scripting languages such as JavaScript and VBScript (3). Scripting tags used with some forms of code injections include: SCRIPT, APPLET (deprecated, inserts a java applet) and OBJECT; although SCRIPT is the most common due to incompatibility issues with the latter two (4) (3). Otherwise script may be inserted 'inline' to the HTML elements, for example this code is contained directly within the HTML script:

JavaScript inserted within HTML Anchor	<a href="JavaScript:[scripting code]"

Cross-Site Scripting (XSS)

With around 80% of all web application vulnerabilities (5), Cross-Site Scripting (referred to as XXS) is one of the most predominant web application injection attacks and a mount to other injection vulnerabilities. Referring to a class of string based injection attacks (6), XSS mainly occurs due to inadequate input filtering procedures exerted by the web application host (5). Like most web injection attacks, successful XSS exploits may lead to compromised authentication information, privilege escalation and possible disclosure of confidential information (6).

While this type of attack is achieved as a result of vulnerabilities on the server-side of the application, exploits are within the client-side web browser, further adding to the complexity of detection and gathering evidence of a successful attack (6). Prevention techniques associated with XSS have not been applied mainly due to their development overhead and degradation in client side application performance (7).

Reports suggest more than 100 XSS variants now exist, however XSS can be categorised into three main injection methods: non-persistent or reflected, persistent or stored and DOM injections. Usually these techniques are independently executed, although it is not uncommon for an attack to consist of multiple variants (7).

Non-Persistent

Non-persistent or reflected XSS is a HTTP exploit, where parts of the incoming HTTP request are simply echoed directly into the HTML of the HTTP response, resulting in unsecure output by the web browser. More specifically this exploit is executed by a tampered URL address containing malicious code with the HTTP GET request parameters. This vulnerability is particularly difficult to detect when masqueraded by encoding techniques or used in conjunction with URL shortening services (such as 'tinyurl.com'). Although this form of attack is not harmful within the server-side application, it is caused by inadequate filtering standards and may lead to further application exploits (6). Please refer to Appendix C for a breakdown of how non-persistent/reflected XSS can be performed.

Example:

URL containing malicious code	http://website.com/?search=<script>alert(document.cookie)</script>
Script echoing	<input name="search" value="<?php print REQUEST['search']?>">

Persistent

Persistent or stored XSS is a more devastating form of web application exploit intended to permanently inject malicious code into the web application's server-side storage, typically in the form of a database. Subsequently this attack will disseminate the stored malicious payload to all users accessing the infected page (6). For example, an attacker injects the malicious code into a blog comment, unfiltered this script is executed by every visit to the page, thus multiple users are affected. Please refer to Appendix C for a breakdown of how persistent XSS can be performed.

In 2006 MySpace was infected by a persistent XSS vulnerability contained with a QuickTime video. The exploit overlaid the video with a JavaScript menu containing links to a bogus login page, targeted at stealing the user's authentication credentials. Once the victim is compromised the exploit uses the now legitimate access route to post spam message, furthermore this script replicates on the victims profile page in attempt to spread further (8).

DOM

DOM XSS is a variant of reflected XSS, a form of web injection that exploits vulnerabilities passed by the URL. Typically this exploit passes JavaScript code to the web browser allowing the manipulation of the DOM tree, including the addition or removal of elements and styling. At one time it was possible for the open application to steal information from other simultaneously open windows, a flaw which Netscape fixed with the introduction of Same Origin Policy (SOP). However this exploit is still possible by forwarding the users' information onto an external site using a 'XML HTTP Request'. (7)

Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF/XSRF) also known as Session Ridding, Hostile linking or Cross-Site Reference Forgery (9) is a client based vulnerability "among the top 5 worst vulnerabilities for web-based programs" (10). The majority of web developers are oblivious to Cross-site Request Forgery attacks, making this form of attack much harder to secure (11). Similar to XSS, CSRF contains two main variants: Persistent and Non-Persistent (10).

Essentially CSRF is an exploit of trust between the web application and the user, as opposed to XSS which exploits the trust between the browser and the web application (9). More specifically CSRF is an attack executed on a web application against another application which the user is currently authenticated, thus CSRF is mainly a form of Social Engineering exploit that performs unauthorised and undetected user actions (9). This form of attack often occurs after the user has authenticated with the legitimate web, by then navigating to another applications controlled by the attacker. Still authenticated with the original web application, the attacker's application executes malicious requests with the original authentication application, consequently these request appear to originate from the authenticated user. Once this authentication has been compromised, the attacker may send arbitrary HTTP requests to the legitimate authenticated web application (11). Please refer to Appendix D for a breakdown of Cross-Site Request Forgery can be performed.

The subsequent diagram within Appendix D shows a real-life example of CSRF. This attack uses an exploit of the source attribute within a HTML image tag. The source path consists of a URL address containing GET parameters, used to perform an input action upon the legitimate web application. The exploit of the image tag provides a simple form of execution, with minimal sign of exploit disclosure. Attempts made by the browser to receive an image with an invalid path name will result within an image error due to the output mismatch of received text/html opposed to the expected image/jpeg (10).

An important deliberation of this exploit is the origin of the second web application. The CSRF attack may originate from another legitimate source compromised by another form of injection attack. For example an application may be compromised by a cross-site scripting attack containing a malicious payload to send requests to another web application similar to the previous example. This form of application attack may be as a result of weaker security constraints upon the second legitimate web application, furthermore this attack may increase the amount of successfully executed exploits.

From the perspective of the browser, Same Origin Policy (previously discussed within XSS) it is not sufficient enough to prevent CSRF attacks. Solutions exist to reduce the likelihood of CSRF attacks, such as checking the HTTP referrer header; however these measures are easily bypassed as shown with attacks upon a second legitimate application, or in this case of execution on an attacker's application – the use of HTML Meta tags (10).

SQL Injection

Structure Query language (SQL) is the de-facto standard for accessing and manipulating web databases and consequently the point of entry for the majority of persistent web injection attacks including XXS and CSRF. However the database language itself is prone to attacks known as SQLIA (SQL Injection Attacks) (12), with 75% originating from China and the United States of America (2). This injection attack consists of unfiltered application input containing SQL code. The attackers endeavour is to change the “logic, semantics or syntax of an SQL query by inserting new SQL keywords” (13). Consequences of SQLIA’s are some of the most severe, with attacks directly on application data, exploits lead to a violation in confidentiality or integrity by disclosing or deleting data.

There exists a number of SQLIA variants used to exploit different SQL vulnerabilities.

Tautology

Aimed at bypassing authentication, an attacker injects SQL tokens that evaluate a comparative statement as true (13). For example when passing authentication details through a login entry form, the attacker may pass an injected SQL code for the password input:

Input	password’ OR ‘1’=’1’
SQL statement	SELECT * FROM employee WHERE id=’9’ AND password=’password’ OR ‘1’=’1’;

Union queries

When SQL is used to retrieve data, the UNION operator is used to join multiple relational database tables together. By inserting code containing the UNION operator, the attacker may return more information than the query intended; therefore this attack aims to compromise data confidentiality. (13)

Piggy-back queries

This form of attack exploits the ‘;’ delimiter used at the end of an SQL statement to separate multiple statements from one another. By exploiting this syntax the attacker may append multiple queries to the original SQL statement. The first query will be legitimate; however succeeding injected and illegitimate queries will allow the attacker a great amount of flexibility with the ability to execute arbitrary SQL statements. (13)

While most languages allow multiple queries within one statement, this is not supported within PHP’s ‘mysql_query()’ function, thus eliminating the risk of this vulnerability (14). It must also be noted some scripting languages do not require the ‘;’ delimiter, thus ruling out prevention techniques such as removing data after the initial statement. (13)

Stored procedures

A database administrator may set an additional abstraction layer in the form of a stored procedure intended to hide some of the applications functionally. Similar to the piggy-back vulnerability, an attacker may manipulate these procedures by appending terminating syntax to the query. (13)

Illegal or logically incorrect queries

Unlike other SQLIA’s, the primary focus of this attack is to disclose weaknesses of the database rather than attempting to manipulate the data. By entering rejected queries, the application will return error messages containing useful debugging information. This information can be used by the attacker to further assist in finding vulnerable query parameters for subsequent injection attacks (12).

Inference

Inference attacks are designed to change the behaviour of the database in order to gather vulnerability information. They differ from illegal/incorrect queries by the output returned by the execution, instead of debugging information the user is often presented with a generic error warning. Detecting vulnerabilities within the applications code instantly becomes more difficult; however there are various techniques that use this limited output to the attacker's advantage.

Blind injections use the error output as a true/false feedback for the attempted query. By executing a series of queries the attackers may learn from application weaknesses. Timing attacks involve the use of injected 'if then' and 'WAITFOR' commands. The success of the injected statement can be determined by observing the time delays between each application response, acting as a true/false mechanism. (12) (13)

Alternate encodings

Encoding techniques may be used to bypass application filtering processes. Using alternate encodings such as ASCII, Unicode or Hexadecimal allow the masquerading of known bad characters.

HTTP Header Injection

Cache Poisoning

Cache poisoning is a mechanism for performing HTTP header injections by exploiting vulnerabilities within the web cache. This vulnerability is particularly significant when a shared cache is in use, for instance those found in proxy servers. Within this case the injected HTTP header will be distributed to all users until the cache entry is purged. Cache poisoning is a particularly difficult attack to perform and requires a number of conditions in order to be successful.

To perform a successful cache poisoning attack, the attacker must flush the cache of its contents by establishing a vulnerable service code to inject the new HTTP header. Once flushed the attacker may send a specially crafted message, stored in the cache to become the response for all subsequent requests. (15) (16) (17)

Carriage Return and Line Feed (CRLF)

Carriage Return and Line Reed (CRLF) refers to a sequence of special input characters that represent End of Line (EOL) markers, used to terminate HTTP header information. In order for EOL exploits to succeed, the application must allow the input of the special Carriage Return (CR) and Line Feed (LF) characters, often given by the syntax '%0d' and '\r' respectively. This form of vulnerability is commonly exploited within many Internet protocols including MIME (email), NNTP (newsgroups) and HTTP. (18) (19)

The CRLF vulnerability contains 4 main HTTP exploits: HTTP Request Splitting, HTTP Response Splitting, HTTP Request Smuggling and HTTP Response Smuggling (15).

HTTP Request/Response Splitting

HTTP Request/ Response Splitting are forms of response hijacking exploits that utilise the CRLF vulnerability (15). Although not a direct form of attack itself, it does provide a mount for other injection attacks including XSS, cache poisoning and Cross-User Defacement (18).

This form of vulnerability allows an attacker to break the original HTTP request into multiple requests; subsequently this allows the attacker to inject arbitrary header or body content including the construction of an entirely new HTTP header. The malicious data contained within this HTTP

payload is consequently included within the HTTP response header, injecting the malicious payload onto the requested page (18) (17). The example below shows an end of line marker contained within the HTTP header string terminating the original HTTP header and starting the new inserted HTTP header.

Malicious string containing EOL characters	Some String \r\nHTTP/1.1 200 OK\r\n…
HTTP header after injection	HTTP/1.1 200 OK . . . Set-Cookie: author=Some String HTTP/1.1 200 OK . . .

The second HTTP response shown above is now under complete control of the hacker and may contain any content including an entirely new HTTP response.

HTTP Request/Response Smuggling

Similar to HTTP Request/Response Splitting, HTTP Request/Response Smuggling also facilitates other injection vulnerabilities. Within this HTTP exploit the attacker smuggles the request using encapsulation techniques to embed a secondary HTTP header within the original HTTP payload, exploiting incomplete parsing carried out on an intermediate HTTP proxy system (20). However HTTP Request/Response smuggling "…does not require the existence of application vulnerabilities" (21). The following example shows a second HTTP header contained within a parameter of the original HTTP header.

HTTP header	GET /some_page.jsp?param1=value1¶m2= Content-Type: application/x-www-form- Content-Length: 0 Foobar: GET /mypage.jsp HTTP/1.0 Cookie: my_id=1234567 Authorization: Basic ugwerwguwygruwy

Session Fixation

Session management is the fundamental principle allowing today's web based applications to track user interactions. The session architecture requires Session Identifies (SIDs) to map each application action to the specific user due to HTTP being a stateless protocol (cannot remember previous requests) (11), thus SIDs are the underlying tool of authentication. As SIDs possess a significant control over the user's interaction with the web application, they evidently become a natural target for malicious attack. SIDs can be stored a number of ways including hidden form fields, cookies and URL arguments; although the use of cookies is prominent and arguably the most secure. (22)

Session Hijacking is the process whereby an attacker obtains the session identifier of the victim, performing a replay attack by re-submitting the authentication parameter to gain authenticated access to the web application. Session fixation on the other hand does not concentrate on obtaining the users SID, instead this attack alters the victims SID to an authenticated SID held by the attacker, commonly known as the 'trap session'. Upon further authentication actions by the user (such as bank verification), the attackers account now has access to the newly authenticated resources. (23) (22)

Cross-Site Cooking

Cookies provide the most "convenient, covert, effective and durable means of exploring session fixation vulnerabilities" (22). Under Same Origin Policy (SOP) (11), browsers will only accept cookies assigned by the originating server or domain; however there are a number of injection methods which can be used to carry out an exploit of cookie data. This includes the injection of Meta tags containing the 'Set-Cookie' attribute or the HTTP 'Set-Cookie' header response utilising CRLF exploits; embedded client-side scripts such as JavaScript to exploit the 'document.cookie' DOM; and even XSS flaws within the URL handling. (22)

In 2008 the CVE (Common Vulnerabilities and Exposures) reported a flaw within Apple's Safari web browser (named CVE-2008-3170) (24), allowing websites to set country specific top level domains including '.co.uk' and '.com.au' (23). The successful exploitation of this vulnerability directed towards session fixation attacks.

Cross-Sub domain Cooking

Unlike Cross-Site Cooking, the malicious attacker originates from a sub domain hosted upon the root domains server example: 'badsite.goodsite.com'. This exploit relies on the ability of a sub-domain to set wildcard cookies, valid upon either the root domain or another sub-domain (11).

Detection and Prevention Techniques

From the analysed literature and the injection discussion, it is a common observation that insufficient validation and sanitisation standards are the prominent cause among the successful execution web injection vulnerabilities. This is most commonly due to negligence on the development side of the web application and can be related to a substantial lack of awareness regarding web injection vulnerabilities.

A number of techniques may be used when trying to secure web applications against the injection of malicious code. The three main forms of application detection techniques for web injections include filtering, validation and escaping. These methods aim to alert the application of unexpected entry through the validation methods, or manipulate the data into a format that can be safely executed by the application. These methods are insufficient prevention techniques when used on their own and most effective when used in conjunction with one another.

It is important to understand the theory of 'zero day attacks' when considering prevention techniques, even the most comprehensive combination of the techniques outlined will never guarantee the application is completely secure. Security measures can be taken against known injections attacks, whilst mechanisms must implemented to allow the addition of new detection and prevention techniques.

Validation

Validation procedures are used to check data against a defined, expected criterion; returning a Boolean value according to the success or failure of the operation. This may include checking the validity of the data's type, length or format. This validation process is only a detection method and does not manipulate the data.

For example, an application containing a registration form may wish to validate an input string against an email address criterion. This will include the correct format, inclusion of special characters '@' and '.' along with a valid host domain-name. When applied to injection attacks, data validation may be used against input strings, checking for scripting syntax or unexpected encodings. Upon finding a false entry, the validation method should raise an exception.

Although these methods provide feedback to the web application regarding the cause of the raised exception, application developers must be careful when implementing application output. As previously discussed with blind injection attacks, not only may this form of output provide detailed debugging information, it may also provide a true/false response through other attacks such as SQL timing attacks.
(25)

Filtering

Data filtering is a process of sanitisation through data normalisation by the transformation of data values. This process is achieved by altering and often removal of certain characters within the data. For example, if a required input is an integer and the data processed by the application contains non-integer characters, filtering procedures may remove extra values in order to try upholding the validity of the input, therefore this method can therefore be seen as an automatic correction mechanism. In context of injection attacks, filtering methods may remove special characters used within browser or client-side scripting for example <>/ characters used within HTML and XML tags.

Escaping

For some web applications the input of special characters may be desired, therefore they must undergo other forms of filtering processes in order to eliminate the risk of illegal execution. Instead of removing special characters, escaping introduces special escape values inserted within the data. These escape values remove the meaning of subsequent special character, treating the value as ordinary text when performing further operations on the data.

For example, an application allowing textual input will commonly allow for the input of commonly used characters such as quotations marks. When data is processed by filters, special characters are removed resulting in punctuation errors, whereas escaping this data will maintain the original data, removing the escaped values when output to the user.

Encoding

Character encoding is the process of how characters are understood by computer systems and involves substitution of each character by an encoded value. There are numerous standards of character encoding varying by the characters available. For example the use of special punctuation symbols, Diacritic marks and even Latin characters are possible with different types of encoding.

As discussed in previous sections, many web injection vulnerabilities use character encoding to bypass sanitisation and validation methods. When performing data filtration, all input data must be fully de-encoded to eliminate this risk of bypassing the filter. It is also necessary to include the possibility of multiple encodings when de-encoding data input. Encoding may also be used as a form of character escaping to avoid malicious code from being executed. (25)

Encryption

For information processing, encryption is the process whereby the input data is converted into a format that does not resemble the original data. Normally this methods is used for the safe transfer and storage of sensitive information, however this project proposes the use of encryption to reduce the risk of server-side web injection attacks, by removing the meaning of script syntax.

Encryption may be an appropriate solution to solve many forms of web injection attacks whist fulfilling other standard practice requirements when dealing with information security.

Tokenisation

When performing data filtering methods, it can often be difficult to process an exorbitant amount of information at once. Tokenisation is the process of breaking the input data stream into separate strings, allowing for an easier processing load on the data filter. However the process of tokenisation does produce a number of problems when extracting an individual word from an exceptionally long string. Additionally this function may prove error prone when scripting keywords only have meaning in a larger context.

Issues with filtering methods

While filtration methods are crucial towards the prevention of web injection attacks, their use can become complicated when implemented alongside a verity of complex web applications. Filtering requirements will differ not only by the purpose of web application, but also by each individual application interface. The application requirements may range from simple input forms with strict filtering methods, blog posts allowing for minimal coding such as html picture or hyperlinks, or the use of back-end content management systems allowing the arbitrary storage of code.

The most significant issue when implementing filtration methods is to develop an understanding of the applications filtering requirements, particularly if the filtration methods are called before the web applications business logic. Subsequently, appropriate methods should be put into place allowing for the customisation of those filtration methods on an input specific basis. Strict rules should be applied by default with the addition of overriding methods allowing for the adaption of filtration rules.

Existing Methods and Tools

Some web application development tools contain either built-in methods to help counter web injection attacks, or methods to assist when developing a filtration tool. This section looks at the most popular web development languages, their methods and how effective these tools are towards preventing web injection attacks. Some of these methods may not provide a direct influence upon preventing injection attacks; however they may present valued functionality when producing a detection or prevention solution.

Development languages

When producing web applications, there are an ample amount of development languages varying by their functionality; however two of the most renowned and widespread include PHP and Java. This section looks briefly at these languages and some of their methods which may be used when developing a solution to detect and prevent web injection attacks.

PHP

Hypertext Pre-processor: PHP; is the most widely-used server-side web development language. They claim to be currently implemented on more than 20million domains. This development language is popular mainly due to its easy inline integration within the applications HTML content, along with a large development community and extensive online documentation. At time of writing PHP v5.3 contains a number functions to help prevent web injection attacks. These functions consist of various methods allowing the user to specify input data, for example: strings, arrays and HTTP inputs such as GET, POST and COOKIE data. The input method then takes a second parameter of either a validation or sanitation method.

> Validation methods – The PHP validation methods shown within Appendix E allow for the validation of data types including Boolean, email, floating point number, integer, IP addresses, regular expressions and URL addresses (according to RFC 2396 specification). The

function then returns a Boolean true/false response according to whether the input string matches the validation type.

Sanitation methods – Appendix E shows the sanitation methods provided by PHP. These methods are invoked the same way as validation methods, although they return a sanitised value rather than the Boolean true/false response. These methods include the ability to sanitise an input string against a particular data type, for example: when sanitising against an integer, all non-integer characters are removed. Furthermore, these methods also include the ability to remove encode values and special characters.

Flags – The flag functions shown within Appendix E and F are declared along with the validation or sanitation method as a subsequent parameter. Their purpose is to add greater flexibility to the these methods, for example: FILTER_FLAG_STRIP_LOW can be used within the sanitise method FILTER_SANITIZE_STRING to remove all ASCII characters with a numerical value less than 32 (please refer to Appendix F), FILTER_FLAG_IPV6 is used within the validation method FILTER_VALIDATE_IP to specify the IP address must conform to the v6 standard. The accessibility of these flags varies according to the purpose of the validation or sanitisation method; consequently all flags are not accessible to every method.

(26)

Java

Java is one of the most popular and powerful programming languages, variations such as JSP (Java Servlet Programming) are similar in construction to PHP allowing server-side scripting within the HTML code. Java also contains useful built-in functions when implementing a solution to prevent web injection attacks.

Prepared Statements – Aimed at preventing SQLIA's, Java contains a built-in function called 'PreparedStatement'. This function uses SQL statements pre-defined by their internal function, while still allowing for the insertion of input variables. This function succeeds against SQLIA's by ensuring the entirety of the statements purpose and syntax, therefore the input variables are isolated from the SQL syntax and processed independently, thus do not interfere with the original SQL statement. (27)

doFilter – This built-in java function intercepts the users request before any information is processed by the business logic. This function is also aimed at allowing data manipulation including filtration functions, before the data is sent back to the user.

Validation – Within Java data can be validated by using the exception handler. A 'type' is explicitly defined when assigning an input value to a variable. If the input variable does not match the defined type, an exception is triggered which may be caught by the exception handler. However this method does not allow for the validation of more complex data types such as IP and email addresses; these types will be validated by using regular expressions.

(28)

Summary

Both PHP and Java contain many useful methods to assist with preventing web injection attacks. PHP offers the most comprehensive arrange of validation and sanitation functions allowing easy integration with the web application. Java contains built-in methods for data handling processes, however lacks the sanitation methods available within PHP. However Javas method for SQL handling provides an extensive solution against SQLIA's, not present within PHP. Both contain the ability to validate data against regular expression, allowing a flexible validation solution.

None of the languages discussed contain a complete solution against web injection attacks; all contain different methods providing different forms of protection. Furthermore all of the investigated methods are not initiated by default, therefore must be explicitly evoked by the developer. Throughout the project these inconsistencies must be taken into consideration when attempting to provide a web injection solution.

Development techniques and other methods

Although this project looks into ways of securing already established web applications, attention must be paid towards guiding web application at different stages of their development life cycle. This section outlines some recent techniques changing how web applications are developed, furthermore how these methods could help to protect applications against web injection attacks.

Frameworks

Over the last decade Frameworks have become increasingly popular when developing web based applications. They provide a platform for developing advanced, large web applications typically involving a team of developers. This includes a number of tools which may become beneficial to preventing web injection attacks.

- The **Model View Controller** paradigm is the separation of the applications business logic, data retrieval and view elements. The aim of this segmentation is to allow easier maintenance of the application as developers may work upon separate aspects of the application simultaneously. For example the back end business logic may be programmed by some developers whilst graphics designers may work on the application views, without causing conflict. This concept may prove useful when educating the right development team about the risks of injection attacks; furthermore this application separation may provide an essential feature when implementing new filtering and validation methods.

- **Bootstrapping** is a technique used within frameworks to perform multiple standard actions on each application request. Essentially all application requests are diverted to the bootstrapping file, performing a number of actions before loading the required application content within the MVC architecture. Bootstrapping may become one of the most important development techniques, allowing for the central application of a data sanitisation function, performed upon every application request before the business logic and data retrieval.

(29)

Libraries

Libraries such as PEAR provide web applications with pre-built object orientated classes, aimed at speeding up development time. This method of deployment will prove valuable when producing, issuing and updating a web injection prevention solution.

SQL - CRUD

Other techniques within the development architecture allow further prevention measures against attacks such as SQLIA's. CRUD (Create, Read, Update and Delete) is a term often given to an object querying the database; however the Update and Delete aspects provide unnecessary data access for normal application actions. Alternatively, the database can enter a new record for every operation, distinguishing through a time stamp and retrieving only the latest version. This process would eliminate the need the UPDATE and DELETE functions whilst maintaining the same functionality, furthermore application access rights can be changed to deny UPDATE and DELETE functions.

This method of eliminating some of the application database functions does create the issue of the storage system becoming bloated, due to a large number of stored records. To counteract this issue, a second user may be defined with full CRUD privileges. This user can be used within the application for processes that do not take user input. Furthermore to reduce the effects of the bloated database, old entries may be removed using the account with CRUD access. (29)

Validation based methods

As previously discussed, some of the injection techniques perform undisclosed form submissions on the user's behalf such as Cross-site Request Forgery attacks (30). There are a number of other techniques to validate the authenticity of input data and try to distinguish human input from non-human based input.

CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) have become a common standard when validating for human input. A number of random characters are displayed upon the web application, which the user must input as part of the form submission. However as attack methods have become more sophisticated to counteract this security measure, the complexity of this validation method has grown. To prevent attacks using tools such as Optical Character Recognition, CAPTCHAs have evolved to include distorted background images along with skewed characters both varying in size and font. Other forms of CAPTCHAs have also arisen compromising of advanced browser features such as sound and video output. (29)

Other forms of validation methods include the use of email to verify the user's entry point into the system. This often involves the use of URL links with embedded GET parameters containing random values. These values are stored on the application database and validated against the received parameters. Although special links provide some form of authentication, there use is often limited to actions such as password reset and account variation, due to the lengthily time involved with retrieving the email link.

Conclusions

The Background Research section of this report discussed a number of injection attacks, based on their use and severity. It provides a proof of concept towards the detection and prevention of these attacks. Due to an exhaustive amount of web injection vulnerabilities, only a sample of these attacks can be covered within this report. These detection and prevention methods mainly focused around string evaluation including filtering techniques such as sanitization and validation function. This section also looked at other prevention techniques surrounding the development environment including recommended development standards and practices. Although these methods may prove a time consuming and costly prevention against injection attacks for already deployed web application, they provide important guidance for new web application developments.

With the existence of many other web injection attacks, recommendations for further projects may include the security of individual client and server-side languages. The recent development of HTML5 could play a significant role within the future of web injection attacks, due to the greatly improved functionality of this language supporting more complex forms of digital media including sound, video and voice activation. Prevention techniques may also look into client side solutions and how they may interact with the information held on the server-side application.

Research Methods

The background research focuses on the most common web injection methods, the literature analysed and discussed conveys one primary cause of injection vulnerabilities - the people who use the IT system. Employees are often regarded as the biggest security risk, injection vulnerabilities ultimately occur from insufficient development standards and practices outlined by development team managers and employees. The risk associated with web injections can be significantly reduced when a formal security standard is applied to the development process.

This section discusses the objectives of the project, along with a plan for both the Technical Requirement Specification and the plug-in library.

Objectives of the project

Technical Requirement Specification

As mentioned, the most common flaw within web application security is the application development practices put in place by the development team. Generally, this is due to a lack of awareness towards web injection vulnerabilities including their risks, cost and impact of their successful execution.

The first phase of the project will involve the production of a Technical Guidelines Specification. This formal standard will be aimed at helping the organisation secure their web application against web injection vulnerabilities by informing technical specialist about web injection attacks. There are many advantages towards using standards when securing IT system; they often are a more widely understood form of policy from which multiple users may follow simultaneously, also as they become more widely used their effectiveness increases with refined versions incorporating feedback from real life instances. This will become one of the key goals towards the successful development of the Technical Requirement Specification – through user participation and accreditation by real working environments. It must also incorporate the organisations adoption strategy, with guidance on periodic reviews and achieving an effective return on investment.

The purpose of this standard is to educate an organisation about web injections attacks. This includes presenting the various types of web injection attack, the flaws they try to exploit and how they are successfully executed. Ultimately this specification endeavours to convey the risk associated with these attacks and the consequences of negligence towards securing their system against web injection vulnerability. This will include reference to other standard requirements and applicable legislation the organisation must adhere to. The consequences of these risks are so severe, the specification must be presented to all employees involved within the application development with personal responsibility applying to those in breach of these terms.

The detection and prevention of these attacks will become the definitive goal of this specification. As organisations may be at different levels of the application development process, these solutions must allow for easy integration into already established applications along with prevention techniques for new application starting the development life cycle. Guidance will be provided for each prevention solution along with implementation requirements. Prevention methods will also include standard practices and procedures for associated organisation employees regarding how they can develop and maintain the injection prevention system.

Upon completion of the Technical Guidelines Specification and following successful guideline appraisal, the specification will be published via a number of sources including the National Computing Centre, available for download.

The final dissertation report will aim to bridge the gap between the Technical Requirement Specification and a high enterprise level view. Ultimately this will include information regarding how an organisation can provide the best development strategy and achieve the most effective return on investment. Due to the severity of negligence towards securing these forms of attack, the organisation must seek an enterprise wide acceptance of the new practices and procedures.

Injection tool

Along with the Technical Requirement Specification, the project will include the development of a solution to show a proof of concept when trying to prevent against some of the most common web injections encompassing the most severe consequences.

The prevention solution will consist of a plug-in library to be deployed within the web application. The purpose of the library will be to handle the input of the applications data and perform a number of filtration processes. This will consist of inputs from all sources including, but not limited to: GET requests, POST requests, stored cookie data and sever session data.

Initially the data will pass through a decoding process, removing all hidden encoding techniques. Once decoded a number of sanitation processes will be applied to the data, detecting and removing injected coding. The input will be filtered before the execution of application data and business logic, thus providing an invisible abstraction layer. Some applications may allow for the input of special characters including coding data. For this occurrence the library should provide functions to override the sanitation process or invoke another sanitation method such as character escaping rather than removal of the entities. Although the application sanitises the input data, the already deployed applications may hold previously input data stored within a database or file; the solution must therefore provide some functionality to sanitise this data. This may include a batch processing method connecting to the database, discovering the database tables and fields before processing the sanitation methods upon each record.

The solution will require compatibility upon a number of web environments; therefore this may require the library to be coded using a number of scripting languages. As frameworks play an important role within today's web application development, integration should be considered with the major framework providers along with a standard version for those applications running without framework controls. A substantial focus will be paid towards integration, once implemented the library should allow some functionality without modification of the current web application. Ease of use is also a desirable trait aiming to achieve little to no training requirements. This will also be reflected by the libraries usability with no required understanding of its inner workings. Validation is another important functionality requirement of this solution; appropriate validation solutions should be provided which are consistent through the various platform implementations.

The library will be deployed on the project website (http://thewebjunkie.com) for download. One important requirement of this solution is the easy expansion to include new sanitation rules, validation procedures and user controls. The application will also be deployed using version control, allowing for developers to contribute towards the project.

Deliverables

The Technical Requirement Specification, tested by a number of contributors and deployed for download by a number sources including the National Computing Centre and the project website (thewebjunkie.com)

The plug-in scripting library will consists of a number of library implementations using a range of the most popular web development languages. This will also include a number of formats for framework implementations and standalone versions. For contributors to the project, the library will be controlled by sub versioning and deployed along with adequate test files. Both implementations will require some form of documentation. For the standard user version this will consist of installation procedures along with details of how to control the advanced features of the package. For contributors this will require extensive documentation on coding standards and testing procedures.

The project website will be the primary source for all application and requirement specification downloads. This will also include a blog to assist with all areas of development. Pages will provide the primary source of documentation for both users and developers, news articles will contain information regarding the current version and development news, for develops this will also provide information on required updates and the voting of new system features.

Development methodologies

Design & Implementation

Development will be carried out simultaneously between the Technical Requirement Specification and the prevention library. A modular, process orientated approach will be taken when developing the library. This methodology is commonly used within projects where entities of the organisation develop separate parts of the system, thus will be ideal when developing a number of independent decoding, sanitation and validation solutions to counter a wide range of injection attacks. Furthermore the use of a modular implementation may be carried past initial development to release further updates, another critical aspect when developing this solution.

The resources required for this section of the development life cycle include a web development playground. Due to the nature of this research, development will be carried out upon a safe local environment to avoid the risk of damage to live web applications if hosted upon a live architecture. The web server will consist of a local install of Apache with the required programming library extensions including PHP and Tomcat (for running Java applications). A database system is also required to develop against SQLIA; this will consist of a MySQL server along with a database control panel such as ‘PHPMyAdmin’.

Testing

The core focus of project testing will be through penetration testing in order to determine the success or failure of the prevention library. This testing strategy will include initial development of a test web application, used as a testing platform when researching web injection techniques. This application will be tested against a pre-defined test plan. The same testing strategy will then be applied to the application after the integration of the prevention library. This may also include a number of test sub-iterations where more advanced features requiring application manipulation are activated. Furthermore this testing strategy will also include the following of other prevention practices and procedures outlined as part of the Technical Requirement Specification. Testing may include the use of purpose built web injection penetration tools like discussed within (25) to provide a more rigorous test solution. All testing procedures will be performed upon a secure local environment to avoid the potential risk of exposing other applications to attack.

Required resources for the testing phase within the development life cycle will include the local installation of the web development platform as noted from the Design & Implementation section. Tools specifically designed for injection simulation may be used throughout this phase, consequently using third party applications carries risk toward their authenticity, precaution's must therefore be taken to ensure the testing strategy is carried out upon a safe closed environment.

Evaluation & appraisal

The evaluation of the Technical Requirement Specification will include external appraisal by the National Computing Centre to determine whether the specification is compliant and appropriate for external release. The prevention library will be evaluated based upon the results of the testing strategy; however the project will try to seek evaluation by external entities and trials within live application environments.

Project plan

This report outlines a substantial amount of research regarding web injection attacks, detection and prevention methods. Further to this report, the development of a test application will be used to try and execute these attacks, in order to gain further understanding of how these attacks are performed. This test web application, vulnerable to injection attacks will be used as a benchmark against all development and testing procedures. Further to the benchmark web application, the production of a test plan will allow comparisons to be taken against the various solutions throughout the project.

Firstly, existing solutions including native programming controls for both PHP and Java development languages will be examined, implemented and tested against the benchmark web application using the pre-defined test plan. Once these results have been analysed and compared, work will begin on producing a prevention library to accompany these techniques.

Initially work will be carried out upon the background of the Technical Requirement Specification, outlining the various injection attacks. Once completed the remainder of the Technical Requirement Specification and the prevention library will be developed simultaneously. Implementing and testing each feature with the benchmark web application and test plan. Following the successful development of these techniques, the Technical Requirement Specification will document their findings.

Upon completion of both project deliverables and after approval, both deliverables will be released for external use and appraisal. During this phase the deployment entity must be configured to enable items to be downloaded. Documentation will be completed before applying version control to the prevention library. A blog will be implemented upon the deployment agent to allow for project discussion and information regarding the project and its solutions.

Following completion of the development phase, write-up will commence for the final dissertation report, aimed to document the entire project. Initial writing will include an overview of the project, an outline of the scope and context of the investigation and why this project is important. The project background section will include a literature survey containing information on injection research carried throughout the project, along with existing solutions including the initial testing against the benchmark web application. The development of the prevention library and Technical Requirement Standard will be documented, discussing how the project arrived at the final solution. Testing will include a discussion upon the testing procedure and how the prevention library performed against the test plan. Conclusions will be drawn from the project and testing phase to

determine the success or failure of the project, including recommendations on future work and possible project expansion.

For a time breakdown of the project plan, please refer to Appendix G. Please note, the exact requirements for the project dissertation report have not yet been discussed, therefore the information presented is a rough outline of the sections to be covered.

Bibliography

1. *Defending Against Injection Attacks Through Context-Sensitive String Evaluation.* **Pietraszek, Tadeusz and Berghe, Chris.** 2006.
2. **McAfee Labs.** *Threats Report Q4.* s.l. : McAfee, 2010.
3. *Web Injection Attacks.* **Morgan, David.** 2006.
4. **W3C.** Objects, Images, and Applets. *W3C.* [Online] http://www.w3.org/TR/html401/struct/objects.html.
5. *Identifying Cross Site Scripting Vulnerabilities in Web Applications.* **Di Lucca, G A, et al., et al.** 2005.
6. *XSSDS: server-side detection of cross site scripting attacks.* **Johns, M, Engelmann, B and Posegga, J.** 2008.
7. *Optimized Client Side Solutions for Cross Site Scripting.* **Tiwari, S, Bansal, R and Bansal, D.** 2008.
8. **Kirk, Jeremy.** My Space Worm. *Computer World.* [Online] December 2006. http://www.computerworld.com/s/article/9005607/MySpace_worm_uses_QuickTime_for_exploit.
9. **The Open Web Application Security Project.** Cross Site Request Forgery. *OWASP.* [Online] 7 April 2009. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).
10. *Client-Side Detection of Cross-Site Request Forgery Attacks.* **Shahriar, H and Zulkernine, M.** 2010.
11. *Web Security - The University of Vienna.* **Platzer, C, et al., et al.**
12. *Evaluation of SQL Injection Detection and Prevention Techniques.* **Tajpour, A and Shooshtari, M.** 2010.
13. *SQL Injection Detection and Prevention Tool Assessment; 2010.* **Tajpour, A, et al., et al.** 2010.
14. **PHP.** PHP: mysql_query - Manual. *PHP.* [Online] http://php.net/manual/en/function.mysql-query.php.
15. *Presentation on: HTTP Message Splitting, Smuggling and Other Animals.* **Klein, A.** 2006.
16. **The Open Web Application Security Project.** Cache Poisoning. *OWASP.* [Online] 23 April 2009. https://www.owasp.org/index.php/Cache_Poisoning.
17. *HTTP Response Splitting.* **Klein, A.** 2004.
18. **Acunetix Web Application Secuirty.** *acunetix.* [Online] 4 May 2010. http://www.acunetix.com/blog/web-security-zone/articles/crlf-injection-http-response-splitting/.
19. **The Open Web Application Security Project.** HTTP Response Splitting. *OWASP.* [Online] 7 April 2009. https://www.owasp.org/index.php/HTTP_Response_Splitting.
20. *HTTP Request Smuggling.* **Linhart, C, et al., et al.** 2005.
21. **The Open Web Application Security Project.** HTTP Request Smuggling. *OWASP.* [Online] 7 April 2009. https://www.owasp.org/index.php/HTTP_Request_Smuggling.
22. *Session Fixation Vulnerability in Web-Based Applications.* **Kolsek, M.** 2002.
23. **Claburn, Thomas.** Apple's Safari Vulnerable To 'Cross-Site Cooking. *Information Week.* [Online] 29 July 2008. http://www.informationweek.com/news/internet/browsers/209800452.
24. **Common Vulnerabilities and Exposures.** CVE-2008-3170. *CVE.* [Online] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3170.
25. **Veracode.** *Eradicate Cross-Site Scripting.* 2011.
26. **PHP.** Data Filtering. *PHP.* [Online] April 2011. http://www.php.net/manual/en/book.filter.php.
27. **Oracle.** PreparedStatement (Java 2 Platform SE v1.4.2). *Oracle | Hardware and Software, Engineered to Work Together.* [Online] 2010. http://download.oracle.com/javase/1.4.2/docs/api/java/sql/PreparedStatement.html.
28. **Schildt, Herbert.** *Java, The Complete Reference.* s.l. : Osborne, 2007.
29. **Vaswani, V.** *Zend Framework, A Beginner's Guide.* s.l. : McGraw-Hill, 2010.
30. *Cross-site Request Forgery: Attack and Defense.* **Alexenko, T, et al., et al.** 2010.
31. **LookupTables.com.** ASCII. *LookupTables.com.* [Online] 2010. http://www.asciitable.com/.

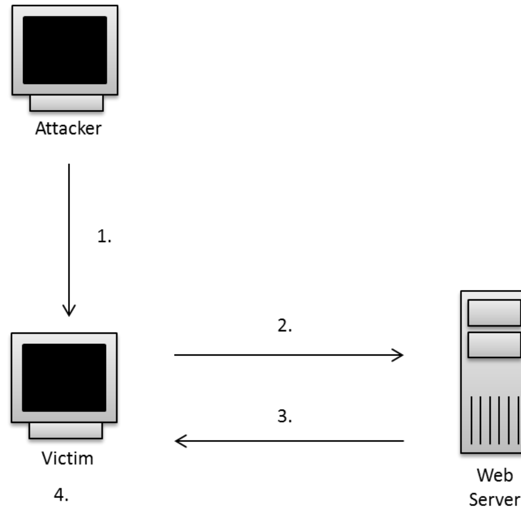
Appendix

Glossary of Terms

Term	Description
Namespace	A conceptual space to avoid conflicts with already defined elements containing the same names.
HTTP	Hyper Text Transfer Protocol – the protocol used to for communication between the browser and the web server
GET, POST	Requests sent within the HTTP header, these requests are generated from form submissions or contained within URL addresses
Turing complete	A term given to programming languages that can perform calculations.
DOM	The Document Object Model is a model for representing and interacting with objects within HTML and XML documents
SQL	Structured Query Language - Language used to query the database
XML HTTP Request	A DOM API used to send and receive HTTP requests directly from a browser-based scripting language e.g. JavaScript
iFrame	A HTML frame which is contained within the web application and may be hidden from view.
PHP	PHP: Hypertext Pre-processor (recursive acronym) - A server-side web language used for business logic and database connections
MySQL	A form of SQL language
ASCII	American Standard Code for Information Interchange - A common encoding format used within web applications
Apache	Apache is a type of HTTP web server used to communicate with the browser

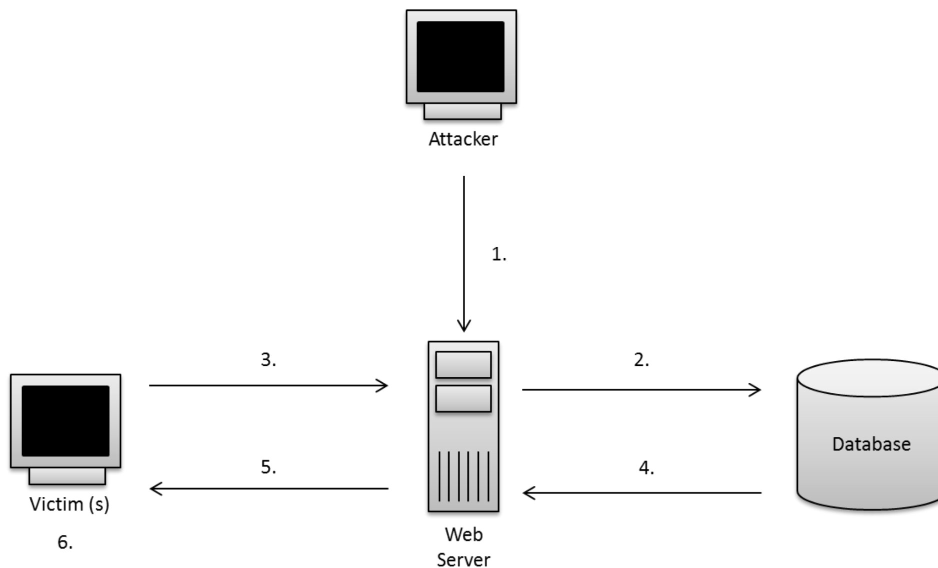
Cross-Site Scripting

Non-Persistent/Reflected Example:



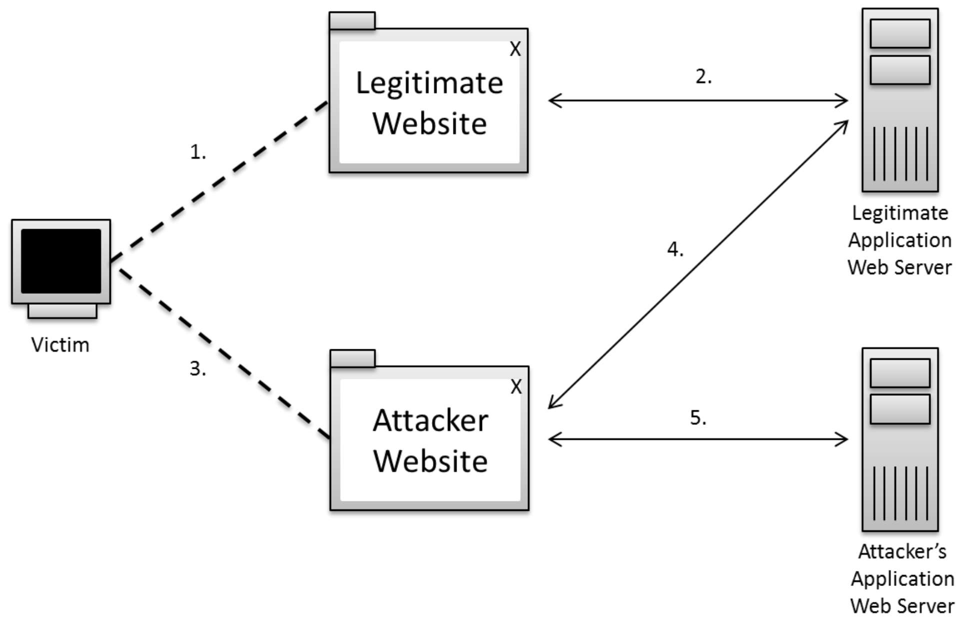
1. Attacker sends a legitimate URL to the victim containing an embedded malicious script contained with GET parameters.
2. The victim requests the URL page from the legitimate web application passing the malicious code.
3. The server returns a response, reflecting the malicious script without filtering the content.
4. The browser executes the malicious script.

Persistent Example:



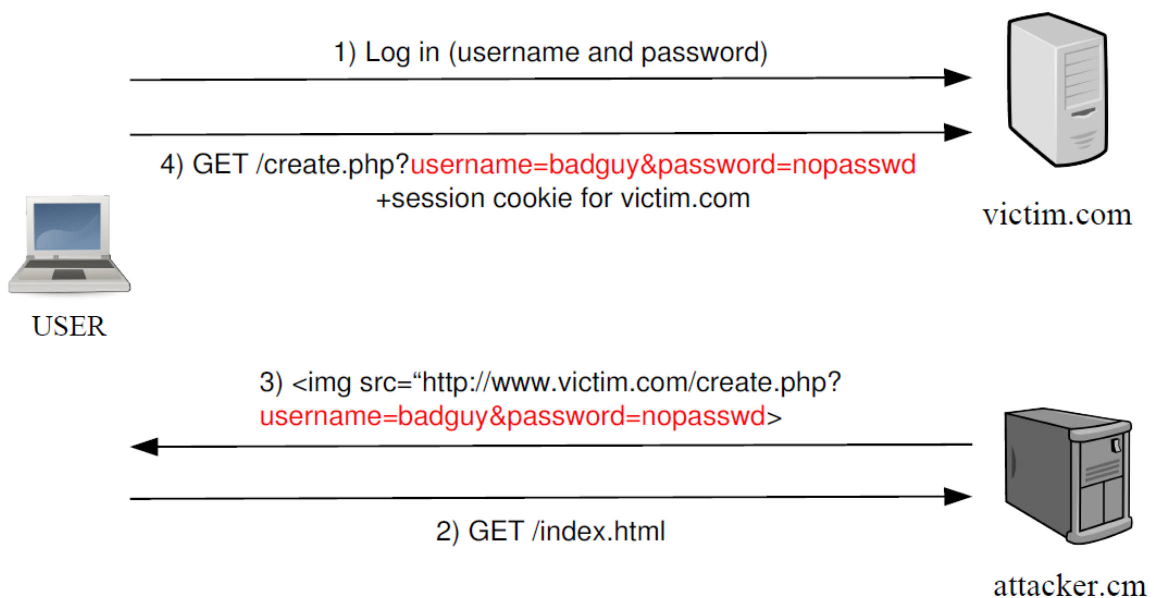
1. The attacker sends the malicious script to the web application
2. The web application stores this malicious script, typically within a database.
3. A user requests a page from the web application
4. The server fetches content from the database, in this case containing the malicious script previously inserted by the attacker.
5. The web application returns the page to the user containing the un-escaped malicious script.
6. When received the users browser executed the malicious script, perceived by the web browser to be legitimate content sent by the web application.

Cross-Site request forgery



1. Victim navigates web browser to a legitimate web application
2. The victim authenticates with the legitimate web server, this will include some form of login process.
3. The victim navigates to the attackers web application
4. Still authenticated with the legitimate web application, the attacker's application secretly sends requests to the legitimate applications web server, in return the response is passed back to the web browser.
5. In some cases this response may include information which is then passed onto the attacker's application web server.

Cross-Site request forgery Example:



PHP Sanitation methods

Filter

ID	Description
FILTER_VALIDATE_BOOLEAN	Returns TRUE for "1", "true", "on" and "yes". Returns FALSE otherwise. If FILTER_NULL_ON_FAILURE is set, FALSE is returned only for "0", "false", "off", "no", and "", and NULL is returned for all non-Boolean values.
FILTER_VALIDATE_EMAIL	Validates value as e-mail.
FILTER_VALIDATE_FLOAT	Validates value as float.
FILTER_VALIDATE_INT	Validates value as integer, optionally from the specified range.
FILTER_VALIDATE_IP	Validates value as IP address, optionally only IPv4 or IPv6 or not from private or reserved ranges.
FILTER_VALIDATE_REGEXP	Validates value against regexp, a Perl-compatible regular expression.
FILTER_VALIDATE_URL	Validates value as URL (according to » http://www.faqs.org/rfcs/rfc2396), optionally with required components. Note that the function will only find ASCII URLs to be valid; internationalized domain names (containing non-ASCII characters) will fail.

(26)

Sanitisation

ID	Description
FILTER_SANITIZE_EMAIL	Remove all characters except letters, digits and !#$%&'*+-/=?^_`{|}~@.[].
FILTER_SANITIZE_ENCODED	URL-encode string, optionally strip or encode special characters.
FILTER_SANITIZE_MAGIC_QUOTES	Apply addslashes().
FILTER_SANITIZE_NUMBER_FLOAT	Remove all characters except digits, +- and optionally .,
FILTER_SANITIZE_NUMBER_INT	Remove all characters except digits, plus and minus sign.
FILTER_SANITIZE_SPECIAL_CHARS	HTML-escape '"<>& and characters with ASCII value less than 32, optionally strip or encode other special characters.
FILTER_SANITIZE_STRING	Strip tags, optionally strip or encode special characters.
FILTER_SANITIZE_STRIPPED	Alias of "string" filter.
FILTER_SANITIZE_URL	Remove all characters except letters, digits and $-_.+!*'(),{}|\\^~[]`<>#%";/?:@&=.
FILTER_UNSAFE_RAW	Do nothing, optionally strip or encode special characters.

(26)

Flags

ID	Description
FILTER_FLAG_STRIP_LOW	Strips characters that has a numerical value <32.
FILTER_FLAG_STRIP_HIGH	Strips characters that has a numerical value >127.
FILTER_FLAG_ALLOW_FRACTION	Allows a period (.) as a fractional separator in numbers.
FILTER_FLAG_ALLOW_THOUSAND	Allows a comma (,) as a thousand separator in numbers.

FILTER_FLAG_ALLOW_SCIENTIFIC	Allows an *e* or *E* for scientific notation in numbers.
FILTER_FLAG_NO_ENCODE_QUOTES	If this flag is present, single (') and double (") quotes will not be encoded.
FILTER_FLAG_ENCODE_LOW	Encodes all characters with a numerical value <32.
FILTER_FLAG_ENCODE_HIGH	Encodes all characters with a numerical value >127.
FILTER_FLAG_ENCODE_AMP	Encodes ampersands (*&*).
FILTER_NULL_ON_FAILURE	Returns **NULL** for unrecognized Boolean values.
FILTER_FLAG_ALLOW_OCTAL	Regards inputs starting with a zero (*0*) as octal numbers. This only allows the succeeding digits to be *0-7*.
FILTER_FLAG_ALLOW_HEX	Regards inputs starting with *0x* or *0X* as hexadecimal numbers. This only allows succeeding characters to be *a-fA-F0-9*.
FILTER_FLAG_IPV4	Allows the IP address to be in IPv4 format.
FILTER_FLAG_IPV6	Allows the IP address to be in IPv6 format.
FILTER_FLAG_NO_PRIV_RANGE	Fails validation for the following private IPv4 ranges: *10.0.0.0/8*, *172.16.0.0/12* and *192.168.0.0/16*. Fails validation for the IPv6 addresses starting with *FD* or *FC*.
FILTER_FLAG_NO_RES_RANGE	Fails validation for the following reserved IPv4 ranges: *0.0.0.0/8*, *169.254.0.0/16*, *192.0.2.0/24* and *224.0.0 .0/4*. This flag does not apply to IPv6 addresses.
FILTER_FLAG_PATH_REQUIRED	Requires the URL to contain a path part.
FILTER_FLAG_QUERY_REQUIRED	Requires the URL to contain a query string.

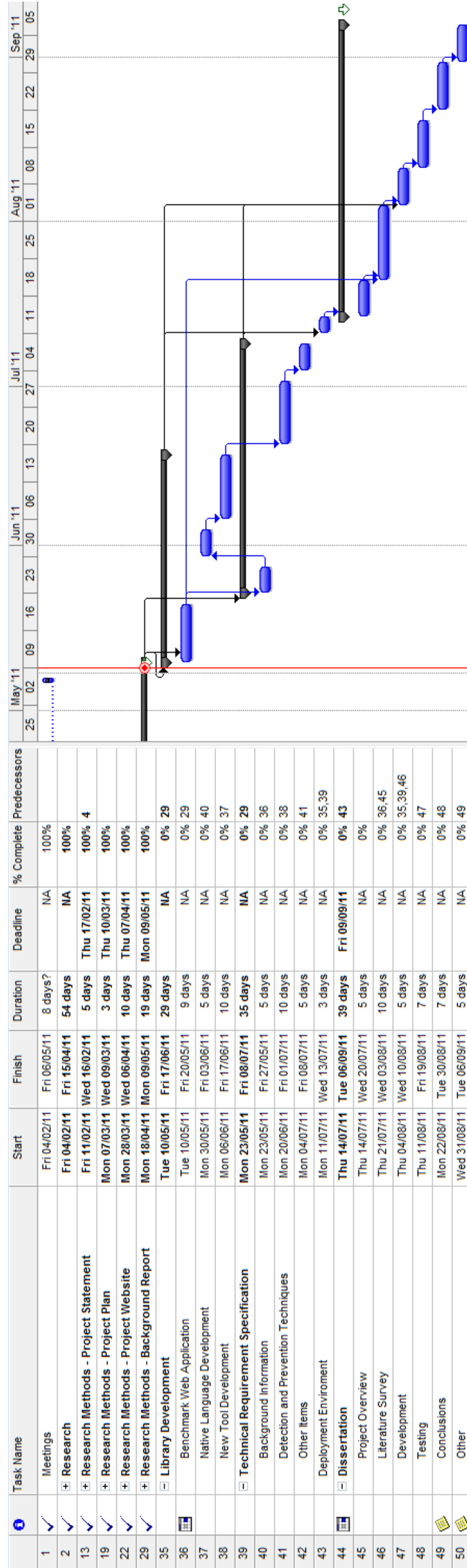
(26)

Flag – ASCII filter

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

(31)

Project Plan



ID	Task Name	Start	Finish	Duration	Deadline	% Complete	Predecessors
1	Meetings	Fri 04/02/11	Fri 06/05/11	8 days?	NA	100%	
2	Research	Fri 04/02/11	Fri 15/04/11	54 days	NA	100%	
13	Research Methods - Project Statement	Fri 11/02/11	Wed 16/02/11	5 days	Thu 17/02/11	100%	4
19	Research Methods - Project Plan	Mon 07/03/11	Wed 09/03/11	3 days	Thu 10/03/11	100%	
22	Research Methods - Project Website	Mon 28/03/11	Wed 06/04/11	10 days	Thu 07/04/11	100%	
29	Research Methods - Background Report	Mon 18/04/11	Mon 09/05/11	19 days	Mon 09/05/11	100%	
35	Library Development	Tue 10/05/11	Fri 17/06/11	29 days	NA	0%	29
36	Benchmark Web Application	Tue 10/05/11	Fri 20/05/11	9 days	NA	0%	29
37	Native Language Development	Mon 30/05/11	Fri 03/06/11	5 days	NA	0%	40
38	New Tool Development	Mon 06/06/11	Fri 17/06/11	10 days	NA	0%	37
39	Technical Requirement Specification	Mon 23/05/11	Fri 08/07/11	35 days	NA	0%	29
40	Background Information	Mon 23/05/11	Fri 27/05/11	5 days	NA	0%	36
41	Detection and Prevention Techniques	Mon 20/06/11	Fri 01/07/11	10 days	NA	0%	38
42	Other Items	Mon 04/07/11	Fri 08/07/11	5 days	NA	0%	41
43	Deployment Enviroment	Mon 11/07/11	Wed 13/07/11	3 days	NA	0%	35,39
44	Dissertation	Thu 14/07/11	Tue 06/09/11	39 days	Fri 09/09/11	0%	43
45	Project Overview	Thu 14/07/11	Wed 20/07/11	5 days	NA	0%	
46	Literature Survey	Thu 21/07/11	Wed 03/08/11	10 days	NA	0%	36,45
47	Development	Thu 04/08/11	Wed 10/08/11	5 days	NA	0%	35,39,46
48	Testing	Thu 11/08/11	Fri 19/08/11	7 days	NA	0%	47
49	Conclusions	Mon 22/08/11	Tue 30/08/11	7 days	NA	0%	48
50	Other	Wed 31/08/11	Tue 06/09/11	5 days	NA	0%	49