**Secure Web Application Development**

**M2M Connect; SMS => PHP Processing**

**Vinay Tailor, Mykolas Kaminskas and Matthew Baber**

Contents

[Introduction 2](#_Toc534661865)

[Security Techniques 2](#_Toc534661866)

[Specification 4](#_Toc534661867)

[General Usage Specification 4](#_Toc534661868)

[Security/Authentication Specification 4](#_Toc534661869)

[Implementation 5](#_Toc534661870)

[Testing 5](#_Toc534661871)

[Testing Strategy 5](#_Toc534661872)

[Testing Plan 6](#_Toc534661873)

[Testing Results 6](#_Toc534661874)

[Analysis and Recommendations 6](#_Toc534661875)

[User Guide 6](#_Toc534661876)

[Bibliography 6](#_Toc534661877)

# Introduction

For this assignment with have been tasked with creating a web application in PHP that will receive and securely download SMS messages, more commonly known as text messages, from a M2M server. Once the message has been downloaded, it will need to be parsed by the web application, and any content contains validated. Finally, users should be able to log into the web application, or register a new account if they do not have one, and view the message’s metadata, which will contain info about its source, and the message’s content, which will contain information from a circuit board being used to send these messages.

# Security Techniques

When building a web application such as this one it is imperative to consider the security techniques and aspects that will be used to deter any users for using it for their own malicious purposes, as well as to preclude any attempts to do so. To this end, it has been deemed necessary to investigate what is considered the 3 most potent threats to the security of a web application that has been created in PHP, as well as techniques that can be implemented to reduce the chance of these threats being utilised and even remove them entirely.

The first, and debatably the most potent, of these potential threats is something called SQL Injection (David Shirey, 2012), which has been used to gain access to the servers of multiple major organisations such as the Wall Street Journal in 2014 (Jeremy Kirk, 2014) and even the World Trade Organisation in 2015 (Pierluigi Paganini, 2015). An SQL Injection attack occurs when the user enters SQL commands, such as the “SELECT” command, into an input box on the web application in an attempt to manipulate the database that is associated with the web application to give themselves access to it, bypassing any security and authentication that may have been put in place on the database. Naturally, the user gaining unauthorised access to the web application’s associated database via SQL injection is exceptionally bad and is definitely something that will want to be avoided.

There are a number of security techniques that can be utilised in the prevention of SQL injection attacks, each with varying degrees of difficulty and effectiveness. However, it is widely agreed that the most effective countermeasure to these attacks is the usage of a “prepare” statement. This is due to the fact that using a prepare statement will separate out the query that is being sent to the SQL server, and the data that the user is attempting to enter into the server. This will ensure that, if the user starts entering SQL commands into the input box, these commands won’t be sent to the server as a part of the query and thus executed. Instead, they will be sent to the server as the data that will be used by the query which will have already been sent to the server, meaning any commands the user has entered will not be carried out.

The second of the potential threats that could be utilised against the web application for malicious purposes is known as Cross Site Scripting (XXS) (WordFence, 2018), a famous, albeit harmless example of this being the Samy worm that was used on Myspace in 2005 (Samy, 2005). Cross site scripting works in a similarly to the aforementioned SQL Injection where the user injects their own malicious code into the web application, usually in the form of JavaScript. However, unlike SQL injection attacks, these attacks affect the web application’s users rather than the web application itself as the malicious script will be run every time they open the web page. How this occurs depends on the type of Cross Site Scripting. A reflected XSS attack occurs when the malicious code is “reflected” off the server being used to host the application and is sent to the user as part of a request, for example, the user opening a malicious link in a phishing email (OWASP, 2018). Whereas, a stored XSS attack occurs when the malicious code is stored in the web server itself, usually in a database, to be sent out to any users that attempt to access the web application (Margaret Rouse, 2018).

There are 3 main techniques and methods that can be employed when attempting to prevent cross site scripting: Escaping or Validating the User Input and Sanitization (Sarah Vonnegut, 2017). The first of these techniques, escaping, involves replacing the potentially harmful characters, for example <, with their equivalent in ASCII characters to prevent them being inserted into the HTML or JavaScript of the application for malicious purposes (Richard Ishida, 2010). The second technique, validating user input, is essentially filtering out any characters that could potentially be used for malicious purposes from a user’s input. For example, filtering out the characters < and > from a text box used to submit a user’s name as those characters are not used within first names or last names and could potentially be used in malicious script that the user may be attempting to inject into the web application. Naturally, these filters will need to be context sensitive and allow the usage of certain characters in certain scenarios, eg the @ character for an email address. The final technique, sanitization, is rather similar to the second technique but with one key difference. Instead of simply filtering out the potentially malicious characters, sanitization “corrects” the user’s input by automatically removing or changing any of these characters (oracle, No Date Given).

The final threat of what is considered the 3 most severe threats to a web application’s security is called session hijacking (David Shirey, 2012). As the threat’s name implies, session hijacking involves taking control of a user’s session through their session ID and then pretending to be said user, allowing all sorts of malicious things to be done to the user’s account or the web application itself if they turn out to be an administrator (Margaret Rouse, 2006). There are 2 main ways in which the session ID of a user can be obtained for a session hijack: session sniffing and cross-site scripting. Cross site scripting is simply using the previous security threat to access the cookie that contains the session ID of the user, thus allowing the session to be hijacked. Naturally this is particularly worrisome if the session ID is being obtained through stored XSS as that will enable the hijacking of every session that is using the server. The other way the ID can be obtained is by using something called a sniffer to search for sessions that are connected to the web server, then attempting to “capture” the session ID before it can be sent to the web server (OWASP, 2014).

The most effective way to prevent session hacking is to prevent it before it can occur, that is, prevent the hacker from accessing the session ids of the web application’s users. Naturally, by protecting the web application from cross site scripting using the previously mentioned methods, you will also be protecting its users from one of the methods used to obtain their session ids. As such, the only other way the user’s session ID could potentially be obtained is through the usage of a sniffer. However, this can also be quite easily prevented by simply encrypting the cookie and the session ID that it contains. This means that, even if the cookie was captured by someone before it reached the web server, said person would not be able to use it as they would not have the decryption key required to decrypt the encryption on the cookie and thus view the session id (Mike Chapple, 2009).

While each of these techniques are, by themselves, not particularly effective at preventing all of the threats to the web application’s security, when used in conjunction with one another, they prove to be an effective prevention method and deterrent for anyone that would potentially make and malicious changes to the web application.

# Specification

Before designing and implementing a system or application, it is a good idea to produce a specification for it as this will help give us, the developers, and idea of what is required by the system and thus how we should design it. Below is a general specification for the web application we will be producing using PHP.

## General Usage Specification

* The web application must be able to connect to the M2M server
* The web application must be able to download the SMS messages from the M2M server
* The web application must parse (breakdown) these messages
* The web application must store the message’s metadata (message info eg the time sent) in its database
* The web application must store the message’s content in its database
* The user must be able to create and account on the web application
* The user must be able to log into their account on the web application
* The user must be able to view the stored message metadata
* The user must be able to view the stored message contents

## Security/Authentication Specification

* The session ID of the web application’s user must be protected using encryption, as must the connection to the server the messages are being downloaded from
* The web application must be protected from SQL Injection attacks through the usage of techniques such as prepare statements
* The web application must be protected from XSS through the use of techniques such as sanitization
* The web application must be validated by W3C validation

# Implementation

As specified in the assignment’s specification, the web application will be built using the PHP language with the utilization of the “SLIM framework” to assist in the web application’s creation. The web application will also utilise the “Twig template engine” to help build the HTML web pages that its users will be viewing by, as the name suggest, creating premade HTML templates for the web app to use when it creates the web pages for the user to view. Both this framework and the template engine will be used in a MVC (Model, View and Controller) format with the SLIM framework acting as the controller for the web application.

# Testing

## Testing Strategy

As the testing of the system is a key part of the project, to ensure that the web application is functioning and secure, it is important to organise it through the use of a testing scope to define key aspects of said testing. As agreed by the team during our first meeting after being given the project, Matthew will be responsible for the testing of the application and thus will be creating the plans for the testing, as well as doing the actual testing itself. The testing itself will mostly be white-box testing, which is the testing of the code used within the application rather than the application as a whole, as is required by the project’s specification, and will particularly focus on the security aspects of the web application and the prevention of the threats that were mentioned in the security techniques portion of the document. The testing itself will not take long, a day at the most, as most of the tests will be relatively simple and should be very short provided that nothing goes wrong or there are no errors with the coding, which is what the will be tested anyway.

The approach that will be used in the testing of the application will be a relatively simplistic one, as not to overcomplicate the process and make it easier to reproduce should future testing be required. The approach itself will simply involve the creation of a testing plan with written instructions on how to carry out each of the tests on the application, for future ease of use. Once a test has been conducted, its results will be recorded in a table and conclusions drawn once all the tests have been completed. The testing of the application will mainly focus on its security, looking at how it prevents some of the aforementioned threats to its security, but will also look at the application’s functionality and investigate whether it has met the requirements that were specified for it there.

The environment of the web application’s testing is an important thing to take into account as different environments can have different effects on the application, due to how the different programs being used to host the application are coded. That said, during the its testing, the application will most likely be hosted using a program called UwAmp, which acts as a WAMP server and thus allows the tester (Matthew) to host the application wherever he is. The application itself will be viewed using a web browser such a Mozilla Firefox.

As the testing approach is so simple, not many tools will be required during the testing of the application, and the tools that are required are relatively simple to use, making the testing easy to replicate. As previously mention, UwAmp will be utilised as a WAMP server to host the web application, and Mozilla Firefox will be used to view it. The testing plans and results of said test will be record in a word document, most likely this one.

As the testing will be occurring in an enclosed environment, there is not really any notable risk to the testing procedure. The only potential risk would have been corruption of the files used by the application, however this is complete nullified by the fact that the group is using github to work on the project and thus a backup version is always available in case this does occur.

## Testing Plan

M2M connection test

SMS download, parse and storage test

Account creation test

Login test

Message viewing (data retrieval) test

Viewing session id test

XSS test (?)

SQL Injection test

W3C validation test

## Testing Results

# Analysis and Recommendations

# User Guide

# Bibliography

<https://www.sitepoint.com/top-10-php-security-vulnerabilities/>

<https://www.pcworld.com/article/2457240/sql-injection-flaw-in-wall-street-journal-database-led-to-breach.html>

<http://securityaffairs.co/wordpress/36528/hacking/anonymous-breached-wto-db.html>

<https://www.wordfence.com/learn/understanding-php-vulnerabilities/>

<https://samy.pl/myspace/>

<https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)>

<https://searchsecurity.techtarget.com/definition/cross-site-scripting>

<https://www.checkmarx.com/2017/10/09/3-ways-prevent-xss/>

<https://www.w3.org/International/questions/qa-escapes>

<https://download.oracle.com/oll/tutorials/SQLInjection/html/lesson1/les01_tm_ovw3.htm>

<https://searchsoftwarequality.techtarget.com/definition/session-hijacking>

<https://www.owasp.org/index.php/Session_hijacking_attack>

<https://searchsecurity.techtarget.com/answer/How-to-prevent-network-sniffing-and-eavesdropping>