Securing web applications
with secure coding practices and integrity verification

by

Arafa Anis

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
March 2018

Copyright © Arafa Anis, 2018
Abstract

The concept of security in web applications is not new. However, it is often ignored in the development stages of the web applications. Being multitiered and spread across different domains, it is challenging to come up with a security solution that works for all web applications. Moreover, developers are more inclined to produce working features for the applications in the rapid development approaches that are chosen, than providing security for the code and often do not practice secure coding. Therefore, countless web applications are launched with security vulnerabilities which manifest later in their life cycle. Integrating security features should be part of the development process for these web applications to prevent unwanted attacks. Along with cross-site scripting, injection attacks and resource alterations; code tampering on the client side is a serious security risk for web applications. Without practicing secure coding and having an integrity verification system in place, it is difficult to defend these unwanted attacks. We present a system that integrates security measures on the client side code based on the best practices of secure coding and an integrity verification system. The proposed approach can be integrated with both new and existing web applications to provide security against prevalent attacks and make the client side tamper resistant. We implement our approach for JavaScript-based applications for which the security flaws are the most common.
Acknowledgments

I would like to thank my supervisor, Dr. Mohammad Zulkernine, for his guidance and support during my graduate career. I am forever in his debt for affording me this opportunity. I would also like to thank my thesis committee members, for their comments and guidance. I want to thank my lab mates, for their continued source of inspiration. Additionally, I would like to thank my good friends Carrick I., Ian M. and Tania N. for their unwavering support, which has been of great importance to me. I would also like to thank my parents and my siblings for their unyielding love and support throughout the last two years.

Moreover, I would also like to express my deepest gratitude and appreciation to Clifford Liem for his guidance and mentorship both at Irdeto during my internship and afterwards. Finally, I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) and Mitacs for their financial support for this thesis work.
Statement of Originality

I, hereby certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

(Arafa Anis)
(March 2018)
# Contents

Abstract i  
Acknowledgments ii  
Statement of Originality iii  
Contents iv  
List of Tables vi  
List of Figures vii  

## Chapter 1: Introduction  
1.1 Motivation .................................................. 2  
1.2 Overview ..................................................... 3  
1.3 Contribution .................................................. 5  
1.4 Organization of the Thesis ................................... 6  

## Chapter 2: Background and Related Work  
2.1 Introduction to Web Applications ............................. 7  
2.2 Web Application Security ...................................... 9  
2.2.1 Web Application Attacks .................................. 10  
2.3 Client Side Security .......................................... 13  
2.4 JavaScript Latency In Browsers ............................... 16  
2.5 Related Work .................................................. 17  
2.5.1 Existing Problems in Purely Server Side Mechanisms ...... 18  
2.5.2 Existing Web Application Security Mechanisms .......... 18  
2.5.3 Existing Solutions for Solving JavaScript Latency ......... 23  
2.6 Summary ...................................................... 24  

## Chapter 3: Secure Coding Practices and Integrity Verification  
3.1 Overview and Design Decisions ............................... 25
<table>
<thead>
<tr>
<th>Chapter 3:</th>
<th>Working Principle of the Proposed Approach</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Security Policies</td>
<td>26</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Security Policies for Attack Prevention</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>Integrity Verification Module</td>
<td>37</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Working Environment</td>
<td>40</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Integrity Verification Module Design</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>Integration of Components</td>
<td>48</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Summary</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4: Evaluation</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Experimental Setup</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Selection of Tools for Evaluation</td>
</tr>
<tr>
<td>4.2</td>
<td>Experimental Environment</td>
</tr>
<tr>
<td>4.3</td>
<td>Evaluation Criteria</td>
</tr>
<tr>
<td>4.4</td>
<td>Experiments and Results</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Pre-integration Protection Evaluation</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Protection Evaluation with Security Policies</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Protection Evaluation with Integrity Verification Module</td>
</tr>
<tr>
<td>4.5</td>
<td>Discussion</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5: Conclusion and Future Work</th>
<th>66</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Summary</td>
</tr>
<tr>
<td>5.2</td>
<td>Limitations</td>
</tr>
<tr>
<td>5.3</td>
<td>Future Work</td>
</tr>
</tbody>
</table>

| Bibliography                         | 71 |
List of Tables

3.1 Types of Attacks to Policy Mapping ........................................ 28
3.2 Summary Characteristics of Web Applications Used in the Evaluation 49
4.1 Evaluation Tools ................................................................. 53
4.2 Increase in Attack Prevention Rate ....................................... 64
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>High-level work flow for securing web applications</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Web application architecture</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Parametrized SQL query example</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Servlet code for getting and posting user input</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Subresource integrity confirmed with HTML attribute</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Communications involved for verifying integrity</td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>Challenge-response protocol in the integrity verification module</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Attack prevention rate prior to integration of the approach</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Attack prevention rate after the integration of the approach</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>Time required for IV with varying number of web workers</td>
<td>61</td>
</tr>
<tr>
<td>4.4</td>
<td>Time required for the IVM with varied attack initiation times</td>
<td>63</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Web-based applications are being rapidly deployed through the Internet. Owing to the fact that these applications are easy to develop and deploy, people are using them for both personal and professional reasons. However, information security and privacy issues are not always taken into consideration properly when these applications are developed. Hence, they fall victim to cyber-attacks and can be prone to giving out valuable information. The most vulnerable among these are e-businesses and applications that deal with a user’s personal information such as credit card information and insurance records. According to Verizon’s 2016 data breach report, 89 percent of web attacks had a financial or espionage motive [1]. Attacks can be conducted and sensitive information may be gathered while leaving little or no forensic evidence [2]. It is mostly after a security breach occurs, when security becomes a priority. Therefore, it is becoming increasingly important to develop web applications that have proper security measures in place and protect the applications that have already been deployed.
1.1 Motivation

In the past few years, businesses and establishments have invested in cyber security and anti-malware software. This has shown a rise in the number of security personnel maintaining the network and firewalls of a particular institution [3]. Security has been taken into consideration more and more in the development stages of a web application [4].

Web applications can be attacked on both the client side and the server side including any third parties that are involved in the process of getting the information to the client’s web browser. Even though all web applications should have security in place on both ends, it is not always the case. As can be seen in the research literature, more often than not security is enhanced on the server side while keeping the client side open to threats [5]. Client side data cannot be trusted and should always be scrutinized before usage. Security measures such as cryptographic algorithms can be put in place to fend off attackers. However, they too can be altered by these attackers at runtime of the web service. Therefore, the code that is added to make the client side secure requires more security of its own. Two industry-driven surveys that focus on web application security, namely OWASP (Open Web Application Security Project) and CWE (Common Weakness Enumeration), have illustrated how the attackers gaze has shifted from the server-side to the client side [6, 7]. Among the vulnerabilities mentioned in their most recent reports, SQL injection and cross-site scripting (XSS) are prevalent in both the reports. These present the need for proper web application security measures that take into account both server and client side security.

A web application also needs to verify that code included for security purposes is not altered during runtime. Due to this reason, the client side code needs to be checked
for integrity and verified against known good versions. Verification mechanisms that are put in place can weigh down the responsiveness of the application [8]. This makes it a challenge for developers to design a mechanism to check for integrity while making sure that the responsiveness is not harmed. It also leads to some establishments opting not to put any heavy-duty protection in place because they do not want to risk slowing down the responsiveness of their applications. This makes the application more prone to getting attacked. It has been assumed by developers for a long time that since client side code executes on the client’s machine, it is impossible to provide complete protection [9]. Web applications that use JavaScript are the most popular among developers and also the most vulnerable when it comes to experiencing attacks [10]. The lack of runtime checks for the client side increases the gaping hole of vulnerabilities. However, timed verification checks between a server and a client can minimize the threat of attacks [11]. Ongoing checks that can be run in the background can provide dynamic protection against code altering attacks.

The primary motivation of this thesis lies in the need to get secure web applications by following secure coding practices. We are interested in securing web applications against some named attacks by providing developers guidelines to follow. We also intend to keep alterations from happening in the client code by providing a module to verify the integrity of the code.

1.2 Overview

According to the needs of an organization and its web applications in question, security policies are customized when developed. For this reason, they are not interchangeable and very often not reusable. The client side of a web application requires
security policies defined to fight attacks similar to the server side. Also, the integrity of any system must be verified constantly to shorten the window of tampering. This needs to be done effectively to not wear down the responsiveness of the system.

For a list of secure coding methods, we have consulted OWASP and recent literature for recommendations [12]. From these recommendations, we derive security policies that can help secure a web application against certain attacks. Furthermore, to verify the integrity of the code during runtime, the system needs its own integrity verification process. To meet this requirement, our approach has an integrity verification module.

In our approach, a repository of open source JavaScript-based web applications is built. This repository hosts web applications that have both been deployed and are under development. A number of security enhancing guidelines, which are referred to as security policies from here on, are defined through secure coding practices to protect these web applications starting from the client side. To ensure that the client side code does not get altered during runtime, an integrity verification module (IVM) is designed that runs on separate threads in the background on the client side. Meanwhile, the main user interface (UI) thread is free to do other work. This maintains the responsiveness of the client side and increases security.

Figure 1.1 illustrates a high-level work flow of our approach. Our approach takes in web applications in order to secure it. To achieve this goal, at first, guidelines are provided for integrating security policies to the web applications. After that to ensure that the client side code cannot be altered, the integrity verification module is added. These two components work together to secure the client side of the web application.
1.3 Contribution

In this thesis, we present how web application client sides can be enveloped in a more wide-ranging security scheme through the practice of secure coding and integrity verification. Following are the main contributions of this thesis:

1. *The development of security policies for web applications.* This component shows how secure coding practices, when implemented properly, can provide security to web applications. Derivations of individual secure coding practices and secure design principles are organized into separate guidelines that are termed as security policies here. All the security policies presented are assembled into a single component. Developers can use this information as a guideline when they build their web applications as it advocates some policies for attack prevention. This will serve as a reference point for developers for incorporating security functions to their web applications in the future.

2. *The development of an integrity verification module that prevents code tampering during runtime of web applications.* In this module, JavaScript code on the client side is protected from alteration. This is also intertwined with the security code on the client so that if it is blocked or taken off by attackers, the application will lose some of its functionality. Client side code is signed and checked at regular intervals.
1.4. ORGANIZATION OF THE THESIS

during runtime to validate its authenticity. A challenge-response protocol is used in the verification mechanism.

In short, with the added security policies and verification mechanisms, the client side will have a new layer of protection. Developers can use this information to secure the client sides of web applications.

1.4 Organization of the Thesis

The rest of the thesis is structured as follows:

Chapter 2 presents an introduction to web applications and describes the security aspects in client-server based web applications. It also discusses the current solutions when it comes to web application security.

Chapter 3 presents our proposed approach. It discusses the design decisions and working principle followed by the two components of our approach.

Chapter 4 discusses the evaluation process and presents the results.

Chapter 5 presents the conclusion with a description of the limitations and future work.
Chapter 2

Background and Related Work

2.1 Introduction to Web Applications

Web applications can have different definitions depending on different stakeholders. This is not surprising since it bears similarities to desktop applications and websites. Also, depending on if someone is an end-user or a developer, the way web applications present themselves can be very different. To understand the inner workings of a web application, at first, we have to understand the basic differences between a web application and a desktop application. Desktop applications, for example, an image viewer or a file reader resides entirely on the PC. They are installed and run on the desktop and other than the occasional update through the Internet, the code for these applications can be run independently and offline. This brings us to the major difference with web applications. Popular web applications that we frequently use are email applications (i.e., Outlook mail, Gmail), instant messaging applications, online shopping applications among others. These applications do not reside on our PCs but rather on web servers that we access over the Internet. Web browsers serve as the portal through which we can access and use web applications. Moreover, the
updates for web applications do not happen on the user end, rather web applications are updated on the server and users get access to it through their browsers.

Web applications are also confused with websites and sometimes the words are wrongly used interchangeably by users. From a developers perspective, there are a few differences that can be used to separate one from the other. In essence, websites can be made of a single page built with HTML [13], CSS [14] and JavaScript [10] that remain constant with time. These websites might have no interaction with the user. A website does not require a database or even a programming language to build it. Web applications on the other hand are dynamic applications that modify their outputs according to user interactions. They deal with user inputs and data processing and have a database in the back end. An online retail web application will have a store front for the user to see on their browser, a form that the user can fill up to send to the server, and a database that stores inventory and client details. Different programming languages such as PHP [15], Ruby [16], Python [17] and frameworks such as Rails [16], Django [18] are used to build and operate a web application. These differences are important when it comes to securing a web application.

To understand the constraints of web application security we have to understand how a web application operates. For this, we refer to Figure 2.1 which shows a deployment diagram for a typical three-tier web application. Although there are other architectural models that can be used, this figure gives us a basic view of the different layers and stakeholders involved in a web application. It can be noted that our approach does not change for different architectures as long as the web applications in question have separate client sides and server sides.
2.2. WEB APPLICATION SECURITY

A web application is deployed on a web server which has different files and code needed for the application. The database is usually stored on a separate server and connected to it. In this three-tier architecture, we have clients or the end users, web servers, and a back-end database system. The clients can get access to web applications through their web browsers once it has been served. Each part of this system has security challenges that have to be dealt with. The next section discusses some of these security issues in more detail.

2.2 Web Application Security

The security of web applications has been a constant source of concern for web application developers and their clients. Attacks on web application client sides are far more prevalent than before and the security systems put in place are having a hard time keeping up with the nontrivial attacks.

Till date, most web application developers are also in charge of the security aspect of their applications. Even though most of the prevalent attacks on web applications
have well known solutions, they are not implemented by developers during the development of the application. There is also the added fear of weighing down the application with too many security features, therefore making it less responsive. However, that has not stopped researchers and security analysts from working for solutions for these attacks. There are recommendations circulated by researchers working to provide security to web applications for specific attacks and vulnerabilities. Communities such as OWASP (Open Web Application Security Project) and CWE (Common Weakness Enumeration) are acting as platforms where researchers and professionals come together to provide information about best practices when it comes to secure coding and securing web applications [19, 7]. Sometimes the product of the research becomes a framework for developers [20], whereas at other times it could feature a secure system that would need to be integrated with an existing application [21].

2.2.1 Web Application Attacks

In regards to web application attacks, at first, we have highlighted attack types from OWASP Top 10 2017 list of most prevalent attacks [19]. They are SQL injection attacks and cross-site scripting attacks (XSS). Along with these attacks, we aim to prevent resource alterations.

Injection attacks have been the top attack type in the OWASP list for past several years. It is the first type of attack that is used when an application is targeted by miscreants. The purpose of this attack is to provide inputs to the application that can cause unexpected responses to assist the attacker. If the data provided to the application is used without sanitizing or filtering, this can lead to an injection attack. Not removing ambiguity in user queries (escaping) can also cause injection attacks.
SQL, NoSQL and OS injection attacks are some of the most common types of attacks. To prevent these attacks, especially SQL injection attacks, the first line of defense is to always check and filter user inputs. Input sanitization and output validation are key factors in keeping a web application secure. OWASP also recommends using client side input validation by utilizing a whitelist along with server side input validation. Context aware escaping also helps prevent injection attacks.

A web application deals with large amounts of information both private and public when it operates. Some of this data might be sensitive such as keys that are exchanged, challenges that are sent in different protocols and passwords and credentials that are saved. This can also include vulnerable code on the client side especially when it is used for security reasons. If sensitive data in web applications are not stored or transmitted properly keeping security in mind, it can fall into the wrong hands leaving the application and its users vulnerable to attacks. This is why proper encryption and obfuscation systems need to be put in place when dealing with data. We employ multiple layers of encryption and obfuscation mechanisms on client side code.

A prevalent attack our approach aims to prevent is the cross-site scripting (XSS) attack. This attack happens due to a vulnerability in web applications when they allow unsanitized and unvalidated code to appear in their output. Taking advantage of this oversight, the attacker can execute malicious scripts into a web application and steal private information through cookies, among other things. Although strict server side security is enough to prevent most XSS attacks, it does not include all types of this attack. Without client side security functions, not all XSS attacks can be prevented.
To analyze the prevention methods for XSS attacks, we have to understand different types of XSS attacks. The three main types of XSS attacks are persistent XSS attacks, reflected XSS attacks and DOM-based XSS attacks (where DOM stands for Document Object Model). For all three types of these attacks, the attacker injects a malicious payload into a web application but uses different ways and resources to do so. Among these, the persistent attack is the most common attack. Here, a web application is injected with a malicious string with the use of a form on the web application. The attacker sends the malicious string as a response on the form and it gets stored in the web application’s database. When a user requests a page, the malicious string can be sent over as an output. By clicking on this link, the victim’s browser would execute the script and this would send private information including the victim’s cookies to the attacker’s server.

The next kind of XSS attack is the reflected attack. In this instance, the attacker tricks the user into requesting a malicious URL from the web application. When the web application responds, it sends the malicious string URL. This executes on the user’s browser and sends the confidential information and cookies to the attacker’s server.

The third type of attack that requires client side security is the DOM-based XSS attack. Here, the malicious code is actually the client side code and not the server side code. In this type of attack, since the web application is not checking the user inputs for safety, the malicious script is executed after the page is loaded. This is different from the last two attacks. In those attacks, the malicious script is executed when the page is loading. The way this attack works initially is similar to the first two as the attacker makes use of a malicious URL and tricks the user into requesting
it. However, in this case, the malicious script is only inserted into the page after the page’s script is executed. Then, the user’s browser executes the malicious script and the attacker receives the victim’s information and cookies. In the first two attacks, if the server-side code was not vulnerable, the attacks could have been prevented. However, that is not the case for DOM-based XSS attacks.

Resource alteration occurs when servers and Content Delivery Networks (CDNs) fail in delivering scripts and stylesheets in their original state. These resources get altered by the attackers supplying malware. To keep the web applications secure, care has to be taken to check whether a content that is asked for is actually what is presented. To verify that there needs to be a known good state saved beforehand. This has to be kept in an unalterable state or obscured from view. Cryptographic measures have been of great use in this regard. Having a hash value of a known good state to compare with during runtime assists in proving no alterations have happened. Our approach delves into solving this problem as well.

2.3 Client Side Security

A client-server themed web application is usually multitiered and spreads across multiple domains. However, more often than not security systems are placed on the server side leaving the client side with vulnerabilities [22, 23]. The security measures such as cryptographic algorithms, user verifications and session validations can be changed by attackers at runtime as well [24]. Moreover, there are specific attacks (e.g., XSS attacks) that are targeted towards taking advantage of a client side vulnerability. Therefore, without proper policies to maintain the security of the client side, web applications can and will be compromised. In terms of security, the most important
2.3. CLIENT SIDE SECURITY

issue to acknowledge is that client side data should always be inspected both before and during usage.

To put things into perspective, we now see web applications updating their pages without communicating with the server. The need to remain fast paced has increased the pressure for data to be generated on the go. In most cases, JavaScript is generated on the client side at runtime. This can happen after an AJAX (Asynchronous JavaScript And XML) request [25] and can put the web application in danger if the content does not go through a security check. This increases the room for vulnerabilities in the client side code. This can also happen if the client side code includes unsafe user inputs in a DOM update after a page is loaded. These practices can lead to an attack even if the server side code is secure.

Prevention methods for these attacks are mostly targeted for the server. Encoding, sanitization and validation of the inputs are some of these methods. However, they can be used to secure both ends if dealt with intuitively as we present. Since inputs taken from the user is not to be trusted, there has to be measures put in place to check for safety and vulnerability. However, we also need to take into consideration where the inputs have been entered on the page. This brings up the concept of context in a web application.

In web applications, context refers to when and where user inputs are taken. This has to be taken into account when checks are performed to ensure that the inputs are safe to use. A check for a JavaScript value is different from a check for a URL query or a HTML element, depending on the context of how it is entered. Here, if a user input is used without sanitization, the attacker’s job would be to end the context by using the closing delimiter ( e.g., commas, semicolons, quotes, and braces) and inserting
the malicious payload. Having quotation marks in the output of the web application makes it vulnerable to a type of attack in a certain context. Therefore, removing them before they are sent over should make the page less vulnerable. However, if the delimiter is different for another context, this solution fails. This is taken into account in our proposed solution.

When it comes to access control on web pages, the most widely used policy is the Same-Origin Policy (SOP). According to its implementation, SOP requires that JavaScript can only acquire information about iframes and properties of windows if their origins are the same as the page containing the script. This implementation varies from browser to browser which is one of the first challenges in implementing this policy. An origin is defined by a protocol, a domain and a port of an URL. This definition holds true for most browsers except Internet Explorer, where an origin does not require port information. In other browsers, there remain dissimilarities in the implementation of the SOP. In Internet Explorer and Opera browsers, a JavaScript file can access other local files through DOM by the use of the “file” protocol. However, this is not the case in most other browsers. This creates inconsistencies when using SOP as the primary security policy against prevalent attacks [26]. Moreover, it has also been brought up in several research experiments that SOP does not provide adequate leakage prevention for sensitive information. If a malicious JavaScript can get access to the information, it can send this information to other remote servers [27, 28, 29]. This can lead to cookie-stealing, location hijacking, history sniffing, and behaviour tracking [30]. The focus of a security solution therefore needs to be consistent across browsers and also needs to include security policies that can deliver coverage for different attack scenarios.
Another issue with client side security is the alteration of JavaScript files during runtime. Even after following secure coding practices, some attacks might be difficult to thwart. Therefore, it is essential for the security of the application that JavaScript files and their functions are tracked and verified. The verification should be done repeatedly and in a way that is not visible or predictable to the attacker. To ensure that no changes have been made to the code, we can check the integrity of the files and functions. This can be accomplished using cryptographic methods, such as hash functions. Most developers choose to have a trusted version of the file on the server for verification. We will discuss why that is an issue in the next section. The client side files need to be tracked and checked for changes during runtime. This creates a tough problem to address as everything on the client side is visible to the attacker and debugging is an option. To counter this problem, we have to come up with an approach that is able to keep track of the client side code and notify the user if any content has been changed. Since it will be on the client side, the implementation details will have to be obfuscated to avoid tampering. The security features of the web application can therefore be secured using this verification system.

2.4 JavaScript Latency In Browsers

A survey of web applications has shown that 99.6 percent web applications use JavaScript on their client side [31]. The problem that arises on client sides that use JavaScript is the inherent latency. As a programming language, JavaScript was designed to run in a single-threaded environment. In a web application, all JavaScript code is executed in a single thread by the web browser. Thus, multiple scripts cannot be executed at the same time. Web applications have to handle UI events and process
large amounts of API data. Events such as mouse clicks and form submissions are handled one by one non-preemptively by the web browser on the UI thread. These events cannot be altered or interrupted at runtime. There is also no way to pause or restart the event handling process once it starts. This often leads to the main thread becoming slow and unresponsive at times. Therefore, care needs to be taken to ascertain that the UI thread is used efficiently and released after the job is done.

Adding security features on top of application features for a JavaScript application adds latency to the single threaded process. This issue has to be handled by the developers by using event handlers and setting timeouts to make the processes run together while providing security. However, this is not always possible and our approach presents a solution to this problem.

2.5 Related Work

Research has been done for defending web applications from different angles. However, most of these work are for prevention and detection of attacks [32] and blackbox testing of vulnerabilities [33]. Web application firewalls (WAFs) are promising as primary defense mechanisms for web applications. They depend on anomaly detection techniques\(^1\) to protect web applications but do not always provide separate client side mechanisms against named attacks [34]. Moreover, WAFs can be bypassed by knowledgeable intruders which is why there needs to be a system in place on the web applications [35]. To detect and protect web applications from vulnerabilities, a significant amount of research has focused on static and dynamic analysis of the

\(^1\)Anomaly detection depends on recognizing dissimilar components or behavior in a more or less heterogenous dataset.
application code [36, 37, 38]. These solutions can be used periodically by the developers to check for vulnerabilities in the web application and make changes accordingly. However, these would not offer runtime protection of the client code. Moreover, some solutions require that policies to check for vulnerabilities must be specified in a specialized language, which may not be developer-friendly [39].

2.5.1 Existing Problems in Purely Server Side Mechanisms

A crucial part of protecting the client side is verifying untrusted content on the web application. Most defense mechanisms in the research literature focus on server side mechanisms only. This is mainly due to the fact that servers are more protected by themselves and have more operational power. Security policies are enforced on the servers for detecting and dealing with attacks [40]. The difficulty with these solutions is that they depend heavily on the fact that processing on the client end is in sync with the server. This assumption has been proven to be costly and dangerous on several occasions. Since browser behavior changes according to browser implementations, servers cannot predict client side processing and it is not always possible to mimic results [33]. Even then, server side defense mechanisms are used to specify and enforce security policies for the browser [41]. Without real-time cooperation and assimilation from both ends, these solutions do not meet their goals.

2.5.2 Existing Web Application Security Mechanisms

Recent work on web application security involve end-to-end security schemes and protection of the web application through code instrumentation [42]. Web application scripts are among the most vulnerable parts of a web application. To get information
2.5. RELATED WORK

about what script is used, where it is from and from where it is loaded, web clients need to be examined. Web application defenses, however, do not always flow from the client end since it is vulnerable by itself. Fine-grained authentication and verification systems are implemented on the server side as mentioned above. However, purely client side defense mechanisms do exist that enforce policies without the help of the server [43, 44]. Detection of malicious code is a mechanism that can be successfully carried out on this side [45]. Another method that is used to enhance security is code instrumentation. Browser-based attacks rooted in JavaScript are thwarted by instrumentation techniques where the code is rewritten according to given policies [46]. This is similar to information flow security techniques [47] where the flow of processes in the web application is protected through tracking and maintaining type systems. The problem with some of these techniques is the high overhead cost of integration [48] as rewriting code or tracking the flow of information both require a system in place that has to be programmed beforehand by developers.

For preventing SQL injections through user inputs, researchers have proposed many defense mechanisms and frameworks. Here, we discuss some of the techniques presented in the recent literature. Blackbox testing and code checking have been the front runners when it comes to detecting SQL injection attacks among researchers. Although blackbox testing does not require knowledge of the code base, like all black-box approaches, it does not assure completeness while checking for vulnerabilities [33]. Combined static and dynamic analysis as seen in AMNESIA [49] employs model-based code checking to detect and prevent SQL injection. This technique uses static analysis to build models of different types of SQL queries. During runtime, AMNESIA intercepts all queries to the database and verifies them against the statically built
2.5. RELATED WORK

models. If the queries do not pass the verification test, they are prevented from executing. The limitation with this technique is that if the primary models are not accurate, the success rate for this approach drops significantly. If code obfuscation is used while sending in queries, the models will not be able to identify actual attacks during runtime. SQLGuard and SQLCheck also use runtime query checks to see if they can be verified through models [50, 51]. These approaches also use a secret key to delimit user inputs during parsing through the runtime checker. Therefore, if the key is discovered, then these approaches would fail to offer security. More recent dynamic prevention of SQL injections include CANDID where a programmer intended SQL query structures are learnt by the system [52]. During runtime, CANDID dynamically constructs the structure of the intended queries and compares them to the benign learnt ones. This approach needs its own language and query structure for formal analysis and transformation. It limits itself as a solution when a benign query is wrongly detected as an attack query, if transformed improperly. This leads to other similar benign queries to be wrongly judged. Moreover, each type of query needs to be transformed before the solution can be used. Although the above approaches can limit the number of SQL injections, they are not developer-friendly. The solutions provided in the above research works, are aimed at web applications that are already deployed and needs a substantial amount of technical knowledge and research for integration purposes.

Another way to defend the web application against attacks that is seen in recent literature is through defensive programming. In [53], the authors present an approach that uses whitelists and blacklists to automate prevention of SQL injections. This is a low cost mechanism that can check inputs against given lists and that can be
integrated by developers with minimum changes to the web application code. It does not involve any new static or dynamic transformations. However, this procedure can lead to cross-site scripting attacks through SQL injections. To deal with XSS attacks, further security policies need to be put in place.

In the context of cross-site scripting attacks, the defense mechanism depends on the kind of attack. For stored and reflected XSS attacks, the prevention methods are usually on the server side. This is done by inspecting and verifying data that is exchanged between the client and the server. Tainted flow analysis in the Perl interpreter is a popular server side defense mechanism. Here, tainted data is marked and tracked through the program and any attempt to use it for a critical command is aborted with an error. The commands involve modification of files, directories, processes and invoking subshells which are caught and terminated instantly. Interpreter-based approaches for PHP are presented in [54]. With boundary injection and policy generation, Shahriar et al. [55] present an automated framework to detect server side XSS attacks. This framework comes with a tool to automatically insert boundaries and generate policies for JavaServer Page (JSP) programs. However, this is still a server side mechanism and lacks the functions required to prevent client side XSS attacks.

There are some approaches similar to ours that employ defense mechanisms on the client side. An auditing system on the Mozilla web browser is presented in [45]. This system monitors the execution of JavaScript code and uses intrusion detection techniques to detect malicious behaviour. However, this does not ensure security against DOM-based XSS attacks. In [56], application level firewall keeps sensitive information from leaking by analyzing HTML pages that have already been browsed.
2.5. RELATED WORK for hyperlinks. This technique is used to prevent XSS attacks. By analyzing HTTP traffic exchanged between the client and the server, Ismail et al. [57] present a way to detect XSS attacks. Even though it shows promise against reflected attacks, it does not prevent stored or DOM-based XSS attacks. Moreover, the above approaches work with web applications that have been deployed. They do not have separate policies for developers to follow if they do not intend to integrate the whole system.

Resource alteration can occur when files and stylesheets are hosted on third party servers and Content Delivery Networks (CDNs). If the servers are compromised, the content delivered are altered with malintent. To prevent resource alteration, web developers use HTTPS to ensure that the connection between the web server and web browser is encrypted. However, that does not ensure the integrity of the content provided. No prevention mechanism that is specific to fighting resource alteration was found in the recent literature.

Code tampering on the client side has been addressed through information flow analysis [58] in recent research work. This is mostly due to the fact that it is easier to secure the server side and provide security to the client through extension of the server side methods. JavaScript integrity has been overlooked for document structure integrity and integrity of content caching [59]. These do not protect the client code from alteration during runtime. This has been a long running problem in web applications. Any solution that runs security functions on the client side, has to make use of the main UI thread and therefore, would slow down the responsiveness of the application. Wan et al. [60] present a solution for verifying the integrity of application cache in android runtime to defend it against attacks. However, similar solutions for JavaScript runtime are not available.
The security mechanisms discussed here are built as separate systems that can be used on web applications. It is not possible to implement parts of the solutions and achieve a secure web application. Web application developers are not helped by these security mechanisms if they do not adopt the entire system as a whole and implement it. Developers would need to integrate entire solutions to their native code for security purposes. This integration might even lead to bigger security holes.

The primary focus of our approach is to provide the power of attack prevention to the developers. We present security policies derived from secure coding practices that developers can refer to and integrate to their web applications without the need to learn about an external approach. There is no need for integration of an analysis and transformation system. Our approach relies on developers reevaluating and changing their coding practices to ensure more security for their web applications in terms of SQL injection, XSS attacks and resource alteration. In our approach, we present two security enhancing components for web applications to protect web application client sides.

### 2.5.3 Existing Solutions for Solving JavaScript Latency

JavaScript is an event driven language that runs on a single core. This is a cause for latency in web applications that use JavaScript. There are no solutions that can change how web browsers interact with JavaScript files without making changes to the code. Mozilla has a JavaScript implementation written in Java that places continuation\(^2\) as its first class object. It is named Rhino and can be used with the Mozilla browser [62]. This is an attempt to help with the latency. However,
continuation is not a first class object in JavaScript standard specifications. Rhino can be useful for code conversion. However, since it is not included in any other web browsers, it cannot always stand as an effective solution. There has been work on building frameworks for asynchronous processing of JavaScript. Maki et al. [63] present a framework suitable for AJAX applications.

The approach that we propose is to use background threads to secure web applications. These background threads would be responsible for lengthy computations needed to confirm code integrity. The JavaScript feature that is used in this case is the web workers API [64]. The worker threads run in the background while leaving the main UI thread of the web application free to execute other files on the web application or stay responsive to the user. The worker threads also provide entanglement in the verification process for added security. This research uses these web workers to perform the integrity verification of the client side code for securing the code during runtime.

2.6 Summary

To date, the client side of a web application is protected through the extension of server side mechanisms. Traffic is restricted via firewalls and proxies. However, this only takes notice of network traffic and once traffic is allowed, the web application is at risk from exposure to attackers. The filtering solutions that are available are limited to their customization and do not provide security from a wide range of attack perspectives. This makes web application attacks through the client side a popular and easy way to gain access and cause harm.
Chapter 3

Secure Coding Practices and Integrity Verification

3.1 Overview and Design Decisions

When broaching the aspect of web application security and integrity, we must keep in mind the many languages, platforms, techniques, and protocols which are in use. The security mechanisms that might work for a web application to share artwork written in PHP and JavaScript might not work for e-business web applications written in ASP. NET. These aspects are taken into consideration while presenting security recommendations [5, 7, 8]. Moreover, thought has to be given to the development process of the application and the compatibility of these unique security schemes [18]. As discussed previously, our approach is focused on securing the web application against some named attacks and verifying the integrity of the JavaScript security code during runtime.

To secure the client side against third party attacks, we propose some security policies. These policies are presented as guidelines to make web applications secure against SQL injection attacks, cross-site scripting (XSS) attacks and resource alterations. These policies are derived from secure coding practices present in industry
3.2. WORKING PRINCIPLE OF THE PROPOSED APPROACH

Driven research [12]. In terms of verifying the integrity of the code, we present a module with functions to accomplish that. Since this verification module encompasses the client side code, it is implemented separately.

3.2 Working Principle of the Proposed Approach

The proposed approach for integrating security in web applications has two components in it. At first comes the security policies and then the integrity verification module. The security policies work alongside each other to provide security for the client side code and protect it against SQL injection, XSS attacks and resource alteration attacks. This is further discussed in Section 3.3. The integrity verification module consists of implementations on both client and server sides. It works during runtime to prevent code changes on the client side. This module uses a challenge-response protocol along with hash-based message authentication codes to ensure integrity.

The security policies deal with code security and they employ methods to ward off some named attacks. The integrity verification module confirms that the client side code is not tampered with during runtime. Both the components are required to confirm that the web application is secure against above mentioned attacks.

3.3 Security Policies

The first component in our approach is made up of several security policies that work toward preventing attacks on web applications. These policies are derived from secure coding practices and can be defined as guidelines that should be followed to protect web applications. They are targeted towards specific parts of web applications. The policies are integrated in the order that they are mentioned below. The policies deal
with how data on web applications are used and how vulnerable the applications are
to the client or even an attacker. Here, we assume that the client is benign and
the attacks occur through a third party attacker. The policies proposed are input
sanitization, output validation, principle of least privilege, subresource integrity, and
content security.

3.3.1 Security Policies for Attack Prevention

Each of the security policies have specific targets it must achieve. The initial aim is
to ascertain that no user inputs can be received or sent back without going through
escaping and encoding mechanisms. The first two policies deal with this issue. This
is essential for defense against SQL injection and XSS attacks. Then we move on to
giving and restricting access to the web application data. To ensure that the different
parts of the web applications are only given the least amount of access it needs, we
integrate a policy called the 'Principle of Least Privilege' inspired by OWASP [65].
This policy assists in the prevention of all three above mentioned attacks along with
any code alterations. Web applications also use scripts from third party resources
including servers or Content Delivery Networks (CDNs). If these scripts are not
checked for integrity before execution, they might be the cause of security breaches.
Therefore, we have integrated a way to confirm that all the resources used by the
web application are checked for integrity. This policy is aimed at preventing resource
alterations. We have also developed and customized a content security policy [66]
for web applications. This policy ensures that the behaviour of the web application
is restricted to some rules. This makes it possible to prevent harmful changes to
the application if a successful malicious injection occurs. This aims at preventing
cross-site scripting attacks. Table 3.1 presents how each of the policies are mapped to specific attacks.

Table 3.1. Types of Attacks to Policy Mapping

<table>
<thead>
<tr>
<th>Policy</th>
<th>Attack type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Sanitization</td>
<td>SQL injection, XSS attack</td>
</tr>
<tr>
<td>Output Validation</td>
<td>SQL injection, XSS attack</td>
</tr>
<tr>
<td>Principle of Least Privilege</td>
<td>SQL injection, XSS attack, Resource Alteration</td>
</tr>
<tr>
<td>Subresource Integrity</td>
<td>Resource Alteration</td>
</tr>
<tr>
<td>Content Security</td>
<td>XSS attack</td>
</tr>
</tbody>
</table>

Through the enforcement of these policies, our aim is to provide a line of defense against these attacks in web applications. Our first priority is to ensure proper input sanitization and output validation. This will always remain the most important defense to prevent attacks. The rest of the policies are customized to deal with different privileges and behaviours in web applications. Working together, these policies provide a robust line of defense against attacks.

**Input Sanitization**

One of the main problems apparent in web applications is insufficient input sanitization. When it comes to secure coding practices, this is the top priority in building a secure application. Without it, the application is open to attacks such as injection attacks, cross-site scripting (XSS) and even phishing attacks. The first security policy that is integrated into the web applications in our repository is input sanitization. This is also because malformed inputs can cause web applications to collapse by wreaking havoc on the security components. It can redirect user’s browser to their own malicious site, steal the user’s private data and exploit cross-site scripting. Although most approaches have server side prevention mechanisms, we are interested...
in sanitizing the data submitted by the client in HTML forms and URLs. Since both SQL injection and XSS attacks are caused by mishandling user inputs, the first policy in this component is input sanitization.

As soon as any input is taken by the web application, it has to be checked for syntactic and semantic relevance. There are default filters in some languages that are in use in web applications. Frameworks such as Apache Commons Validator [67] and Django Validator [68] are used for data type validation. However, we do keep in mind that filters can be bypassed by intruders and there are other security policies in place if that happens. Our guideline ensures that the length and the fields are checked for each input. We aim to deny all attempts at putting untrusted data in HTML documents unless it is required by the application. Following our guideline we start with escaping HTML before inserting it into the element content and HTML common attributes. Dynamically generated JavaScript code is treated as data value as the code cannot be trusted.

We aim to validate all incoming responses from the web application while verifying that they conform to certain rules and constraints. We use validate.js library [69] to emulate primary validations on web applications. In addition to the validators provided, we add more for the inputs required for each web application. The most crucial ones are for username, email, password, birthday, country, zip code, and any numerical inputs. For example, when validating a password for user registration, our approach verifies that there is an upper case letter, a lower case letter, an alphanumeric character, and that it is not similar to the username\(^1\). It does not let the user proceed without meeting these constraints.

\(^1\)According to the latest password guidelines from the National Institute of Standards and Technology, the above constraints for passwords need to be updated for better protection [70].
Our proposed approach uses a whitelist system to protect web applications against injection attacks. This is different for each application according to the genre. We define the whitelist and characteristics for each input depending on expected entries. The inputs only go through the application if they match. The list can be hosted on the client or the server and checked for every input. The minimum and maximum values for numerical parameters are reinforced. There is also a check for minimum and maximum length of Strings.

Most importantly, this approach recommends using parametrized queries when querying a database. Accordingly, each query is prepared as a SQL statement and the parameters are passed to it later. This prevents attackers from altering a query to inject code and cause an attack. Through the use of prepared statements, the database is able to differentiate between code and data. Consequently, there is no need to manually escape inputs. With the use of parametrized queries, there are also fewer String concatenations and smaller Strings sent back and forth between the client and server. This is the leading option for the prevention against SQL injections [71]. Figure 3.1 presents a parametrized query. Here, placeholders are used for name and email instead of actual values. The value for mentioned name and email will be supplied at runtime. This avoids String concatenations and constructs a query plan on the server at runtime. This simple change in how the query is stated can help prevent SQL injection attacks.

```
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, email) VALUES (:name, :email)" );
$stmt->bindParam(':name', $name);
$stmt->bindParam(':email', $email);
```

Fig. 3.1. Parametrized SQL query example.
This policy also assists in protecting the web application from threats such as injection attacks, buffer overflow attacks and file system attacks by checking each input every time. This policy works best if it is integrated to web applications in the development stage as it needs to be configured to every input on every page. For conforming to the guidelines of this policy, developers will need to add checks for every input and redefine SQL queries.

**Output Validation**

After input sanitization, the next policy for ensuring security is output validation. This is regarded as escaping or securing the output. This is a coding practice to be followed for building secure and robust web applications. For ensuring proper security, this policy needs to be enforced on both the client and server sides. Any input received from the client is checked and verified before using it as an output. This policy makes use of escaping techniques with the knowledge of what is expected as an output. This works in accordance to the needs of the web application and might not be needed for all pages or parts. The use of the output is taken into account when integrating and enforcing this policy. The length of every output is checked before it is presented. An output that is longer than expected can be used to cause harm to web applications. Moreover, all outputs to the client are checked for script tags and only scripts that are from expected sources are sent through. Quotes (both double and single) are screened and formatted before it is sent to the client. All kinds of brackets are also investigated during output validation. Every starting bracket is checked for an ending bracket in the same context. In our approach, we want to make sure that no input is echoed back in the HTML context without proper filtration.
In Figure 3.2, we see there is a `getResponse()` function that takes the user input. Here, the user can submit a response and the `postResponse()` function will get the content of the response, store it and send it to a Get request to get it posted on the page. Since there is no check or filtering system in these functions, they can be exploited to cause an attack. Instead of responding with a String, an attacker can send in HTML code and it would still be rendered. For example, “<h1>you have been exploited</h1>” can be submitted and the HTML would be rendered. To prevent this scenario, our approach does not allow any content to be used without going through a filtering process. Every input is escaped to ensure HTML is rendered as text. `<p>My <script>exploit()</script> response</p>` is altered to

```
&<?php; My<&script;>exploit();&</script;&amp;gt;response&amp;lt;/p&amp;gt;
```

The way the output is escaped in terms of validation depends on what it is to be used for, i.e., if it is for an AJAX response or a CSV file. This policy seeks to confirm that everything that is sent over the application is checked and verified before a client can interact with it.

```java
public void GetResponse(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    request.setAttribute("content", content);
    request.getRequestDispatcher("WEB-INF/jsp/render.jsp").forward(request, response);
}

public void PostResponse(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    this.content = request.getParameter("content");
}
```

**Fig. 3.2.** Servlet code for getting and posting user input.
This policy is also designed to prevent brute force attacks. The attackers usually use automated ways to guess passwords and try every combination until they get the right one. They look for some commonly known terms to see if the password succeeded or failed such as 'success', 'error', 'fail' or 'try again'. We include these words in the HTML comments of the source code both in the login page and the successfully authenticated page to thwart the attacks. This confuses the automated brute forced attacks and derails the attackers. We also give the error messages a certain amount of randomness, so that they are not using the same phrases each time. This also helps confuse the automated attacks as they hope for predictable response messages each time.

This policy helps against XSS attacks and brute force attacks. It is ideal for this policy to be integrated in the developmental stage of a web application. Developers are required to code for the customization of this policy on web applications. Escaping and encoding for different contexts on web applications must be done by developers before the web applications are launched.

**Principle of Least Privilege**

The principle of least privilege is a policy derived from OWASP security by design principles [65]. The goal of this principle is to restrict access, rights and privileges to the application in order to secure it. It only allows individual users the level of access they require to perform their work efficiently and not more. This prevents the application from getting attacked by users with undefined rights as that leads to more harm and loss of security. This policy is derived from security by design principles which is an essential element in secure coding.
This concept lends itself well to securing web applications especially in the development stage. Web applications are notorious for the amount of individual users they must keep track of. They also integrate embedded content and features from third parties which cannot be trusted. Therefore, the privilege of different users and functions should always be checked. It means that the application components and the users in it are only given the least amount of privilege they require to complete the work. In essence, our approach initially blocks off features that are not required for a web application and its users. This also leads to the protection from SQL injection attacks [72]. The functionalities of this policy include turning off plugin support if the page does not require using plugins, preventing access to `window.open` depending on the application, and deny root access to databases whenever possible. This approach also highly encourages that privileges are set to roles instead of users. This makes it easier to keep track of users and alleviates the risk of users having privileges they do not need from past roles. In this case, they can be assigned to a new role without having to edit their individual privileges.

This policy needs to be customized for web applications by the developers. This can be done in an iterative fashion starting from the time of development where customizations are done every time new technologies or new roles are added to web applications.

**Subresource Integrity**

For a web application to be functional with options for clients, it has to serve content from several third party resources. Often, these contents are hosted on third party servers and CDNs and the developers have no control over the servers. If the servers
are attacked and the content are tampered with, the web applications can inadvertently serve malware to their clients. This cannot be prevented by secure downloading as the content to be downloaded is now corrupt. Moreover, the most popular prevention rule for cross-site scripting (XSS) attacks is preventing untrusted data to be placed in the HTML document [73]. This rule can help eliminate most XSS attacks. However, scripts that are executed on the application need to be checked for integrity to conform to this rule. This is where subresource integrity (SRI) comes into place. By practicing secure coding in the development stage of a web application, we can ensure that unwanted malware are not served to web applications through servers or CDNs. It is a process through which a user agent can confirm that the downloaded content is indeed what was requested for. To enforce this security policy, every time a third party resource is added to the web application, a String is added to the HTML element to confirm its integrity. This String contains the sha256 cryptographic hash value of the script that is to be requested. Figure 3.3 presents a script tag with the SRI attribute.

![Fig. 3.3. Subresource integrity confirmed with HTML attribute.](https://code.securesource.com/hmac243.js)

The browser plays an important part in enforcing this policy as it compares the cryptographic String that is in the HTML document header with the hash value of the document. It will only execute the files if the two hash values match and will throw an error if they do not. For example, a resource file will have a base64-encoded hash
value that is included in the fetch requests. The hash value is inserted in the integrity attribute of a `script` and a `link`. This is used for JavaScript files and links in the web applications. It arms the client side with a contingency plan to stay protected even if the third party resources have been compromised. This policy can be integrated during the development stage or added afterwards as long as the developer knows what the third party resource is delivering. This protocol needs the developer to add the hash values of third party scripts in the HTML header before launching the web application.

**Content Security**

Finally, we present the content security policy or CSP. Since a web application might have content from different sources, there needs to be a list to check against content sources before using them. CSP provides this with the means of a whitelist. It is a HTTP header that is recommended by the Web Application Security Working Group [66]. With the help of this header, we create a whitelist which the browser can use to choose whether to allow a script to execute or a content to be served. This can be done through the practice of secure coding by developers. This is especially important when it comes to cross-site scripting attacks. Here, even if the attack has been executed and a script injected, CSP will not find a match of the script with the ones on the whitelist and therefore it will not be allowed to execute.

This policy allows developers to declare sources to be verified. It can also be customized to allow and prevent certain behaviours. Moreover, the directives in the policy are used for micromanaging certain behaviours. Some of the directives in CSP are pre-request check, post-request check and inline check. These directives are used
to design customized content security policies.

The content security policy is integrated to each application in terms of resources and content it requires. Due to its customizable features, prior information about the resources needed by the web application is required. This policy is not designed to be the first line of defense against attacks. It is to make sure that if an attack occurs, the damage can be mitigated. This policy does not need the developer to implement new features in the code base. Rather, this will require the present code to be reevaluated and the checks for this policy to be put in place.

All of these policies, when enforced and working in sync, have the capability to thwart almost all unwanted cross-site scripting, SQL injection and resource alteration attacks. Each policy requires a different amount of code integration. Here, input validation and output sanitization require the most attention from developers due to the thoroughness of the checks required. The other policies have significant effect even though they do not require as much attention. This still leaves developers with some attacks which go through. To make sure that harmful code changes on the client code caused by attacks are mitigated, we have another component in our approach to verify the integrity of the code at runtime as described in the following section.

3.4 Integrity Verification Module

The integrity verification module or IVM is used to ensure that the security code on the client side is not tampered with during runtime. The security code and the application code modules in the web application are tagged and labeled beforehand. There are three things principal to the confirmation of integrity in the IVM: having a signed result of the code calculated on the server, randomizing the signing procedure
with different challenges, and verifying the signed code on the client and the server end. This is provided in the proposed verification module. To understand how this module works, we have to be aware of how a client communicates with the server in terms of providing linear, single-threaded integrity verification of its code.

Figure 3.4 features a web application verifying the integrity of its code in a linear manner without using multithreading. In the figure, we can see that there is an Actor or in this case, a user of the web application. To check if the client side has been compromised, the user initiates an integrity verification request. This can happen when the user is logging in, making a payment or filling up an HTML form to submit.

![Diagram of web application integrity verification](image)

**Fig. 3.4.** Communications involved for verifying integrity.
The series of events that generally follow after the request are discussed below. The request contains a seed or key that is used for hashing the client code. The seed or key can differ according to the verification mechanism. For instance, it can be a unique String or a set of numbers. Upon receiving this seed or key, the application code module then sends an HTTP request to the security code module of the application. Here, the integrity verification module (or a similar mechanism that is present) is put into work. The seed or key that is sent with the request is used to calculate the hash (or other cryptographic) value of the client side code. This value is then sent to the server in a message to be verified. The server performs the same calculations with the known seed or key and matches the two hash values to check for any discrepancies. The functions performed on the server are not shown in the figure. After calculating the hash value, the server then responds with the result from the verification. In this step, the server can also respond with a new seed or key for the next iteration on the verification module. Upon receiving the response, the web application is assured that the client side code has not been changed. The code to secure the application extracts the new seed or key and sends a response to the application code which can assure the user with a message on screen. This whole single threaded generic process however happens sequentially and takes a considerable amount of time during which the main UI thread on the client is left waiting. It also opens up options for man in the middle attacks and timed attacks as the process can be predicted and messages can be intercepted.

The integrity verification module presented in this thesis acts preemptively to perform checks on client side code before a request can be initiated for verification. Our approach makes sure that the integrity verification process is not sequential and
can work in the background. The IVM also takes away the predictability of the process and makes sure that it is safe from man in the middle attacks, timed attacks and malicious reverse engineering through code change.

3.4.1 Working Environment

The IVM in our approach is implemented in such a way that it can be integrated with web applications that have some security measures in place with minimum interference with live code. Before starting the process, we identify the client side code we intend to protect. In our case, it is JavaScript functions containing security policies. The communications between different modules are monitored and verified in a timely manner to produce reports of a system if it is tampered with. The clients can then decide whether they want to proceed or stop using the web application. The server is also notified of this report to take further action. In the IVM design, we use the web workers API. These web workers act like multiple threads that run scripts in the background and keep the main UI thread from running every script linearly. Each worker is capable of running a JavaScript file. The workers run the scripts on a separate global context from the main window. To make use of this API, some initial changes are made on the client side code.

The functions on the client side JavaScript files are divided into groups and placed on separate web workers. These functions are signed and hashed to check for integrity. This is done using a keyed hashing algorithm which is HMAC SHA256 [74]. The code for this algorithm is placed on a separate worker script. All the different scripts include a function that retrieves all the function bodies on that script and packs it into a String. This String is then sent to the worker containing the hashing function
3.4. INTEGRITY VERIFICATION MODULE

Communication among workers and the main UI thread is achieved through the use of message channels that are configured to unique ports on each worker. Messages are sent using `postMessage()` method on the workers. To respond to messages, the workers use the `onmessage()` event handler. The design for this module is described in the next section.

3.4.2 Integrity Verification Module Design

The module is designed based on the challenge-response protocol as depicted in Figure 3.5. It is designed to secure the verification module and to maintain obfuscation during runtime. The three major elements of the figure are the main page running the main UI thread, the web worker with the hashing function and the multiple web workers denoted with the number (#) sign.

![Challenge-response protocol in the integrity verification module.](image)

Fig. 3.5. Challenge-response protocol in the integrity verification module.
In this module, the key for the hashing function is sent as a challenge for each iteration. The server has knowledge of the challenge and hash value of the known good code for each script. The sequential process is described below.

- The process starts with the web worker with HMAC SHA256 hash function sending a random 4 bit challenge to the main page (1).

- The main page then proceeds to obfuscate the challenge and sends it over to the web workers with the security code over message channels (2). We use the main page here to communicate with the workers.

- When the web workers receive the challenge they retrieve their function bodies as a String according to two predetermined bits in the challenge. The bits dominate how the functions are arranged within the String. The four combinations from the two bits (00, 01, 10, 11) change the order of the retrieved Strings. According to the last two bits, the Strings retrieved can be ordered in different ways. They can be placed in an alphabetical order- ascending or descending. They can also be placed on the String according to their length- short to long functions or long to short functions.

- For further obfuscation, the function Strings are then XOR-ed with a rolling XOR key and sent to the main page over the channels as a response (3).

- The main page then retrieves its own function bodies as a String according to the two bits in the challenge and applies the rolling XOR to the String.

- The main page then sends over the Strings from the workers and itself over to the web worker containing the hashing function (4).
3.4. INTEGRITY VERIFICATION MODULE

- The web worker with the hashing function (HMAC) receives the message with the masked Strings. The web worker uses the embedded function previously mentioned to retrieve another String. This String contains the web worker’s hashing function.

- The web worker with the HMAC function then performs the hashing computations on all the Strings and saves the hash value with the two bits of the challenge that was sent.

- Consequently, another challenge is sent to the main page by the web worker with the hashing function and the above process keeps repeating in the background (5).

If the web worker with the hashing function receives Strings that are masked with the same two bits as previous challenges, the new hash value is compared to the saved hash value before getting saved. A dissimilarity will lead to an error output. If the Strings are masked with different bits, then it saves it on the web worker in a similar procedure as before.

When the application requires an integrity verification of the client side, the following steps take place.

- The main page requests the worker with the hashing function for a verification check along with a random two bit number which indicates the hash value to be chosen.

- The web worker with the HMAC function and hash values proceeds to send in the pre-computed result without any delay, in essence, speeding up the whole verification process.
3.4. INTEGRITY VERIFICATION MODULE

- After the main page gets the hash result, it sends the result to the server.

- The server computes the same steps as the client, therefore, the server will have the same result.

- The server can check with its own version of hash results and report on the application’s integrity.

The offline computation speeds up the whole process even with the many challenges that are sent and computed for. The web application does not have to wait for a user interaction to run the verification check for the application. This design is able to run the checks on a frequent basis with minimum interaction with the main UI thread. Since the scripts are imported and hashed on different web workers, the main thread is not halted for each verification cycle. The communication that goes through the main thread does not slow it down significantly as opposed to hashing the scripts on the thread itself.

Moreover, a different random challenge is sent in every iteration which protects it against replay attacks. Another feature of this module is that the arrangement of the Strings from each script depends on two bits of a unique challenge each time which is difficult for an attacker to locate without prior knowledge of the system. Furthermore, all the messages are masked before sending and unmasked when needed. When the application requires an integrity check the required result can be sent over to the server without delay as the process computes in the background repeatedly. To ensure that the added JavaScript files do not slow down the download times, all the scripts are minified before integration. The minification process removes all unnecessary code and reduces the size of the JavaScript file. This helps to reduce
bandwidth consumption of the web application and the reduced file sizes improve execution time.

This integrity verification module ensures that the client side code cannot be altered when the web application is live. The whole check is done during runtime and the amount of randomness makes it unpredictable.

Mitigation of Attacks Against the IVM

The integrity verification module adds onto the source code and has to be secured much like the other parts of the web application. The different attack scenarios that the IVM is prone to is discussed below.

The first and most important attack is overwriting functions in the IVM or preventing them from working. Since the code protected by this module is put into place with the help of functions and message passing, preventing them from working is beneficial to an intruder. Stopping the encryption and obfuscation process would mean that the Strings with the security functions would be sent without having been masked. This can have hazardous effect on the security of the module. Also, the web worker threads can be stopped after they are initialized with a security enabling function. Since workers run as threads, it is essential to keep track of all the threads during the operation to make sure that they have not been tampered with by an intruder. Trying to debug these background computations will also change the timing of the requests and responses. This is also taken into account in the integrity verification module.

The second scenario that could occur is that the messages are siphoned in transit. Since the application code and security code are kept separately, there has to be
communication between them to maintain the features of the code. These communications involve important messages with function parameters and results from the decryption functions. It is of absolute importance that these messages do not fall into the hands of the intruder before they are encrypted.

Another common attack type is tracking and debugging the challenge function. This is the function that produces the random seed or key to hash the Strings of the other functions. If this function can be stopped or changed, the security of the system will be undermined. Therefore, the challenge function has to be kept secure and live at all times. This can include adding separate tamper resistance functions to it.

Finally, we confirm that the module does not fall under a replay attack since it is a common form of attack scenario. In this scenario, the attacker uses past challenges and responses to trick the system into thinking that the intruder is a valid user. If the server verifies challenges from a preexisting list without factoring in time and sequence of the challenges, then it can fall under replay attacks.

The assets that need to be protected have shaped the type of attacks. The protection and mitigation strategies that are used to counter these attacks are discussed below.

**Overwrite/prevent functions.** The IVM will keep the functions from getting overwritten. The verification module is integrated in a way that it signs all the functions including the signing function continuously. Any changes made to the code is caught because no matter how inconsequential the change is, it would cause the hash values to mismatch. If the hash values do not match, the process is stopped with an error message. Moreover, preventing the hashing function from working is not beneficial to the intruder. If the intruder prevents the hashing functions that are
in the JavaScript file of the IVM, the system does not get the hash value that the server wants. This causes the server to throw an error stopping the intrusion process. Basically, any change to the code is recognized and can be stopped with the help of this approach.

**Siphon in transit.** The messages can be siphoned when they are sent out from the main page or the workers. The system deals with them by masking them before they are sent out. They are obfuscated to provide secondary security.

**Pin on challenge function.** Care is taken when implementing and integrating the challenge function as it is the first place to attack for an intruder to gain control. Since the challenge function is using a Math.random() function, it can be overwritten and an attacker can point it at their own function. Their function can then give them privileges over the entire system. However, all other Math.random() functions are deleted other than the system function which we use. Therefore, before calling our challenge function, we call a function that deletes all other overwritten functions with the same name. This protects the challenge function from getting pinned.

**Replay attack.** An attacker can try to use previous challenges to get known responses. They can also try to create a rainbow table of precomputed hash values. However, using a nonce to save each hash value prevents this attack. The values get stored with the nonce and unmasked once the server asks for a particular hash value related to a particular challenge. This also prevents man in the middle attacks and confirms that anybody without a valid session will not get access to the system.
3.5 Integration of Components in Web Applications

The integration of the two components to web applications is influenced by application types. In all cases, the security sensitive APIs and libraries used are taken into consideration before our components are added. These are left in as primary defense mechanisms.

Table 3.2 presents the various web applications that are integrated with new components. The 22 open source web applications vary from each other both in their functionalities and sizes. Since our approach encapsulates the JavaScript files during runtime to provide security, the average size of JavaScript files are presented here.

Our approach takes information from each web application to include in the input, output and whitelist policies. This was done in an iterative fashion. In every iteration, more APIs, functions, scripts are added to the whitelist. Once they are all accounted for, the security policies are added on top of the application’s present security code. It is then integrated into different parts of the application. Connections are forged for functions to trigger these policies both automatically and upon request depending on the implementation of the policies.

For each web application, our approach takes note of the different forms that are used on it. The input validation policy enforcement is customized for each input. Afterward, the output sanitization policy is addressed according to what data the web application shares with the user. For subresource integrity, all the third party scripts are taken account of and their hash values are added to the HTML header. Developers can customize the principle of least privilege policy according to their needs. However, for the purpose of this research, we ensure that users are assigned to different roles and only users with admin privileges have root access to databases.
Table. 3.2. Summary Characteristics of Web Applications Used in the Evaluation

<table>
<thead>
<tr>
<th>Type of web application</th>
<th>Average JavaScript file size (in KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Online reservation platform</td>
<td>348</td>
</tr>
<tr>
<td>Blogging application</td>
<td>600</td>
</tr>
<tr>
<td>Deployment tool</td>
<td>654</td>
</tr>
<tr>
<td>Podcast system</td>
<td>802</td>
</tr>
<tr>
<td>Status page system</td>
<td>532</td>
</tr>
<tr>
<td>Inter-network communication application</td>
<td>670</td>
</tr>
<tr>
<td>Note-taking application</td>
<td>409</td>
</tr>
<tr>
<td>E-commerce platform for online merchants</td>
<td>670</td>
</tr>
<tr>
<td>Greeting-sending application</td>
<td>349</td>
</tr>
<tr>
<td>Watsapp clone application</td>
<td>321</td>
</tr>
<tr>
<td>Pokemon go clone application</td>
<td>1270</td>
</tr>
<tr>
<td>E-learning application</td>
<td>491</td>
</tr>
<tr>
<td>Restaurant management system</td>
<td>1123</td>
</tr>
<tr>
<td>Donation application</td>
<td>359</td>
</tr>
<tr>
<td>Online tutorial application</td>
<td>352</td>
</tr>
<tr>
<td>Cross-platform chat application</td>
<td>432</td>
</tr>
<tr>
<td>Office attendance application</td>
<td>296</td>
</tr>
<tr>
<td>News outlet application</td>
<td>236</td>
</tr>
<tr>
<td>Data analysis application</td>
<td>243</td>
</tr>
<tr>
<td>Personal dash board</td>
<td>414</td>
</tr>
<tr>
<td>Event polling application</td>
<td>367</td>
</tr>
<tr>
<td>Meeting scheduling application</td>
<td>398</td>
</tr>
</tbody>
</table>

Plugin support is turned off when not needed and checks are turned on for every script used. The content security policy is customized for every web application. This is the last step for integrating the policies into the web applications. In this step, we ensure that the browser is informed about all the verified sources to get scripts from. The native web application protocols that are allowed and required to be used (e.g., HTTPS, SSL) are specified. The integration of this policy depends largely on how a developer would like to implement it according to the requirements of the web applications in question. For all the policies discussed here, developers need to
3.5. INTEGRATION OF COMPONENTS

configure them according to the needs of their web applications. The amount of extra work involved to integrate and enforce each policy differs from one policy to another as discussed above.

The integrity verification module acts on top of the security policies. Therefore, it is integrated after the security policies are in place. In our implementation of this module, the web workers are used to put different scripts of the security code on separate workers. The amount of code that is required to be protected by the IVM can be chosen by the developer. The main priority of this module is to sign and check the JavaScript code using web workers in the background. Therefore, a separate method is given to each script to get all the functions in them in a String form to pass to the signing worker. Also, the known good hash value is calculated for each file and saved on the server. Once these steps are completed, the two components of our approach are considered integrated and the web applications can be served. The extra work required by the developers to integrate this module involve dividing the JavaScript functions to be run on separate scripts and saving the hash values of the scripts on the server for comparison purposes.

3.5.1 Summary

This chapter presents the two components of our proposed approach that work together to secure web applications. The security policies work to fend off SQL injection attacks, XSS attacks and resource alteration attacks. The integrity verification module makes use of web workers to ensure client side code integrity during runtime. Integrating both these components on web applications protects them from some prevalent attacks and code tampering on the client side.
Chapter 4

Evaluation

The goal of this chapter is to showcase the experimental approach we undertook to compare the security of the web applications in question. It also presents the results we achieved for each component individually and together. The main criteria of the experiment was to make the attacks as practical as possible to get the results for our prevention methods. Throughout the evaluation process, we ask ourselves the following questions:

*How effective are the two proposed components in securing the web applications?*

*How is the proposed approach preventing client side code alteration during runtime?*

For the first question, we evaluate the web applications in terms of security. A comparative evaluation method is needed. For the second question, we need to check for changes to the code during runtime. Therefore, attacks performed and prevented during runtime are detected and prevention rates are recorded.

In Section 4.1, we define our experimental setup with a discussion on some of the
tools that are used. We outline our experimental environment in Section 4.2 and our evaluation criteria in Section 4.3. Our experiments and results are given in Section 4.4. A discussion of the results is presented in Section 4.5.

4.1 Experimental Setup

In this section, we present our experimental setup to acquire the comparative results. The setup is targeted towards taking advantage of the vulnerability of the web applications. At first, our proposed approach aims at preventing SQL injection, cross-site scripting, and resource alteration attacks through the security policies. Our experiments emulate these attacks on web applications.

To emulate these attacks, we used several open source tools. Below, we describe which tools were used along with why they were used to evaluate the web applications.

4.1.1 Selection of Tools for Evaluation

To run experiments and check for vulnerabilities, we choose tools to attack the web applications. These tools are chosen based on their ability to take advantage of vulnerabilities that might be present in the web applications and mimic attacks such as cross-site scripting, SQL injection and code tampering.

In Table 4.1, we present the tools that are used along with their purposes. All the tools in the table are used with a specific target in mind. At first, we check the web applications for overall security and vulnerability. This is done using Skipfish. This report is used to review and integrate security policies for SQL injection, cross-site scripting and resource alteration in the security policy component. Zed Attack Proxy is used for its vulnerability reports for the same purpose. It is also utilized for fuzzing
purposes. Fuzzing is defined by the act of providing web applications with different kinds of inputs with the intention of making it fail or crash so that vulnerabilities can be found. For fuzzing purposes, we employ attack scripts generated by a crawler. This is done with the use of the fuzzing tools. The tools use the scripts to create HTTP requests and records the responses for inspection. For more detailed injection and cross-site scripting attacks, along with fuzzing, Vega and JBroFuzz are employed.

**Table. 4.1. Evaluation Tools**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Intended Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vega [75]</td>
<td>SQL injection, header injection, cross-site scripting</td>
</tr>
<tr>
<td>Zed Attack Proxy [76]</td>
<td>Automated scanner, passive scanner, forced browsing, fuzzer</td>
</tr>
<tr>
<td>Skipfish [77]</td>
<td>Security threats, vulnerability report</td>
</tr>
<tr>
<td>JBroFuzz [78]</td>
<td>Automated fuzzing</td>
</tr>
</tbody>
</table>

**Vega**

This is an open source tool that is used to check for cross-site scripting vulnerabilities. It is written in Java and has a GUI. It can be employed to scan a single web page and customized to scan a whole web application. It presents us with a report about web application vulnerabilities and the attacks they might succumb to. The attacks we are interested in are XSS attacks and SQL injections.

**Zed Attack Proxy**

This tool is made by OWASP and it is an open source web application scanner. It is automated to find web application vulnerabilities. It was given the second place at the Bossie awards for the best open source networking and security software in 2015.
4.1. EXPERIMENTAL SETUP

It provides a set of manual tools along with the automated scanning system. It is used to get reports about vulnerabilities on the web applications.

Skipfish

This tool is used to check for vulnerabilities and create a report. The tool makes use of recursive crawling and dictionary-based probes to check for security holes. It then creates a sitemap with the active security checks. The report generated by Skipfish is used to gather data for the comparison of prevention rates against attacks. It also supports most popular web frameworks and mixed-technology sites which make it possible to evaluate all the open source web applications in our repository [80]. It has automatic learning capabilities and also makes use of differential security checks. This makes it very useful when checking for security holes in web applications of different functionalities.

JBroFuzz

This is another tool from OWASP that is used to secure web applications. It uses fuzzing to inspect vulnerabilities. JBroFuzz is customized for mimicking SQL injection and XSS attacks on all the web applications. This tool is also employed to verify that the input validation and output encoding that is integrated are up to the mark.

The use of these four tools along with manual inspection is aimed at evaluating the security policies that are integrated in the web applications. By employing automated attacks, we are able to do an exhaustive evaluation with possible inputs and outputs.
4.2 Experimental Environment

The environment setup for the experiments involved having the means to open, run and serve web applications. The machine used for this experiment has a core i7 intel processor with 4GB of memory. It has a 64 bit operating system with Windows 8.1 Enterprise edition and Kali Linux running.

To serve the web applications, we make use of the Apache Tomcat server. We use version 8.5.23 of this web server [81]. The web applications are run on the machine with the help of the web server application. We visit different parts of the web application to perform attacks and calculate the prevention rate.

4.3 Evaluation Criteria

In this section, we discuss the criteria on which we are evaluating our approach. The goal of the two components is to provide security for the web applications against prevalent attacks and prevent changes to the code. We chose prevention against vulnerabilities as the main criteria for the evaluation. Our experiments mimic real life attacks and the evaluation is done on the applications based on whether they are able to prevent these attacks.

The secondary criteria for the evaluation is to check if the IVM component can detect changes to the code during runtime. This is calculated by the number of attempts to make a change which is compared against the number of times it has been reported.

The attacks that our approach intends to protect the web application from are cross-site scripting attacks, SQL injection attacks and resource alteration. These attacks are initiated with the tools discussed in Section 4.1.1. The experiments are
conducted once before and once after the security policies and the integrity verification module are integrated into the web applications. The results from the change in attack prevention rates are discussed in the following section.

4.4 Experiments and Results

The experiments to show the success rate of this approach is conducted through the vulnerability scanner and attack tools. Each of the tools stated above are used on the web applications. This is done before any of the changes are made to the web applications. The results from the experiments are then saved as the “before” version of the experiments. Afterwards, the web applications are put through our approach and the security policies and integrity verification module are integrated to it. This makes the web applications less vulnerable and they are able to protect themselves from SQL injection attacks, cross-site scripting attacks and resource alterations. The web applications are put through the same experiments again after each component is integrated.

To emulate SQL injection attacks and XSS attacks, we use the fuzzers discussed above. For attack purposes, we choose attack inputs from two widely used sources to detect XSS and SQL injections which are XSS Cheat Sheet and SQL Injection Cheat Sheet [82, 71]. For each web application, the attacks are performed one hundred times, each time with a unique input. The number of times the web applications can prevent the attacks (out of one hundred) is calculated. The average of the number of preventions across all the web applications is presented as the prevention rate. For resource alterations, we alter the scripts the web application is expecting both before and after the page is loaded. This is attempted one hundred times for each
4.4. EXPERIMENTS AND RESULTS

The number of times each web application detects the alteration and prevents the scripts from executing (out of one hundred) is calculated. The average number of times the attacks are prevented across all the web applications are presented here. The average prevention rate is for all the web applications in the repository, each of which have been attacked one hundred times separately.

The protection evaluation is divided into three parts. The first part discusses the main flaws and vulnerabilities found in the web applications. The second part comes in after the security policies are integrated. It presents the added benefits and shows the results from the experiments. The third part reveals the results after both the components are integrated. This is considered as the final results for our approach. The three levels of evaluation are discussed below.

4.4.1 Pre-integration Protection Evaluation

To keep track of the vulnerabilities the web applications have before integrating our approach, we run our experiments on them. These experiments consist of the attacks we later secure the web applications from. The attacks in question are SQL injection, cross-site scripting and resource alteration attacks.

Figure 4.1 summarizes the initial experiment results. For each type of attack, we calculate how many were prevented. As discussed above, the attacks were performed one hundred times with unique parameters. The prevention rate was calculated by how many attacks from the hundred attacks were prevented. From Figure 4.1, we can see that the web applications have massively succumbed to attacks in the first round of experiments. The prevention rate is below 55 percent for all of the attacks. This includes common SQL injection attacks which are the most popular type of
injection attacks performed on web applications. For evaluation purposes, resource alteration is done by changing the scripts and files the web application is fetching. The prevention rate is the lowest for this kind of attack. This is mostly due to the fact that the web applications in question did not have a mechanism to check for altered scripts. Most web applications still let the scripts execute after alteration.

![Attack prevention rate prior to integration of the approach.](image)

**Fig. 4.1.** Attack prevention rate prior to integration of the approach.

These initial results are needed to calculate how well the applications have been secured after our approach has been integrated. After this step, we move on to the integration of our approach to the web applications in two steps. At first, the security policies are integrated and afterward the integrity verification module is added to the web application.

### 4.4.2 Protection Evaluation with Security Policies

The first part of our approach to securing the web applications is to integrate the security policies. The policies begin with input sanitization and output validation.
However, this component also includes policies to decide privilege and check for integrity. Lastly, the content security policy makes sure that the browsers have power over what to execute. All of these policies are integrated into the web applications in order.

To check if the integration of the security policies provide the web application with more security, we do another round of experiments. The protection evaluation is still based on one hundred unique attacks per attack type. The results are shown in Figure 4.2.

![Figure 4.2. Attack prevention rate after the integration of the approach.](image)

After the integration of our approach, the prevention rate of the web applications against prevalent attacks increases. For each of the unique attacks involved, the web applications detect and deflect them more than before. The prevention rate for resource alteration attacks shows the most positive change. This is due to the
subresource integrity policy ensuring that the external scripts cannot be altered. The resource alteration attacks are mostly successful when the security policies are not able to load before the attacks are performed. This happens when the attacks are initiated before the page with the security policies has loaded. Both SQL injection and cross-site scripting attacks are better defended than before. This is due to the exhaustiveness of the security policies. It can be mentioned that these two policies do not require that the developers are knowledgeable in web application security. These policies require that the developers have information about the expected behaviour of the web application in question.

4.4.3 Protection Evaluation with Integrity Verification Module

After the security policies have been integrated and evaluated for results, we shift our focus to protecting the client code with the integrity verification module.

The primary purpose of the IVM is to verify the integrity of the code while keeping the main UI responsive. To achieve this result, we employ web workers working in the background to sign code. However, with the variable number of web workers, the time to compute the integrity differs. Figure 4.3 presents the results of using multiple web workers in the web applications.

We start with just one web worker and increase it to 25. All of the workers communicate with the main page and have access to the cryptographic HMAC SHA256 signing code. Some might assume that having more workers would always speed up the process of verification. However, from the figure we can see that it is not the case. The fastest speed is at 16 web workers after which the times start to show inconsistencies. Adding on more web workers increases the time needed for computation.
The initial experiment is done with the two components in place but no web workers. From all the web applications put through this experiment, we take the average time it takes to do the integrity verification (IV). We find that on average it takes 4.3 seconds. This is a long time for the main UI to be waiting for results. This also sheds light on why developers, for the most part, do not have separate integrity checks in place. It slows down the user interface and for this 4.3 seconds, the user has to wait for a verification check before they can proceed with the next steps of the web application. As discussed in the introduction, this lag causes users to switch to other web applications and is not ideal for the owners of the applications. However, it can be seen from Figure 4.3 that integrating even one web worker to the verification process significantly lowers the time needed. Even with two web workers, time taken for the verification process is 3.7 seconds which is a 16 percent decrease in time. The
speed increases gradually with the added web workers. The highest average speed with the lowest number of web workers is 0.04 seconds with 16 web workers. After which, we see a slow decrease in speed of the IVM with added web workers. This change is in the millisecond range and goes up to 0.17 seconds with 25 web workers which is the highest number of web workers we use for this experiment. Even at 0.17 seconds, it is substantially lower than not using any web workers which would leave us at 4.3 seconds per round of integrity verification process.

Now that we know the optimum average number of web workers that can be used to get the fastest results, we alter the client side code during runtime to check how long it takes for the IVM module to report an attack. Here, sixteen web workers are used each cycle as was shown to be an optimum amount in our last experiment. For this experiment, we use multiple files of average 350KB size for checking the integrity. However, the time the attacks are initiated differs from one round of attacks to another. This is done because the first couple of iterations of the integrity verification process needs to happen before the hash value is ready for the client side. If the server requires the hash value for a challenge that has not been computed yet, the client side will have to compute it before it can be sent. After which, it can be compared and an error is reported, if any discrepancy exists. Here, we alter the client side code starting at 2 seconds after the page has been loaded. Figure 4.4 shows that for this attack, it takes the IVM 4.5 seconds to report the attack. The results are computed for any one web worker reporting an error. The first time a web worker reports an error is taken into account and recorded.

There are three things that need to happen in one cycle of integrity verification. They are completing a challenge-response cycle, storing the hash value in the hash
table and verifying the hash value against the known good value. The time to perform integrity verification depends on all the three. As the attacks occur later in time compared to when the scripts are loaded, the error reports come in sooner. This happens because the verification mechanism gets time to perform several cycles before a check is required. Therefore, the hash results are precomputed and sent in without delay when there is an integrity verification request.

![Graph showing time required for IVM with varied attack initiation times](image)

**Fig. 4.4.** Time required for the IVM with varied attack initiation times.

Initially, when the attack happens at 2 seconds, it takes the IVM 4.5 seconds to report it. However, when the code alteration attack is initiated at 5, 10 or even 20 seconds after the page load, we can see a steady decline in the report times. Here, time taken to report takes around 3 seconds. This goes down to around 2.5 seconds when the attack happens at 25 seconds and continues to go down. The results show that with the increase in time after the page load the integrity verification module can report the attacks faster. This is due to the fact that the challenge-response protocol
has been run multiple times at this point and the verification process can report on the integrity with the results stored. Even though the application takes more time to report if the attack is initiated early on, the application is still protected from being tampered with.

### 4.5 Discussion

The experimental results show that the proposed approach for securing the web applications with the use of the two components is promising. Table 4.2 shows the percentage increase in prevention rates of all the individual attack scenarios. The average percentage increase is around 32 percent from just integrating the security policies. This is a major change from the initial state of security of the web applications. The prevention rates are calculated from the average rates across multiple web applications which range in varying sizes, usability and complication levels. These results show how the introduction of our approach to secure the web applications makes them less vulnerable to the above mentioned attacks.

<table>
<thead>
<tr>
<th>Attacks</th>
<th>Increase in prevention rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL injection</td>
<td>24</td>
</tr>
<tr>
<td>Cross-site scripting</td>
<td>31</td>
</tr>
<tr>
<td>Resource alteration</td>
<td>43</td>
</tr>
</tbody>
</table>

The integrity verification module adds its own security features. It makes sure that all the client code is signed and checked during runtime. If any of the client code is altered, this module lets the user know immediately. This verification process on
top of the security policies secure the web application client side in a comprehensive manner. This module helps secure web applications by making sure that the client side code cannot be tampered with. The results from the experiments also show how web workers can help with gaining speed while still verifying the integrity of the client side code. Multithreading also improves the web application to be less susceptible to attacks through the increased obfuscation it provides in the verification process.
Chapter 5

Conclusion and Future Work

In this chapter, we summarize our research work and contributions. We also discuss the limitations of the proposed approach and present our future plans.

5.1 Summary

The primary goal for this thesis is to provide an approach to secure web application client sides. To achieve this goal, we make use of secure coding practices to protect the application against some named attacks using predetermined security policies. The secondary goal is to verify the integrity of the client side code during runtime. To attain this goal, we design and implement an integrity verification module that checks code integrity during runtime. The two components have different functions for providing security. From secure coding practices and secure design principles, we come up with security policies. These policies work together to protect the web application against some prevalent attacks namely SQL injection, cross-site scripting and resource alteration. These policies include input and output sanitization, principle of least privilege, sub resource integrity and content security policy. The integrity of the JavaScript files is also checked in this component. Not having the same hash
value as the known good hash value or not having the right permissions prevents the JavaScript files from executing. This ensures that attacks such as cross-site scripting and injection attacks are prevented. An integrity verification module is presented in this thesis along with its working environment and model design. Equipped with its own hashing function, it confirms that the code on the client side is not tampered with. The web workers API is used to deal with the verification process concurrently in the background. The security provided by the applications is evaluated and the results are presented. From the analysis of the results, we can say that the security provided for the prevention of attacks and code tampering on client side is significantly better with the use of our approach. We have also presented the optimum number of web workers to be used to get speedy results from the verification module in our experiments. The experiments have also shown average time taken to report a code tampering attack depending on attack initiation times.

The contributions of this thesis include:

1. The development of security policies with the help of secure coding practices and secure design principles. These policies act as a guideline for web application developers to protect their applications from different types of attacks. Although not all the policies are novel, the manner in which they are organized with implementation guidelines for specific web application attacks is original. Moreover, the evaluation information that is presented is meant for developers to use in factoring in what approaches to use to secure their web applications. The results can act as a guide for web application developers in the prevention of similar attacks.

2. The development of an integrity verification module (IVM). This module uses the application of message authentication code tied with a challenge-response protocol
to secure the code of web application client sides. The use of web workers adds to the novelty of this design. The continuous check in the background provides web applications with prevention methods for code tampering.

The proposed approach incorporates both static and runtime checks with constraints on both the client and server sides. The communication channels designed for exchanging information between different parts of the web application were put into place as an improvement to the existing systems. As it is evident from the initial experiments done on the web applications in our experimental section, most applications lacked a secure library or policies to counter common attacks such as XSS attacks, SQL injection or code tampering attacks. The code in use in the web applications can be integrated with new components that are part of our approach to provide better security. The components which contain security policies and an integrity verification module help protect web applications against unwanted attacks and also keep the client side code secure by signing and checking them continuously. This is done with the help of web workers in the background. Every aspect in the system that was added was analyzed for attacks and the results have been presented.

The protection provided by our approach shows an average of 33 percent increase in attack prevention rates. The integrity verification module reports code tampering attacks as fast as around 2.5 seconds with proper multithreading.

5.2 Limitations

The web applications that we used for this research work are all open source. They range from inventory managers to invoicing systems and are built on different platforms with JavaScript used on the client side. However, throughout the experiments,
we did not keep track of how our approach affected each web application in practice. The average attack prevention rate results across all the web applications in our repository were presented instead of each individual web application. In this work, we assume that developers know the expected inputs and outputs for the web applications. Without this, integrating the security policies will be complicated and might cause security holes. For our approach to be useful, the client side code needs to be loaded without errors or changes. If the files containing the security policies and integrity verification module are prevented from loading, then this approach will fail to secure the web application. Moreover, the client side of web applications will always be more vulnerable and difficult to secure than server sides. The security policies provide the first line of defense against attacks on the web applications. The policies are designed to thwart attacks for web applications that do not have more comprehensive security systems in place. However, for a well rounded security system, policies should be placed on both the client and the server side that work simultaneously to provide security to web applications.

5.3 Future Work

We present two areas of future work that will increase the usability of this approach and the security it provides.

As discussed in the contribution section of this thesis, the security policies that are presented cover an array of attack prevention methods for web applications. The policies are chosen from relevant research and OWASP secure coding practices [12]. There was no specific optimization done in terms of integrating these policies with the web applications in mind. In the future, more information can be collected for each
5.3. FUTURE WORK

type of web application to put in specific sets of rules and constraints when integrating the policies. This will lead to more secure applications with fewer development patches needed.

For the integrity verification process, web workers are made to communicate with each other over message channels. Strings made up of entire classes of JavaScript code are passed back and forth in these channels. The signing process is not possible without the use of at least one of these channels. However, only the minimum number of these channels are used to connect different files of the web application to the signing code and main page. Introducing more message channels that carry encrypted code back and forth can make the system more scalable. This can also be used as a means for obfuscation. Compartmentalizing the client side code and using message channels for communication can be explored more in the future.
Bibliography


