Securing Web Applications with Secure Coding Practices and Integrity Verification

Arafa Anis*, Mohammad Zulkernine*, Shahrear Iqbal*, Clifford Liem†, Catherine Chambers†
*Queen's University, Kingston, Ontario, Canada
{iqbal,mzulker,anis}@cs.queensu.ca
†Irdeto, Ottawa, Ontario, Canada
{clifford.liem,catherine.chambers}@irdeto.com

***Abstract*—The concept of security in web applications is not new. However, it is often ignored in the development stages of the 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 implement features and often do not practice secure coding. Therefore, countless web applications are launched with security vulnerabilities like cross-site scripting, injection attacks and resource alterations. In addition, code tampering on the client side is a serious security risk for web applications. In our opinion, integrating security features should be a part of the development process. Without practicing secure coding and having an integrity verification system in place, it is difficult to defend security attacks. In this paper, we present a system that helps developers to implement security measures on the client side code based on the best practices of secure coding. We also develop an integrity verification module to prevent code tampering attacks on the client side. The proposed approach can be integrated with both new and existing web applications. We implement our approach for a number of JavaScript-based applications and the results show that our approach increased the security of the applications and prevented any modifications performed on the client side.**

I. Introduction

Web-based applications are being rapidly deployed through the Internet as these applications are easy to develop and deploy. However, information security and privacy issues are not always taken into consideration properly. 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 [1], 89 percent of web attacks had a financial or espionage motive. Attacks can be conducted and sensitive information may be gathered while leaving little or no forensic evidence [2]. Often, security becomes a priority only after a security breach. Therefore, it is very important to develop web applications that have proper security measures in place and protect the applications that have already been deployed.

Web applications can be attacked on both the client side and the server side including any third parties that are involved in the process. However, security is often enhanced on the server side while keeping the client side open to threats [3]. 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 attackers at runtime. Therefore, the code that is added to make the client side secure requires 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 focus has shifted from the server-side to the client side [4], [5]. Among the vulnerabilities mentioned in their most recent reports, SQL injection and cross-site scripting (XSS) are prevalent in both the reports. Another type of attack is resource alteration which occurs when servers and Content Delivery Networks (CDNs) fail to deliver scripts and stylesheets in their original state. These resources get altered by the attackers.

A web application also needs to verify that the code included for security purposes is not altered during runtime. For this, the client side code needs to be checked for integrity and verified against known good versions. However, client side verification mechanisms can compromise the responsiveness of the application [6] which is a challenge when designing mechanisms for ensuring integrity.

In this paper, we introduce an approach to secure web applications by providing guidelines to the developers to prevent SQL injection, XSS, and resource alteration attacks. For a list of secure coding methods, we have consulted OWASP and recent literature for recommendations [7]. From these recommendations, we derive security policies that can help secure a web application against above mentioned attacks. Furthermore, to verify the integrity of the code during runtime, we propose an integrity verification module.

The integrity verification module (IVM) is designed to prevent client side code modifications. It executes on separate threads in the background to maintain the responsiveness of the web application.

In particular, the contributions of this work are as follows:

- We develop security policies for web applications to prevent prevalent attacks. We show that how secure coding practices, when implemented properly, can provide security to web applications.

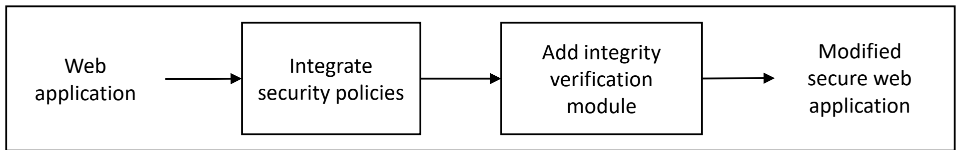

Figure 1. High-level work flow for securing web applications.

- We develop an integrity verification module that prevents code tampering at runtime. 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 during runtime to validate its authenticity.

The remainder of the paper is structured as follows. We discuss the related work in Section II. Section III describes our approach and Section IV presents the evaluation results. We conclude the paper in Section V.

II. Related Work

Recent works on web application security involve end-to-end security schemes and protection of the applications through code instrumentation [8]. Web application scripts are among the most vulnerable parts of a web application. Web application defenses, however, do not always flow from the client end since it is vulnerable by itself. However, purely client side defense mechanisms do exist that enforce policies without the help of the server [9]. Detection of malicious code can be successfully carried out on this side [10].

For preventing SQL injections through user inputs, researchers have proposed many defense mechanisms and frameworks. Black-box testing and code checking have been the front runners when it comes to detecting SQL injection attacks. Although black-box testing does not require knowledge of the code base, it does not assure completeness while checking for vulnerabilities [11]. Combined static and dynamic analysis as seen in AMNESIA [12] employs model-based 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 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 the queries, the models will not be able to identify actual attacks during runtime.

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 deployed on the server side. Interpretor-based approaches for PHP are presented in [13]. With boundary injection and policy generation, Shahriar et al. [14] presented 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.

Code tampering on the client side has been addressed through information flow analysis [15] in a 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 extensions of the server side methods. JavaScript integrity has been overlooked for document structure integrity and integrity of content caching [16]. These research do not protect the client code from alteration during runtime. This has been a long running problem in web applications. Wan et al. [17] presented a solution for verifying the integrity of application cache in android runtime to defend 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 benefited 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. In contrast, 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 addition, the integrity verification module ensures that the integrity of the client side protection code.

III. Secure Coding Practices and Integrity Verification

The proposed approach for integrating security into web applications has two components. First, we have the security policies and then the integrity verification module. Figure 1 illustrates a high-level work flow of our approach. At first, guidelines are provided for integrating security policies. After that to ensure that the client side code cannot be altered, the integrity verification module is added.

A. Security Policies

The security policies deal with how data on web applications are used and how vulnerable the applications are to an attacker. Here, we assume that the client or the user 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. Table I presents how each of the policies are mapped to specific attacks. Below, we describe these policies.

Table I
TYPES OF ATTACKS TO POLICY MAPPING

Policy	Attack type
Input Sanitization	SQL injection, XSS attack
Output Validation	SQL injection, XSS attack
Principle of Least Privilege	SQL injection, XSS attack, Resource Alteration
Subresource Integrity	Resource Alteration
Content Security	XSS attack

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. 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 [18] and Django Validator [19] 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 to put untrusted data in HTML documents unless it is required by the application. 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 [20] 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.

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, we recommend 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.

Output Validation. 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 the application. 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.

Principle of Least Privilege. The principle of least privilege is a policy derived from OWASP security by design principles [21]. 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. 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 [22]. 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. We highly encourage 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.

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 [23]. 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 2 presents a script tag with the SRI attribute.

```
<script src=“https://code.securesource.com/hmac243.js”
Integrity=“sha256-
161ec3c2b4036a4a54aa85de18e8c3fd9be5e6e5e6647a74
2771Bd2c95d78b7”
crossorigin=“anonymous”> </script>
```

Figure 2. Subresource integrity confirmed with HTML attribute.

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. 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. Since a web application might have content from different sources, there needs to be a list to check against content sources before using them. Content Security Policy (CSP) provides this with the means of a whitelist. It is a HTTP header that is recommended by the Web Application Security Working Group [24]. 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.

All of these policies, when enforced and working in sync, increase the security against cross-site scripting, SQL injection and resource alteration attacks.

B. *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 module is designed based on the challenge-response protocol as depicted in Figure 3. It is designed to secure the verification module and to maintain obfuscation during runtime. The three major elements of the figure are the page running the main UI thread, the web worker with the hashing function and multiple web workers denoted with the number (#) sign.

In this module, the key for the hashing function is sent as a challenge in 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 a web worker with the 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 to the web workers with the security code (2).
- 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 as a response (3, Response_i 1).
- 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 the Strings from the workers and itself to the web worker containing the hashing function (4, Response 1||Response_i 1).
- 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.

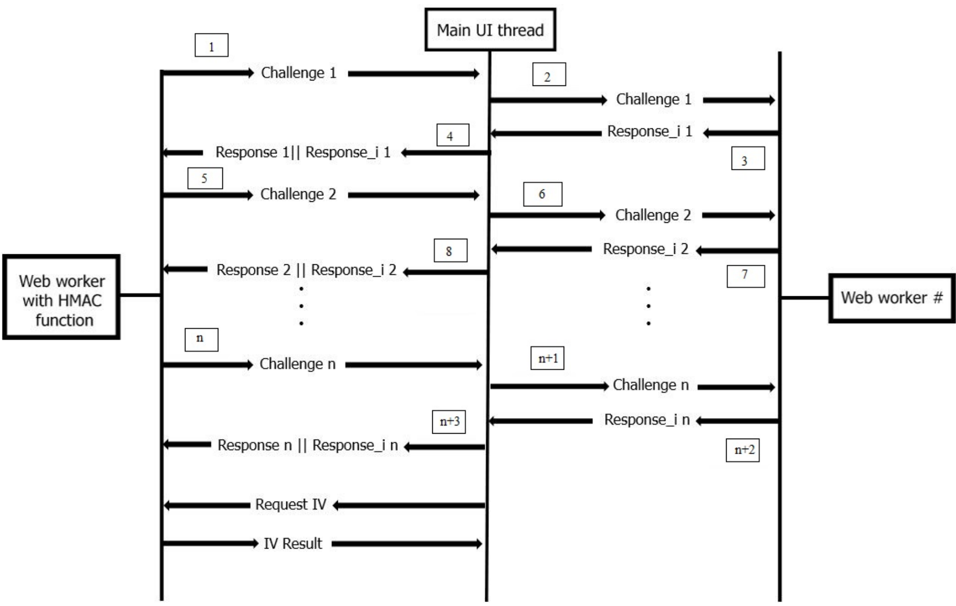

Figure 3. Challenge-response protocol in the integrity verification module.

- 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 the web worker saves it in a similar procedure as described 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.
- 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 user interactions to run the verification check. This design is able to execute the checks on a frequent basis with minimum interaction with the main UI. Since the scripts are imported and hashed on different web workers, the main thread is not halted in the verification cycles. This ensures that the responsiveness of the application is not compromised.

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 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 to the server without any 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.

IV. EVALUATION

In this section, we present our experimental setup and evaluation results. To emulate attacks, we used several open source tools. 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 II, we present the tools and Table III presents 22 web applications that are modified using our approach. 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, the average size of JavaScript files are presented here.

Table II
EVALUATION TOOLS

Tool	Intended Use
Vega [25]	SQL injection, header injection, cross-site scripting
Zed Attack Proxy [26]	Automated scanner, passive scanner, forced browsing, fuzzer
Skipfish [27]	Security threats, vulnerability report
JBroFuzz [28]	Automated fuzzing

Table III
WEB APPLICATIONS USED IN THE EVALUATION

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

A. Experimental Environment

The machine used for this experiment has an intel core i7 processor and 4GB of RAM. To serve the web applications, we use the Apache Tomcat server (version 8.5.23) [29].

B. Experiments and Results

The experiments to show the attack prevention rate of our approach is conducted through the vulnerability scanner and attack tools. 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 [30], [31]. For resource alterations, we alter the scripts the web application is expecting both before and after the page is loaded. 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.

1) Protection Evaluation with Security Policies: Figure 4 summarizes our experiment results. For each type of attack, we calculate how many of them are prevented. As discussed above, the attacks were performed one hundred times with unique parameters. From Figure 4, we can see that the web applications have massively succumbed to attacks before we integrate our approach. The prevention rate is below 55 percent for all the attacks. 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.

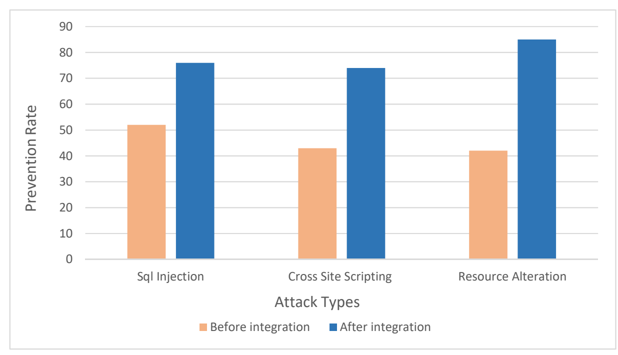

Figure 4. Attack prevention rate before and after the integration of our approach.

After the integration of our approach, the prevention rate of the web applications against the attacks increases. 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.

2) Evaluation of the Integrity Verification Module: Figure 5 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 the workers communicate with the main page and have access to the cryptographic HMAC SHA256 signing code. Someone might assume that having more workers would always speed up the process of verification.

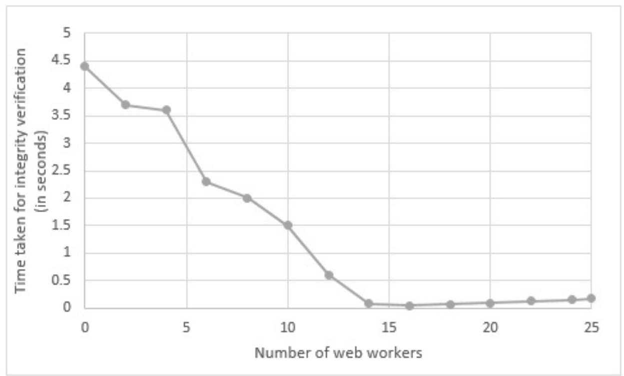

Figure 5. Time required for integrity verification (IV) with varying number of web workers.

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.

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. However, it can be seen from Figure 5 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.

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. 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 need 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 6 shows that for this attack, it takes the IVM 4.5 seconds to report the attack. 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.

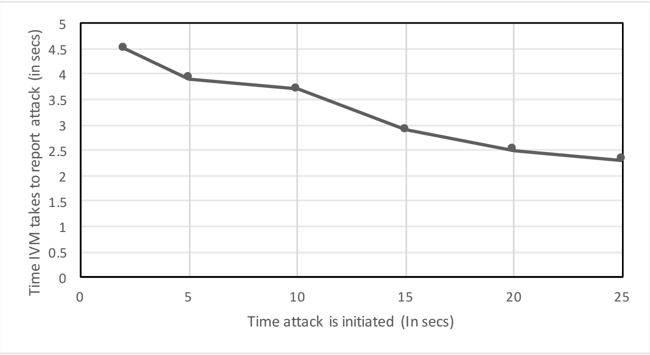

Figure 6. Time required for the IVM with varied attack initiation times.

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. The results show that the integrity verification module can report the attacks faster with the increase in time after the page load. Even though our approach takes more time to report if the attack is initiated early on, the application is still protected from being tampered with.

C. Discussion

The experimental results show that the proposed approach for securing the web applications is promising. Table IV 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. The integrity verification module makes sure that all the client code is signed and checked during runtime. If the client code is altered, this module lets the user know immediately. The results also show how web workers can help with the responsiveness while still verifying the integrity of the client side code.

V. Conclusion

It is becoming extremely important to secure web application client sides. To achieve this goal, we make use of

Table IV
INCREASE IN ATTACK PREVENTION RATE

Attacks	Increase in prevention rate
SQL injection	24
Cross-site scripting	31
Resource alteration	43

secure coding practices to protect the application against prevalent attacks using predetermined security policies. The secondary goal of this work 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 security policies work together to protect the web application against SQL injection, cross-site scripting and resource alteration attacks. The policies include input and output sanitization, principle of least privilege, sub resource integrity and content security policy. The integrity of the JavaScript files is checked by the integrity verification module. The protection provided by our approach shows an average of 33 percent increase in attack prevention rates. Also, the integrity verification module reports code tampering attacks as fast as around 2.5 seconds with proper multithreading.

ACKNOWLEDGMENT

This work is partially supported by Mitacs Canada and Irdeto Canada.

REFERENCES

[1] "2016 DBIR: Understand Your Cybersecurity Threats." [Online]. Available: http://www.verizonenterprise.com/verizon-insights-lab/dbir/2016/

[2] J. B. D. Joshi, W. G. Aref, A. Ghafoor, and E. H. Spafford, "Security Models for Web-based Applications," *Communications of the ACM*, vol. 44, no. 2, pp. 38–44, Feb. 2001.

[3] P. De Ryck, L. Desmet, F. Piessens, and M. Johns, *Primer on Client-side Web Security*. Springer, 2014.

[4] "Category:OWASP Top Ten Project - OWASP." [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#tab=OWASP_Top_10_for_2010

[5] B. Martin, M. Brown, A. Paller, and D. Kirby, "CWE - 2011 CWE/SANS Top 25 Most Dangerous Software Errors." [Online]. Available: http://cwe.mitre.org/top25/

[6] R. C. Marchany and J. G. Tront, "E-commerce Security Issues," in *Conference on System Sciences*. IEEE, Jan. 2002, pp. 2500–2508.

[7] "OWASP Secure Coding Practices Checklist - OWASP." [Online]. Available: https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_Checklist

[8] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring," in *Conference on World Wide Web*. ACM, 2003, pp. 148–159.

[9] L. A. Meyerovich and B. Livshits, "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser," in *Symposium on Security and Privacy*. IEEE, May 2010, pp. 481–496.

[10] O. Hallaraker and G. Vigna, "Detecting Malicious JavaScript Code in Mozilla," in *Conference on Engineering of Complex Computer Systems*. IEEE, Jun. 2005, pp. 85–94.

[11] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the Art: Automated Black-Box Web Application Vulnerability Testing," in *Symposium on Security and Privacy*. IEEE, May 2010, pp. 332–345.

[12] W. G. J. Halfond and A. Orso, "Preventing SQL Injection Attacks Using AMNESIA," in *Conference on Software Engineering*. ACM, 2006, pp. 795–798.

[13] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically Hardening Web Applications Using Precise Tainting," in *Conference on Security and Privacy in the Age of Ubiquitous Computing*. Springer, May 2005, pp. 295–307.

[14] H. Shahriar and M. Zulkernine, "S2xs2: A Server Side Approach to Automatically Detect XSS Attacks," in *Conference on Dependable, Autonomic and Secure Computing*. IEEE, Dec. 2011, pp. 7–14.

[15] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: Policy-reduced Integrity Measurement Architecture," in *Symposium on Access Control Models and Technologies*. ACM, 2006, pp. 19–28.

[16] Y. Nadji, P. Saxena, and D. Song, "Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense." in *Symposium on Network and Distributed System Security (NDSS)*, Jan. 2009, pp. 1–20.

[17] J. Wan, M. Zulkernine, P. Eisen, and C. Liem, "Defending Application Cache Integrity of Android Runtime," in *Conference on Information Security Practice and Experience*. Lecture Notes in Computer Science, Springer, Dec. 2017, pp. 727–746.

[18] "Validator Commons Validator." [Online]. Available: https://commons.apache.org/proper/commons-validator/

[19] "Validators - Django documentation." [Online]. Available: https://docs.djangoproject.com/en/2.0/ref/validators/

[20] "Validate.js." [Online]. Available: https://validatejs.org/

[21] "Security by Design Principles - OWASP." [Online]. Available: https://www.owasp.org/index.php/Security_by_Design_Principles#Principle_of_Least_privilege

[22] D. Wichers, J. Manica, M. Seil, and D. Mishra, "SQL Injection Prevention Cheat Sheet - OWASP." [Online]. Available: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

[23] S. Gupta and B. B. Gupta, "Cross-Site Scripting (XSS) Attacks and Defense Mechanisms: Classification and State-Of-The-Art," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 1, pp. 512–530, Jan. 2017.

[24] "Content Security Policy CSP Reference & Examples." [Online]. Available: https://content-security-policy.com/

[25] "Vega." [Online]. Available: https://tools.kali.org/web-applications/vega

[26] "OWASP Zed Attack Proxy Project - OWASP." [Online]. Available: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[27] "Skipfish." [Online]. Available: https://tools.kali.org/web-applications/skipfish

[28] "JBroFuzz - OWASP." [Online]. Available: https://www.owasp.org/index.php/JBroFuzz

[29] "Apache Tomcat." [Online]. Available: http://tomcat.apache.org/

[30] "XSS (Cross Site Scripting) Prevention Cheat Sheet - OWASP." [Online]. Available: https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

[31] "SQL Injection Prevention Cheat Sheet - OWASP." [Online]. Available: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet