Detection and Prevention of Changes in the DOM Tree

by

Junaid Iqbal

Bachelor of Science in Information Technology, BZU, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor:	Natalia Stakhanova, PhD, Faculty of Computer Science
Examining Board:	Rongxing Lu, PhD, Faculty of Computer Science, Chair
	Dima Alhadidi, PhD, Faculty of Computer Science
	Rickey Dubay, PhD, Department of Mechanical Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

August, 2018

©Junaid Iqbal, 2018

Abstract

The current generation of client-side Cross-Site Scripting filters are mostly *browser-based* tools and do not allow the web developers to control authorized or unauthorized modifications of the web page's Document Object Model (DOM). In this thesis, we propose a *policy-based* and *browser-based* protection mechanism to detect and prevent unauthorized tampering with the DOM. To examine the efficiency and feasibility of our approach, we implement the proposed solution in an open source web browser, Chromium. Our proposed approach has little performance overhead and effectively detects malicious modifications of the DOM. We also conduct a thorough analysis of the current state-of-the-art *policy-based* MutationObserver API and uncover a set of limitations.

Dedication

This thesis work is dedicated to my wife, Mariam, for supporting me through the ups and downs of life. I am truly blessed for having you in my life.

This work is also dedicated to my parents for their unconditional love, numerous sacrifices, and blessings.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Natalia Stakhanova for her guidance, support, and encouragement throughout the entire process. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly. I am grateful to my thesis committee members, Dr. Rongxing Lu, Dr. Dima Alhadidi, Dr. Rickey Dubay, and Dr. Patricia Evans, who were more than generous with their expertise and precious time. A special thanks to my colleague, Dr. Ratinder Kaur for the contribution of her ideas in this research. I would also like to thank the members of the faculty for their support. Finally, I must express my profound gratitude to my parents, and my wife who endured this long process with me. This accomplishment would not have been possible without them. Thank you.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent times, client-side web applications have become more popular because of faster response time and low burden on servers' resources. The JavaScript language has become a de-facto standard for developing these types of applications. It can be used to build games, browsers or even an operating system [1, 4, 16, 18]. The most popular applications such as Gmail or Google Docs are fully developed using JavaScript. In 2017, it was the most popular language on the GitHub platform [17]. Figure 1.1 shows the fifteen most popular languages in 2017 on GitHub by opened pull request.

Due to a tremendous increase in the development and usage of JavaScript applications, it attracts the attention of attackers. Since users' personal and sensitive information is transferred through these applications, security becomes an important concern. Currently, JavaScript based applications have many security holes including Cross-site Scripting (XSS). The XSS attacks

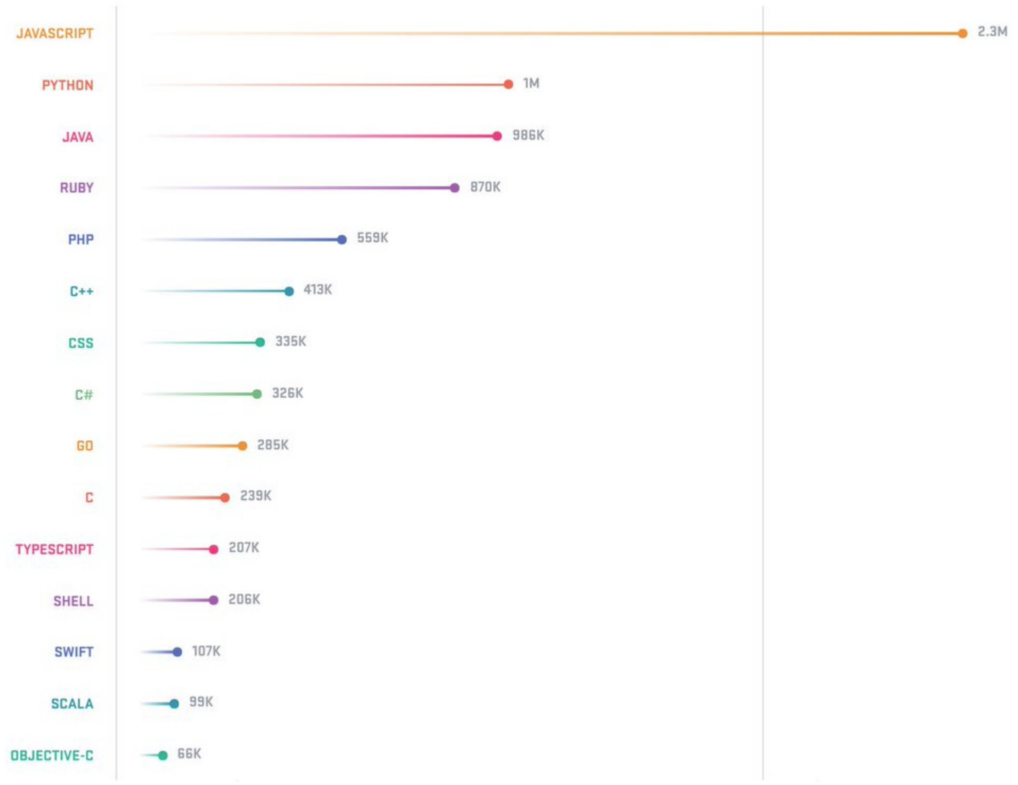
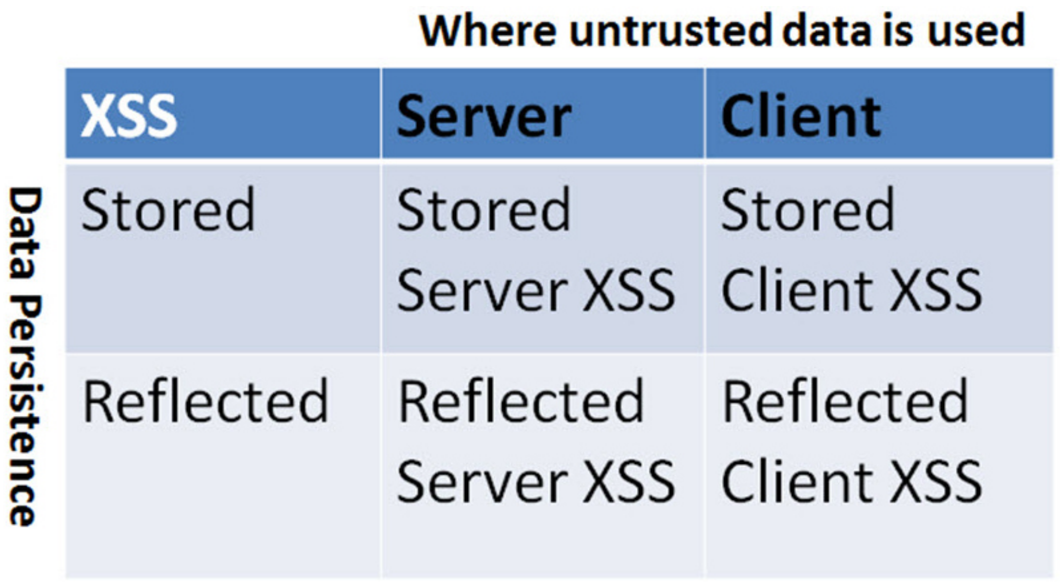


Figure 1.1: Fifteen most popular languages on GitHub in 2017 by opened pull request [17]

occur when user supplied data are included in a web application without proper validation.

Ever since its initial discovery in 2000 [33], XSS has been a very common security vulnerability and is found in around two thirds of all web applications [20]. The task of detecting these attacks is becoming more challenging as attackers can use various JavaScript obfuscation techniques [45] to hide the malicious payload. Obfuscation is a process of transforming the original code into a form that is difficult to read, understand, or reverse engineer while retaining the actual functionality of the code. Transforming the malicious JavaScript requests helps attackers to bypass certain XSS defensive filters [27].

XSS can be broadly categorized into two types [22]: Server XSS and Client XSS. Server XSS attacks occurs when user supplied data that could be malicious are included in the HTML response generated by the server. Whereas, the Client XSS attacks occurs when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call. Both Server XSS and Client XSS can be Stored and Reflected as shown in Figure 1.2.



Where untrusted data is used

	XSS	Server	Client
Data Persistence	Stored	Stored Server XSS	Stored Client XSS
	Reflected	Reflected Server XSS	Reflected Client XSS

Figure 1.2: Types of Cross-Site Scripting (XSS) [22]

Reflected XSS attacks generally occur when the attack payload is sent to the web application and immediately reflected back to the web browser in the form of search results or error messages without the data being sanitized. In this type of XSS, the data are not stored on the client or server. Stored XSS attacks occur when untrusted data are stored on the client or server side for later use. Data are not properly validated and when the victim requests the stored information, he is able to retrieve the malicious data.

Server XSS have been known for a number of years while Client XSS, also termed DOM(Document Object Model)-based XSS, was first introduced in 2005 [36]. The Document Object Model (DOM) is a programming API for accessing and modifying HTML and XML documents. DOM is a way JavaScript code can communicate with HTML elements on a web page. In DOM-based XSS, the attack payload is executed as a result of modifying the DOM environment in the victim's browser used by the original client-side script, so that the client-side code runs in an unexpected manner [19].

DOM-based XSS attacks are different from Server XSS attacks because the HTML source code and response of the attack stays the same, but the client-side code executes differently due to malicious modifications by the attacker, whereas in Server XSS, vulnerable payload can be seen in the response page. DOM-based XSS vulnerabilities exist entirely in the client-side JavaScript and are becoming more and more of a threat as web applications are providing increasingly richer client-side functionality.

DOM-based vulnerabilities are very difficult to detect because the target of the malicious payload is the browser, as compared to other attacks where target is the server. In a recent study, it was shown that nearly 10% of Alexa Top 5000 websites contain at least one DOM-based XSS vulnerability [37].

Most of the popular web applications such as Google+ and Twitter have been vulnerable to DOM-based XSS attacks [21, 2, 8].

Server-side attack detection and prevention systems are not able to provide protection against DOM-based XSS attacks because there are some cases in which attack payload cannot be seen by the server, for example, in an HTTP request like *http://www.example.com/ test.html#⟨script⟩alert(1)⟨/script⟩*, the malicious payload which appears after a # character is not sent by the web browser to the server. Figure 1.3 shows the graphical representation of DOM-based XSS attack.

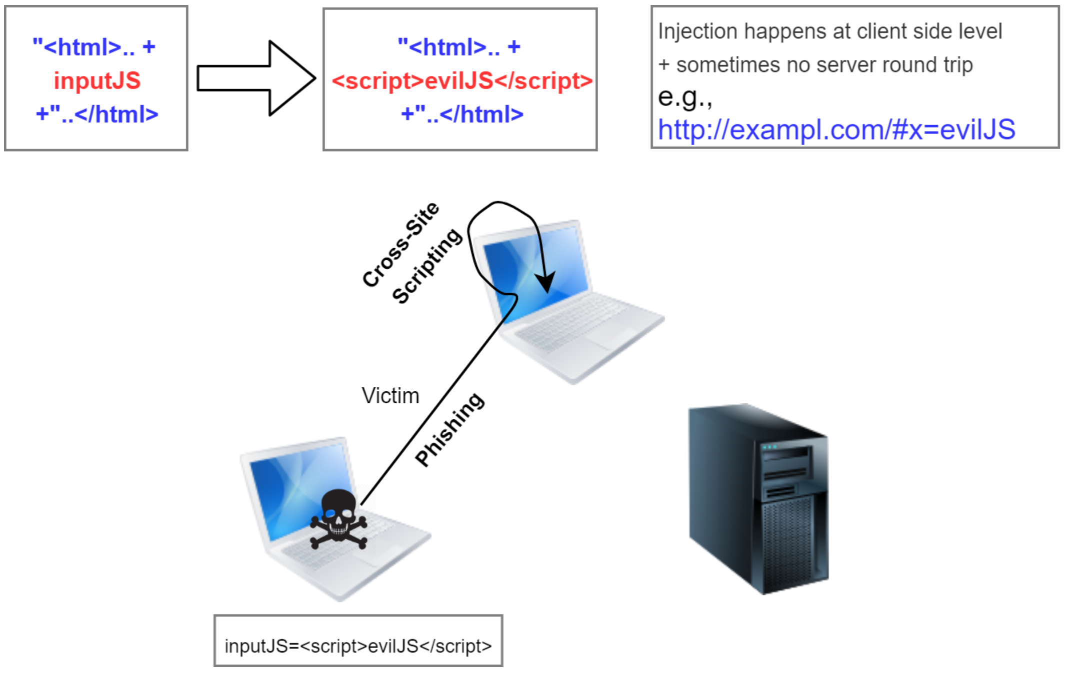

"<html>.. +
inputJS
+"..</html>
"<html>.. +
<script>evilJS</script>
+"..</html>
Injection happens at client side level
+ sometimes no server round trip
e.g.,
http://exampl.com/#x=evilJS
Cross-Site Scripting
Victim
Phishing
inputJS=<script>evilJS</script>

Figure 1.3: DOM-based XSS Attack

To provide protection against this type of attacks, many state-of-the-art *browser-based* tools are designed and implemented, e.g., XSS Auditor [26] and DOMinator [29]. XSS Auditor is a browser-based tool that automatically detects and prevents XSS attacks. The Auditor employs string comparison to detect the injection of malicious strings. However, it has several shortcomings related to DOM-based XSS [44] including not being able to detect attacks if malicious code was not found during the initial parsing of HTML response or allowing the attack to mislead the string-matching algorithm to bypass the filter. To evade DOM-based XSS, inspections at DOM level are necessary. However, the Auditor works only at the parsing level and therefore is not able to fully detect DOM-based XSS attacks.

DOMinator is a Firefox browser-based tool that utilizes dynamic taint tracking to identify DOM-based XSS vulnerabilities. MutationObserver [13] is another solution focus on detecting changes in the DOM. However, it notifies after the change has already occurred. Both DOMinator and MutationObserver focus only on the detection part rather than prevention. In this work, we address this gap.

The existing browser-based tools do not have the capability to differentiate between authorized or unauthorized DOM modifications. They automatically detect and prevent the change without considering the security requirements of web applications. They lack server-side control and do not allow the web developer to specify the authorized or unauthorized modifications in the DOM.

In this work, we have pursued the following research question:

- Can we protect web applications from unauthorized tampering with a web page's DOM by providing server-side control as well as automatic client-side protection?

1.1 Our Approach

To answer the above research question, we aim to provide a novel protection mechanism to detect and prevent unauthorized tampering with the DOM. The protection mechanism makes use of the enriched policy language *DOM Security Policy* (DSP) that can be used by web developers to define security policies. The policies can be applied on HTML elements as per the security requirements of the web application. The policies are delivered to the web browser through a new HTTP header named "DOM-Security-Policy." The *DOM Monitoring Module* is designed and implemented to enforce policies on the client-side to either allow or block the incoming requests. The proposed solution provides developers better control of a web page's DOM by specifying policies. The solution can be easily deployed in all web browsers which is made evident by our prototype implementation in an open source browser, Chromium. We propose to detect and prevent the malicious request even if the attacker payload is obfuscated. The proposed solution provides an added layer of security against DOM-based XSS attacks. We have evaluated the performance of our approach with two popular JavaScript performance benchmarks: Dramaeo [28] and Speedometer [31].

1.2 Contribution

Our main contributions are as follows:

- Design and development of a protection mechanism for mitigation of DOM-based XSS attacks with the following characteristics:
  - Server-side control as well as automatic client-side protection
  - Compatible with existing web applications
  - Resilient to obfuscated malicious requests
  - Less performance overhead
- Demonstration of practical applicability of the proposed technique by:
  - Modifying an open source browser to show the feasibility of our approach.
  - Performing experiments that demonstrate the effectiveness of our mechanism in detecting and preventing unauthorized modifications to the web page's DOM.
  - Conducting experiments that evaluate the performance of the proposed approach, specifically the overhead, the solution brings to the web browser.

The remainder of this thesis is structured as follows: Chapter 2 provides background, related work, and analysis of MutationObserver API. Chapter 3 explains the proposed approach in detail. Chapter 4 describes implementation, evaluation results, and comparison of the proposed solution with popular tools and techniques. Finally, we summarize our findings, conclusions, and future work in Chapter 5.

Chapter 2

Background and Related Work

2.1 Background

A web page is a document which can either be displayed in a browser window or as the HTML source. Both representations refer to the same document. Document Object Model (DOM) is a programming API for accessing and modifying HTML and XML documents. The DOM is a way JavaScript code can communicate with HTML elements on a web page. It is an object-oriented representation of the web page which can be manipulated through a scripting language such as JavaScript. Figure 2.1 shows an example of DOM representation of a basic web page.

DOM represents an HTML document as a tree structure where each node is an object representing a part of the document and the topmost node named ”Document object.” When a web page is loaded, the browser creates the

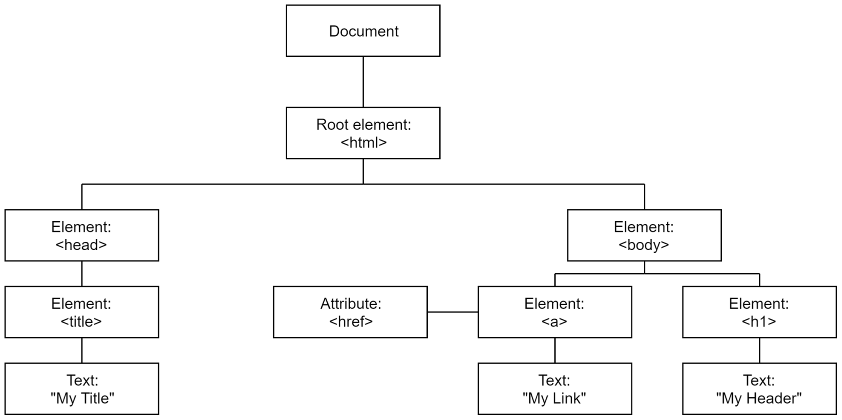


Figure 2.1: Example of DOM representation of a web page

DOM of the page that acts as an interface between JavaScript and DOM itself for the creation of dynamic web pages [11]. Web browsers depend on layout engines to parse HTML into DOM. Some layout engines including Blink [3], WebKit [25], and Gecko [9] are shared by a number of browsers such as Google Chrome, Chromium, Opera, Safari, and Firefox.

Any unauthorized changes in a DOM can have adverse effects on the web application causing security breach and loss of sensitive information. It is essential to ensure the integrity of DOM to protect users' personal information such as username/password, credit card number etc.

2.2 Related Work

The related work in the area of DOM-based XSS can be divided into three categories: *1) Browser-based*, in which the solution is available as an automatic attack detection and prevention tool embedded in a web browser. *2)*

External Tools, in which the solution is available as an external tool to analyze web applications to identify DOM-based XSS vulnerabilities *3) Policy-based* approach, that allows web developers to specify policies on web pages to detect or prevent any unauthorized modifications of the DOM.

2.2.1 Browser-based

These studies focussed on designing and implementing a solution that is embedded in a web browser to automatically detect and prevent DOM-based XSS attacks.

Lekies et al. [44] designed a filter for DOM-based XSS that utilizes runtime taint analysis and a taint-aware parser to stop the execution of attacker-controlled syntactic content. The filter is composed of two interconnected components: Taint-enhanced JavaScript engine [37], and Taint-aware JavaScript HTML parser. The JavaScript engine tracks the data flow from attacker controlled sources such as document.URL, document.cookie. The parser detects the generation of malicious code from traced values, resulting in rejection of the code from being executed. The authors implemented the filter in an open source browser, Chromium, and performed experiments using a set of 1,602 real-world vulnerabilities, achieving a rate of 73% successful filter bypasses.

Parameshwaran et al. [41] developed a system called DEXTERJS based on auto-patching. DEXTERJS can be embedded in a standard web browser and automatically patches DOM-based XSS vulnerabilities. The system works by performing dynamic analysis to detect, modify, and repair DOM-based bugs.

The patches can be applied directly to the web application via hot-patching. DEXTERJS first analyzes the given JavaScript application, identifies the positioning of all dynamic code evaluation (DCE) points and checks whether they are exploitable or not. If they are exploitable, the system marks them as patch points. DEXTERJS utilizes dynamic taint-tracking to identify the attacker-controlled bytes in the string. The hot-patching includes adding hooks at the patch points, so the program will execute the patch function instead of the original code.

The browser-based tools do not have the capability to differentiate between authorized and unauthorized modifications. They automatically detect and prevent the attack without considering the security requirements of the web application. They are embedded in web browsers and cannot provide server-side control.

2.2.2 External Tools

These studies provide developers a way to perform the security assessment of web applications to identify DOM-based XSS vulnerabilities.

Nguyen et al. [38] introduced a distributed scanning tool for crawling modern dynamic websites generated by JavaScript. The tool consists of two components: crawler and scanner. The crawler uses hooking techniques to crawl websites, and the scanner detects the suspicious code. The study analyzes Alexa's top 1000 websites using two benchmarks: IBM JavaScript test suite and Google's Firing Range. The evaluation results show zero false negative rates.

Saha et al. [42] proposed a DOM-based XSS vulnerabilities detector that utilizes data-flow analysis technique [32]. The detector works by accepting a JavaScript embedded HTML file as an input and building its abstract syntax tree (AST). The control flow graph is created by converting AST into a linearized form resembling three-address code [39]. The detector then analyzes the data from malicious sources (URL, cookie, location) to sink (write) and checks whether data is filtered before reaching the sink. Lastly, it shows a message if the given HTML page is vulnerable or not.

External tools are helpful for web developers to separately analyze the web application for the identification of DOM-based XSS vulnerabilities. It requires modifications in the web application's source code to fix the identified vulnerabilities. The developers usually do not have much security background to repair web applications against DOM-based XSS attacks.

2.2.3 Policy-based

A policy-based approach allows developers to detect or prevent unauthorized modifications to web pages.

MutationEvents [12] is an API that allows web developers to be notified of and react to changes made to the DOM. The API cannot be used to prevent unauthorized changes as it notifies after the change has already occurred. This API has been depreciated because of poor performance and lack of support among web browsers.

MutationObserver [13] is proposed as a replacement of MutationEvents, providing better performance, reliability, and cross-browser support. MutationObserver identifies changes which are already occurred in the DOM. MutationObserver cannot prevent modifications of the DOM, but it provides developers with a way to react to those modifications that has already occurred. The developers can use MutationObserver inside the web application providing better control as compared to other tools that are used outside the web application environment. Our analysis revealed several limitations in MutationObserver which are discussed in Section 2.3.

Content Security Policy (CSP) [43] is a whitelist, and per-page policy language that helps prevent XSS and data injection attacks. CSP is a W3C standard and is now supported by all major web browsers. It allows the developer to control contents of a web page by specifying policies via directives. The proposed approach also has directives but with a more adjustable fine-grained control over the web page's DOM. At the same time, the proposed solution allows specifying policies per-tag, per-id, and per-class of HTML elements while CSP operates only on per-page level.

Oda et al. [40] proposed Security Style Sheets (SSS) whose syntax is similar to Cascading Style Sheet (CSS). CSS is used for adding style to a web document. SSS has three directives: *domain-channels*, *page-channels*, and *execution*. The *domain-channels* directive focuses on the whitelisting of domains for loading third party contents, *page-channels* directive controls the whitelisting of HTML elements within a webpage, and *execution* directive controls the execution of JavaScript. SSS directives are not fine-grained and cannot provide protection against DOM-based XSS attacks. The proposed approach is an extension of SSS with more focus on detecting and preventing unauthorized tampering with the DOM.

SIACHEN [34] is a whitelist, per-id, per-class policy language for the mitigation of XSS attacks. SIACHEN syntax is similar to CSS and its semantics are based on CSP. SIACHEN has several directives that allow developers to create whitelists of trusted domains for loading third party resources. SIACHEN mainly focuses on reflected XSS and provides protection against malicious third party contents. It cannot prevent DOM-based XSS attacks.

The above policy-based tools can protect web applications against reflected or stored XSS attacks, but they are not enough against DOM-based XSS attacks. The proposed solution combines the *browser-based* and *policy-based* approach to protect web page's DOM from unauthorized modifications that provides an added layer of security against DOM-based XSS attacks.

2.3 Analysis of MutationObserver

MutationObserver is a powerful web API that can be used to detect different types of changes in the DOM. It has provision to specify a target node to monitor for changes. The target node can be any HTML element e.g., div, input, button etc. It has several options that can be activated to provide notifications about the changes occurring inside the specified target element.

The options are:

- *attributes:* Set to true if target's attribute modification needs to be observed.
- *attributeOldValue:* Set to true if target's attributes old value before modification needs to be recorded.
- *attributeFilter:* Set to an array of attribute names whose modifications need to be observed.
- *childList:* Set to true if addition and removal of target's child nodes need to be observed.
- *characterData Set:* Set to true if target's data need to be observed.
- *subTree:* Set to true if target's descendants need to be observed.
- *characterDataOldValue:* Set to true if target's data before modification need to be recorded.

2.3.1 Limitations of MutationObserver

MutationObserver is a popular API and is now being used in over 128,000 websites [15]. To evaluate the effectiveness of this API, several experiments were performed on Google Chrome, version 67.0.3396.99; Mozilla Firefox, version 60.0.2; Safari, version 11.1; and Opera, version 54.0.2952.51. It is observed that MutationObserver API is a very effective way to detect changes in the DOM and is supported by all major web browsers. However, there are some scenarios where this API does not notify about DOM modifications. Once the web page is loaded inside the web browser, there is a limited set of JavaScript DOM API functions [7] that can be used to update the DOM. We tested all functions to evaluate the MutationObserver API. Some basic DOM operations such as adding or deleting nodes, changing attribute values etc, are tracked by MutationObserver. There are a few advanced features such as shadow DOM and graphic APIs that can also affect the DOM which cannot be tracked by MutationObserver. Table 2.1 shows the experimental results of our analysis.

Mutation Type	**Result**	**Mutation Type Supported by**			
Node insertion	Able to detect	Google Chrome	Mozilla Firefox	Safari	Opera
Node removal	Able to detect	Yes	Yes	Yes	Yes
Attribute change	Able to detect	Yes	Yes	Yes	Yes
Change in sub tree	Able to detect	Yes	Yes	Yes	Yes
Removing target node	Not able to detect	Yes	Yes	Yes	Yes
$\langle canvas \rangle$, $\langle svg \rangle$ graphics	Not able to detect	Yes	Yes	Yes	Yes
Shadow DOM Attachment	Not able to detect	Yes	NA	Yes	Yes
Shadow DOM Changes	Not able to detect	Yes	NA	Yes	Yes

Table 2.1: Experimental Evaluation of MutationObserver API

Our results show that MutationObserver API is very effective in detecting various changes in the DOM. It has the following limitations:

- **Removing target node:** MutationObserver does not have the mechanism to report if the target node itself is removed, for example, if the target node is a login form inside a web page to observe changes e.g., $\langle div \rangle$. If somehow that login form is removed and replaced by an attacker through execution of malicious JavaScript, there is no notification from MutationObserver. This can result in sensitive information of the user being stolen.

- **$\langle canvas \rangle$, $\langle svg \rangle$ graphics:** The $\langle canvas \rangle$ tag is used to draw graphics on the fly via scripting. Drawing graphics using scalable vector graphics (SVG) and embedding them into the $\langle canvas \rangle$ tag is not trackable by MutationObserver. Moreover, modifying the graphics in SVG e.g., change in image size, or replacement of image cannot be observed. This limitation can have a severe effect if malicious JavaScript is being injected into the SVG by the attacker.

- **Shadow DOM Attachment:** The shadow DOM allows hidden DOM trees to be attached to elements in the regular DOM tree. The subtree created by shadow DOM is rendered on a web page but does not become part of the document's DOM tree. The main purpose of the shadow DOM is to hide the implementation details by encapsulating part of DOM tree. Shadow DOM is supported by all major web browsers except Firefox which will be added in the next release of Firefox version 63 [23]. MutationObserver has no mechanism to track the attachment of the shadow DOM to the target element. The attacker can exploit this limitation by attaching a shadow DOM over sensitive area of a webpage e.g., login form, to steal the user's login credentials.

- **Shadow DOM Changes:** MutationObserver does not have the capability to track changes occurring inside the shadow DOM.

2.3.2 MutationObserver and Other APIs

There are some other APIs similar to MutationObserver that also detect changes in the DOM but most of them are built on top of MutationObserver to provide some additional functionalities, e.g., providing better log reporting or compatibility with specific platform like NodeJS. These APIs include Mutation-Summary [14] and WatchDOM [24]. Similar experiments were performed on these APIs to understand whether or not MutationObserver limitations were correlated. The results of the experiments are shown in Table 2.2.

Our results show that Mutation-Summary and WatchDOM also have the same limitations as MutationObserver. As the shadow DOM was not supported by Firefox, experiments were not conducted.

Mutation Type	**Mutation-Summary**	**Watch-DOM**	**Mutation Type Supported by**			
			Google Chrome	Mozilla Firefox	Safari	Opera
Removing target node	✗	✗	Yes	Yes	Yes	Yes
⟨*canvas*⟩, ⟨*svg*⟩ graphics	✗	✗	Yes	Yes	Yes	Yes
Shadow DOM Attachment	✗	✗	Yes	NA	Yes	Yes
Shadow DOM Changes	✗	✗	Yes	NA	Yes	Yes

Table 2.2: Experimental Evaluation of other similar APIs

Chapter 3

Proposed Solution

In this chapter, we propose a protection mechanism against unauthorized tampering with a web page's DOM providing an added layer of security against DOM-based XSS attacks. The proposed solution allows developers to have better control of a web page's DOM by defining security policies. The proposed mechanism is a hybrid of *browser-based* and *policy-based* approaches. It contains built-in policies and also allows developers to define their own security policies according the security requirement of the web application. Figure 3.1 shows the high level diagram of the proposed solution. The solution is composed of two main components: *DOM Security Policy* (DSP) and *DOM Monitoring Module* (DMM). The DSP provides server-side control by allowing developers to specify security policies on HTML elements. The DMM provides automatic client-side protection by enforcing built-in and developer specified policies. The DMM detects and prevents unauthorized

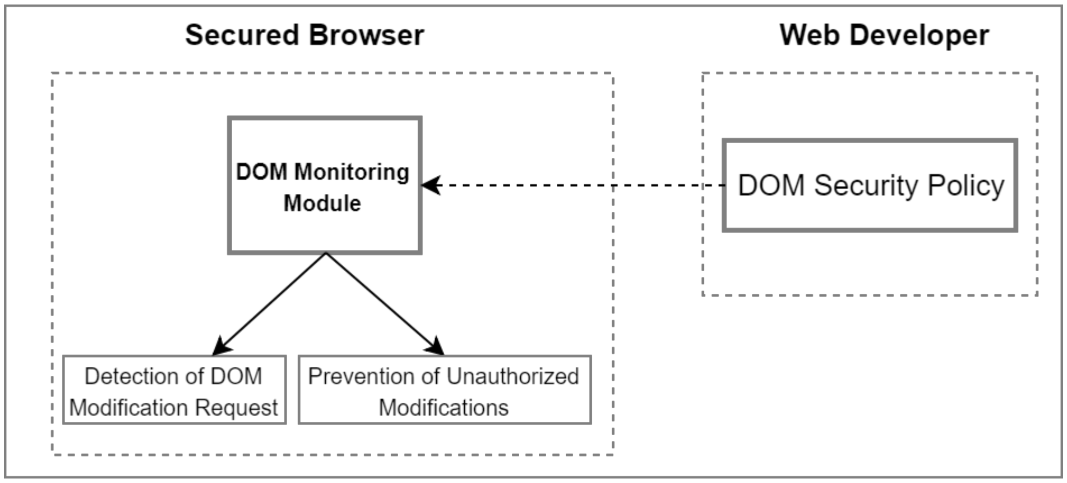

Secured Browser
Web Developer
DOM Monitoring Module
DOM Security Policy
Detection of DOM Modification Request
Prevention of Unauthorized Modifications

Figure 3.1: High Level Diagram

modifications in the DOM. It analyzes each incoming DOM modification request and allows or blocks requests based upon built-in policy or developer specified policy. It is designed inside the web browser's rendering engine and monitors native DOM APIs for detecting the DOM modifications. It does not depend upon JavaScript code that could be obfuscated, to detect changes of the DOM. Due to the designing nature of the proposed solution, the obfuscated JavaScript requests are not able to bypass the DOM Monitoring Module.

3.1 DOM Security Policy

DOM Security Policy is a fine-grained, blacklist/whitelist policy language. DSP operates *per-id*, *per-class*, or *per-tag* of a web page's HTML elements. It provides adjustable control over the web page's DOM and a clear separation of security policy language from the web page's contents. The generic syntax of DSP is shown in Figure 3.2.

```
// using one selector to apply policy directive
#ElementSelector {
--DSP Directive Name: value;
}

// using multiple selectors to apply policy directive
#ElementSelector, .ClassSelector, TagName {
--DSP Directive Name: value;
}
```

Figure 3.2: Generic Syntax of DOM Security Policy

DSP syntax is similar to CSS. The idea of using CSS syntax as a security policy language is not new. Oda et al. [40] proposed Security Style Sheets (SSS) based on CSS syntax. DOM Security Policy is consistent with SSS in terms of using CSS like syntax for defining a security policy language. We have extended this work with a focus on detecting and preventing unauthorized tampering with the DOM. We developed several directives that can be used to specify authorized and unauthorized modifications of the DOM.

CSS provides different types of selectors which are used to select elements for styling [6]. DSP operates on *per-id*, *per-class*, or *per-tag* selector. Table 3.1 shows the selectors available in DOM Security Policy.

3.1.1 DSP Directives

DOM Security Policy is composed of several directives that web developers can use to specify policies on a HTML element or set of elements. Each

Selector	**Syntax**	**Example**	**Description**
Per-id	#id	#Menu	Selects all elements with id="Menu"
Per-class	.class	.paragraph	Selects all elements with class="paragraph"
Per-tag	tagname	img, script	Selects all $\langle img \rangle$ and $\langle script \rangle$ elements

Table 3.1: Selectors available in DOM Security Policy

directive has a name and a value and controls a specific type of modifications in the DOM. These directives allow web developers to specify what part of the DOM should be modified or not. DSP directives are based on CSS custom properties [5], which requires (--) sign before any property name.

DSP directives are divided into two main categories: *General Directives* and *Tag-specific Directives*. Each directive has a default built-in policy. If the developer does not specify the DSP directive, the default policies can be automatically enforced by the DOM Monitoring Module.

3.1.1.1 General Directives

The general directives can be applied to any HTML tag. The directives are:

- **allow-event-modification:** The user interaction with HTML elements happens through events. HTML DOM events allow JavaScript to detect when a user performs a certain action e.g., clicking an element, or hovering over an element. Events are an important part of DOM. Each HTML tag has a set of events that execute JavaScript code. The security of the legitimate events is very important as they can be used by attackers to steal user sensitive data.

The *allow-event-modification* directive protects the DOM from any unauthorized attachment or modification of events. The value of the directive must be *true* or *false*. If this directive is present in policy with a *false* value, the DOM Monitoring Module blocks all event modification requests on the specified elements. If the policy is present without this directive, then the default value is *true* which allows all the events to be attached and modified without any restrictions.

Figure 3.3 shows an example of the *allow-event-modification* directive blocking event modification requests in all the images on a web page.

```
// HTML code
<img src="security.png" onclick="alert('I am legit event!')" />

// Corresponding DOM Security Policy
img {
--allow-event-modification: false;
}
```

Figure 3.3: Example of allow-event-modification

- **event-blacklist:** contains a blacklist of event names which are not allowed to be modified. The value of the directive must be a comma separated list of the event names. The DOM Monitoring Module blocks all requests that try to modify the events present in this directive value. All the other events are allowed to be modified.

Figure 3.4 shows an example of *event-blacklist* directive to prohibit the *click* event to be modified for all the input elements on a web page.

```
// HTML code
<input type="button" value="Legit button" onclick="alert('I am legit!')" />
<input type="checkbox " onclick="alert('I am legit!')" />

// Corresponding DOM Security Policy
input {
--event-blacklist: click;
}
```

Figure 3.4: Example of event-blacklist

- **event-whitelist:** contains a whitelist of event names which are allowed to be modified. The value of this attribute must be a comma separated list of the event names. The events listed in this directive value can be allowed to be modified, all the other events modification requests are blocked.

- **allow-attribute-modification:** HTML attributes are an integral part of DOM. When the browser parses the HTML elements and creates the DOM, it recognizes the standard attributes and creates DOM properties from them. Every HTML tag contains a set of attributes.

  The *allow-attribute-modification* policy directive protects the DOM from any unauthorized attribute addition, deletion, or change in value. It controls the requests that modify the attributes of HTML tags. The value of the directive must be *true* or *false*. If this directive is present in policy with a *false* value, the DOM Monitoring Module blocks all attribute modification requests on the specified elements. If the policy is present without this directive, then the default value is *true* which allows all the DOM attributes to be modified without any restrictions.

```
// HTML code
<div class="Secure" id="login"></div>
<p class="Secure" id="paragraph">some text</p>

// Corresponding DOM Security Policy
.Secure {
--allow-attribute-modification: false;
}
```

Figure 3.5: Example of allow-attribute-modification

Figure 3.5 shows an example of *allow-attribute-modification* directive to block attribute modification requests of all the tags with class="Secure".

- **attribute-blacklist:** contains a blacklist of attribute names which are not allowed to be modified. The value of this directive must be a comma separated list of the attribute names. The DOM Monitoring Module blocks all requests that try to modify the attribute present in this directive value. All other attribute modifications are allowed.

```
// HTML code
<div id="login" name="restricted"></div>
<p class="Secure" id="paragraph" name="text">some text</p>

// Corresponding DOM Security Policy
div, .Secure {
--attribute-blacklist: name, id;
}
```

Figure 3.6: Example of attribute-blacklist

Figure 3.6 shows an example of *attribute-blacklist* directive to block the modification of name, and id attribute of div tags and all other tags with class=”Secure”.

- **attribute-whitelist:** contains a whitelist of attribute names which are allowed to be modified. The value of this attribute must be a comma separated list of the attribute names. The attributes listed in this directive value can be modified, while all other attribute modification requests are blocked.

- **allow-style-modification:** controls whether the styling of specified elements can be changed or not. The value of this directive must be *true* or *false*. If this directive is present in policy with a *false* value, the DOM Monitoring Module blocks all styling requests (e.g., change color, background, font etc) on the specified elements. The default value of this directive is *false*, which blocks the execution of all DOM style modification requests.

- **allow-shadow-attachment:** controls the attachment of the shadow DOM on HTML elements. The shadow DOM allows hidden DOM trees to be attached to elements in the regular DOM. It has several benefits e.g., encapsulating part of the DOM tree, and hiding the implementation details from the user but if it is not controlled properly, it can be used by an attacker to steal sensitive user information. The value of the directive must be *true* or *false*. By default, it is *false*, which means the shadow DOM cannot be attached to any element of the web page unless specifically allowed by the web developer. Existing techniques such as MutationObserver do not have the ability to track the attachment of the shadow DOM.

- **protected:** controls any type of modification on a specified element e.g., restricted areas like password input fields, or login forms. This directive specifies the area as fully protected. The value of this attribute must be *true* or *false.* If this directive is present in a policy with a *true* value, the DOM Monitoring Module blocks all requests related to the attribute modification, style modification, and shadow attachment on the specified elements. By default, the value of this directive is *false.* This means no elements are protected by default unless specified.

3.1.1.2 Tag-Specific Directives

The tag-specific directives control where a type of resource may be loaded. These directives can only be applied to ⟨*a*⟩, ⟨*img*⟩, ⟨*script*⟩, ⟨*audio*⟩, ⟨*video*⟩, ⟨*source*⟩, ⟨*track*⟩, and ⟨*object*⟩ tags. This set of tags are mostly used in common websites and if not handled properly could result in making the website vulnerable to DOM-based XSS attacks. As these tags can be used to load third-party contents, so they should be monitored properly. The DOM Security Policy provides the web developer an easy way to control the resources coming from third parties to prevent the execution of any malicious content such as malicious JavaScript, malicious URLs etc. The directives are:

- **domain-blacklist:** contains a blacklist of domains which have been disapproved for loading third party contents. The value of this directive must be a comma separated list of domain names which are not allowed for loading contents such as image, JavaScript, URLs etc.

```
// DOM Security Policy
img {
--domain-blacklist: www.example.com;
}
```

Figure 3.7: Example of domain-blacklist

  Figure 3.7 shows an example of the *domain-blacklist* directive to block the loading of images from www.example.com.

- **domain-whitelist:** contains a whitelist of domains which are allowed to be used as a source for loading contents in the tags listed above. If no domain is specified, the default built-in policy allows resources to come from the web page's own domain only.

  Figure 3.8 shows an example of the *domain-whitelist* directive to load all the images, audios, and videos only from www.example.com.

```
// DOM Security Policy
img, audio, video {
--domain-whitelist: www.example.com;
}
```

Figure 3.8: Example of domain-whitelist

Table 3.2 summarizes all the DOM Security Policy directives.

Category	**Directive**	**Accepted Value**	**Default-Value**
General	allow-event-modification	true or false	true
	event-blacklist	comma separated list of event names	none
	event-whitelist	comma separated list of event names	none
	allow-attribute-modification	true or false	true
	attribute-blacklist	comma separated list of attribute names	none
	attribute-whitelist	comma separated list of attribute names	none
	allow-style-modification	true or false	false
	allow-shadow-attachment	true or false	false
	protected	true or false	false
Tag-Specific	domain-blacklist	comma separated list of domain names	none
	domain-whitelist	comma separated list of domain names	self domain

Table 3.2: DOM Security Policy Directives

3.2 DOM Monitoring Module

The DOM Monitoring Module enforces the policy on HTML elements as specified by the web developer through an HTTP header. Once the policy is received and parsed in the browser, this module is activated and starts monitoring every request that tries to modify the DOM. The DMM allows only those requests to execute which comply with the policy rules. If no policy rules are set, the default rules are automatically set to limit the effect of DOM-based XSS attacks.

The DMM is activated only if the web page has an HTTP header named "DOM-Security-Policy." Existing web applications that do not implement this HTTP header can operate properly in a browser supporting our proposed solution.

Figure 3.9 depicts how the DOM Monitoring Module works. Once any DOM modification request comes, the module first identifies the type of request, the element(s) associated with that request, and checks if the element matches with any selectors specified by the web developer in the policy. If the match is found, the monitoring module retrieves the corresponding policy and checks if the request violates any of the policy rules. If it does, the request is rejected, its access to the real DOM is blocked, and an error message is sent to the browser console.

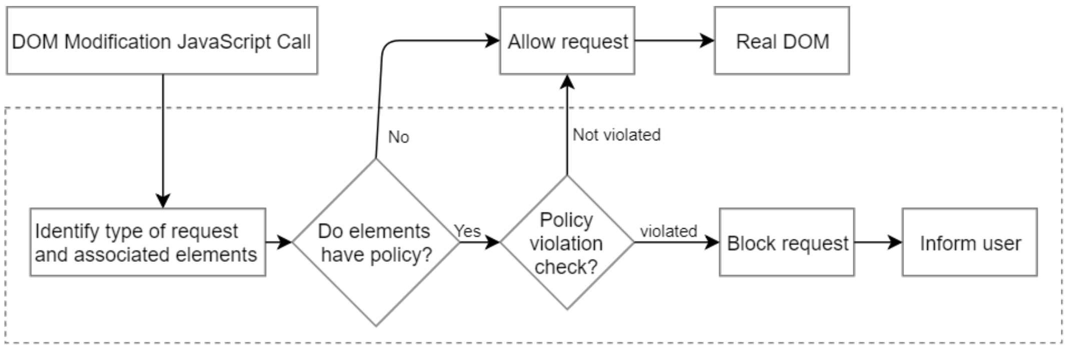


Figure 3.9: Overview of DOM Monitoring Module

The protection mechanism of the DOM is put in action by patching the native DOM APIs. The mechanism can handle any type of incoming request (through third-party JavaScript, web page's parameters, or obfuscated). If any request calls the DOM API functions, the request is monitored by DOM

Monitoring Module before reflecting the change on the real DOM.

3.3 Client-side working of Proposed Solution

In this section, we briefly describe how the proposed solution works at the client-side as shown in Figure 3.10. The whole process can be divided into 3 parts. 1) Deliver Policy, 2) Parse Policy, 3) Apply Policy

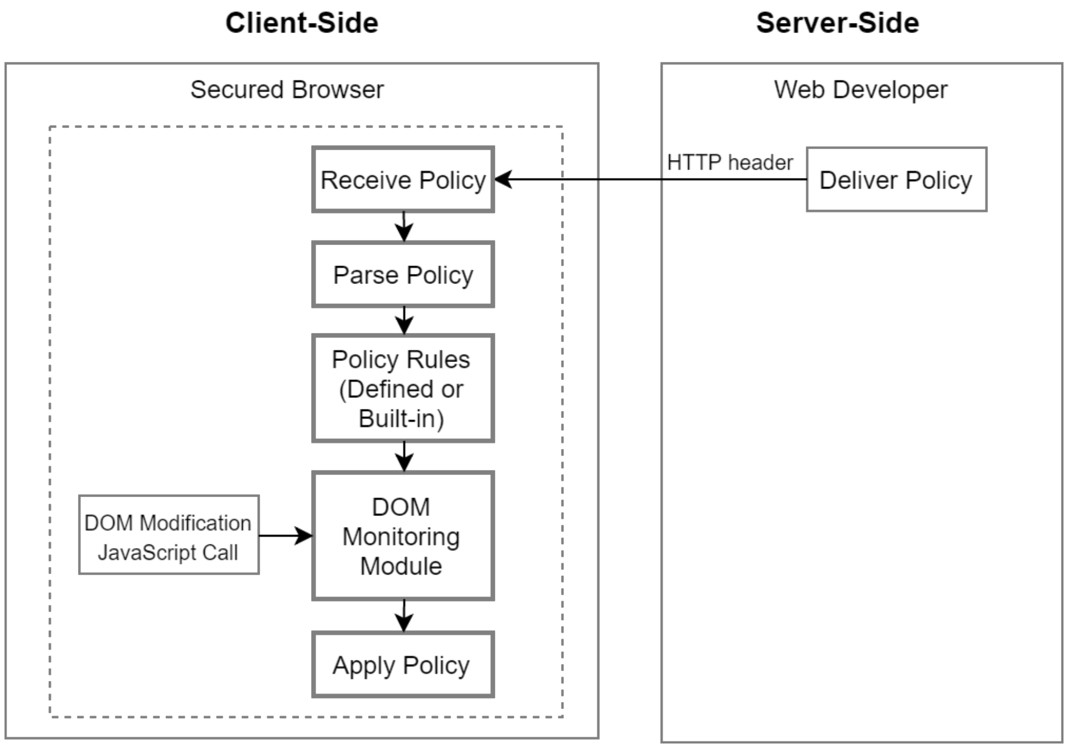


Figure 3.10: Client-side working of proposed solution

Deliver Policy

The first step to ensure that our protection mechanism works is to find a way for the policy to be delivered to the browser, so it can be enforced properly. Web developers can specify policy directives on HTML elements via a new HTTP header named "DOM-Security-Policy."

The reason why we use the HTTP header to deliver the policy is because the HTTP header is received by the browser before the DOM is being parsed and rendered. We want to make sure that we receive the policy before any unauthorized request modifies the DOM. In this way, we can prevent any changes occurring to the DOM which are forbidden by the policy. Figure 3.11 shows how the web developer can specify policy on the server side using the PHP header function.

```
<?php
header("DOM-Security-Policy: #ID {--allow-attribute-modification:false; } iframe, img,
    script {--domain-whitelist: www.example.com; }");
?>
```

Figure 3.11: Example of policy specified in HTTP header

Parse Policy

Once the policy has been delivered to the browser through the HTTP header, the modified browser can receive and parse the policy to identify DSP directives. The unknown directives are simply discarded. The policy is delivered, received, and parsed by the browser before the DOM elements begin to parse and render. The policy stays active in the browser's memory as long as the corresponding web page is active.

Apply Policy

Once the policy has been received and parsed by the browser, the *DOM Monitoring Module* enforces the policy on HTML elements. This module allows only those requests to get executed that fulfill the policies specified by the web developer, or according to the default built-in policy.

Chapter 4

Implementation and Experimental Results

In this chapter, we discuss implementation of the proposed solution. To practically validate the feasibility of our protection mechanism, we implement our approach in an open source web browser, Chromium. We also discuss the performance evaluation with different test cases and compare our approach with existing tools and techniques.

4.1 Implementation

We implemented the proposed solution in Google Chromium, version 69.0.3481.0, by patching various parts of the code inside Blink [3]. Blink is a rendering engine used in Google Chrome. It was developed as a part of a Chromium project with contributions from Google, Opera Software ASA, Adobe Systems, Intel, Samsung and others [30]. It is written in the C++ programming language. It is based on WebKit [25] that is shared by Google Chrome, Opera, and Safari web browser. We modified different parts of WebKit inside Blink to accommodate the following components:

- **Receive DOM Security Policy:** We modified the Document Loader component inside Blink to receive the HTTP header named "DOM-Security-Policy" containing security policies.
- **Parse Directives:** As mentioned earlier, the DSP syntax is similar to CSS, so we modified the built-in Chromium CSS parser to parse DSP directives. As DSP directives are not actually CSS properties so they are unknown to the CSS parser, and therefore dropped by the CSS parser. We modified the parser implementation to ensure all the directives are processed correctly.
- **DOM Monitoring Module:** This is a new module introduced in the web browser. The module is implemented by hooking several DOM APIs inside Blink.

4.2 Experimental Results

To understand the effectiveness of the proposed solution, we performed several experiments. The experiments are broken down into four phases:

1. Analysis of our approach ability to prevent unauthorized requests
2. Analysis of the implemented solution overhead on the client's system
3. Analysis of the proposed system ability to handle obfuscated requests
4. Comparative analysis of our approach to existing tools and techniques

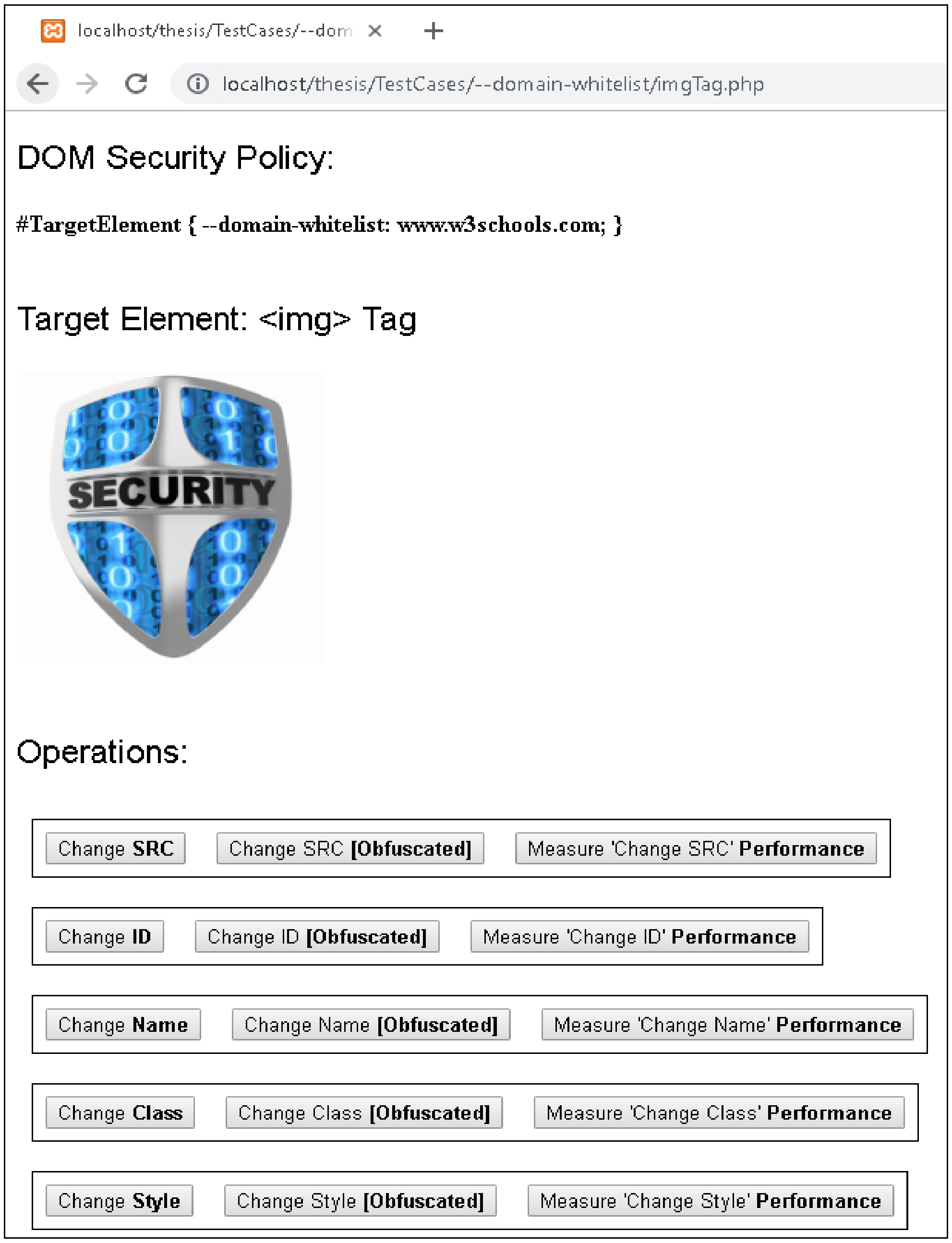
4.2.1 Analysis of our approach ability to detect and prevent unauthorized requests

We developed test cases for the 6 most commonly used tags: $\langle a\rangle$, $\langle audio\rangle$, $\langle iframe\rangle$, $\langle img\rangle$, $\langle object\rangle$, and $\langle video\rangle$. We tested all DSP directives on these tags. The test page contains 150 lines of HTML, CSS, and JavaScript code. Figure 4.1 shows an example of the test case on $\langle img\rangle$ tag.

Each test case has different operation buttons that generate requests to modify the target element. If the request fulfills the policy specified in the test case, it can be accepted or blocked. The sample output of a blocked and accepted request is shown in Figure 4.2 and Figure 4.3 respectively.

4.2.2 Analysis of the implemented solution overhead on the client's system

In order to evaluate the performance of our implementation, we conducted two types of evaluations: *Manual Evaluation*, and *Benchmark Evaluation.*


localhost/thesis/TestCases/--dom

localhost/thesis/TestCases/--domain-whitelist/imgTag.php

DOM Security Policy:

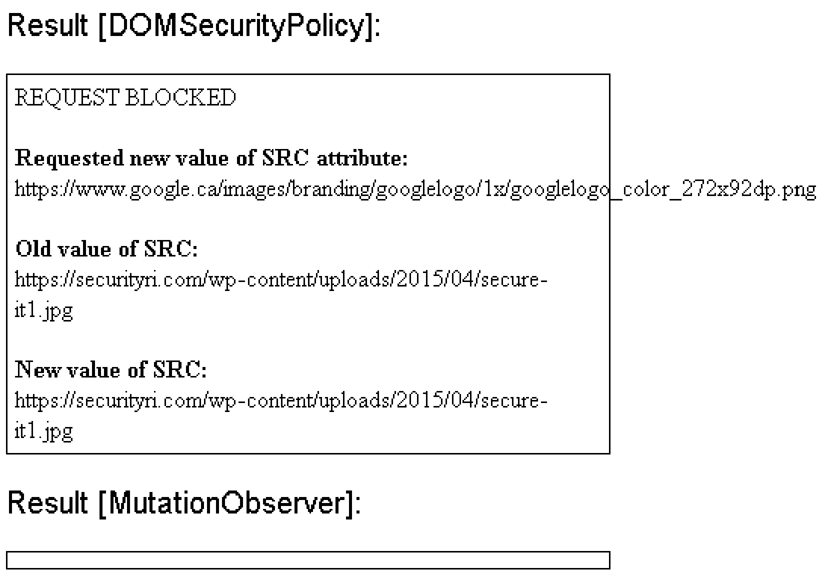
#TargetElement { --domain-whitelist: www.w3schools.com; }

Target Element: <img> Tag

Operations:

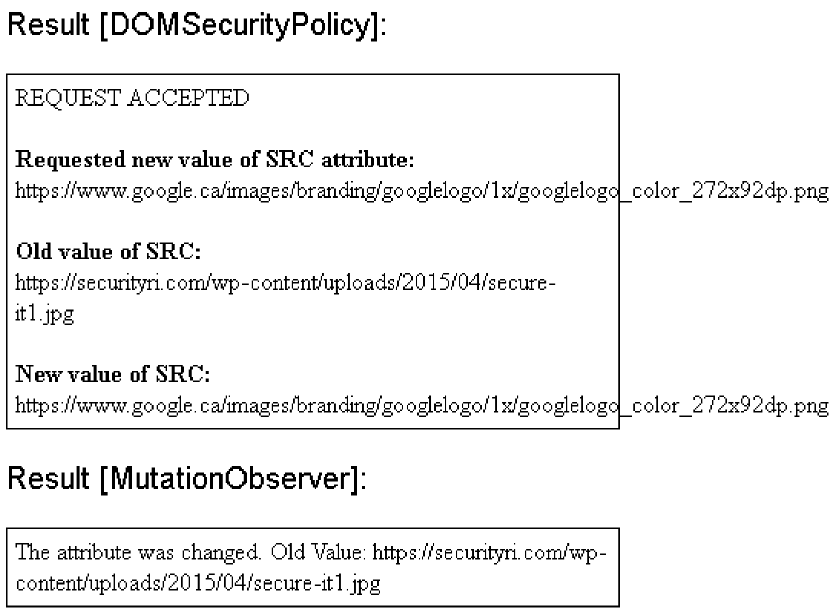
Change **SRC**
Change SRC **[Obfuscated]**
Measure 'Change SRC' **Performance**
Change **ID**
Change ID **[Obfuscated]**
Measure 'Change ID' **Performance**
Change **Name**
Change Name **[Obfuscated]**
Measure 'Change Name' **Performance**
Change **Class**
Change Class **[Obfuscated]**
Measure 'Change Class' **Performance**
Change **Style**
Change Style **[Obfuscated]**
Measure 'Change Style' **Performance**

Figure 4.1: Test Case Example



```
Result [DOMSecurityPolicy]:

REQUEST BLOCKED

Requested new value of SRC attribute:
https://www.google.ca/images/branding/googlelogo/1x/googlelogo_color_272x92dp.png

Old value of SRC:
https://securityri.com/wp-content/uploads/2015/04/secure-
it1.jpg

New value of SRC:
https://securityri.com/wp-content/uploads/2015/04/secure-
it1.jpg

Result [MutationObserver]:
```

Figure 4.2: Blocked Request



```
Result [DOMSecurityPolicy]:

REQUEST ACCEPTED

Requested new value of SRC attribute:
https://www.google.ca/images/branding/googlelogo/1x/googlelogo_color_272x92dp.png

Old value of SRC:
https://securityri.com/wp-content/uploads/2015/04/secure-
it1.jpg

New value of SRC:
https://www.google.ca/images/branding/googlelogo/1x/googlelogo_color_272x92dp.png

Result [MutationObserver]:

The attribute was changed. Old Value: https://securityri.com/wp-
content/uploads/2015/04/secure-it1.jpg
```

Figure 4.3: Accepted Request

In *Manual Evaluation*, we calculated the timing of different DOM modification operations in an unmodified and modified version of a web browser and compared their results. In *Benchmark Evaluation*, we conducted experiments with the popular JavaScript benchmarks Dromaeo [28] and Speedometer [31] to evaluate DOM APIs performance. We ran these experiments using an Intel(R) Xeon(R) CPU 2.20GHz (2 processors) Windows 10 machine with 32GB of RAM.

4.2.2.1 Manual Evaluation

In manual evaluation, we performed different DOM modification operations on $\langle a \rangle$ and $\langle img \rangle$ tag in an unmodified and modified browser. We calculated the execution time of each request using *now()* function of High Resolution Time API [10]. Table 4.1 shows the results of our experiments. To eliminate side effects e.g., the operating system or network latency, we ran the same operation ten times and took the median execution value as a performance measure.

	$\langle a \rangle$ **tag attributes**			$\langle img \rangle$ **tag attributes**		
	href	*id*	*class*	*src*	*id*	*class*
Unmodified browser	0.2	0.1	0.1	1.0	0.1	0.1
Modified browser	0.7	0.3	0.6	1.3	0.2	0.6

Table 4.1: Results of Manual Performance Evaluations (in miliseconds)

4.2.2.2 Benchmark Evaluation

For benchmark evaluation, we used two popular benchmarks to evaluate the performance of the modified browser. Dromaeo is a JavaScript Performance Test Suite developed by Mozilla. It has two experiment sets: JavaScript, and DOM. We ran the DOM test as we have modified DOM APIs for the implementation of our solution. Speedometer is a benchmark for web app responsiveness. It simulates user interactions in web applications and evaluates the performance by executing different DOM API operations using popular JavaScript frameworks and libraries including jQuery, AngularJS, React, Ember etc. Table 4.2 shows the results of Dromaeo and Speedometer.

	Dromaeo (runs/s)	**Speedometer (Arithmetic Mean (runs/min))**
Unmodified browser	217.27	6.27
Modified browser	219.99	6.58

Table 4.2: Results of Benchmark Performance Evaluations

4.2.3 Analysis of the proposed system ability to handle obfuscated requests

As shown in Figure 4.1, each test case has an operation button to send the modification request in an obfuscated format. As mentioned earlier, the DOM Monitoring Module is implemented by hooking DOM APIs inside the Blink rendering engine. Blink runs on an abstract platform inside a sandbox.

The obfuscated requests are automatically de-obfuscated by the Chromium's content layer before reaching the DOM APIs. Since the DMM operates inside the Blink, our system is automatically able to handle the obfuscated requests. We experimented with different obfuscated requests as shown in Figure 4.4 and Figure 4.5.

Figure 4.4, Line 2 shows the code to modify the *src* attribute of TargetElement to load malicious JavaScript. Line 5 represents the same code but in an obfuscated form. The DOM Monitoring Module is able to detect both normal and obfuscated requests.

```
1  // normal JavaScript code
2  document.getElementById("TargetElement").src = "www.maliciouswebsite.com/malicious.js"
3
4  // Obfuscated JavaScript code
5  eval(function(p,a,c,k,e,d){while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+c+'\\b','g')
       ,k[c])}}return p}('2.1("0").3="4.7.6/5.8"',9,9,'TargetElement|getElementById|
       document|src|www|malicious|com|maliciouswebsite|js'.split('|')))
```

Figure 4.4: Normal vs. Obfuscated JavaScript request

```
1  // normal JavaScript code
2  document.getElementById("TargetElement").href = "www.maliciousweb.com/login"
3
4  // Obfuscated JavaScript code
5  eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/^/,String)){while(c
       --){d[c]=k[c]||c}k=[function(e){return d[e]}];e=function(){return'\\w+'};c=1};while(
       c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('
       2.1("0").3="4.6.5/7"',8,8,'TargetElement|getElementById|document|href|www|com|
       maliciousweb|login'.split('|'),0,{}))
```

Figure 4.5: Normal vs. Obfuscated JavaScript request

Figure 4.5, Line 2 shows the code to modify the *href* attribute of TargetElement with malicious URL. Line 5 represents the same code but in an obfuscated form. The DOM Monitoring Module is able to detect both types of requests.

4.2.4 Comparative analysis of our approach to existing tools and techniques

In this section, we compare our approach to the existing popular tools and techniques.

4.2.4.1 MutationObserver and other similar APIs

The proposed solution provides the developer an easy way to detect and prevent unauthorized changes in the DOM, whereas MutationObserver focuses only on the detection part. We compare the result of each test case with MutationObserver. MutationObserver is asynchronous and only notifies if the request gets accepted and DOM has been modified. However, DOM Monitoring Module can monitor the request before it affects the DOM and allows or blocks the request depending on the policy specified by the developer or built-in policy. As we showed, the MutationObserver has several limitations including detection of shadow DOM, and detection of removal of the target node. The proposed solution overcomes these limitations and provides additional control to the developer to prevent unwanted changes in the DOM. Table 4.3 summarizes the comparison of DSP and MutationObserver with other similar APIs for detecting changes in the DOM.

Mutation Type	**Mutation-Observer**	**Mutation-Summary**	**WatchDOM**	**DOM Security Policy**
Removing target node	✗	✗	✗	✓
$\langle canvas \rangle$, $\langle svg \rangle$ graphics	✗	✗	✗	✓
Shadow DOM Attachment	✗	✗	✗	✓
Shadow DOM Changes	✗	✗	✗	✓

Table 4.3: DOM Security Policy, MutationObserver and other similar APIs

- **Removing target node:** The proposed approach detects and prevents the removal of target node. Whenever the removal request comes, the DOM Monitoring module blocks the request if the node has any policy associated with it specified by the developer.

- $\langle canvas \rangle$, $\langle svg \rangle$ **graphics:** The proposed solution supports the detection and prevention changes in graphics added through SVG.

- **Shadow DOM Attachment:** The proposed solution has the capability to detect the attachment of shadow DOM trees. By default, the monitoring module prevents the attachment of shadow DOM, but it can be disabled through *allow-shadow-attachment* directive.

- **Shadow DOM Changes:** The proposed solution is capable of detecting and preventing the changes that have been taking place inside the shadow DOM. The policy specified by the developer on the webpage can be applied to all the shadow DOM trees attached to any element on the webpage. The shadow DOM can be attached to HTML element using *open* or *closed* mode. With *open* mode, the elements inside the shadow DOM tree can be accessible from outside. On the other hand, the *closed* mode do not allow the shadow DOM tree elements to be accessible externally.

The proposed approach can detect changes occurring inside the shadow DOM if it is attached using *open* mode only, because the *closed* mode completely isolates the shadow DOM tree from outside.

4.2.4.2 Other Related Approaches

Table 4.4 shows the comparison of the DOM Security Policy with other popular policy-based approaches. The Content Security Policy focussed on XSS and data injection attacks in general, whereas the DOM Security Policy focussed on detecting and preventing unauthorized modifications in the DOM and provides fine-grained control. At the same time, the CSP only provides page-level control while DSP operates per-id, per-class, or per-tag. SIACHEN lacks per-tag control and only focussed on Reflected XSS. MutationObserver has several limitations and it only detects the changes in the DOM and lacks prevention.

	Content Security Policy	**SIACHEN**	**Mutation Observer**	**DOM Security Policy**
Detection	✓	✓	✓	✓
Prevention	✓	✓	✗	✓
Per-id control	✗	✓	✗	✓
Per-class control	✗	✓	✗	✓
Per-tag control	✗	✗	✗	✓
*F*ocussed on	XSS, Data Injection Attacks	Reflected XSS	Detecting DOM Changes	Detecting and Preventing Unauthorized DOM Modifications

Table 4.4: Comparison of DOM Security Policy with other related approaches

Chapter 5

Conclusion and Future Work

5.1 Conclusions

In this thesis, we present the design, implementation, and evaluation of a protection mechanism to detect and prevent unauthorized modifications of the web page's DOM. Our proposed solution is quite promising in many respects. Our results show that we can protect the web page's DOM from unauthorized modifications by using the DOM Security Policy specified by web developers or according to our default built-in policy. Our mechanism relies on DOM Security Policy and DOM Monitoring Module to provide server-side control as well as automatic client-side protection against unauthorized tampering with the DOM. The proposed solution provides an added layer of security against DOM-based XSS. Our approach reliably detects the DOM modification request even if the malicious payload is obfuscated.

In order to mislead our proposed protection mechanism, the DOM Security Policy can be changed if the communication between the server and client is not secure. As the policy is delivered through the HTTP header, it can be modified by an attacker through a Man-in-the-middle attack. However, such a case is out of the scope of our analysis. Our performance measurements show that our implementation incurs some extra performance overhead. We believe there is always a trade-off between security and usability.

5.2 Future Work

The goal of our proposed approach is to provide web developers with fine-grained access control over web page's DOM. Our mechanism effectively detects and prevents unwanted changes in the DOM. However, it does not report developers when the malicious request tries to modify the DOM. We leave the implementation of a reporting module as a part of future work.

Bibliography

[1] *A new modular browser*, https://github.com/breach/breach_core.

[2] *Analyzing a Dom-Based XSS in Yahoo*, https://www.exploit-db.com/docs/english/24109-domsday---analyzing-a-dom-based-xss-in-yahoo!.pdf.

[3] *Blink*, https://www.chromium.org/blink.

[4] *Crafty JavaScript HTML5 Game Engine*, http://craftyjs.com.

[5] *CSS Custom Properties*, https://developer.mozilla.org/en-US/docs/Web/CSS/--*.

[6] *CSS Selectors, W3Schools*, https://www.w3schools.com/cssref/css_selectors.asp.

[7] *DOM Element Object*, https://www.w3schools.com/Jsref/dom_obj_all.asp.

[8] *DOM XSS on Google Plus One Button*, http://goo.gl/ohRAkM.

[9] *Gecko*, https://developer.mozilla.org/en-US/docs/Mozilla/Gecko.

[10] *High Resolution Time API*, https://www.w3.org/TR/hr-time/.

[11] *JavaScript HTML DOM*, https://www.w3schools.com/js/js_htmldom.asp.

[12] *MutationEvents*, https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Mutation_events.

[13] *MutationObserver*, https://developer.mozilla.org/en/docs/Web/API/MutationObserver.

[14] *MutationSummary*, https://github.com/rafaelw/mutation-summary.

[15] *NerdyData*, https://nerdydata.com/.

[16] *node-os: First operating system powered by npm*, http://node-os.com.

[17] *Octoverse 2017*, https://octoverse.github.com.

[18] *OS.js: JavaScript Cloud/Web Desktop Platform*, http://osjsv2.0o.no.

[19] *OWASP DOM XSS*, https://www.owasp.org/index.php/DOM_Based_XSS.

[20] *Top 10-2017 A7-Cross-Site Scripting (XSS)*, https://www.owasp.org/index.php/Top_10-2017_A7-Cross-Site_Scripting_(XSS).

[21] *A twitter domxss, a wrong fix and something more*, http://blog.mindedsecurity.com/2010/09/twitter-domxss-wrong-fix-and-something.html.

[22] *Types of Cross-Site Scripting*, https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting.

[23] *Using Shadow DOM*, https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM.

[24] *WatchDOM*, https://www.npmjs.com/package/watchdom.

[25] *WebKit*, https://webkit.org/.

[26] *XSS Auditor*, https://www.chromium.org/developers/design-documents/xss-auditor.

[27] *XSS Filter Evasion Cheat Sheet*, https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

[28] *Dromaeo: JavaScript performance testing*, http://dromaeo.com/, 2010.

[29] *Stefano Di Paola. DominatorPro: Securing Next Generation of Web Applications. [online]*, https://dominator.mindedsecurity.co/, 2012.

[30] *Google, Opera Fork WebKit. Samsung Joins Firefox to Push Servo*, `https://www.infoq.com/news/2013/04/Google-Blink-Mozilla-Servo`, 2013.

[31] *Speedometer 2.0: A Benchmark for Modern Web App Responsiveness*, `https://browserbench.org/Speedometer/`, 2018.

[32] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman, *Compilers, principles, techniques*, Addison Wesley **7** (1986), no. 8, 9.

[33] CERT Coordination Center, *Cert advisory ca-2000-02 malicious html tags embedded in client web requests*, CERT/CC Advisories **3** (2000).

[34] Ashar Javed, Jens Riemer, and Jörg Schwenk, *Siachen: A fine-grained policy language for the mitigation of cross-site scripting attacks*, International Conference on Information Security, Springer, 2014, pp. 515–528.

[35] Ratinder Kaur, Yan Li, Junaid Iqbal, Hugo Gonzalez, and Natalia Stakhanova, *A security assessment of hce-nfc enabled e-wallet banking android apps*, 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), IEEE, 2018, pp. 492–497.

[36] Amit Klein, *Dom based cross site scripting or xss of the third kind*, http://www. webappsec. org/projects/articles/071105. shtml (2005).

[37] Sebastian Lekies, Ben Stock, and Martin Johns, *25 million flows later: Large-scale detection of dom-based xss*, Proceedings of the 2013 ACM

SIGSAC conference on Computer & communications security, ACM, 2013, pp. 1193–1204.

[38] Trong Kha Nguyen and Seong Oun Hwang, *Large-scale detection of dom-based xss based on publisher and subscriber model*, Computational Science and Computational Intelligence (CSCI), 2016 International Conference on, IEEE, 2016, pp. 975–980.

[39] Flemming Nielson, Hanne R Nielson, and Chris Hankin, *Principles of program analysis*, Springer, 2015.

[40] Terri Oda and Anil Somayaji, *Enhancing web page security with security style sheets*, Carleton University (2011).

[41] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena, *Auto-patching dom-based xss at scale*, Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 272–283.

[42] Suman Saha, Shizhen Jin, and Kyung-Goo Doh, *Detection of dom-based cross-site scripting by analyzing dynamically extracted scripts*, The 6th International Conference on Information Security and Assurance, 2012.

[43] Sid Stamm, Brandon Sterne, and Gervase Markham, *Reining in the web with content security policy*, Proceedings of the 19th international conference on World wide web, ACM, 2010, pp. 921–930.

[44] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns, *Precise client-side protection against dom-based cross-site scripting.*, USENIX Security Symposium, 2014, pp. 655–670.

[45] Wei Xu, Fangfang Zhang, and Sencun Zhu, *The power of obfuscation techniques in malicious javascript code: A measurement study*, Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on, IEEE, 2012, pp. 9–16.

Vita

Candidate's full name: Junaid Iqbal

University attended:
Master of Computer Science, University of New Brunswick, 2016-2018
Bachelor of Science in Information Technology, Bahauddin Zakariya University, Multan, 2010-2014

Publications:
Ratinder Kaur, Yan Li, Junaid Iqbal, Hugo Gonzalez, and Natalia Stakhanova, *A security assessment of hce-nfc enabled e-wallet banking android apps*, 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), IEEE, 2018, pp. 492–497

Conference Presentations: none