A

Ph.D Thesis

on

Attacks and Mitigation in Web Browsers

Submitted for partial fulfillment for the degree of

Doctor of Philosophy

(Computer Science and Engineering)

in

Department of Computer Science and Engineering

(2015-2016)

Supervisors:
Dr. Manoj Singh Gaur
Dr. Vijay Laxmi

Submitted by:
Anil Saini
(2011RCP7124)

MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY JAIPUR

Declaration

I, Anil Saini, declare that this thesis titled, "Attacks and Mitigation in Web Browsers" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Ph.D. degree at MNIT.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at MNIT or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: ______________________________

Date: ______________________________

Abstract

A Web browser is an important component of every computer system as it provides the interface to the Internet world. The browser allows users to view and interact with content on the web pages. With the rapid increase in the number of users and utility in every facet of life, browsers are becoming the potential source of attacks. Browser attacks over the years have stormed the Internet world with so many malicious activities. These attacks have resulted in an unauthorized access, damage or disruption of the user information within or outside the browser.

This research mainly examines the browser attacks occurring due to (a) Browser extensions and (b) Misjudged User clicks. In former, we observed that high privileges in browser extension and information flow among critical operating resources are main causes of attacks. Based on this insight, we propose semantic analysis model to detect suspicious information flows in the browser extensions. This model analyses information propagation among browser's resources. In addition, we identified new browser attacks that circumvent all detection mechanisms proposed yet for browser extensions. We demonstrate new attacks using collusion of two or more browser extensions leading to privacy leakage. In particular, we illustrate an important weakness in Firefox browser architecture and its XPCOM interface. We observed that detecting a malicious flow in an extension is a partial protection against extension-based attacks. Consequently, we propose a sandbox and isolated environment to protect operating system resources from such attacks. Our sandboxed policies enforce restrictions on web browser extensions in accessing the operating system resources. To understand the attacks targeting clicks of a user, we study new classes of clickjacking attacks in web browser. We find that most of the attacks against users of web application are caused by exploiting the fact that human visual system may not perceive minor changes caused due to blurring or filters used in image processing. We develop a novel detection method for such attacks based on the behavior (response) of a website active content against the user clicks (request). We also present an extensive comparison of our approaches with the related work in the area.

Dedications

This thesis is dedicated to my family for their endless support and encouragement..

Acknowledgements

This doctoral thesis would not have been possible without the contribution, encouragement, and guidance of a number of individuals.

I especially want to thank my principal advisor, ***Dr.Manoj Singh Gaur***, for her guidance during my research and study at MNIT, Jaipur. He has been a constant source of inspiration and guidance. His insights and feedbacks have directly shaped several ideas in this thesis. His passion for excellence in research has motivated me to work harder and has greatly influenced my personality.

My special thanks to my co–supervisor ***Dr. Vijay Laxmi*** for his invaluable guidance on both academic and personal level. I will forever be obliged for her support throughout my PhD life and guides me into the ocean of computer security and privacy. Both supervisors set an example of world–class researchers with their rigour and passion for research. They were always accessible and willing to help their students with their immense knowledge.

I would like to thank my committee members, ***Dr. Dinesh Gopalani***, ***Dr. Preety Singh*** and ***Dr. Lava Bhargava***, who have provided numerous opinions during my thesis proposal. I thank them for sparing time in providing me with valuable comments to give the required direction to my work.

I would also thank ***Dr. Mauro Conti*** from University of Padua, Italy, who provided me technical and non-technical support for the thesis.

My special thanks to my wonderful friends and fellow researchers, Smita, Chhagan Lal, Rimpy, Manoj Bohra, Gaurav, Sonal, Ashish for revitalizing each day. I cherish the prayers and support extended by them during low phases.

I will be forever indebted to my wife, my family who have been pillars of support.

For any glitches or inadequacies that may remain in this work, the responsibility is entirely my own.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A Web browser is an important component of every computer system as it provides the interface to the Internet world. The browser allows user to view and interact with content on the web pages. It provides user the interface to perform the wide range of activities, such as personal financial management, online shopping, social networking and professional business. Hence, the web browsers are becoming an increasingly adequate and important platform for millions of Internet users. With the rapid increase in the number of users, browsers are becoming the potential source of attacks.

Browser attacks over the years have stormed the Internet world with so many malicious activities. It provides an unauthorized access, damage or disruption of the user information within or outside the browser. The appearance of various browser attacks executed on web browser causes real challenges to Internet user in protecting their information from an attacker. For example, suppose an attacker can inject malicious scripts that do not change the website's appearance, but silently redirect you to another website controlled by an attacker without your notice. This redirected malicious website might execute some malicious program to download a malicious file on your machine [1, 2]. The major goal of such attacks is to allow remote access to your machine to the attackers and to capture personal information, often related to obtaining a credit card, banking information and data used for identifying theft.

Like other software, web browsers are vulnerable to attack and exploit if appropriate updates and security patches are not applied. Moreover, a fully patched web browser can still be vulnerable to attack if the browser plug-ins and add-ons are not fully patched. The plug-ins and add-ons are third party software used to enhance browser functionality, but at the same time they can be malicious and vulnerable. The vulnerabilities and dangerous nature of browser extensions have been mentioned in the literature [3, 4], where the risks associated with the Firefox extension have been explained. The plug-in and add-on software are not automatically patched with the browser updates. Instead, they require some extra support from third party for updating their versions and patching vulnerabilities

Early versions of the browser are implemented with monolithic architecture [5] that combines the browser components into single memory and process space. For Example, a browser kernel and rendering engine run into a single process space. If an attacker can exploit one of a browser component, it can easily compromise the other browser components because all components run in the single process space. The vulnerabilities in this design cause an attacker to execute malicious code with full browser privileges. Older versions of popular browsers, such as Internet Explorer 7, Firefox 3, and Safari 3.1 were executed in a single operating system protection domain. Since the inception of vulnerabilities [6] in web browsers design, the browser research communities and developers have modified and extended the browser architecture to minimize the browser attacks.

The Chromium browser allocates the rendering engine into sandbox environment to provide isolation from browser kernel [7]. The architecture allocates high-risk components, such as the HTML parser, the JavaScript virtual machine, and the Document Object Model (DOM), to its sandboxed rendering engine. This feature helps to reduce the critical attacks on the Web browser. Internet Explorer 8 browser allocates separate process for tabs, each of which runs in protected mode. This architecture is designed to improve reliability, performance, and scalability [8].

This research mainly examines the browser attacks occurring due to (a) Firefox browser extensions and (b) User clicks. In former, we observed that high privileges in browser extensions and information flow among the critical browser and operating resources are the major cause of attacks. To understand the attacks caused by user clicks, we study new classes of clickjacking attacks in the web browser. Figure 1.1 illustrates the work flow of our research.

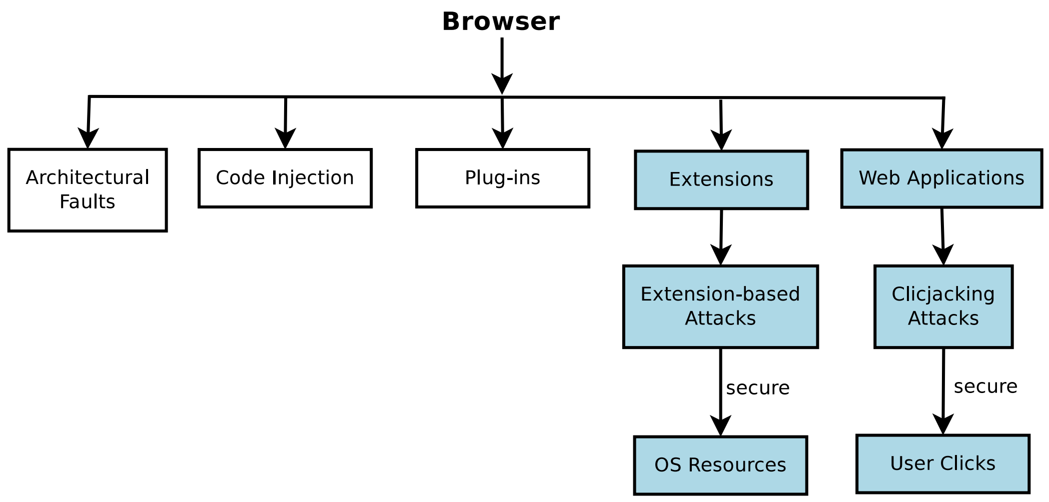


FIGURE 1.1: Overview of the research work.

1.1 Motivation

Over the last three decades, the web has rapidly transitioned from a set of interconnected static documents into a platform for the feature-rich dynamic and interactive web applications. Consequently, the web browsers have also evolved from mere user interfaces for remote documents into systems for running complex web applications. In other words, web browsers have become full-fledged operating systems for web-based programs. But the advancement in the technology leads to various complicated and more sophisticated attacks on Web browsers.

The browser-based attacks are initiated different attack vectors apart from malicious websites. The attacks may arise from trusted and legitimate web applications. Since all web applications, developers are not security experts and due to poor security coding the vulnerabilities occurs in these web applications. The attacker can exploit vulnerabilities present in trusted or legitimate websites to deploy attacks. For instance, an attacker can take advantage of vulnerabilities within the browser to run arbitrary code, which can steal user's sensitive information or install malware.

Traditionally, browser-based attacks are commonly originated only from malicious web sites [9]. However, the attackers have been introduced attacks that are beyond the malicious websites [10, 11]. These attacks not only exploits browser resources but also operating system resources. In particular, the third-party softwares such as Plug-ins and extensions can also be exploited by an attacker to initiate browser-based attacks.There are several questions, which could help to characterize an attacker: Who is the attacker? What source an did attacker use to enter into the system? What Vulnerabilities did he exploit? By answering these questions, we can get the clear picture of an attacker and what should be done next to protect the information.

1.2 Objectives

Browser attacks provide an unauthorized access, damage or disruption of the user information within or outside the browser. The browser add-ons (or extensions) runs with full browser privileges, which can be the major source of browser attacks. We explore new attacks caused with such high privileges in the web browser and devise new methods for detection of newly identified attacks. The objectives of the thesis are as stated below:

1. To identify new browser attacks arise from highly privileged browser extensions, which impacts privacy leakage in the browser by exploiting operating system resources.
2. To explore the vulnerable areas in web browsers that allow an attacker to execute privacy leakage and privilege escalation attacks.
3. To devise a method for detecting suspicious flow in high privileged browser extensions that can effectively identify the malicious behavior of browser extensions.
4. To identify colluding extensions attack in web browsers that results in privacy leakage attack in the browser.
5. To devise a method for advanced clickjacking attacks using finite state machines build from request and response characteristics yield from user click.
6. To built an isolated and sandbox environment for web browsers that protects operating system resources from being exploited by browser attacks.

1.3 Contributions of Thesis

In this thesis, we have identified new browser attacks occurring due to (a) Firefox browser extensions and (b) Misjudge User clicks.

1. We observed that high privileges in browser extension and information flow among critical operating resources is a major cause of attacks. This model analyzes information propagation among browser's resources (browser, web page, and host-OS components accessed through JSE). Using the results of our analysis, we can identify maliciousness in JSEs and provide comprehensive severity reports on their behavior.

2. We found that detecting a malicious flow in the extension is the partial protection against extension-based attacks. Therefore, we propose a sandbox and isolated environment to protect operating system resources from such attacks. Our sandboxed policies enforce on the web browser extension that restricts and limits their access to operating system resources.

3. We identified new browser attacks originate from collusion of two or more browser extensions. We demonstrate new attacks leverage this concept and causing privacy leakage in the web browser. In particular, we illustrate an important weakness in Firefox browser architecture and its XPCOM interfaces [12]. This weakness permits two extensions to collude with each other and share objects that are allocated in a same address space.

4. To understand the attacks caused by misjudged user clicks, we study new classes of clickjacking attacks in the web browser. In our analysis, we find that most of the attacks against users of the web application are caused by misjudged human perception on web pages. We demonstrate that current defense techniques are ineffective to deal with these sophisticated clickjacking attacks. Furthermore, these attacks are browser agnostics. Subsequently, we develop a novel detection method for such attacks based on the behavior (response) of a website active content against the user clicks (request). In our experiments, we found that our method can detect advanced Scalable Vector Graphics (SVG)-based attacks where most of the contemporary tools fail. We explore and utilize various common and

distinguishing characteristics of malicious and legitimate web pages to build a behavioral model based on Finite State Automaton (FSA). Our results demonstrate that proposed solution enjoys the good accuracy and a negligible percentage of false positives of 0.28% and zero false negatives in distinguishing clickjacking and legitimate websites. We also present an extensive comparison of our approaches with the related work in the area.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 provides the relevant background knowledge. We describe the background of browser ecosystem and various classes of attacks in the web browser. We also describe how extensions can execute attacks on the browser and operating system. The chapter also surveys the related work of extension-based browser attacks, secure browser design, securing browser extensions and securing misjudged user clicks. Chapter 3 presents our proposed approach based on the semantic analysis of browser extensions. In particular, we describe how a suspicious information flow among critical operating resources causing browser attacks can be detected. Chapter 4 describes our newly identified browser attacks using collusion of two or more browser extensions. We demonstrate new attacks leverage this concept and causing privacy leakage in the web browser. In Chapter 5, we proposed a security model for Firefox browser using isolated and sandbox environment. In Chapter 6, we build a FSA model to detect hijacked user clicks in the web browser. We present a novel detection method for such attacks based on the behavior (response) of a website active content against the user clicks (request). Finally, thesis conclusion and scope of future work is covered in Chapter 7.

Chapter 2

Browser Attacks and Countermeasures: An Overview

This chapter provides an overview of background information and the work related to the thesis. We first briefly discuss the browser components and how these components work together to provide the standard web browsing features (Section 2.1). Section 2.2 discusses various class of browser attacks. In Section 2.3, we discuss the taxonomy of browser-based attacks along with the working principles of current browser security model. In Section 2.4, we discuss the extensibility feature of web browsers and how they are susceptible to browser-based attacks. We then show how a combination of widespread availability, usage of standard web technologies, and powerful privileges can pose browser extensions as significant security threats to web browsers.

2.1 The Browser Ecosystem

The Web browser has become the dominant interface to a broad range of applications, including online banking, Web-based email, digital media delivery, social networking, and commerce services. Early Web browsers provided simple access to static hypertext documents. In contrast, modern browsers serve as de facto operating systems that must manage dynamic and potentially malicious applications. In particular, web browsers have also evolved from mere user interfaces for remote documents into systems for running complex web applications. Modern web browsers are complex multi-component systems providing many features analogous to operating systems. For example, web browsers extend its features using extensions (third-party software), which provide access to system level resources (e.g., files, network etc.).

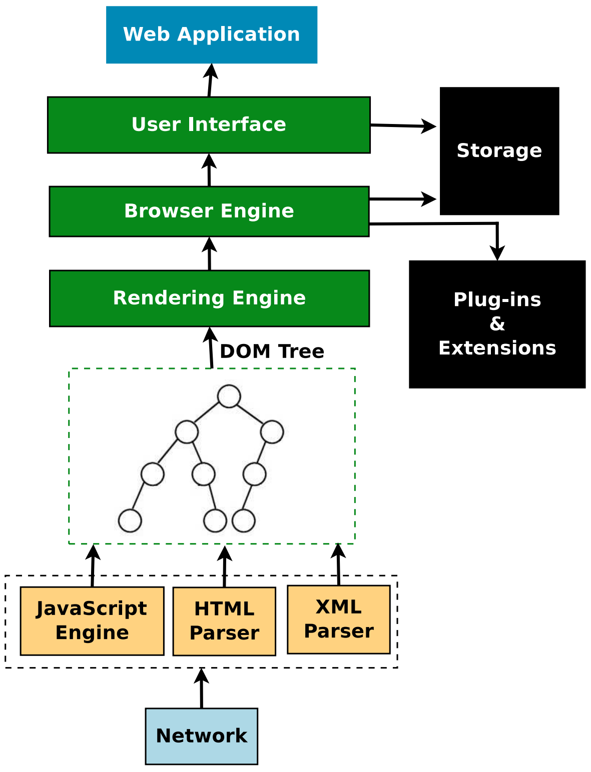


Figure 2.1: Major components of a web browser.

All major web browsers consists of various core components that interact in intricate ways to provide the web browsing experience. Figure 2.1 illustrated a simplified view of the major browser components. The Networking component handles fetching HTML documents over various network protocols. An HTML document is parsed using the HTML Parser and XML Parser, which produces a Document Object Model (DOM) [13] tree. The Rendering Engine uses this DOM tree to calculate the document layout for presenting on the screen. The Browser Engine controls the events performed by the web browser. The Rendering Engine is the "backend" of the Browser Engine. It loads the given URI and manages the session. The JavaScript Engine is a standalone component that is used by browsers to execute JavaScript code present on web pages. For this purpose, web browsers make a copy of the DOM tree available to the JavaScript Engine. The User Interface (UI) is the "frontend" of the Browser Engine. It displays

HTML documents based on the layout information received from the Rendering Engine. Additionally, the User Interface contains UI Widgets such as navigation buttons, menus, and the address bar.

Over the years, the browser research communities and developers have modified and extended the browser architecture to minimize the browser attacks. The OP web browser [14] runs multiple instances of a rendering engine, each in a separate protection domain isolated using different trust labels. The isolation namespace implemented in OP browser provides a protection domain to each web page. The Chromium browser allocates the rendering engine into sandbox environment to provide isolation from browser kernel [7]. The architecture allocates high-risk components, such as the HTML parser, the JavaScript virtual machine, and the DOM, to its sandboxed rendering engine. This feature helps to reduce the critical attacks on the web browser. Internet Explorer 8 allocates separate process for tabs, each of which runs in protected mode. This architecture is designed to improve reliability, performance, and scalability [8].

2.2 Browser-based Attacks

To better understand the security issues and challenges faced by the Internet community while using the web browser, we first need to understand the browser-based attacks on browser ecosystem. An attacker can execute various attacks on the web browser to target user information and system resources. The major goal of an attacker is to find the vulnerable targets in a browser itself, web programs or the user itself (e.g., Social Engineering Attack). We briefly discuss browser-based attacks as follows.

1. *Attacks through Web Applications.* Web applications are the most common way to provide services, data and rich features to the Internet users. Unfortunately, with the increase in the number and complexity of these applications, there has also been an increase in the number and complexity of vulnerabilities. Web applications run in the browser, any security loop hole in the browser will lead to exploiting the vulnerability in the web application. For example, an attacker who knows an unpatched security vulnerability in the victim browser, and somehow can convince the victim browser to render malicious web content. This activity

will allow an attacker to inject malicious script into the browser page, this attack is known as Cross-site scripting attack [15].

The common web application attack is code injection attack, such as HTML injection, Cross Site Scripting, and SQL Injection [16] allows an arbitrary script to execute in a browser. Other attacks originated from web applications are Cross-site request forgery (CSFR) vulnerability

2. *Attacks through Browser Extensions.* The extension-based browser attacks are the serious concern to user information, browser, and system. In browser, the extension scripts receive greater privileges than web application scripts. As browser extension code runs with much higher privileges, the malicious effects can be devastating. The extensions in the browser are allowed to read cookies, get access to browser APIs, open a network connection to gain access to the victim machine, and spawn OS process. For example, an attacker can execute privilege escalation attacks [17] on the victim using code injection attack. It then can access cross-domain network information, sensitive browser APIs and user's file system [10, 11].

3. *Attacks through Plug-ins.* The plug-in based attacks arise from the vulnerable plug-in installed in a browser. The plug-in are installed as third party software to support the additional feature in the browser. A user can install Plug-in whether for document reading, interactive content, Java run-time environment or ActiveX controls can be subjected to attack. The attackers look for vulnerabilities in plug-ins to carry out attacks like drive-by-download and Clickjacking attack. The attacker exploits vulnerabilities in JavaScript code running under Java run-time environment, one of the most susceptible languages to attack. For example, many attacks do spawn a pop-up message from Java asking for permission to execute a malicious Java file, but it's often too hard for users to tell which browser window created the pop-up. One accidental allow click is all it takes to start an attacker to control victim browser. Once the malicious Java applet is running, it takes only seconds for the malware payload to execute an attack.

4. *Attacks through Architectural Vulnerabilities.* The browser architecture consists of several sub-systems, such as user interface, browser engine, rendering engine, networking, JavaScript interpreter, XML parser, display backend and data persistence subsystem [5]. At runtime, all components are instantiated and executed

in the same protection domain. Consequently, a fault in any of them can compromise all the others. For example, if an attacker can craft a successful attack on the JavaScript interpreter, it can take advantage of the data persistence subsystem and access available information. The attacker might also exploit the core component vulnerabilities of web browser. The authors have reported various architecture vulnerabilities in the web browser, which are patched by browsers. However, attackers have adopted different new ways to exploit the web browser. For instance, the vulnerability in rendering engine can be exploited, which allows an attacker to render a malicious page inside web browser through which an attacker can execute arbitrary malicious code with high privileges.

2.3 Taxonomy of Browser-based Attacks

This section discusses the taxonomy of browser-based attack that characterizes the flow of an attack in the browser. The taxonomy describes the complete attack action steps that an attacker carries out to execute an attack in the browser. The classification is based on the attacker and browser side sequence of actions. The attack starts with an attack vector, vulnerability exploitation and finally reach a target. The browser security model acts as a barrier to stop an attacker in successfully executing an attack. In particular, these classifications represent the attack actions performed by an attacker to produce a browser-based attack and browser actions to stop an attacker. The taxonomy in Figure 2.2 shows the classification chart. The taxonomy describes how an attacker enters into the browser by adopting attack vectors, how it propagates by exploiting vulnerabilities at different levels of browser, what area it targets. The security model plays its act if an attack is within its scope. Otherwise, the attacker successfully targets the victim user.

1. *Attack Vector.* An attack process starts with an attack vector that an attacker uses as an entry point to the victim browser and reaches its target. An attack vector is defined as a path by which an attacker can gain access to a victim host. Attack vector classifies various entry points for the attackers. The most common trick adopted by an attacker is social engineering tricks in which an attacker lure a victim user to perform some task and gets infected. Our understanding of the

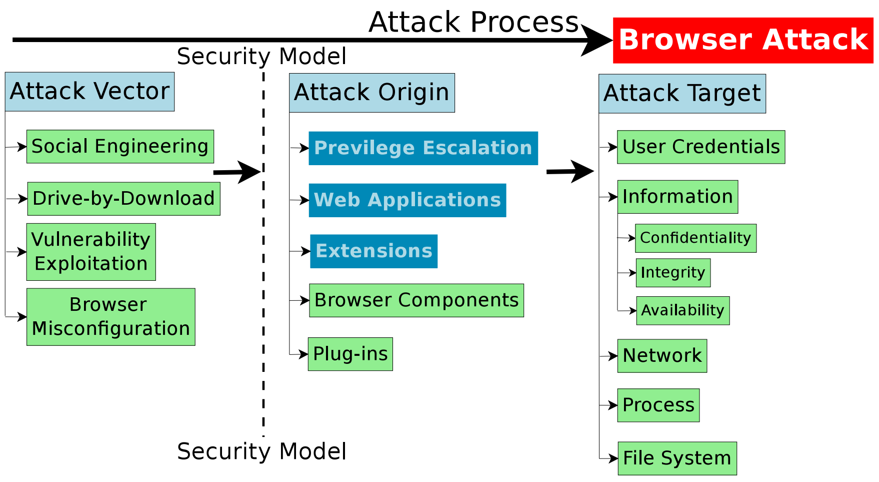


FIGURE 2.2: A Taxonomy of attacks in web browsers.

attack vector motivates us to analyze weak points in the system which can be used by an attacker to route into the system.

2. *Browser Security Model.* The web browser enforces various security control techniques to encounter attacks in the browser. The key to attack web browser and applications is to find the vulnerabilities in the browser security model or circumvent one of the policies. Each security control attempts to be independent of the others, but if an attacker can inject a little JavaScript in the wrong place, all the security controls break down. Figure 2.3 illustrates the classification of various security policies and mechanisms that are enforced on the web browser.

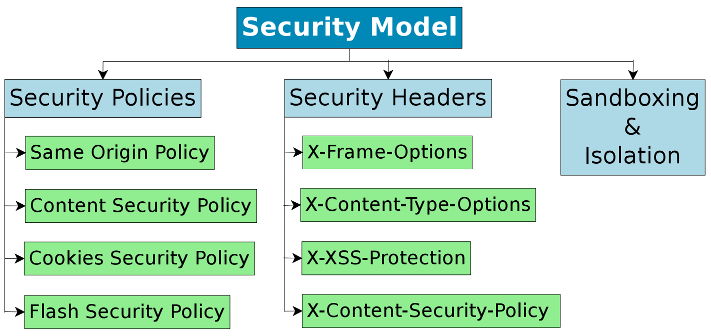


FIGURE 2.3: Browser Security Model.

3. *Exploiting Browser Vulnerabilities.* The taxonomy describes various vulnerable points in the web browser that an attacker can exploit. The vulnerabilities can originate from browser architecture, browser components, plug-ins/extensions and web applications. For instance, an attacker can gain system-level privileges

through the browser extensions to modify the browser configuration to taint the normal functioning of running the browser.

4. *Attacker Target.* The attacker target(s) are the logical entities of the browser, such as user credentials, web page information, file system and running the process. Furthermore, an attacker can also target physical entity, such as, network protocols. For instance, an attacker can steal cookies to execute Session Hijacking attack. A malicious extension can access to network APIs to transfer user data via networks. For example, extension based malicious scripts can send network requests to arbitrary web servers using `XMLHttpRequest` API.

2.4 Extending Browser Functionality

Browser functionalities can be widely extended by browser extensions. Nowadays, modern web browsers support a modular architecture that allows third-party extensions to enhance the core functionality of the browser [18]. Browser extensions enjoy high privileges, sometimes as high as those of the browser itself. Extensions have access to browser resources not usually available to scripts running on web pages. High privileges allow extensions to read and modify arbitrary web pages, accesses browser components, and even customize browser interfaces. Furthermore, extensions are also not subject to the same origin policy [19] that applies to scripts on the web applications. With access to these and other capabilities, malicious extensions might put browser at risk of information breach and privilege escalation attacks. In particular, malicious extensions can misuse these privileges to compromise confidentiality and integrity, e.g., by stealing sensitive information from web pages, such as cookies and passwords, or executing malicious code on the host.

Each major browser has its unique extension system. For example, Mozilla Firefox and Internet Explorer use the Component Object Model (COM) approach. The com-based architecture facilitates separation between the design and the implementation of components. In this section, we briefly discuss the extension systems of the three most popular web browsers: Internet Explorer, Mozilla Firefox, and Google Chrome.

1. **Chrome extension model.** Google Chrome extensions are written using technologies like HTML, JavaScript, and CSS. Google Chrome extensions can use all the APIs that are restricted to web pages, for example, `XMLHttpRequest`, JSON and HTML5 local storage. Extensions with high privileges are allowed to modify the user interface of Google Chrome browser. The security architecture of Google Chrome gives minimal required privileges and restricts access to the file system and other resources to the extensions. Furthermore, in Google Chrome, extensions run in the separate process. The extension process is isolated from the browser process and other operating system processes. Such an isolation restricts Google Chrome extensions in accessing core components of browser, such as kernel, rendering engine, and thus an extension cannot compromise core components of the browser.

   Google Chrome uses extension manifest file that describes the extension's privileges. Extensions specify their resources and the capabilities they require in an extension manifest file. When a user tries to install an extension, Google Chrome reads the extension manifest and asks for user permission to allow or deny extension privileges. Furthermore, Google Chrome isolates memory of browser kernel, web pages, extensions, and plug-ins. The isolation is achieved by using a separate process for every extension.

2. **Internet Explorer (IE) extension model.** Internet Explorer (IE) supports several mechanisms for browser extensibility. The most commonly used tool is Browser Helper Objects (BHOs). BHOs (usually native binaries) can be used to add additional functionalities to the browser such as modifying the user interface, adding toolbars, explorer bars, and shortcut menus.

   The security architecture of IE is based on privilege separation and the ability to create processes with lower privileges. In particular, it allows to run the browser with limited rights, which protects the operating system in case the browser gets compromised. This isolation is necessary, as the extensions run with the same privileges as the browser. The new versions of IE (after IE7) run in protected mode, which helps protect users from attack by running the IR process with greatly restricted privileges. Protected mode significantly reduces the ability of an attacker to write, alter or destroy data on the users machine or to install malicious code.

Unlike Google Chrome, which uses entirely isolated process, IE run extensions (BHOs) in the process where they are called. The isolation depends on the number of created tabs. As the number of tabs grows, new web pages are forced to share the process with other web pages. Thus, the extension called from shared memory space have direct access to the other websites and extension in that process.

3. **Firefox extension model.** Firefox extensions are developed using widely used web technologies such as JavaScript, CSS, XUL Overlays, RDF, and XUL templates [20]. This extension system was originally intended for expert browser developers, but its use has expanded widely. The developers familiar with basic web technology can quickly develop Firefox extensions. The extensions in Firefox browser interact with browser components through XPCOM framework [12], the Cross Platform Component Object Model, which provides a variety of services within and across the browser.

   Due to unrestricted access, powerful privileges, and ease in development, the XPCOM interfaces are very popular among extension developers. As these developers are often not security experts, there is a high chance of bugged/vulnerable or malicious code that is liable to exploitation by an attacker. The Firefox JSEs have critical functionalities provided by XPCOM interface and APIs that may pose security risks in browsers. For example, a password stealing from web page's password input field by accessing browser DOM [13], accessing cookies to steal session, is a critical security threat. JSEs exploit this threat to leak privacy. Also, the interfaces also allow JSE to access arbitrary files from a file system and invoke new process on the host system.

   When comparing with Google Chrome and IE, Firefox uses single process memory model. All the windows, tabs, plug-ins and extensions run inside the same process. Threads that run inside the browser process share the address space with extensions, plug-in, tabs and thus there are no borders between threads. With a single process model, Firefox extensions can access and modify the stack of a random thread inside the process. Furthermore, Firefox does not provide any isolation between extensions. As a result, an extension can change the functionality of other extensions.

2.4.1 Privilege Escalation Attacks through Extensions

To enhance the browser functionalities and get customizable features, the extensions are required to execute with the full chrome privileges. The COM [21] interface include services such as file system access, process launching, network access, browser components and APIs access. These interfaces allow browser extension to have full access to all the resources browser can access. For instance, Firefox supports the XPCOM APIs that are implemented in C++ or JavaScript. In Firefox, JavaScript code of extensions receives higher privileges [22] than JavaScript code of web applications.

The JavaScript code of JSEs and chrome can access any browser and system resources through XPCOM interface. These high privileges in browser may result in privilege escalation attacks, which allows an attacker to execute malicious scripts with extra privileges space. The authors have shown that malicious extension could spy on users and install malware [23] [24]. Furthermore, these elevated privileges are not bounded with any browser policy such as same-origin policy (SOP) [19]. In particular, the web applications are bound with SOP while extensions can override the SOP and access cross-domain components as well. Table 2.1 summarizes the privileges of browser extension.

Table 2.1: Privileges associated with browser extensions.

Privileges	**Description**
Modifying DOM	modify Page contents, add new contents
Accessing browser components	browser password manager, cookies, bookmarks, history, preferences, etc.
Accessing OS	processes, files system, network
Modifying browser	user interface, chrome
Accessing network	sending same and cross-domain XMLHttpRequest

2.4.2 Firefox Extension System

In this thesis, we focus on JavaScript-based extensions of the Mozilla Firefox browser. We opt Mozilla Firefox for some reasons.

1. First, Mozilla Firefox is one of the earliest browsers to support JavaScript-based extensions. As a consequence, Mozilla Firefox now supports a very robust and mature JavaScript-based extension system.

2. Second, as of this writing, the Mozilla Firefox browser is the popular browser and its extensions are used by millions of users[1]. The Mozilla add-on repository hosts over thousands of extensions, providing us plenty of opportunities for experimentation.

3. Some public disclosures about Mozilla Firefox extension vulnerabilities are also available, providing us the opportunity to investigate and analyze the common patterns in those vulnerabilities. Finally, Mozilla makes the source of the Firefox browser publicly available, which allows us to experiment, implement, and evaluate our solution in a real browser.

In the rest of the thesis, "extensions" and "JavaScript-based extensions of Mozilla Firefox" are used synonymously unless stated otherwise.

XPCOM platform is similar to Microsoft COM [21], which provides a set of core components, classes related to file and memory management. This platform also provides core elements of threads, basic data structures (strings, arrays, variants). Figure 2.4 illustrates the interaction of various Firefox components and the extension. The extensions in Firefox browser interact with browser components through XPCOM framework. This interaction provides a variety of services within the browser, such as file system access, process launching, network access, browser components and APIs access.

The JavaScript in extensions uses XPconnect [25] to invoke XPCOM components. XPConnect acts as a bridge between JavaScript and XPCOM. The user interface of Firefox extension is programmed using XUL (XML User Interface Language). Firefox extensions can randomly change the user interface of the browser via a technique known as overlays [26] written in XUL. CSS are used to add the presentation and visual styles of the Firefox extension.

The main weakness in Firefox extension system is the unrestricted privileges it assigns to the extensions. Moreover, the system has not implemented any security policy to restrict

[1]http://www.w3schools.com/browsers/browsers$_s tats.asp$

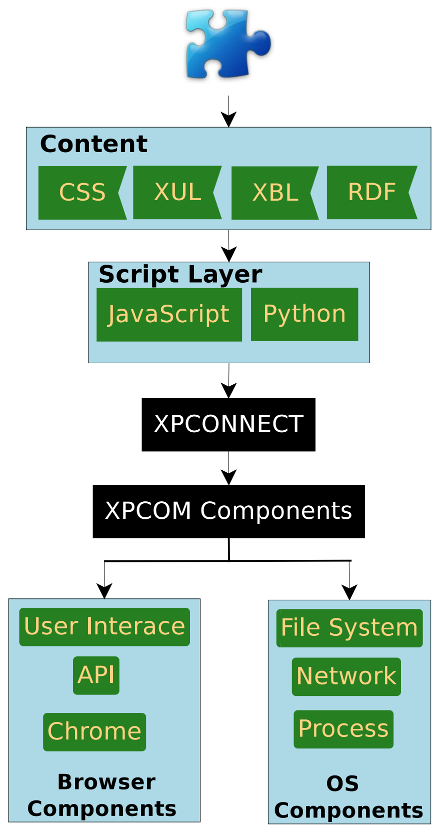


FIGURE 2.4: XPCOM Architecture View in Firefox extension system.

the rights of extensions. In [27], we have discussed various attack vectors showing how an inexperienced user might install a malicious browser extension without suspecting it is malicious. We also discussed various weaknesses in Firefox extensions arise due to unrestricted privileges that can be used for malicious purposes.

2.4.3 Security Risks with XPCOM Interfaces

Table 2.2 illustrates various attack vectors used by an attacker. We have illustrated the resources that are exploited along with the interfaces used to invoke these resources. Also, Table 2.2 defines the severity of an attack action in terms of critical, high, moderate and low ratings. For example, the information accessed from a web page password field, steal cookies to hijack session is always essential to the user, and hence we have rated it critical. Interfaces that allow to access arbitrary files and processes on the host system are also very critical because using them an attacker can initiate a malware injection to alter user les. Firefox and Chromium define the security severity ratings [28], [29] based on the information flows from various resources. We have adopted their rules to define severity rating for different information flows in the browser.

Table 2.2: Attack Vectors for Extension-based Attacks.

Attack Class	**Resources Exploited**	**Interface Used**	**Rating**
Accessing password from web page	DOM	`nsIDOMNode`, `nsIDOMElement`	Critical
Launching arbitrary local application	Invoke process	`nsIProcess`	Critical
Cross-domain access and violation	Network channel	`nsIXMLHttpRequest`, `nsIHttpChannel`, `nsITransport`	High
Profile attack [27]	File system	`nsIFile`, `nsILocaFile`, `nsIOutputStream`	Critical
Accessing confidential data	DOM	`nsDOMNode`, `nsIDOMElement`	High
Stealing local files	File system (OS)	`nsIInputStream`, `nsIFileInputStream`, `nsILocalFile`, `nsIFile`	High
Accessing browser history	Browser component	`nsIbrowserHistory`, `nsIGlobalHistory`	Moderate
Accessing stored passwords	Password manager	`nsILoginManager`, `nsILoginManagerStorage`	Moderate
Accessing events	Keyboard & Mouse events	`nsIEventListenerService`	Moderate
Session stealing	Cookie manager	`nsICookieManager`, `nsICookie`, `nsICookie2`, `nsICookieService`	Critical
Accessing bookmarks	Bookmark service	`nsINavBookmarksService`	Low
Setting browser preferences	Preference system	`nsIPrefService`, `nsIPrefBrach`	High
Setting extension preferences	Preference system	`nsIPrefService`, `nsIPrefBrach`	High
Accessing page Information (Images/text)	DOM	`nsIDOMNode`, `nsIDOMElement`	Low
Turn on/off private browsing mode	Browser Component	`nsIPrivateBrowsingService`	Moderate
Access to windows registry system	Windows registry	`nsIWindowsRegKey`	High

The next few sections discusses a range of work related to our thesis. We begin by discussing research efforts that propose browser attack models and security risks pose by the browser attacks (Section 2.5). The researcher had highlighted the alternative browser designs for addressing the security and reliability problems for web browsers. In particular, there has been a large body of work that aim to isolate browser components and provide a sandbox environment for web applications to execute in the web browser(Section 2.6). Although some of these work address architectural issues and plugin-based extensibility, we do not find any approaches among these which directly address the system-level attacks arises from the malicious flow in JavaScript-based extensions. Our approach also uses a static taint analysis for analyzing the JavaScript code of extensions. Therefore, we discuss work that utilize static analysis techniques to analyze JavaScript programs (Section 2.7).

In recent years, a number of research efforts have been made to secure the browser and user privacy from extensions-based attacks in the web browser. However, none of work ever addressed the security risk caused by the collusion of two or more extensions. Apart from the extension-based attacks in the web browser, we discuss research efforts that address the click-hijacking problem in web browsers (Section 2.8).

2.5 Extension-based Attacks in Browser

In [10, 30, 31], the authors have taken a practical approach and demonstrate examples of possible attacks on Firefox extensions. They discussed the possible vulnerability in reviewed existing Firefox extension, which could be used to exploit extensions and launch concrete attacks such as remote code execution, password theft, and file system access. But the author has not mentioned about the attack vector, which an attacker used and how these malicious extensions spreads. We use attack scenarios inspired from these two works to create our malicious extensions as proof-of-concept to show vulnerable points in Firefox extension system and show how extensions install and spread.

The authors have studied the security and privacy in browser extensions for different Browsers. Martin Jr. et al [32] investigate privacy issues in IE 6 extensions, where they found some extensions monitoring users' behaviors or intercepting and disclosing SSL-protected traffic. Some author had studied the security and privacy issues in Firefox extension, the work by Ter Louw et al. [11] highlights some of the potential security risks posed by Firefox extensions. In [33], the author investigate privileges in 25 Firefox extensions that are necessary for extensions' functionalists, and found that only 3 out of the 25 extensions would actually require the most powerful capabilities of the privileges Firefox extensions all have, violating the least privilege principle. Liu et al. [34] assess the security features of the Google Chrome extension system. They demonstrate that it is possible to write powerful malicious extensions in spite of the security options provided by the browser.

In [35], raised serious concerns about the privacy of users. The authors have conducted a comprehensive empirical study to assess the feasibility and accuracy of inference attacks that are launched from the extension API of Social Network Systems. Recently in [36], the authors have presented Hulk, a system to perform dynamic analysis for Google Chrome extensions. They have demonstrated the effectiveness of HoneyPages and event handler fuzzing to elicit malicious behavior in browser extensions. To the best of our knowledge, no other work has considered an attack scenario due to inter-communication between objects of two extensions. In our work, we have demonstrated the attacks on user privacy using two colluding extensions in which individual extension is claiming to be legitimate.

The extensions in the browser are similar to Apps in Android devices. The concept of inter-component communication is not new in Android devices. The attacks using inter-application communication system in Android devices deal with standard threats that apply to all messaging systems. For example, eavesdropping, spoofing, denial of service, etc. The researchers have discussed various privilege escalation attacks with colluding application on mobile (Android) domain using inter-application collusion [37–40]. As we have shown in this paper, the similar threats are also present in browser extensions. Problems in XPCOM interfaces in sending/receiving notification (collusion) lead to various privacy leakage attacks.

Researchers have shown how traditional browser security model (Same Origin Policy), which isolates the content from different origins. However, the implementation of this principle tends to be error prone due to the complexity of modern browsers [41]. Furthermore, the same-origin policy is too restrictive for use with browser extensions and plug-ins, which allows an attacker to execute attacks [42–44].

2.6 Incorporating Security in Browser Design

Since the inception of vulnerabilities (Silic Delac, 2010) in web browsers design, the Browser research communities, and developers have modified and extended the Browser architecture to minimize the Browser attacks. Researchers have proposed a number of alternative browser designs for improving security and reliability. Most of these work view a web browser like an operating system and restructure the browser to enforce protection principles inspired by operating systems.

The most closely related work are the building a secure browser environment was presented in [45–47]. These researchers are all focus on designing new browser architecture that relies on the underlying OS policies (e.g., file system permissions) to enforce browser security. Tahoma [46] shares many of the same design principles as discussed in [45]. Tahoma uses Virtual Machine Monitors (VMMs) to provide isolation for different web-based applications, and a manifest to help craft their network policy. In this architecture, the web applications are treated more like desktop applications. The web application runs inside a virtual machine is represented by a browser instance. This architecture isolates web applications from each other; however, it is unclear whether the browser instances themselves could be extended and how it might affect the security of the architecture.

The OP [45] use OS-level mechanisms for isolation various browser components from the Operating System. In particular, each browser components runs in a separate OS process isolated from each other. The communication between the components is managed by a lightweight browser kernel. OP tracks interactions of components at a finer level of granularity. Moreover, OP allows browser-specific security policies to be specified which prevents the DOM of a web page from being accessed by an untrusted plugin. Although OP facilitates plugin-based browser extensibility, it does not provide any other extension features, such as JavaScript-based browser extensions.

In [47], the authors propose a new browser architecture that relies on the underlying OS policies (e.g., file system permissions) to enforce browser security. They present a secure browser design that leverages an OS specific sandboxing system. The browser runs on top of SubOS, an operating system that restricts the privileges of a process based on the object that process is executing. When the browser tries to execute objects downloaded over the internet, the browser process is sandboxed to restrict the downloaded objects access to sensitive resources. Although their approach sandboxes helper applications invoked by the browser to execute unknown content, plugins or extensions is not addressed in their work.

MashupOS [48] proposes new abstractions to facilitate improved sharing among multiple principles hosted in the same web page. In addition, the solutions are implemented to provide powerful security policies for web Mashup application that communicates with each other in browser [49], [50]. However, these solutions can be applicable only to web applications. An obvious drawback of these solutions is that these policies do not apply to browser extensions or plug-ins.

The idea of sandboxing web browsers has been used in the past. For example, VMware released a virtual-machine based "Web browser appliance", containing a check pointed image of the Firefox browser on Linux [51]. As another example, GreenBorder [52] augments Windows with an OS level sandbox mechanism similar to BSD jails [53], in order to contain malicious content arriving through Internet Explorer or Outlook. Current research efforts to retrofit today's web browsers help to improve security but fail to address the fundamental design flaws of current web browsers.

The major theme of these projects is to restructure web browsers to support OS primitives in order to enhance security and reliability. Except OP and Google Chrome, none of these approaches addresses browser extensibility. Moreover, OP only addresses plugin-based extensibility problems, not JavaScript-based extensions. In contrast, we focus on the security issues associated with JavaScript-based browser extensions. Unlike these approaches, our solution does not require a redesign of the browser, rather leverages the existing components of the browser.

Moreover, the solutions mentioned above provide sandboxing policies at the web-application level, which is too coarse-grained. These policies fail to isolate different scripts and objects within the same web application. Combining current fine-grained isolation techniques with sandboxing systems does not provide a complete solution since it would still rely heavily on the underlying browser itself. In contrast to these previous work, we have not modified the browser architecture. Instead, we proposed a sandbox environment that is not applicable to a web application. Instead, we develop a sandbox policies using open-source SELinux policies, which restricts OS resources from the browser process.

2.7 Security against Browser Extensions

Information flow tracking and control has been the basis of many operating systems and programming language designs over the past several decades. There are few approaches that uses static analysis of browser extensions differing in precision, runtime, scope and focus. There are primarily two classes of threat models associated with browser extensions described in the literature.

The first one is the malicious extension threat model. In this model, the attacks originate from malicious extensions. The malicious extension authors are attackers themselves, and try to trick users into installing a fake or altered extension to attack victim user.

The second model is the benign-but-buggy (vulnerable) extension model. These extensions are well-intentioned but contain exploitable vulnerabilities that let an attacker execute malicious code with the privileges of the extensions. In particular, this extension behaves legitimately but contains malicious flow that can be exploited by malicious web site operators and active network attackers. In this section, we discuss the research effort addressing both kinds of threats.

Malicious JSEs. Attackers themselves write malicious JSEs, and simply by installing the JSE a user becomes a victim. Zhuowei et al. [54] proposed a SpyShield, which dynamically monitors for malicious Browser Helper Objects(BHOs) [55]. SpyShield enforces access control rules for untrusted BHOs based on the sensitivity level of user's information.

Ter Luow et al. [11] were the first to highlight some of the potential security risks posed by Firefox JSEs. They proposed run-time monitoring of XPCOM calls for detecting suspicious activities in a manner akin to spyshield [54]. The suspicious access by JSE to sensitive browser components and resources are monitored and are permitted or denied based on predefined policies. Our technique is complementary to these techniques since, we statically analyze the JSE source code before installing on the browser and avoid the risks caused due to false negatives in the discussed approaches. In addition to that, it is difficult to detect accurately attack flow patterns at run-time because it is difficult to distinguish the flow caused due to genuine browser activity or malicious JSE.

Benign-but-buggy JSEs. The attacks may also be induced by the vulnerabilities exposed in benign JSEs. These benign JSEs contain exploitable vulnerabilities that can allow an attacker to execute malicious code with the privileges of the extensions. However, the vulnerability can sometimes be introduced by an ill-intentioned JSE developer so that it can later target the victims Browser. There are several prior work [56–59] presented for detecting vulnerable and malicious JSEs and web applications through information flow and taint analysis methods.

VEX uses a static information flow analysis on the JavaScript code of Mozilla JSEs to identify vulnerable information flow patterns. VEX primarily focuses on vulnerable flow induced by JavaScript objects. In addition to that it also checks few unsafe programming practices that could lead to security vulnerabilities. In particular, VEX is a tool for vetting JSEs to analyze security vulnerabilities in JSEs. But the JSEs can access various XPCOM interfaces to interact with critical browser and system resources, and hence has many other flow patterns causing maliciousness. In contrast, our work analyzes various other flow patterns to detect maliciousness in non-vulnerable extensions. Since, VEX do not check every flow in their analysis, it may have the possibility of false positives. On the contrary, our security model covers the larger set of flow patterns that originates from JavaScript as well as XPCOM APIs.

Researchers have also proposed dynamic analysis techniques to offer runtime protection. In [58], the author, has proposed a policy enforcer for the Firefox browser that gives fine-grained control to the user over the actions of existing JavaScript Firefox extensions. However, their work is limited to only ten legacy JSEs and they define policies based on only four attacks and hence the approach is not scalable due to limited policies.

Another related runtime approach SABRE [57] tracks the flow of JavaScript objects from sensitive sources to sensitive sinks inside the Mozilla Firefox browser. SABRE also includes the flow between browser components by tracking XPCOM objects. SABRE assigns a label to every JavaScript object, which marks whether the object contains sensitive information. However, it is difficult to detect accurately attack flow patterns since many legitimate extensions also demonstrate flows similar to attacks. Furthermore, a major drawback of the dynamic analysis approach is the high-performance overhead due to whole-system tracking of objects. In contrast, in past, the static analysis on JSEs have only tracked information flows within resources accessed through JavaScript objects. This analysis is partial if JSE also contains XPCOM objects. The browser provides access to resources through XPCOM objects, which have not been statically analyzed in the past. Therefore, we advocate a static analysis based approach, as opposed to dynamic ones, for its complete code coverage and scalability.

Another approach, which is based on Hidden Markov Model (HMM) is presented in [59]. HMM model uses potentially relevant features for constructing the models followed by their probability distribution from a given set of extensions. The model consumes these features to generate HMM models for benign, vulnerable, and malicious extensions separately. It characterizes the features based on the presence of JavaScript and XPCOM APIs in benign, vulnerable and malicious extensions. However, differentiating JSEs with these features may not produce accurate results because it may be possible that a legitimate JSE may use sensitive API, which is used by malicious and vulnerable JSE. Also, the dataset that this work have used to extract features is not justified. So we can not distinguish the JSEs just by looking at the presence of APIs. In contrast, our work finds the malicious flow in a JSE to distinguish it as a malicious JSE from a legitimate JSE. Moreover, we also assign the security ratings to each flow that is found in a JSE, which is helpful in estimating the severity of a JSE.

Djeric et al. [60] also adopted a dynamic taint analysis based approach for vulnerable browser extensions. In this, the authors have studied the vulnerabilities caused by JavaScript in extensions and suggested solution against them. Their approach taints all the data received from untrusted sources and prevents the execution of this unprivileged data from being compiled into privileged bytecode. A major drawback of the dynamic analysis approaches is the high-performance overhead due to whole-system tracking of objects.

IBEX [61] is a general purpose browser extension development system that provides verifiable security guarantees. IBEX requires the redesign of the extension system. IBEX provides a new language and policies that let extension use common browser functionalities. The extension code can then be verified with respect to the policy using theorem provers. However, to use IBEX, extension developers need to program the extensions in a verifiable language other than JavaScript. Moreover, each browser provides unique APIs for its extension system that is constantly updated with new features, making it challenging to develop extensions using a general purpose system like IBEX.

Table 2.3: Comparison with other extension vulnerability and malicious mitigation approaches.

	Comparison Parameters			
Work	**Technique**	**JavaScript Flows**	**XPCOM Flows**	**Requires Redesign?**
VEX [56]	Static Information Flow Analysis	Yes	Partial	Yes
SABRE [57]	Dynamic Taint Analysis	Yes	Yes	No
Hossain et al. [59]	Classification based on Hidden Markov Model	Partial	Partial	No
SENTINEL [58]	Policy Enforcer	No	No	Yes
IBEX [61]	Formal Verification	Yes	No	Yes
Djeric [60]	Dynamic Taint Analysis	Yes	Yes	Yes
Our Work	Static Information Flow and Taint Analysis	Yes	Yes	No

Table 2.3 compares our proposed approach with the previous approaches. In particular, we note that while some of the approaches require a redesign of the extension system [61], [60] [58], some of the approaches track limited from arises from XPCOM object. In contrast, our work neither redesigns the extension system nor does it require the developers to write an extension in a new language (e.g., IBEX). The dynamic taint analysis approaches need to modify various browser components (e.g., DOM system, privileged components) along with the JavaScript engine for tracking flows across different browser components. This instrumentation costs browsing experience in terms of performance. In contrast to the aforementioned work, our work focuses on the security issues in Firefox extension system. To understand the capabilities of Firefox extensions, we have implemented malicious Firefox extensions using legitimate JavaScript methods and APIs, which has been studied and reported by these authors. Our approach is based solely on the syntax and semantics of the JavaScript language.

2.8 Hardening Browser Clicks against Clickjacking Attacks

Mozilla Firefox browser developers are the first to report the misuse of transparent, or hidden iframes in their bug report [62]. However the term *clickjacking* was first introduced by Hanssen and Grossman in 2008 [63]. The early clickjacking attacks were completely focused on unsafe iframe based web pages. Hansen has presented several attack vectors and proof of concepts for clickjacking attack [63].

In [64], Barth et al. has explored the unsafe use of iframes in the web page, analyzed the frame navigation policies, and advocated a stricter policy to prevent attacks. The attackers are constantly looking for exceptions and vulnerabilities in the browser. For instance, the browser bugs had employed the unsafe frame communication to circumvent the same-origin policy [19] checks with the aim of stealing or modifying sensitive user information. The major research focus has been done on detection and mitigation of iframe based clickjacking attacks. The clickjacking attacks are not limited to the use of invisible iframes, but can be conducted in a variety of different ways. Apart from iframe based attacks, we have also discovered some new attack classes and implemented attack signatures from these classes.

There are several proposals for Clickjacking defense and countermeasures. The Browser offers same-origin policy [19] to tackle cross-domain communication between web pages. But, it fails to stop any of the clickjacking attacks reported in the literature. As a result, several anti-clickjacking defenses have been proposed (many of such ideas were suggested by Zalewski [65]), and some have been deployed by browsers as follows.

1. **Web-key Authentication.** The web-key authentication scheme proposed in [66] uses unguessable secrets in URLs instead of cookies for authentication. This approach can mitigate confused Social Engineering attacks such as clickjacking and CSRF [67]. Unfortunately, this approach degrades the user experience and the benefits a web page get from cookies [68]. It also requires the server side modification to handle the new unguessable secret. In contrast to this, our approach does not need any server-side modification and it compatible with the current web pages.

2. **User Confirmation and UI Randomization.** In [69], the author, has presented a client side defense to prevent clickjacking attack. In this, for every out-of-context click the system would generate a confirmation dialog for end users. Facebook currently deploys this approach for the *"like"* button, asking for confirmation whenever request come from blacklisted domains. Unfortunately, this approach degrades the user experience, especially on single-click buttons. Another technique is to protect a target element by randomizing the UI (GUI element) layout of a legitimate web page. Thus, an attacker page failed to create an exact overlap with the legitimate web page.

3. **Detecting Frame Overlays.** The authors had proposed the client-side solutions to detect frame overlays. One of the solution in which a module is integrated in *Noscript* Firefox extension [70]. The *clearClick* module aims for clickjacking protection by extending the browser's functionality to detect malicious clicks. ClearClick monitors every click on the web page, which occurs during user interaction on framed web sites of different origin. Once a user attempts to click a link on a framed website that appears to be the victim of obfuscation attempts from its parent and cross-origin document, the interaction will be blocked. However, the researchers have proposed the methods to circumvent the clearClick protection [71].

   Balduzzi et al. [72] develop the ClickIDS Firefox extension. It compares the bitmap of the clicked object on a given web page to the bitmap of that object rendered in isolation (e.g., without transparency inherited from a malicious parent element). It alerts users when the clicked element overlaps with other clickable elements. Unfortunately, ClickIDS cannot detect attacks based on partial overlays or cropping. In contrast to these frame overlays solutions, our approach enables a robust handling because our approach considers the complete or partial overlapping of every web page element, which is generating a click event.

4. **Framebusting.** A more effective defense is framebusting or avoids the unauthorized frames from being rendered in iframes. This is achieved using a small snippet of JavaScript code in the target element, which first checks if the page that contains the script is currently framed [73]. Other methods to deploy framebusting is by using X-Frame-Options [73, 74] and CSP's frame-ancestors [19]. A fundamental limitation of framebusting is its incompatibility with target elements that

are intended to be framed by arbitrary third-party sites, such as Facebook *"like"* buttons. Also, if JavaScript is blocked by means such as Noscript, XSSfilter [75], then this framebusting code will not work.

All the defense mechanisms are focused on clickjacking attacks crafted using frame/iframes. However, an attacker can craft more advanced attacks, such as SVG-based attacks, event bubbling, (Cascading Style Sheets) CSS-based attacks. In contrast to the approaches that examine the basic clickjacking attacks in websites, the focus of our work is on the detection of novel advanced types of clickjacking attacks. To the best of our knowledge, this is the first work investigating these novel clickjacking attacks. Our study provides insight into the current prevalence of SVG-based clickjacking attempts on the web pages.

2.9 Summary

In this chapter, we have discussed the background details and a range of work related to our thesis. We see that researchers have proposed secure browser architectures for addressing the general security and reliability problems of web browsers, but none of the work address the risk associated with extension that affects operating system resources. A number of projects address the malicious extension problem. However, the threat model does not address malicious behavior incorporated by extensions developed by well-intentioned developers. We conjecture that, the best way to deal with malicious behaviour of the legitimate extension is to analyse an extension before installing into browser. In particular, we propose an static analysis approach solely based on the semantics of accessing browser and system resources through JavaScript. We present such an approach in the next chapter.

In particular, the attacks through colluding extension are not addressed by any of previous work. A number of techniques have been proposed which isolate browser components from each other. However, these solutions require redesign of web browser architecture and its components. The number of work has been proposed to detect clickjacking attacks. However, all the defense mechanisms are focused on clickjacking attacks crafted using frame/iframes. In particular, none of the work addresses the advanced clickjacking attacks caused due to SVG-based filter and effects.

Chapter 3

Malicious Flows in Browser Extensions

JavaScript Extensions (JSEs) enjoy high privileges, sometimes as high as those of the browser itself. However, this may place the browser under risk of information breach, privilege escalation attacks, etc. In this chapter, we discuss different classes of attacks caused due to browser extensions and show how browser insecure policies and JSEs can be abused by an attacker for malicious purposes. Subsequently, we present a semantic analysis model, BEAM (Browser Extension Analysis Model), to analyse information flow in JSEs and detect maliciousness in browser extensions.

Our model analyses information propagation among browser's resources (browser, web page and host-OS components accessed through JSE). A resource is labeled sensitive when it accesses critical information (directly or indirectly). BEAM classifies information as suspicious when a tainted object accesses sensitive information in an unsafe manner (e.g., if a JSE extracts information from web page, and send it over the network or write it to a host file). Furthermore, our model assigns a rating to every information flow to compute risks associated with a JSE under analysis. Using the results of our analysis, we can identify maliciousness in JSEs and provide comprehensive severity reports on their behavior. We evaluate our work on a substantial body of benign and malicious JSEs. The results of our experiments show that it is better to quantify some of the benign JSEs that are prone to risks due to suspicious information flows.

3.1 Problem

The Firefox JSEs have critical functionalities provided by XPCOM interface that may pose security risks in Browsers. For example, a password can be stolen from the web page password input field by accessing browser DOM, or by accessing cookies to steal session, etc. is a critical security threat exploited by the JSEs to leak privacy [76]. In addition, the interfaces also allow JSE to access arbitrary files from the file system and invoke new process on the host system. These threats are also very critical because using them an attacker can launch a *malware* process or can alter user files.

Browser renders JSEs to run with full chrome[1] privileges, including access to browser components such as browser DOM (Document Object Model) [13], cookie manager, password manager and elements or information present in a web page. In addition, JSEs can access OS resources such as file system, network services, and process system. For example, a popular banking trojan attack, called *Man-in-the-Browser* [77], accesses information from browser DOM to achieve malicious goals such as: stealing user's login credentials, modifying current web bank transaction on the fly, and modifying web pages contents on the fly without the knowledge of the user.

Previously published analysis on Firefox JSEs have proposed static and dynamic techniques for detecting maliciousness and vulnerabilities. VEX [56] analyzes Firefox extensions for suspicious flows using context sensitive and flow-sensitive static analysis. However, VEX only checks for three flow patterns that capture flows from injectable source to executable sinks. SABRE [57] presents a dynamic analysis framework to track the flow of JavaScript objects from sensitive sources to sensitive sinks inside the Mozilla Firefox browser. However, it is difficult to detect accurately suspicious flow patterns at runtime since the browser itself has many pre-installed legitimate extensions, and it is very difficult to differentiate the flow of pre-installed and malicious JSE. Another related approach, which is based on Hidden Markov Model (HMM) is presented in [59]. HMM model uses potentially relevant features for constructing the models followed by their probability distribution from a given set of extensions. It characterizes the features based on the presence of JavaScript and XPCOM APIs in benign, vulnerable and malicious extensions. The HMM model may not produce accurate results because it may be possible that a legitimate JSE may use sensitive API. Another dynamic approach

[1]Chrome is an entity making up the user interface of a specific application or extension.

for preventing legacy JavaScript-based Firefox extensions from the malicious activity is presented in [58]. The approach is experimented only on *ten* Firefox extensions using the *four* attack scenarios they proposed in [10].

3.2 Threat Model

In this work, we focus on finding the malicious intent of legitimate browser extensions. We assume that some developers may have malicious intent while developing an extension and writing the code that has malicious information flow ignored by the Firefox review process. In particular, we focus on finding suspicious information flow in the JSEs, which causes an attack opportunity for an attacker. We do not try to identify the JavaScript vulnerabilities, bugs in the browser itself, or bugs in other browser extensibility mechanisms, such as plug-ins. We consider all the JSEs that are free from such vulnerabilities. We believe that our proposal will help Mozilla JSE reviewers in finding suspicious information flows in legitimate extensions. Our goal is to automate this process so that analysis can be done quickly on particular snippets of code that are likely to contain suspicious flow.

We will use two threat models to illustrate the various aspects of our proposed approach: (i) First, we consider attacks that originate from malicious extensions and we assume that the authors of malicious extension try to trick users into installing a fake or altered extension; (ii) In the second threat model, we consider the legitimate extensions possessing suspicious flow. These extensions are well-intentioned but contain suspicious flow that can be exploited by malicious web site operators and active network attackers.

3.2.1 Attack Scope and Target

JSE having elevated privileges and vulnerabilities are exposed to a wide variety of threats. It is, therefore, important to clarify our threat model, specifically on the nature of protections that we offer and the threats that are outside the scope of this work. We use the term attack target, or simply target, to represent which resources an adversary may target to achieve malicious goals. The browser provides built-in classes and APIs through which an extension can access browser resources and may initiate an attack.

Table 3.1: Activities that introduce extension-based attacks in browser

Activity ID	Activity	Target Resource Exploited
A1	Accessing password from web Page	DOM
A2	Launching arbitrary local application	Invoking OS process
A3	Code injection	Browser chrome
A4	Send/Receive on network	network channel
A5	Violation of Same origin policy	network channel
A6	Profile attack	File system (OS)
A7	Accessing confidential data	DOM
A8	Stealing local files	File system (OS)
A9	Accessing history	Browser Component
A10	Accessing stored passwords	Password manager
A11	Accessing events	Keyboard & Mouse events
A12	Session stealing	Cookie manager, Cookies
A13	Accessing bookmarked sites	Bookmark manager
A14	Modifying browser configuration	Browser preferences
A15	Changing browser and Extension preferences	Modifying arbitrary
A16	Accessing web page information (images/text)	DOM

The browser resources that are potential target includes cookies manager, password, DOM, etc. An attack is accomplished by executing some course of actions, such as modification, gaining access, tainting, and leaking information on the target resources.

In-Scope Threats. We define some key resources that an attacker may target, i.e., *likelihood of attack*. In general, we are trying to determine the impact of extension-based browser attacker on the several browser and operating system resources. Table 3.1 illustrates the activities that an JSE can perform. These activities may result in the threat in a browser. We have addressed the threats pose by given activities in our proposed solution. For example, the most common target for an attacker is web page DOM for stealing user password and other confidential information, and network channel for sending sensitive information from the browser to other domain. Our solution will raise an alert if a confidential information is send over a network channel.

Out-of-scope threats. Many security threats posed by browser JSEs have been identified by the security community. There has been intense research in analysing threat caused due to JSEs, which can complement our approach for protection against specific attacks. We are more focused towards attacks that are initiated by legitimate JSEs or sometimes popular JSE. In particular, our work does not address the threats listed below.

1. *Threats with browser vulnerabilities:* We do not address the threats caused due

to browser vulnerabilities such as drive-by-downloads [78] [79], buffer overflow vulnerability in HTML parser [80], buffer overflow in the bookmarks system [81], threats with plug-in vulnerabilities and so on. Our goal is to prove that the vulnerabilities in the browser are not the only attack vector to initiate privacy leakage attacks in the browser. The attacks can also be initiated with a non-vulnerable browser.

2. *Threats with extension vulnerabilities:* Our approach leverages suspicious information flow in JSE, which determines the malicious intention of a JSE. A non-vulnerable extension is thus sometimes causes a serious threat to user information. Thus, our approach detects attacks that are initiated by non-vulnerable and legitimate JSEs.

3. *Attacks due to code obfuscation:* The extension which uses code obfuscation using hexadecimal and base64-encoded background script, `eval` and `unescape` are not handled by our approach.

4. *Colluding extensions attack:* The attacks can be caused due to a collusion of two or more extensions [82]. Our model do not address the flow that originates from two or more colluding extensions. As a result, the suspicious flow originates from two colluding extensions is the out-of-scope threat for our model.

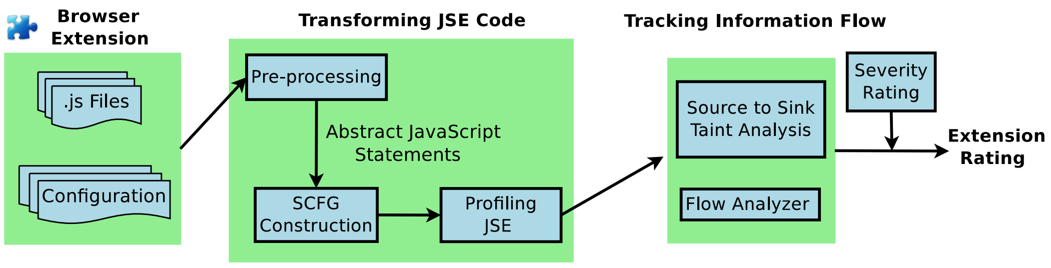


FIGURE 3.1: Overview of the our proposed approach.

3.3 Proposed Approach: BEAM

In this section, we describe the internals of BEAM, our semantic analysis system that identifies malicious behavior in JavaScript-based browser extensions. BEAM takes a source code of a JSE to extract semantic information so as to analyse and deduce the maliciousness in the JSE. In particular, BEAM identifies the interaction of JSE under analysis with the web pages, system, and browser resources. Using a set of heuristics to identify potentially dangerous behavior, it labels extensions as malicious, suspicious, or benign. In the rest of this section, we describe how BEAM works and the challenges that arise in analyzing browser extensions.

Figure 3.1 shows an overview of browser extension analysis, which consists of two phases: a transforming phase and information flow tracking phase. In the first phase, we parse and perform appropriate code transformation on an extension's source code. The transformed source code is checked with the JSE profiles to extract important properties of the JSE. In the second phase, the information collected from the previous phase is analysed for suspicious information flow. Our model, also assigns the severity rating to the suspicious flow found in the JSEs.

Abstraction Model Definitions. We introduce the notion of BEAM model, and, in that context, formally defines an abstraction of JSE code with following definitions.

Definition 1: (Script Block) *JSE contains the set of script blocks,* $\delta = \{\delta_1, \delta_2, ..., \delta_n\}$*, where* δ_k *represents* k^{th} *script block containing set of JavaScript statements.*

Definition 2: (Integrated Script Block) *An Integrated Script Block (ISB) can be defined as,* $\iota = \bigcup_{k=1}^{n} \delta_k$*, where* $\delta_i \in \delta$ *and* δ_{k+1} *is called script block of callee* δ_k*.*

Definition 3: (Sequence Control Flow Graph) *Each integrated script block,* ι_i *consists of Sequence Control Flow Graph (SCFG) G, where,* $G = (N, E)$ *defined as a finite set N of nodes and a finite set* $E = \{e, e_{out}\}$ *of edges with* $N \cap E = \phi$*. e is an edge connecting node* n_i *to its immediate successor node* n_{i+1} *and an edge* e_{out} *is added from* $(\hat{n_a}, n_a)$ *iff the node* $\hat{n_a}$ *is accessing the information of* n_a*.*

Definition 4: (Node) *A node* $n = \{R, C\}$ *is a tuple of two attributes, where R is a resource to be accessed by a statement, and C is a class associated with every resource, such that* $C \in \{source, sink\}$*, which represents mapping of resource, r to its corresponding class.*

The resource itself is a tuple of three attributes $Resource(R) = (O, A, P)$. A resource $r \in R$ is referred to operating system, browser, and web resource that is accessed by JavaScript Statement (JS). An Object $O = \{XpObject, JsObject\}$, where *XpObject*, and *JsObject* are the object instances created by invoking XPCOM interface, and JavaScript Method respectively. An action $a \in A$ is viewed as the method invoked by an object O for accessing resource r, and P is the parameter that is passed when a method is invoked.

3.3.1 Transforming JSE Code

We transform the source code of the JSE to facilitate construction of a SCFG for the JSE. The transformation occurs in three stages. In the first stage, Pre-processing processes all statements of the JSE code (.js files) to make it compatible with static analysis. The next stage constructs the SCFG from the pre-processed code. The nodes in SCFG generated appropriate facts corresponding to statements in the JSE code. We now describe the various stages of the analysis in detail:

Pre-processing. Pre-processing stage processes the entire JSE code statements and transforms to make it compatible to static analysis. The Pre-processing is done in two phases: (i) script block, and (ii) integrated script block.

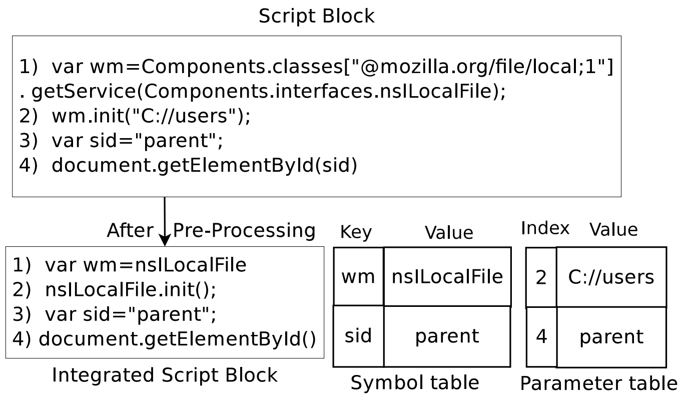
Phase-I (Script Block): In script block the reference of called functions are inserted into callee function. It may be more than one script block for a JSE, but if a JSE code does not have any function call, only one script is created. The reason behind script block creation is to reduce function call overhead.

Phase-II (Integrated Script Block:) In this phase the function reference from script block is replaced with the corresponding function statements. Each script block is transformed into blocks called integrated script blocks (ISB) which contained parser compatible statements for next stage. As illustrated in Figure 3.2 the script block contains the JSE code and is transformed into ISB, and wherever the objects and variables created by API/XPCOM Interfaces are referenced or assigned they are replaced with corresponding API names. For example, a `nsILocalFile` object *wm* is replaced with `nsILocalFile` wherever the object *wm* is invoked (Figure 3.2, Line 2 of ISB). This example code is executed on Windows platform. Pre-Processing stage maintains two data structures for each script block: (i) a symbol table stores the key-value pair, where keys and its values

Table 3.2: Partial list of Sensitive Sinks

Entity	Method of Access
Files/Processes	`nsIOutputStream`, `nsIFileOutputStream`, `nsIFile`, `nsIProcess`, `nsIDownload`
Network	`nsIXMLHttpRequest`, `nsIHttpChannel`, `nsITransport`, `worker.port.emit/on`

are represented by variable name and string value respectively; and (ii) a parameter table stores the parameters invoked by a function call.



```
1) var wm=Components.classes["@mozilla.org/file/local;1"]
. getService(Components.interfaces.nsILocalFile);
2) wm.init("C://users");
3) var sid="parent";
4) document.getElementById(sid)
```

```
1) var wm=nsILocalFile
2) nsILocalFile.init();
3) var sid="parent";
4) document.getElementById()
```

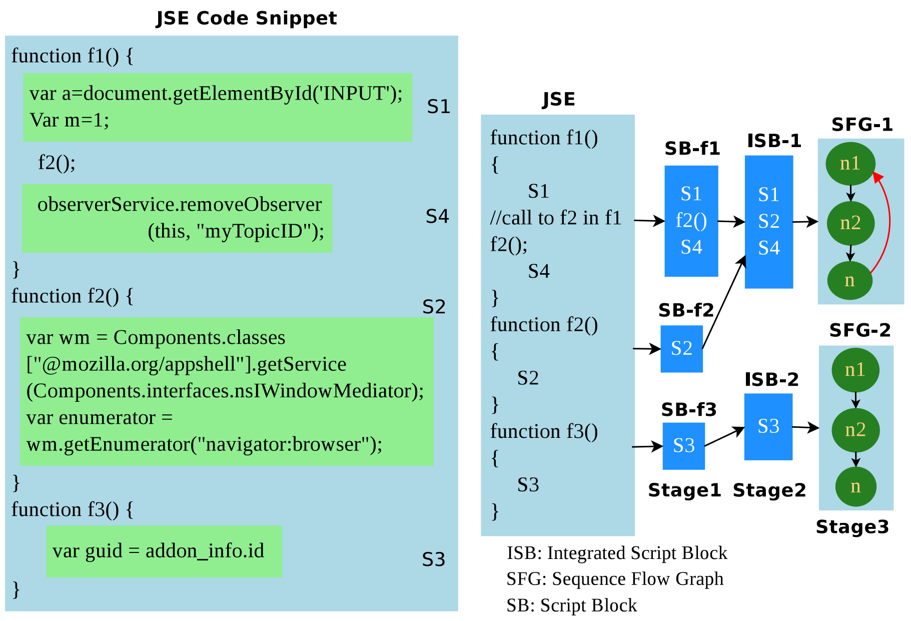
Key	Value
wm	nsILocalFile
sid	parent

Index	Value
2	C://users
4	parent

Figure 3.2: pre-Processing of Script Block

SCFG Construction. For the purpose of statically analyzing the pre-processed JavaScript code, we use an off-the-shelf tool to generate a sequence control flow graph. Figure 3.3 shows the various stages of our SCFG construction of a JSE under analysis. Once the ISB is created for a JSE, every JavaScript statement is represented with a notion of graph node. Each graph node, representing access to a resource, is then analysed for identifying flows, i.e., what information is flowing from one node to another, and in what order. We associate a notion of source and sink with every resource. A *source* is referred as a sensitive resource from which the information originates, and sensitive *sink* is referred to the resource that expose internal application data to the outside world.

Profiling Extensions. After SCFG construction, we generate appropriate facts corresponding to statements in the JavaScript code. In particular, we extract following the set of information from the JSE source code. With this information, BEAM closely approximate JSE execution behavior in the browser.



```
function f1() {
    var a=document.getElementById('INPUT');
    Var m=1;
    f2();
    observerService.removeObserver
                (this, "myTopicID");
}
function f2() {
    var wm = Components.classes
    ["@mozilla.org/appshell"].getService
    (Components.interfaces.nsIWindowMediator);
    var enumerator =
    wm.getEnumerator("navigator:browser");
}
function f3() {
    var guid = addon_info.id
}
```

```
function f1()
{
    S1
//call to f2 in f1
f2();
    S4
}
function f2()
{
    S2
}
function f3()
{
    S3
}
```

FIGURE 3.3: Execution profiling of our proposed method

1. **Cross-domain Interaction.** Most extensions interact with the content of web pages. We search the source code of JSE to extract the list of URLs that it interacts. The list of URLs is useful in determining the cross-domain access performed by a JSE. The malicious JSE may act as a middle layer to initiate communication between the cross-domain websites. The malicious JSE, then, can execute cross-domain scripts and access and share cross-domain information. For example, an extension is capable of violating same origin policy, which may lead to the interaction of cross-domain websites.

2. **Event-based Actions.** The Chrome and Firefox browsers offer the set of APIs to extensions for sending notifications that respond to certain browser-level events. Firefox browser provides certain in-built notification topics, which an extension can listen and take appropriate action. For example, a browser opens a Facebook page to read information from Facebook server. When, the browser interacting with Facebook server is completed, it generates a notification topic to the browser in response to completion of the access (read) event. This notification can be intercepted by an extension that requires read event to complete. Our analysis records all the notifications topics be generated and intercepted by an extension.

3. **Interaction with Network.** Extensions often use network channel to send and receive information to remote server. Browser provides several APIs such

Table 3.3: Partial list of sensitive sources

Entity	**Sensitive Attributes/Method of Access**
Document	`cookie`, `domain`, `forms`, `lastModified`, `links`, `referrer`, `title`, `URL`
Form	action.
Form input	`checked`, `defaultChecked`, `defaultValue`, `name`, `selectedIndex`, `toString`, `value`
Select option	`defaultSelected`, `selected`, `text`, `value`.
Location/Link	`hash`, `host`, `hostname`, `href`, `pathname`, `port`, `protocol`, `search`, `toString`
Windows	`defaultStatus`, `status`
DOM	`getElementbyId`, `getElementbyTagName`
Files/Streams	`nsIInputStream`, `nsIFileInputStream`, `nsILocalFile`, `nsIFile`
Password	`nsIPasswordManager`, `nsIPasswordManagerInternal`
Cookies	`nsICookieService`, `nsICookieManager`
Preferences	`nsIPrefService`, `nsIPrefBranch`
Bookmarks	`nsIRDFDataSource`
Extension's objects	`nsIObservable`
History	`nsISHistory`, `nsIBrowserHistory`

as `XMLHttpRequest`, `worker.port.emit/onto`, `HTTPChannel` to establish communication with remote server. We identify the flow that sends sensitive information over a remote location. In addition, network information also used to track the information that is fetched from the remote server.

4. **Interaction with Host-OS.** We identify the APIs that will allow an extension to access host OS resources such as file system, and processes. Using this information, we can monitor two major activities; (i) the source of information that is written on file system, and (ii) the files from host file system are accessed by the JSE.

5. **Interaction with Browser Components.** Extensions often use browser components such as DOM, Password Manager to access information stored on the browser. We monitor the information that a user input in a web form and stored passwords in a browser. We track this information to identify the flow that looks to be sensitive. For instance, a stored password send over network channel, or input information send to a cross-domain web page, etc.

6. **Interaction with persistent Data.** We track persistent data, such as cookies, history, bookmarks that a JSE can access in the browser. In particular, we monitor if a persistent information stored in the browser is leaked to a remote location. For instance, if the cookies are leaked, then an attacker can hijack a web session [68].

3.3.2 Tracking Information Flow

Our static analysis closely approximates JSE execution behavior in the browser. In particular, first we find how information propagates from one resource to another. The XPCOM interfaces of extension system allow developer (user or attacker) to establish an inter-communication channel between components of operating system, browser components, and web applications. Hence, the information can flow among various components of the browser as well as the operating system in legitimate or illegitimate manner. The Algorithm 1 illustrates the proposed algorithm for analyzing the information flow in a JSE.

Algorithm 1 Proposed Approach Algorithm

```
$\delta = \{\delta_1, \delta_2 \cdots \delta_n\}$   : Set of $n$ Script Blocks (SB)
$\iota$: $\bigcup_{k=1}^{n} \delta_k$ : Integrated Script Block (ISB), where $\delta_i \in \delta$ and $\delta_{k+1}$ is called SB of callee $\delta_k$.
**preProcessing($\delta_i$)**   : Extracts .JS and configuration files of Extension
**constructSCFG(preProcessing($\delta_i$))**: Construct control flow graph
**profileJSE($\delta_i$)**     : Extracting relevant information from the Extension
---
**preProcessing($\delta_i$)**
extract(.js files)
extract(conf files)
**return** $\iota$
**for** $p = 1$ to $\iota$ ***do***
   // scanning all statements in ISB ($\iota$)
   $N$ = ***profileJSE($\delta_i$)***
   $GRAPH$ = ***constructgraph($N$)***
   $flag$ = ***findSourcetoSink($GRAPH$)***
   ***if*** $flag == false$ ***then***
      No Suspicious Flow
   ***else***
      rating = suspiciousFlow()
   ***end if***
***end for***
```

Our goal in this work is to identify the information flows, which may be legitimate or suspicious depending upon the functionality of a JSE. We define suspicious flow with respect to the browser environment in which a JSE executes, which reflects how a JSE can risk, and expose various resources to the attacker. For example, an extension might use JavaScript method `getElementById(id)` to capture information from sensitive source

(DOM), and then use XPCOM interface `nsIXMLHttpRequest` to send this information to untrusted domain through sensitive sink (Network Channel).

This section overviews the core operations of BEAM. Here we discuss (a) source to sink taint analysis, (b) explicit and implicit flows, (c) suspicious flow analysis, native code taint propagation, and (d) whitelisting of flows.

Source to Sink taint analysis. BEAM associates action label with SCFG nodes. Each action label stores two pieces of information: (1) a resource type, which determines resource accessed by a node; and (2) a class, which determines the origin/destination of the information. Hence, each action label is used in determining the sensitivity of information used by the resources. For example, an action label associated with node n_3 in Figure 3.4 defines *network* as resource type and *sink* as its class. In our static analysis, the taint information is modeled as the capability of accessing sensitive sources. In particular, an object carrying information is tainted if (a) it directly accesses any of the sensitive sources or (b) explicitly accessing the reference of an object that accesses sensitive source.

In addition, we incorporated the tainting capability to our static analysis to isolate potentially untrusted values from trusted ones. BEAM places an action label in each of the nodes, based on that label BEAM marks whether the node is tainted or not. Whenever a reference is assigned a value from potentially untrusted sources (e.g., the DOM object window.content.document), it taints the node. The taint propagated as a node is assigned or passed as a function parameter to other references. At the end of the analysis, if we find that the tainted node reference points to a sink resource, we can conclude that the use of such reference might potentially be unsafe.

We have identified several sensitive sources along with their method of access. The partial list of potential sensitive sources and sinks are listed in Table 3.3 and Table 3.2. Rows 1, 2, 3, and 4 of Table 3.3 are derived from Netscape Navigator 3.0 [83], which primarily includes the data elements to which JavaScript engine provides data tainting features. The origin of these data elements is considered as sensitive source, these elements primarily deal with access to DOM elements. The remaining rows of Table 3.3 denotes sensitive sources originate from the browser and operating system resources, such as cookies, passwords, bookmarks, files and preferences. Table 3.2 shows the sinks where the information leaves or exposed to the outside world. One of the critical sink that we encountered during our analysis is network channel. For example, the information flows from internal browser source such as *DOM* to the outside world such as *network*.

In some extensions that have *spyware* or *bot* [84–87] features, can use sensitive sink such as network as a source. For example, a spyware can start a malicious process using `nsIProcess` XPCOM interface on victim's browser. An attacker machine can send a signal through the network (sensitive source) to initiate a process on victim's machine. This example shows the network channel can also be used as sensitive source to execute malicious activity on the browsers. In addition to monitoring sources and sinks, our model also tracks information flow originated from JavaScript APIs such as `eval()`, `innerHTML()`, `wrappedJSObject()`. For instance, some JSEs modifies web page information at runtime. Our approach also monitors DOM information, which can be modified using `innnerHtml`. It raises suspicious flow alert when the information accessed from DOM is modified through `innerHtml` function.

Explicit and Implicit Flows. Our system handles explicit flows by updating left hand side (LHS) labels of an assignment with the labels of values assigned to right hand side (RHS) label. For example, the assignment `var data = frame.document.getElements ByTagName('input')` shown in Figure 3.4 that is used to take out information input by user from web page `INPUT` field. BEAM handles this by updating `data` variables on the LHS with label `getElementsByTagName` on RHS. Similarly, a `info` variable in assignment `info = data[i].value` is replaced with `getElementsByTagName('input')`. The purpose of doing this is to recognise the source of information at each JavaScript statement. In this example, the information to be entered in web page `INPUT` field propagates from `data` to info variables. Thus, BEAM associates same label to `data` and `info` variables. This way the labels are propagated in explicit flows. So if data variable is labeled sensitive so as `info` variable. Implicit flows arise through the control flow in the program. Our approach will handle limited implicit flow on an assumption that if a information checked in the conditional expression is labeled sensitive then both *true* and *false* statements are considered as sensitive. For example, if `(document.cookie.length > 0) then {T} else {F}`, in this case `document.cookie.length` is sensitive and hence both `T` and `F` blocks are considered sensitive.

Suspicious Flow Analysis. Our model will map all the nodes produced from ISB into SCFG. Each node in ISB is referred using a relevant JSE statement. The connectivity between nodes is referred to as the control flow between statements. In this model, we consider two types of edges: (i) a forward edge; and (ii) an outer edge. The consecutive connectivity between nodes in SCFG represents a forward edge, whereas an outer edge defines an edge between two or more nodes. When the information flows from sensitive source to sink resource (defined in Table 3.3 and Table 3.2). For example, a JSE has a functionality to access password field from web forms and send them to third-party domains. BEAM found this flow as suspicious as the critical information flow from the sensitive source (web forms) to sink (network). In addition, we also identified the severity of suspicious flow in terms of critical, high, moderate, and low rating. Mozilla and Chromium define security severity rating on the basis of information accessed by various resources and discussed in [28] [29]. In more generalized way, we define a suspicious flow as follows.

Definition 5: (Suspicious Flow) *A suspicious flow S_f is defined as flow between any two nodes, n_1, n_2, such that,*

$$S_f = e_{out}(n_j, n_i) : n_i \in \{sensitive\ source\} \quad \wedge n_j \in \{sensitive\ sink\}$$

The above logic represents the suspicious flow that occurred between two nodes. The flow between nodes n_1 to n_j is said to be suspicious only when there is an outgoing edge from n_j to n_i and the node n_i to n_j belongs to a set of sensitive sources and sinks respectively. An outer edge indicates that the information flows from node n_i to n_j.

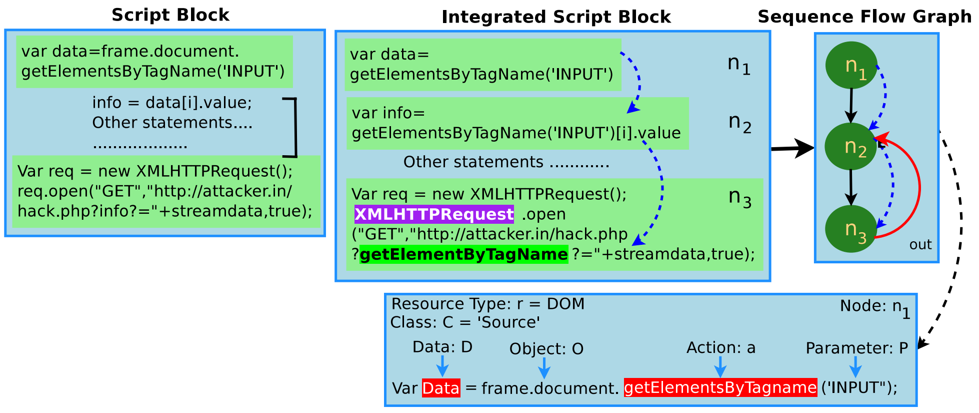
Whitelisting of Flows. Sometimes a benign JSE can contain information flow that may potentially be classified as suspicious by our method. For example, consider *LastPass*, a password manager JSE, which extracts all passwords stored in the browser or input by a user on website login page. It captures these passwords and sends them to a third party site and provides password management functionality. The JSE reads and modifies a sensitive resource (i.e., a password) from browser and web form, which is then transmitted over the network. It raises a critical alert because an untrusted JSE can use a similar technique to transmit passwords to a remote attacker. To handle such flows for popular legacy JSE, BEAM uses whitelisting policies to declassify these flows. BEAM defines trust-specific whitelisting policy, which permits declassification of all flow from a trusted JSE. We trusted the JSE based on its popularity, and number of times a JSE downloaded on Mozilla add-on store. If a newly introduced JSE contain such a flow, BEAM does not whitelist the flow.

3.3.3 An Example

We now demonstrate how the analysis detects flows in JSE code. This section explains an example scenario for the extension-based attack on the browser. An example attack scenario is described as follows: (i) an extension first creates an object to access web page information from $INPUT$ tag using `getElementByTagName('INPUT')`; (ii) an XPCOM object is created to access network channel, and the information is send to the attacker server by calling `XMLHttpRequest.send()` function. To understand the generic view of the above attack scenario, we map it to an abstraction model. Figure 3.4 illustrates a running example of suspicious flow with detailed node view. An overview of an example scenario is as follows:

1. The script block/s are created from the given JSE code.
2. Our model then transforms script blocks into ISB that contains statements that are compatible to B-parser. Our model will replace objects/variables with their corresponding actual API name. As seen in example a statement `info = data[i].value` is replaced `getElementByTagName('input')` and `req` is replaced with `XMLHTTPRequest`. The reason for doing this operation is to track the propagation of information flows from one resource to another, and to reduce the number of backward calls which improves the parser performance.
3. The ISBs are then mapped into SCFG after filtering abstract statements from the ISB. The SCFG contains only those ISB statements that are associated with source and sink resources. In this example, the statement that is forming nodes (n_1, n_2, and n_3) are relevant, and the rest are just ignored.
4. The statements associated with node n_3 in ISB indicates a suspicious flow. It invokes `XMLHTTPRequest.op en()` method and passes `getElementByTagName` as argument, which indicates that the method is sending information extracted from INPUT field of a web page. Our model denotes this with an E_{out} edge from n_3 to n_2.

These steps clearly show an attack scenario, which allows an attacker to illegally acquire critical information, such as, *authentication credentials* from any web page and send it to the attacker over network channel.



```
var data=frame.document.
getElementsByTagName('INPUT')
          info = data[i].value;
          Other statements....
          ..................
Var req = new XMLHTTPRequest();
req.open("GET","http://attacker.in/
hack.php?info?="+streamdata,true);
```

```
var data=
getElementsByTagName('INPUT')                          n1

var info=
getElementsByTagName('INPUT')[i].value                 n2
          Other statements ...........

Var req = new XMLHTTPRequest();                        n3
XMLHTTPRequest .open
("GET","http://attacker.in/hack.php
?getElementByTagName ?="+streamdata,true);
```

```
Var Data = frame.document. getElementsByTagname ('INPUT");
```

FIGURE 3.4: Mapping of Integrated Script Block to Sequence Flow Graph

3.4 Implementation and Evaluation

Our analysis model BEAM uses ANTLR tool [88], which can be used for creating script blocks and grammar rules. We use ECMA Script v.5.0 grammar [89] for creating script blocks. We have implemented B-parser (BEAM parser) component to capture extension's control flow. BEAM is implemented in Java. We have modified ECMA script grammar for parsing JSE code. The existing parser is not fully compatible with all JSE statements because some JSEs contain statements other than core JavaScript syntax, like XPCOM interface calls, *regex* pattern, etc. These statements cannot be parsed with existing parser. We have implemented new grammar rules in ECMA-script for parsing these statements. Our modified ECMA-script is able to parse all XPCOM interface calls invoked in JSE, it also parses `let` statement and *regex* patterns.

To test the effectiveness of our security model, we have evaluated our model with malicious and benign JSEs. To understand the suspicious flows in JSE, we downloaded malicious JSEs reported in the literature and also developed new JSEs. We have evaluated a total of 50 malicious JSEs. To test legitimate JSEs for suspicious flow, we have downloaded 258 JSEs (50 JSEs are popular) from Mozilla Firefox AMO [18], which are claimed to be benign. We performed a preliminary evaluation with some popular legitimate JSEs to explore their suspicious behavior. We have used BEAM to understand the flows and determine whether they are potentially malicious. We have also evaluated each flow, and assign one of the four ratings to each flow based on attack rules defined in Table 3.4.

3.4.1 Evaluation Methodology

We have evaluation our approach using three experiments; (i) evaluation of malicious extensions, (ii) evaluation of popular legitimate extensions, and (iii) evaluation of benign extensions.

Evaluating with Malicious Extensions. We have tested 50 malicious JSEs using BEAM. Some of the malicious JSEs are developed by embedding malicious flow in small benign JSEs. In addition, we also use publicly available extensions that have known instances of malicious flows (e.g., FFSniff). The malicious extensions are implemented such that they exhibit all potential behaviors of attack shown in Table 6.6. We adapted the proof-of-concept given for malicious extensions [10] to implement these malicious JSEs. The goal of this experiment was to understand the nature of information flows in Malicious JSEs and analyse the sensitive source and sink resources they have used. We have also assigned the severity ratings to each suspicious flow extracted from the JSE.

Evaluating with Popular Legitimate JSEs. We have tested 50 popular benign JSE from different categories (mostly from security and privacy category). To determine the JSE's advertised functionality, we ran each JSE and manually exercised its user interface and read the documentation available with it. The goal of this experiment was to understand the nature of information flow between sensitive source to sink in these JSEs and identify flow that are suspicious but not necessarily malicious. BEAM observed all critical and high rated flows in some of these JSEs, since the JSEs are popular, and we assume them trusted, we applied the trust-specific policy to whitelist flows. With manual inspection, it is found that these flows are mandatory needed to accomplish the desired functionality of an extension. However, we found that the information flows in some of these benign JSEs closely resemble to those exhibited by malicious JSEs and hence the question arises why these JSEs are trusted by Mozilla add-on review process.
Results. Our experiments on malicious as well as legitimate extension show suspicious

flows. Table 3.4 shows some of the flows along with the severity that are found in the JSEs.

TABLE 3.4: Sensitive information flows found in evaluated JSEs

Activities Performed	**Description**	**Severity Rating**
A1+A4 A12+A4 A10+A4	Send password from web page, cookies, stored password over network to untrusted domain.	Critical
A2+A3	Injecting and launching malicious script with browser chrome privileges	Critical
A6	Writing malicious file to browser profile folder.	Critical
A7+A4 A8+A4	Send user input fields except password, reading file from host file system and send over network to untrusted domain	High
A5	Read information from one and write it to other domain	High
A9+A4 A11+A4	Record keyboard/mouse events, extracts browser history and over network to untrusted domain	Moderate
A13+A4 A16+A4	Access bookmark sites, page information and end it over network tountrusted domain	Low

To show threat posed by JSEs in the Firefox browser, we identified information flow in the JSEs. Figure 3.5 illustrated the aggregated percentage flow in malicious and legitimate JSEs. As we can see from Figure 3.5, both malicious, as well as legitimate JSEs, have flows, which BEAM has ranked in terms of severity ratings. We observed that some legitimate JSEs had suspicious flows. In addition to that, we also observed that some popular legitimate JSEs have critical advertised flow, which closely resembles the malicious JSEs flow. Most of the legitimate JSEs contains the benign flow, which is labeled as *none*.

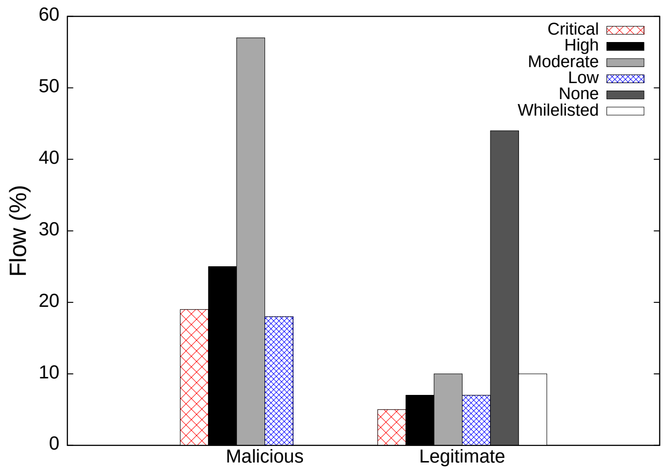


FIGURE 3.5: Information Flow % in Legitimate and Malicious JSEs

Our experiments indicate that several legitimate JSEs exhibit information flows that can be suspicious and misused by an attacker. Table 3.5 has listed the critical flow found in

4 popular extensions. BEAM marks these JSEs as critical and suspicious flow found in them. We further analysed these extensions manually and found out that the suspicious flow is because of JSE's functionality. In such case, BEAM whitelists the flow. For instance, few observations that we found with popular JSEs are listed as follow.

1. It is interesting to notice that *Lastpass* stores all your critical passwords (including email, credit card, banking, etc.) and sends them to third party domain without prompting any warning to the user. We have whitelisted this flow after manually analysing the *Lastpass* JSE. The third party location at which the information is being sent is found trusted by the security community.

2. The basic functionality of Flashgot is to download links, audio and video files from the web page. To achieve this functionality it creates a file in binary format using *nsIFileOutputStream*, and writing to same file using *nsIBinaryOutputStream*. Once the file is created it invokes *nsIprocess.init()* method with parameters to invoke a process. BEAM consider this as critical, suspicious flow as the control flows from File System (sensitive source) to process (sensitive sink). This behavior can be used by an attacker to launch malware script as OS process.

3. The WOT JSE extracts cookies information using *nsIcookieservice.getcookiestring()* from document URI (sensitive source), and it call *XMLHttpRequest* methods to access HTTP channel (sensitive sink). This behavior can potentially be misused to steal cookie information of any URI.

Table 3.5: Critical flows in some popular JSEs

JSE	**Flow Rating**	**Sentitive Source**	**Sentitive Sink**
LastPass V 3.1.1	Critical	`nsIPasswordManager.adduser()`, `nsILogin-Manager.findlogins()`	`XMLHttpRequest.send()`
WOT V 20131118	Critical	`nsICoolieService.getcookiesstring()`	`XMLHttpRequest.send()`
Flashgot V 1.53	Critical	`nsIFile`, `nsILocalFile`	`nsIProcess.init()`
GreaseMonkey V 2.3	Critical	`nsIFile`, `nsILocalFile`	`nsIProcess.init()`

Evaluating with Benign JSEs. We have tested BEAM against 208 randomly selected benign JSEs from different categories. This time the JSEs are less popular but downloaded more than 500 times. Again we determine the JSE's advertised functionality by manually running them. Our analysis model found all kinds of flows in selected JSEs. We have critically rated flow in 3 JSEs, these JSEs mainly has flows like sending credentials, and cookies from web page over the network to a untrusted domain. It has been found that 19 JSEs exhibits highly rated flows as these JSEs takes information from web page other than a password and send them over network channel. We have found 13 JSEs with moderate and 20 JSEs with low rated flows. Furthermore, in some cases we have found that the suspicious flow is a part of JSE's advertised functionality, which is whitelisted by BEAM. We found in total 153 JSEs with such flows.

Table 3.6 summarizes the critical suspicious flows found in two extensions. $FacebookDislike$ JSE maliciously stores web page information (userid and cookies) in JSON objects (stored in local disk). The extension performs queries to Facebook's website using `worker.port.emit/on` APIs to retrieve userid and preference information from *graph. facebook.com* and leaks these information to a remote website *dislikenew.doweb.fr/get2.php* that is not related to Facebook. *CoolPreview* JSE finds stored password and login details from browser and use `XMLHttpRequest.send()` to send them over unknown remote location. This functionality is not given in its documentation and hence cannot be whitelisted.

TABLE 3.6: Suspicious flow in benign JSEs

JSE	**Flow Rating**	**Sentitive Source**	**Sentitive Sink**
Facebook Dislike V 3.3	Critical	`getElementById`, `document.cookies`	`worker.port.emit` `worker.port.on`
CoolPreview V 4	Critical	`nsIPasswordManagerInternal.` `findpasswordentry()`, `nsILoginManager.findlogins()`	`XMLHttpRequest.send()`

3.4.2 Limitations

Our work has three limitations: (i) Currently, our approach analyses JavaScript code based on common DOM and XPCOM APIs. We do not check `XPCNativeWrapper` based protection that limits access to the properties and methods of the object it wraps; (ii) Our approach was only limited to malicious flows in JSE and has not analysed the vulnerabilities in JSE instead we are assuming that the JSEs does not have any vulnerability; (iii) We have tested JSEs against the potential attack rules. The list of rules that we have used for analysis is not an exhaustive list.

3.5 Summary

This chapter introduces Browser Extension Analysis Model (BEAM), a static analysis model for analysing suspicious flows in browser JavaScript Extension (JSE). Our model detects potentially insecure information flows within the sensitive source to sensitive sink resources. In particular, it helps to investigate the suspicious flow in JSEs before installing them onto the browser. We implemented our approach in a prototype tool for Mozilla Firefox extensions, and detected suspicious flows that effect confidentiality and integrity in a browser. Our observations found that the suspicious flows in JSEs can be due to the unprotected and privileged access to critical resources. We also associated severity rating with each flow, which rates the security risks present in a JSE. The experiments demonstrate that BEAM can detect critical flows even in some benign extensions, which closely resemble malicious flow and can be critical to the browser security. In the next chapter, we discuss novel attack techniques using colluding browser extensions.

Chapter 4

Colluding Browser Extensions

In this chapter, we first introduce the concept of colluding extension, and then we demonstrate new attacks are leveraging this concept and causing privacy leakage in a web browser. In particular, we illustrate a significant weakness in Firefox browser architecture and its XPCOM interfaces. This gap permits two extensions to collude with each other and share objects that are allocated in a same address space. Also, we show the way in which colluding extensions can communicate over overt and covert communication channels for executing colluding attacks. In addition, we have tested the effectiveness of newly identified attacks against representative state-of-art techniques for extensions.

We identify the vulnerable points in the extension development framework as (a) object reference sharing; (b) event notification; and (c) preference overriding. We illustrate the effectiveness of the proposed attack using various attack scenarios, and we provide a proof-of-concept implementation for web domains including banking and shopping. We believe that the use-case scenarios we consider in our demonstration underlines the severity of the presented attack. Finally, we discuss possible mitigation techniques to address the given colluding attack.

Contributions This chapter makes the following contributions.

1. We demonstrate that the attacks using colluding extensions can be easily implemented in modern browsers. We test and implement our proposed attack on the

Firefox browser. In particular, we explore several colluding extensions that communicate over overt and covert communication channels. The weakness of Firefox extension system in handling JavaScript object allows the extensions to share objects (carrying sensitive information) between other extensions. The attackers can exploit this weakness to execute privacy leakage attacks on the browser. Firefox browser has been used as an example as it is an open source browser with API support for easy development of extensions. In this work, we have employed Firefox browser versions (3, 9, 12, 25) for implementing extensions towards proof-of-concept.

2. We present a proof-of-concept of colluding attacks showing how a reference of the JavaScript object of one extension can be invoked by another extension for *accessing* and *sharing* sensitive information. This suspicious nature is difficult to capture with known detection approaches. We have addressed the exploitable coding features in two important Cross Platform Component Object Model (XPCOM) interfaces [12], and JavaScript Wrapper method offered by Firefox.

3. We have evaluated our finding by applying colluding attack scenarios on different web domain applications that demonstrates how two legitimate extensions can collude with each other to achieve malicious goals in a browser.

4.1 Problem

Sometimes, extension code needs to send a message to itself or another extension for notifying that a task is completed, and subsequent action(s) need be performed. In order to achieve this functionality, the browser extension system provides message passing techniques using interfaces: i.e., APIs that can send information to the local and global environment. In this chapter, we investigate the Inter-Extension Collusion (IEC) to infer the object sharing and explore communication channels in the browser.

We discuss how various components interact through browser extensions and also determines whether IEC in the browser can be exploited to produce colluding browser attacks. For example, let us consider two legitimate extensions with the following functionalities: the first extension (X) has the functionality to capture information from any web page.

The second extension (Y) can communicate with network channel. Individually, these extensions behave benign, and their information flow, when analyzed individually, cannot be considered as malicious. However, if X can communicate sensitive information through Y, this flow could be malicious. If X and Y collude, together, they can send confidential information captured from a web page to the attacker through the network channel. Since the attack comes from the combined activities of two extensions, it can not be detected by a method that analyzes individual extension statically or dynamically. Our study is a necessary step for a comprehensive security analysis on IEC in the browser.

A number of recent techniques for securing web-based applications and browser plug-in policy are presented in [45], [90]. The authors have designed and implemented secure web browser policies, which provides OS-level mechanisms for isolation. In [45], the authors developed flexible security policies that allow to include browser plug-ins within the security framework defined by them. However, this secure browser architecture does not provide any isolation for the JavaScript code in extensions. The policies that are implemented for a plug-in are restricted to web applications only, and hence an attacker can successfully launch colluding attacks that we are discussing in this chapter. Furthermore, the researchers have also proposed a secure browsing environment for social networking sites such as Facebook. For example, in [91], the authors have proposed a complete architecture and implementation of Virtual Private Social Network (VPSN) for Facebook website. Again, this analysis is restricted to web applications only. The solution does not discuss the policies for browser extension JavaScript code.

The current research for detecting malicious flow in browser extensions is primarily focused on an assumption that a single extension can only be used as a source of the attack. This motivates us to explore the attacks that are originated from two colluding extensions. This chapter demonstrates the weakness of this assumption and shows how two legitimate extensions with benign functionality can initiate privacy leakage attack in the browser. Several static [56], [92] and dynamic [57], [58], [93], [94] analysis methods have been proposed to detect malicious or vulnerable extensions in isolation.

In this work, we analyzed four of the most popular techniques: VEX [56], SABRE [57], H. Shahriar approach [94], and JSFLOW [92]. These methods can detect the vulnerable points or tainted JavaScript objects in a browser extension. These techniques allow browser users to limit the impact that malicious extensions, if installed, have on their system, browser, and their privacy. Unfortunately, these methods check taint object and flow originating within single extension only. In particular, these methods do not examine whether an attack source has originated from some other extension. The primary reason these methods are not effective against the attacks resulting from colluding extensions. We show that colluding extensions can be constructed and can use covert[1] as well as overt channels to execute attacks on a browser. Colluding extensions can therefore indirectly execute operations that these extensions can not perform individually.

4.1.1 Colluding extensions in browser

In this section, we investigate the extensions collusion to infer the locations and substance of all inter and intra-extension collusion in browser. This approach provides high-precision means to study how various components interact through browser extensions, and also determines whether this collusion in browser can be exploited to produce colluding browser attacks.

Through analysis techniques developed for testing if a single extension is vulnerable and/or malicious are efficient in identifying privacy leakage and privilege escalation attacks, these remain fragile against multiple extensions collaborating for such attacks. We show this by introducing the concept of colluding extensions in the browser. Colluding extensions are the extensions that cooperate in a violation of some security property or exploitation of some vulnerability of the browser. The attack by colluding extensions is possible because current security mechanisms are not focusing on the objects that are shared between two extensions. Instead, most efforts have been made in analyzing information flow incorporated by a single extension [56], [57], [92], [94].

Apart from communication between extensions and web pages, the browser extensions often need some mechanism for communicating with the other extensions. For example, RSS reader extension might use content scripts on a web page to detect the presence of an RSS feed on a page. Other extensions can use the RSS feed information in order to send information to the remote server.

[1]we used "covert channel" term in a slightly different sense (also commonly used in the literature), i.e., we used it for the secret communication where hidden exchange of information takes place among extensions [95], [96].

Google Chrome uses message passing for establishing communication between extensions and their components, extensions, and their content scripts. In addition to sending messages between different components within an extension, Google Chrome uses the messaging API to communicate with other extensions [97]. For example, `chrome.runtime.sendMessage` API sends a single message to event listeners within your extension or a different extension [98]. Internet Explorer (IE) on the other hand provides `postMessage()` [99] method to send messages between BHOs (Browser Helper Objects) [55] and web applications. Using `postMessage()` a BHO can establish communication channel with other BHO, and they can collude with each other. Firefox provides XPCOM interfaces in order to establish inter-extension communication. In this chapter, we discuss inter-extension communication in details with reference to Firefox extensions. However, we underline that the collusion is possible in other browsers as well, such as: Google Chrome and IE. In this work, we decided to focus on open source Firefox browser because of its popularity, which allows easily accessible APIs for implementation, and ease in extension development.

Figure 4.1 illustrates the layered architecture of the Firefox extension system. The first layer is browser application layer, which includes pre-installed and third party extensions. The middleware layer includes script layer and XPCOM framework, which allow extensions to access core kernel components. The kernel layer contains the key components of the system that provides essential system services to upper layers, such as process, file system, and networking support. Our primary focus in this architecture is identify the possible channels for establishing communication between browser extensions.

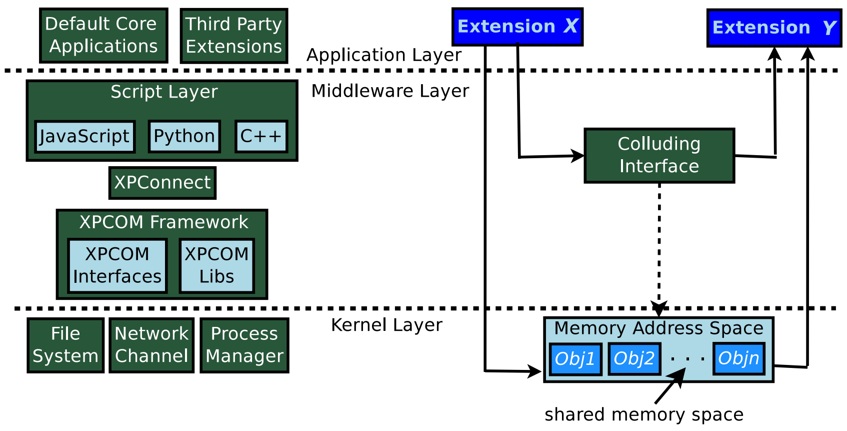


FIGURE 4.1: Illustration of inter-extension communication in Firefox browser.

The components invoked by Firefox extensions can interact within or across components through observer notifications. Our analysis aims at communications with components, both within a single extension and between different extensions. Browser extension system provides an interfaces for accessing different browser components. XPCOM offers an easy way to achieve this functionality through observer service interfaces. The two objects defined in different extensions can share information through observer notification interface to set synchronous communication. Furthermore, the browser does not provide any isolation among multiple extensions running simultaneously and hence all installed extensions use same memory space for variables and objects. A more illustrative scenario is shown in Figure 4.1, where an extension X is accessing some information using Object $Obj1$, and notifies extension Y. Now, Y may have transitive access to information which was accessed through $Obj1$ because both X and Y share same address space in memory.

4.2 Threat Model

Firefox browser contains a vast array of extensions, and a user may install extensions from trusted source with varying trust levels. Besides that, a user may install an extension from unknown source alongside trusted extensions that handle private information such as authentication of data and personal information. For example, a user might install both a highly trusted banking extension and a free game extension. In an ideal scenario, the game should not be able to gain access to the user's bank account information through bank extension.

The biggest disadvantage of Firefox browser extension model is that, it does not provide isolation among extensions. All extensions run in single-process and use shared memory space. An attacker can take advantage of this weakness to set up collusion among extensions, which may lead to privacy leakage and privilege escalation attacks. Collusion can become a primary attack vector. Indeed, if a developer accidentally exposes functionality, then the extension can be tricked into performing an undesirable action. If an extension sends data to the wrong recipient, then it might leak sensitive data. In this chapter, we consider how extensions can collude with each other to execute colluding-based attacks. It should be noted that we do not discuss the attacks initiated by a single browser extension.

The attacks discussed in this chapter are categorized based on two threat models. Colluding extensions use existing or construct new communication channels to perform actions or access resources they are unable to gain access to otherwise. Two threat models for collusion-based attacks in browser extensions are as follows:

1. **Covert Channel Communication.** In the first threat model, we consider attacks that make use of covert channel communication, and we assume the attacker can access shared memory space allocated for extensions. In particular, we focus on attacks where the shared memory objects carrying sensitive information on one extension communicate with another extension through a covert channel.

2. **Overt Channel Communication.** In the second threat model, we consider attacks that originate from overt channel communication. In particular, we focus on attacks where an attacker uses overt communication channel to compromise browsers and other extensions. For example, an attacker can modify extension and/or browser preferences to make them unsecure.

4.2.1 Covert Channel Collusion

We now briefly describe the colluding techniques using covert communication channels to give an intuition of how they work. We examine the security challenges of IEC in browser, from the perspectives of notification senders and notification listeners. We discuss how sending a notification to the wrong application and sharing inter extension objects can compromise sensitive user information. An attacker can exploit IEC functionality of browser extension to deploy privacy leakage attacks, such as eavesdropping, service hijacking, and side channel. This section discusses a detailed overview of achieving collusion using covert communication, leaving the implementation details to Section 4.3. It should be noted that we will not discuss the attacks that are originated using single browser extension.

Observer Notification Collusion. When one component in an extension sends a notification to another component to notify that a task is complete, there is no guarantee that the notification is received by the intended recipient (component). A malicious application can intercept a notification over covert channel simply by registering using an observer notification interface (`nsIObserver`). The malicious application then carries out subsequent actions. We demonstrate how attacks are deployed on the browser by stealing or intercepting the notification.

Our threat model assumes that individual extensions are benign, and they are installed on a browser. The objective of attackers is to compromise the extensions and the intended information access. In particular, the browser extension uses event or observer notification messages to setup collusion among different applications (pre-installed and third party extensions). We further classified this threat model into two sub-categories: (i) intercepting observer notification; (ii) source to sink notification.

1. **Intercepting Observer Notification:** An attacker can intercept the observer notifications to act maliciously. In this model, we assume that one trusted extension and one malicious extension are installed on the browser. Getting the victim to load extension is not very difficult. For example, emails, social networks, and advertisements can be used to load extensions. The objective of attackers is to intercept notification topics on object, which is sent from a trusted domain to a trusted service (web, network, and file system). Figure 4.2 illustrates the model using various steps:

    (a) In Step 1, a trusted extension sends a notification of $Obj1$ to communicate with trusted service, which then replies to trusted extension in Step 2.

    (b) Step 3 shows that a malicious extension intercepting a notification on $Obj1$.

    (c) In Step 4, the information from trusted extension is used through $Obj1$ reference to execute a malicious event (Step 5).

    Through eavesdropping on notification between two processes, an attacker can accomplish malicious tasks. In Section 3, we present more illustrative examples such as eavesdropping and service hijacking attacks caused due to an interception of object notification.

2. *Source to Sink Notification.* The source to sink analysis has been used to track tainted information flow in browser extensions. This analysis is limited to single

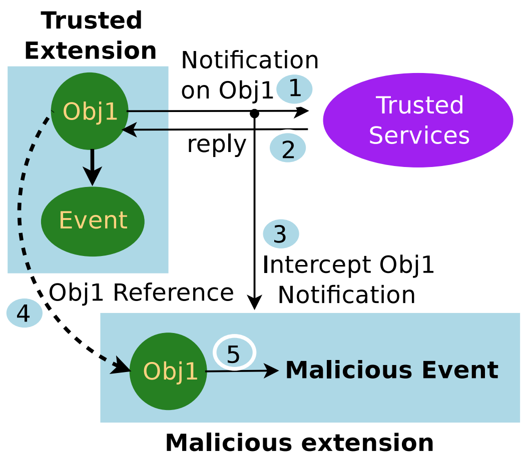


FIGURE 4.2: Interception on user-defined notification.

extension, and hence an attacker can initiate privacy leakage through information using source in one extension and sink in another extension. A component from one extension can notify an object's reference to the component from other extension correspond to flow of information from source to sink. A static list of source and sink APIs is given by [57]. Figure 4.3 illustrates the generic operation by two extensions for sending information from source to sink. The component from extension X first takes sensitive information from source resources, such as DOM, password manager, and then notifies the object's reference to extension Y. Extension Y takes further action to send sensitive information to sink. Examples include leaking information to attacker's domain using network channel, writing information to file system etc.

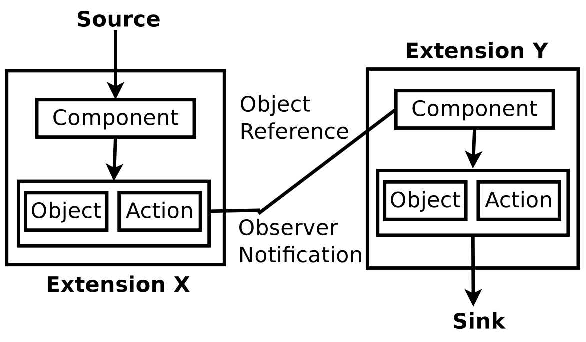


FIGURE 4.3: Source to sink interaction between two extensions.

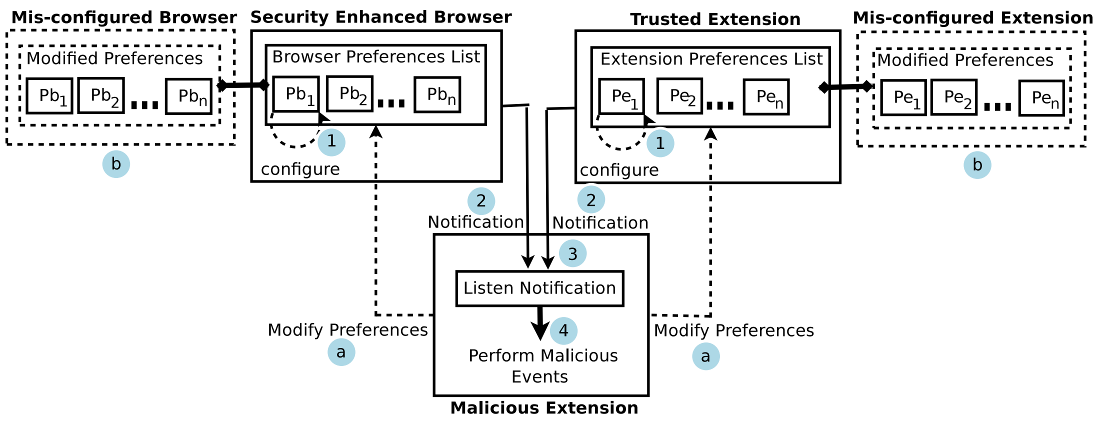


FIGURE 4.4: Illustration of how preferences can be observed and altered by other extension.

4.2.2 Overt Channel Collusion

This section discusses a how overt communication can be used for achieving collusion among extensions and the browser. In the overt channel collusion, we consider the preference overriding attacks. The attacker can change preferences of browser and extensions using another extension over an overt channel, without violating any security policy. We examine the security challenges of the browser from the perspectives of critical preferences. Also, we discuss how altering or listening to preferences can misconfigure the browser as well as the extension.

Preferences Overriding. The preferences associated with either browser or extensions can be misconfigured leading to serious threats. In this threat model, we discuss how preferences can be overridden by an extension to achieve malicious goals. The model assumes that a malicious extension manages to get installed on the browser. Figure 4.4 illustrates two different models to present how a malicious extension can compromise a browser and other extensions by modifying the preferences. We discuss these two models in the following paragraphs:

Observing Preferences. In this model, a malicious extension observes the notification on preference change and executes malicious event afterwards. The browser is enriched with various security related preferences that prohibit the attacker from executing malicious tasks. Some extensions (especially security related) may also have preferences that allow configuration at runtime [70]. To understand the threat which arises from misconfigured preferences, let us consider a set of browser preferences denoted with

$Pb_1, Pb_2, \cdots, Pb_n$ and a set of extension preferences denoted with $Pe_1, Pe_2, \cdots, Pe_n$. A general overview of how an attacker can misconfigure a browser and/or extension(s) by altering preferences is given in Figure 4.4. The steps (numeric representation) taken by user or attacker for altering browser preferences are as follows:

1. Step 1: Suppose a preference Pb_1 represents a preference to enable/disable the pop-up windows for a browser. Initially, Pb_1 value is set to `disabled`, which prevents an extension from performing malicious events via pop-up window, for example, Clickjacking attack [100].
2. Step 2: The moment Pb_1 value changes to `enable`, the browser sends a global notification, which might be observed by malicious extension.
3. Step 3: Malicious extension listens to the global notification sent for Pb_1.
4. Step 4: After intercepting notification, the malicious extension can execute malicious events, such as opening of pop-ups on browser.

Altering Preferences. In this model, a malicious extension alters the default preferences of the browser and/or other extensions. Some preferences are critical and if not configured properly can cause a significant security threat to the browser. Figure 4.4 also illustrates various steps (alphabetic representation) to demonstrate how a security enhanced browser can be breached by a misconfigured preference. More details of altering preferences with the example are discussed in next section. The steps for altering extension preference are as follows:

1. Step a: Malicious extension changes security preference of either the browser or the installed extension(s). For example, let us consider a security extension `noscript`, which has whitelisting preference. If a malicious website acquires this preferences, all security checks are bypassed for the web site.
2. Step b: A browser can be misconfigured by altering security relevant preferences, such as, enable/disable JavaScript, cookies, password storage etc. This may lead to cookie/password stealing, script injection etc. Altering of preferences can weaken the browser security. In a similar manner, misconfiguration of an extension by changing preferences can have security implications.

4.3 Instantiation of Colluding Attacks

In this section, we introduce proof-of-concept example of the colluding attacks over covert and overt communication channels. We describe an attack scenario and provide a detailed description of our attack implementation. In the event notification threat model, we demonstrate the impact of the collusion-based attack and show how these attacks can lead to privacy leakage in the browser. In preference overriding threat model, we demonstrate the effect of insecurely configured browser and extension preferences. We have constructed our example attack scenarios employing security relevant preferences of the browser and/or extensions.

In all our attack scenarios, we have assumed that a user downloads and installs the extension from the Internet. Our extensions provide legitimate functionalities to the user in declared benign mode but shall collude with another extension or service otherwise to execute malicious tasks. Apart from collusion, in this section we also give a thorough discussion of the implementation. First, we briefly discuss the colluding extensions using observers notifications (Section 4.3.1) and source to sink notification (Section 4.3.2), then we present their respective attack implementations with example attack scenarios. In sections 4.3.3 and 4.3.4, we discuss the collusion by altering preferences with the help of an illustrative example.

4.3.1 Attack Technique: Intercepting Observer Notifications

Using notification interception technique, we present an attack in which another extension intercepts the communication between the trusted extension and trusted application. Figure 4.5 illustrates various steps taken by two extensions to achieve the malicious goal of privacy leakage. We have implemented an extension X with the functionality of reading information from trusted Facebook server that provides facilities such as like and share Facebook timelines. Here, we assume that a user has logged into Facebook application and X is legitimate. After X reads information from Facebook page by reading DOM content (Step 1), it then sends a notification to the browser web window to notify that read event is finished. The malicious extension Y (Step 2) listens to this notification and access object's reference read by X in Step 1. More in detail: after Y listens to the notification it can set up a covert communication channel with Facebook server to steal sensitive information such as Facebook user profile (steps 3 and 4). Here, the information extracted by legitimate extension (X) is stolen by malicious extension (Y).

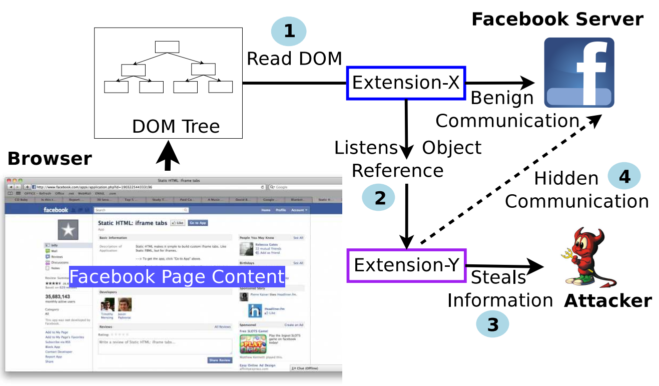


FIGURE 4.5: Attack scenario intercepting observer notifications.

4.3.2 Attack Technique: Source to Sink Notifications

This technique presents the attacks using two colluding extensions in which one extension takes information from a source while other sends this information to a sink. Using the proposed method, we implemented a modified version of the attack that is derived from MitB (Man-in-the-browser) attack, a well known Banking Trojan [77]. The primary goals of this attack are:

1. Stealing user assets, such as login credentials.
2. Modifying current bank transaction on the fly.
3. Modifying the web page's contents on the fly without the victim's knowledge.

The potential attack vector for MitB attack is through malicious browser extension. Once installed into the browser as an extension, this gets activated when a user visits target websites, such as bank websites. This technique demonstrates a new way of launching the MitB attack using two extensions having benign functionality, so that even malicious flow analyzers existing, such as, VEX [56], and SABRE [57] would not be able to detect any maliciousness and vulnerability in either of the extensions.

Object Collusion Attack Scenario 1. First attack scenario for collusion based attack consists of two benign extensions. Extension Y has a functionality to read static and dynamic information from web pages including the information input by the user at run-time. The second extension X can send information over network channel. Using these two extensions with benign functionality, we setup a collusion between them so that one extracts information from web page and other can send this information to a third-party domain. In addition, these extensions can work for every web page open in the browser window. Figure 4.6 illustrates the process of exchanging information between two colluding extension. The steps for achieving malicious goal using X and Y are as follows:

1. Y first read web page information such as user credential from DOM. It creates an object, stored in the global address space (Step 1)
2. The reference to the object stored in the global address space is notified to the browser. This notification is listened to X (Step 2).
3. X then uses this reference to access the DOM information extracted by Y and transmits over the network channel to the attacker's domain (Step 3).

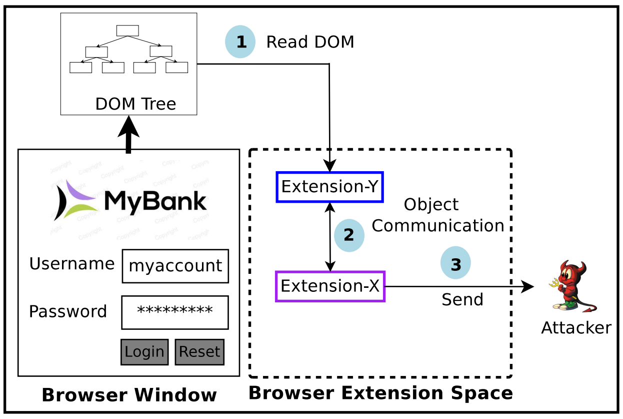


FIGURE 4.6: Scenario-1 showing how credentials can be stolen.

Object Collusion Attack Scenario 2. This scenario demonstrates the modification of web page information on the fly. We construct this scenario using the websites that offer a user to buy download credits (torrents, file hosting, warez sites, etc.) as an example. These sites allow any user to create a *userID* with credentials and payment details. We implemented two extensions to steal user credential in a way the credit provider server does not notice. Figure 4.7 shows various steps taken by two extensions in collusion to modify current user details. At the time of account creation, our attack modifies the user details without the victim's notice. In particular, the behavior is as follows:

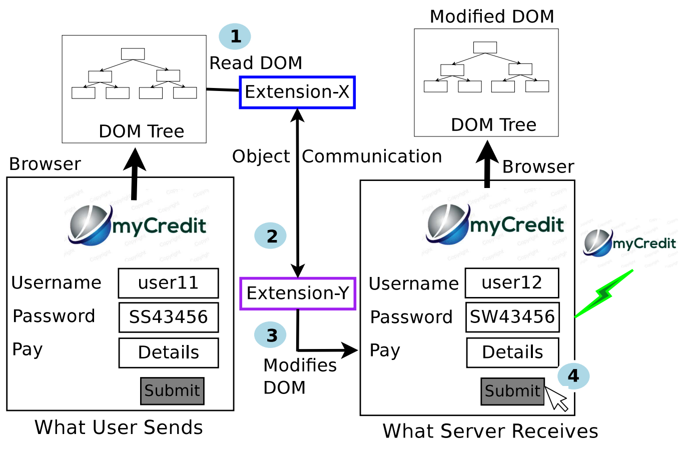


FIGURE 4.7: Scenario-2 showing how a new field can be added and sent to attacker domain.

1. Step 1: X reads the information from the web page DOM.
2. Step 2: X wraps the reference of an object that carries web page information and makes it accessible to extension Y.
3. Step 3: Y modifies the user details.
4. Step 4: Finally when victim user clicks on the `submit` button, the modified information is sent to the credit server, and fake account is created on the server.

Object Collusion Attack Scenario 3. In this scenario, we demonstrate how an attacker can dynamically add new fields on the current page. We have considered on-line shopping websites as an example to demonstrate this attack. Suppose a victim user wants to buy some item he has selected from a shopping site. We have implemented an attack using two extensions having following functionalities: the first extension modifies the shipping address, and mobile number of the purchaser. The second extension adds new field to steal payment details of the victim user. Figure 4.8 shows the various steps taken by two extensions to create this attack.

1. Y reads all information from the page DOM, and at the same time it adds new field to the web page that asks for the payment details (steps 1_a and 1_b).

2. Step 2: Y wraps the reference of an object carrying requisite information and makes this reference accessible to X.

3. Step 3: X modifies the user detail by modifying DOM.

4. Step 4: When the victim user clicks on the `submit` button, the modified information is sent to the credit server, and payment information is sent to the attacker.

X is benign because it modifies the information captured from sensitive source but this information is not sent to sink. Instead, this information is obtained by Y.

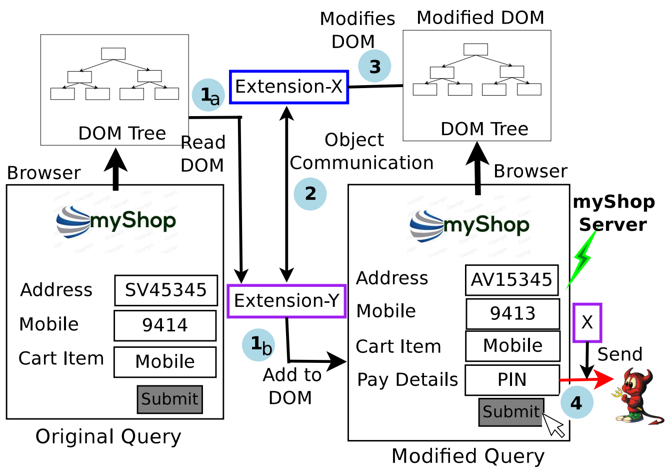
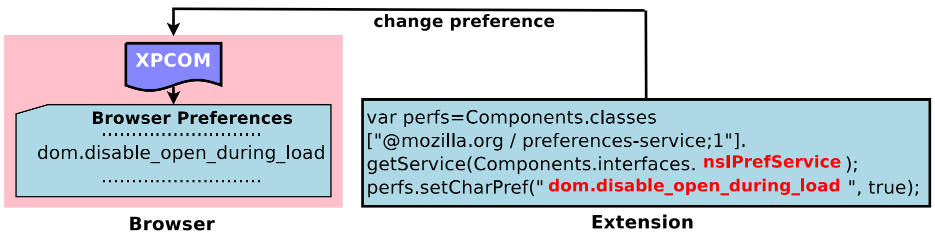


FIGURE 4.8: Scenario-3 showing how transactions can be modified on the fly.

4.3.3 Altering Preferences

From the attacker's perspective, the preferences can be set or modified for achieving malicious goals. In Firefox, an extension has privilege to change the preferences of the browsers, as well as any other installed extension. For example, an attacker, through a malicious extension added to the victim's browser can set a malicious page as the home page. He can also modify the critical preferences of security tools such as *noscript* [70], *adblock*, etc. and also has a capability to alter the browser's privacy settings allowing access to private data. This attack technique shows how two preference management interfaces (`nsIPrefService` and `nsIBranch`) can be used by the attacker to set or reset the stored preferences. We describe two potential attack points that can be exploited by an attacker through preferences system.

Attack Scenario 1: Altering browser's Preferences. A browser has many security related preferences, such as, enable/disable cookies/JavaScript, and privacy settings. An attacker can set or reset critical browser preferences through an extension having privileges to override the default preferences values. For example, we can disable the Firefox pop-up blocking by setting `dom.disable_ open_during_load` preference value to `true`. Figure 4.9 shows an example for changing preference of the browser.



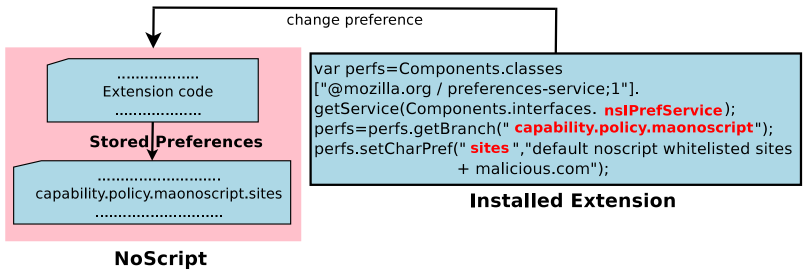
```
var perfs=Components.classes
["@mozilla.org / preferences-service;1"].
getService(Components.interfaces. nsIPrefService );
perfs.setCharPref(" dom.disable_open_during_load ", true);
```

FIGURE 4.9: Changing privacy settings of web browser.

Attack Scenario 2: Altering Extension's Preferences. Some extensions use preferences for customizing. For example, `noscript` has provision for whitlisting an URL so that security checks can be bypassed for this URL. An extension can change these preferences without the user's notice. We have implemented an extension for bypassing `noscript` using `nsIprefService`. Our extension can change critical preferences of `noscript`. We have added a malicious domain (e.g., *malicious.com*) in `noScript` using `capability.policy.manoscript.sites` preference string, so that all security checks provided by `noscript` for that domain are bypassed. Figure 4.10 shows the code snippet of our extension.

4.3.4 Observing Preferences

A malicious extension can take advantage of misconfigured preferences to execute an attack on the browser. An extension remains silent, waiting for user preference to change and then launches the attack. An extension remains silent, waiting for user preference to change and then launches a attack. The technique can be more elaborated with real attack example. For this, we utilize the observer notifications technique to listen to preferences change. Let us consider a simple attack scenario in which a malicious extension is installed on the browser and has a capability of injecting scripts into the browser content window. Initially, the malicious extension cannot execute script in the content window because the preference for the content security policy (CSP) is enabled, i.e., `security.csp.enable` is set to `true`. The extension waits for user to disable this preference, or it can use some social engineering trick enticing user to disable it. Once the user disables CSP, our extension listens to the change and acts immediately.



```
var perfs=Components.classes
["@mozilla.org / preferences-service;1"].
getService(Components.interfaces. nsIPrefService);
perfs=perfs.getBranch(" capability.policy.maonoscript");
perfs.setCharPref(" sites ","default noscript whitelisted sites
+ malicious.com");
```

FIGURE 4.10: Changing preference of NoScript extension.

4.4 Results and Analysis

We crafted prototypes of our attacks and evaluated their effects. The observer notification collusion is evaluated on 300 web pages taken from different web domain categories: banking, mailing, e-commerce domains, and domains that offer download credits. We have collected 50 banking web sites registered in different countries. The Bank websites are randomly selected from the list of top 10 banks for 10 different countries. The other 250 websites from different domains are top 50 websites taken with respect to Google ranking (taken from Jan-Feb, 2014). Due to security reasons, we cannot reveal the name of the banks. The preference overriding is evaluated with five critical browser configurations and three traditional security extensions.

TABLE 4.1: Actions Performed and Parameters taken in the Experiments.

	Actions Performed on Web Pages		
	Banking Domains	**Mailing/e-commerce Domains**	**Buy Credit Domains**
Scenario 1	Extract username, password	Extract username, password	Extract username, password
Scenario 2	Not Applicable	Modify transactions Modify shopping address	Modify transaction amount Modify username
Scenario 3	Not Applicable	Add OTP/password field	Add OTP/password field

Table 4.1 summarizes the actions performed on the web pages along with the parameters extracted and manipulated for the attack scenarios. We have tested our proposed colluding attack using three different attack scenarios. However, we have not applied

scenarios 2 and 3 for Banking domains. Since these two scenarios modify/add fields to the web page, we need a Internet banking account to login in a bank website and facilitate Internet banking, which is not possible.

4.4.1 Evaluation of Collusion Techniques

For attack scenarios 1 and 3, we have taken five legitimate Firefox extensions individually from shopping and social category of Mozilla add-on database. These extension incorporate functionalities desired by our attack scenarios like accessing information from the web page, sending information over the network, etc. We embed the additional desired functionality in these extensions so that they can also collude with each other. All these extensions are installed on different versions (3, 9, 12 and 25) of Firefox. We then apply test case scenarios for selected web sites taken from three web domains discussed above. Table 4.2 summarizes the experimental results for Object Collusion attack. We have tested the attack using three different attack scenarios and found following observations:

Table 4.2: Results for attack scenarios executed on web domains.

	Banking Domains	**Mailing/e-commerce Domains**	**Buy Credit Domains**
	Success%	Success%	Success%
Scenario 1	19%	100%	100%
Scenario 2	Not Applicable	100%	100%
Scenario 3	Not Applicable	77%	80%

- Scenario 1 is 100% successful in mailing, e-commerce and download credit domain, whereas only a few bank websites allow the extensions capture credential information. Our attack can capture username from banking login page, but the password is hashed. In one-third of 50 banking sites we have tested, our attack scenario can extract both username and password from bank login page.
- The second attack scenario is 100% favorable for all the web pages that we have tested. Every website allowed our extension to modify the typed content on the fly. This scenario is critical for every shopping and credit domains that are tested. We have not applied this attack on banking domains.

- The third attack scenario is successfully executed on 78% shopping domains and 80% credit procurement domains. Other domains did not either permit additions of extra field on the page, or suppressed subsequent processing if field was successfully added. The web domains maintain session with every text field using *type=hidden* for HTML input tag. When a new field is added to the web page, the course is changed, and this effectively mitigates the attack. We have not applied this attack on banking domains.

- This section demonstrates how misconfigured preferences can result in critical browser attacks. We have selected five security relevant preferences of browser and three popular browser security extensions: (1) noscript; (2) Web of trust; and (3) adblock. We have implemented one extension, which can change the selected preferences. Our extension is installed on different versions (3, 9, 12 and 25) of Firefox. This extension modifies the selected preferences with insecure values and for each preference we have evaluated browser security. Table 4.3 shows the five critical browser preferences and preferences of three security extensions modified by our extension.

4.4.2 Ramifications of Colluding Attacks

We have implemented our extensions in such a way that even a client side solution would not be able to detect information and data leakage. We have tested our attacks against four popular detection methods (i.e, VEX [56], SABRE [57], JSFLOW [92] and HMM approach [94]). In this section, we present the brief analysis of these methods and discuss how colluding attacks are effective against these solutions.

1. Colluding Attacks and VEX: VEX approach can detect (some of) the known vulnerabilities in Firefox browser extensions using static information flow analysis. In their analysis, they have checked the suspicious flow pattern from injectable sources to executable sinks. Our extension contains either sensitive source or sink but not both. Although, the legitimate functionality of an extension may constitute part of such flows, but these partial flow are not detected by the current framework of VEX.

Table 4.3: Results showing browser security leaks using preferences.

Preference	**Risk after modifying preferences**
Security relevant browser preferences	
`security.csp.enable`	enable/disable the content security policy of browser
`dom.disable_open_during_load`	Allows pop-up windows on browser if set to true
`dom.popup_allowed_events`	Adding entries to this list may allow unwanted pop-ups
`extensions.update.url`	Adding malicious url using this preference will change extension update source
`browser.safebrowsing.malware.enabled`	Do not download malware & blacklists do not check downloads if set to false
`noscript` **extension**	
`capability.policy.manoscript.sites`	Adding url to this preference will bypass all the security checks provided by noScript
`adblock` extension	
`extensions.adblockplus.whitelistschemes`	Using this preference an attacker can
	add and remove whitelisting rules
`WOT` **extension**	
`weboftrust.norepsfor`	Adding malicious domain using this preference will bypass the malicious domains

VEX checks for vulnerable patterns in JavaScript extension on the basis of five source to sink flows: out of these flows, our concern is only towards `wrappedJSObject`. VEX marks an object as vulnerable if the object is obtained using `wrappedJSObject` calls in a method. The source locations are created using `wrappedJSObject` and sink location is the method called by this object. In our attack, we have used `wrappedJSObject` to wrap notification objects. VEX is able to determine that the information has originated from a sensitive source from one extension while in another extension the information flows to a sink. Since a single extension has no suspicious flow and vulnerability, it will be passed as legitimate extensions by VEX. Our colluding attacks are thus effective against VEX.

2. Colluding Attacks and SABRE: SABRE is based on the dynamic analysis of object's flow in JavaScript extensions. SABRE associates a label with each in-memory JavaScript object in the browser, which determines whether the object contains sensitive information. The label tracks the flow of sensitive information and the flow of low-integrity information. SABRE only propagates labels (objects), which are modified by the JavaScript extension and passed between

browser subsystems within a single extension. It raises an alert if an object containing sensitive information is accessed in an unsafe way. However, they have not mentioned the object, which is outside extension's scope, used by another extension. The colluding attacks that we have implemented pass an object from one extension to another extension. Due to this reason the SABRE framework lets execution of colluding attacks.

3. Colluding Attacks and HMM: In [94], the authors present a model-based approach for detecting vulnerable and malicious browser extensions using Hidden-Markov Model (HMM). The authors have analyzed different types of browser extensions (malicious, vulnerable and benign) based on the distinguishing features extracted from these extensions while in operation. These features help determine the behavior of the extension and thereby detect its type automatically. The major limitation of this work is the that they are classifying extensions on the basis of a non-exhaustive malicious and vulnerable signatures and the method also excluded the objects wrapped in wrapper method, such as, `wrappedJSObject`. We have implemented our extensions with benign behavior and used wrapper methods and hence the above-discussed method is not able to categorize our extensions as malicious or vulnerable.

4. Colluding Attacks and JSFlow: This approach is only partially related to our attack domain. JSFlow is a security-enhanced JavaScript interpreter for fine-grained tracking of information flow. The authors have implemented the information flow policies for the full JavaScript language, as well as tracking information in the presence of libraries, as provided by browser APIs. The JavaScript that the authors have analyzed is the script embedded in the web applications. They have not focused or discussed the JavaScript code in extensions. In absence of analysis of JavaScript code in conjuction with extension privileges, the claimed policies may not be the information policies for complete JavaScript language. Since we have implemented attacks using two extensions, and the JavaScript code is more privileged code, thus the policies discussed by JSFlow cannot be enforced on these extensions.

4.4.3 Mitigation Techniques

Solving the confinement problem, and in particular securing all possible covert channels in a system, is known to be a difficult problem [101]. It is further more complex in the case of browser extensions, where extra privileges given to browser and exposed API features are key to user and developer adoption. Mitigation can be achieved either at the design time (by reducing the privileges to sensitive APIs or by limiting communication possibilities) or by analyzing static and dynamic properties of applications and their off-line or at run-time interactions. In this section, we explore possible mitigation techniques for the collusion attacks as discussed earlier.

Design Time Mitigation Techniques. There are number of techniques that could be considered by browser and the developers of the extension:

1. **Sandboxing.** We explored SpiderMonkey, a JavaScript engine for Firefox browser and found few weaknesses in handling the JavaScript objects. SpiderMonkey creates a top-level object called*JSRuntime* object that represents an instance of the JavaScript engine. A program has only one *JSRuntime*, though it may have many threads. The *JSRuntime* is the universe in which JavaScript objects live; they can not be shared with other JSRuntimes. The *JSContext* is a child of the *JSRuntime*.

   A context can run scripts, contains the global object and the execution stack. Once created, a context can be used any number of times for different scripts. Objects may be shared among *JSContexts* within a *JSRuntime*. There is no fixed association between an object and the context in which it is created. Since the memory is shared among all extensions, the objects of one extension are accessible to another extension. We suggest a sandbox environment for extensions so that their memory spaces are isolated from each other, and any communication should be through browser kernel. Sandboxing through virtualization shall mitigate such attacks. Alternately, binding between object and its context can be strengthened and violation of the binding is not allowed by default.

   Few modern Web browser components run in a sandbox environment with restricted privileges. Browser's critical component, such as the rendering engine runs in a sandbox with restricted privileges with no direct access to the local file system or other OS components. For example, in Google Chrome architecture, rendering engine runs in a sandbox and has no direct access to the local file system. Apart from Sandbox environment, the web browser provides isolation among web programs. The browser provides single and shared space for all the extensions. In some browsers, the extensions are isolated from other browsers components but not from each other and share same address space. If each extension executes in its own address space as in Chromium browser that provides separate address space for each tab [102]. If the same policy is applied to the extensions, their objects are also isolated from each other and cannot communicate directly to avoid colluding attacks.

2. **Limiting Privileges.** When designing APIs or interfaces exposed to third-party developers, designers should carefully consider the possibility that the API may create a covert communication channel between extensions and browser. If any overt or covert channel is found, its privileges should be controllable through the browser configurations or policies. Preferably privileges assigned to any communication channel should be less than those of objects carrying sensitive information to be protected. While transmitting contents of these objects across a communication channel, the privileges of the channel may be temporarily upgraded.

3. **Limiting Channels.** Communication channels constructed at browser application level are dependent on the APIs exposed by the underlying operating system. Careful design of permissions used to access data sources as well as data sharing points (i.e., sharing of files or preferences, settings, memory, observer notifications) could be used for identifying extensions that require an excessive number of permissions. In addition, some of these channels could be closed by removing unnecessary APIs after an analysis of the used and unused ones. This would enable strict security while maintaining a reasonable amount of freedom for the extension developers. Yet another way of achieving this is through encryption of the sensitive data.

Extensions Analysis Techniques. There are number of techniques proposed in the literature to analyze single extension that could be tuned to detect colluding communication among extensions:

1. **Black-box Analysis.** One strategy in trying to detect collusion is to add a data monitor between separate extensions on the browser. This technique would remove the need to detect covert channel by only monitoring data leakage from an extension. In such a model, when data from one extension is used, the monitor would store it and track the data sent to the colluding extension. While this approach seems promising, it is inherently limited: malicious extension can encrypt or encode data in a way that it defeats monitoring. While it is clear that black-box analysis may detect some trivial attempts of collusion, it clearly does not provide a complete solution.

2. **Improving client side solutions.** There are number of offline extension analysis solutions. But none addresses the collusion among extensions or detect covert channel communication. We have analyzed four popular client side solutions, VEX [56], SABRE [57], JSFLOW [92] and HMM [94] that are effective against vulnerable and malicious extensions. We found that these solutions are meant for analyzing an extension individually, and hence remain ineffective if a malicious flow originates from two or more inter-communicating extensions. It is likely that when colluding extensions are executed simultaneously on a device, they would show a different behavior than when executed independently. For example, they would detect each other's presence and engage into communication over a covert channel. To mitigate such attacks, behavior analysis could be used to detect such a change of behavior.

4.5 Summary

In this chapter, we went through the concept of colluding browser extension for Firefox that we have recently proposed in the preliminary version of this chapter. To the best of our knowledge, this is a new concept related to security risks and exploits present in Firefox's extension model. We demonstrated the extension collusion attacks with respect to extension communication over covert and overt channels. In particular, we showed how two legitimate extensions can collude to achieve malicious goals and how an individual malicious extension can mis-configure browser and another extensions configurations. Our attacks are undetectable by existing known client side methods used for detecting malicious flow and vulnerability in extensions. We have demonstrated our finding by targeting a malicious goal using two legitimate extensions on three critical web domains; banking, online shopping, and websites that allow users to buy download credits. We also provided a proof-of-concept explaining how multiple extensions can collude with each other for compromising the browser for data leakage.

Chapter 5

sandFOX: Sandbox for Firefox Browser

Browser functionalities can be widely extended by browser extensions. One of the key features that makes browser extensions so powerful is that they run with “high” privileges. As a consequence, a vulnerable or malicious extension might expose browser, and operating system (OS) resources to possible attacks such as privilege escalation, information stealing, and session hijacking. The resources are referred as browser as well as OS components accessed through browser extension such as accessing information on the web application, executing arbitrary processes, and even access files from a host file system.

This chapter presents sandFOX (secure sandbox and isolated environment), a client-side browser policies for constructing sandbox environment. sandFOX allows the browser extension to express fine-grained OS specific security policies that are enforced at runtime. In particular, our proposed policies provide the protection to OS resources (e.g., host file system, network and processes) from the browser attacks. We use Security-Enhanced Linux (SELinux) to tune OS and build a sandbox that helps in reducing potential damage from attacks on the OS resources. To show the practicality of sandFOX in a range of settings, we compute the effectiveness of sandFOX for various browser attacks on OS resources. We also show that sandFOX enabled browser imposes low overhead on loading pages and utilizes negligible memory when running with sandbox environment.

Contributions This chapter makes the following contributions.

1. We present the design of new sandbox environment called “sandFOX” for the browsers on SELinux enabled Linux OS.
2. To the best of our knowledge, we are the first to enforce security policies on OS resources accessed implicitly or explicitly by the browser. We are the first to cope with compromised browser applications (plug-in or extension) while still maintaining the security of browser and host OS resources.
3. We show how SELinux policies, as designed in Linux operating system, can be useful for browser security.
4. We evaluate our sandFOX and security policies using malicious and critical resource eating extensions. We show how these extensions can attack if a browser does not use sandFOX policies.

5.1 Problem

As the advent of web browser technologies, more and more new types of attacks has evolved on the browser and operating system resources. The attacker initiate various attacks on browser such as web-based attacks [66], side-channel attacks [103], and extension-based attacks. The extension based attacks are most critical as they exploit browser and OS resources. The threats against OS resources using browser extensions are discussed by many researchers. These third-party extensions are used to enhance the core functionality of the browser [18] and hence enjoy high privileges, sometimes as high as those of the browser itself. The threats posed by these extensions are very critical, for instance, an attacker can launch a malicious process to compromise an entire system. In particular, Liverani [10] takes a more practical approach and demonstrate examples of possible malicious actions induced by an adversary on Firefox extensions. In our previous work [27], we have provided the proof-of-concept to show vulnerable points in Firefox extensions system and shows various attack vector used by the attacker to install and spread the JSEs.

Browser provides thousands of free JSEs to customize and enhance the look and feel of the browser. Browser renders JSEs to run with full chrome[1] privileges including access to browser components such as cookie manager, password manager, and access to information present in a web page. While the ability to access such privileges within browser and host machine OS, the hosting browser effectively opens itself up to attacks and poor programming practices from every JavaScript library or API it uses. For instance, a vulnerable or malicious extension might access */etc/password* file to gain root access in a Linux based host machine. Consequently, malicious and benign-but-buggy JSEs are significant security threats in browser [24].

Efforts to provide security in this evolved model of web browsing have had limited success. The Same Origin Policy (SOP) [19] is one security policy that most web browsers implement. The SOP restricts the objects or scripts loaded from one origin (or domain) to access objects and scripts from the other domain. However, the SOP is too restrictive for use with browser extensions and plug-ins. The extensions privileges allow a malicious extension to bypass SOP [27]. As a result, extension developers have been forced to implement their security policies [104]. Current research efforts into more secure web browsers help improve the security of browsers in the presence of browser extensions [56, 59, 60, 105]. But ensuring the protection of OS resources from browser extension is still a daunting task.

Component isolation and restrictive access to resources can be used to address OS level attack through browser. The new browser architectures for separating the browser components from OS are proposed in [45, 46, 106, 107]. However, these more secure web browsers are susceptible to serious attacks because all applications including extensions share OS services, which the attackers can compromise. Also, the isolation forces the extensions to loose functionality of accessing OS resources. Furthermore, all previously published solutions require browser redesign.

5.2 sandFOX: Proposed Sandbox

In this chapter, we propose a sandFOX policies, an OS based sandboxing environment that isolates browser from the OS resources to ensure the protection from extension

[1]chrome is the entities making up the user interface of a specific application or extension

based browser attacks. In particular, our security policies allow the extensions to achieve the desired functionality in a restrictive sandboxed environment. In sandFOX, we use SELinux policies to build OS-level sandboxing. SELinux allows to set policies on the critical OS resources that are accessible via browser extensions. sandFOX provides a clean separation between the highly privileged extensions and the OS resources, and it also allows us to provide strong isolation guarantees to prevent threats from vulnerable and malicious extensions. This chapter provides an threat model and proposed sandbox environment for web browser. We first briefly discuss the threats model for Firefox browser (Section 5.2.1). Section 5.2.2 briefly describes an overview of SELINUX in Linux operating system. In Section 5.2.3, we discuss the sandbox environent for web browsers.

5.2.1 Threat Model

Isolation in web browser varies from vendor to vendor. For example, Firefox browser does not use isolated memory and runs on a single process model, whereas Chrome uses isolated multi-process architecture that allows isolation among components. In addition, the extensions can access and modify the stack of a random thread inside the process. However, every browser uses high privileges extension that communications with the OS resources. The vulnerable and malicious extensions in browser may lead to critical attacks.

This chapter presents a threat model for Firefox browser because our analysis observed that Firefox does not use any isolation between browser and OS resources, and hence an extension with malicious intent is able to access OS resources. The property of browser lacking isolation motivates us to drill down more into Firefox extension system. In this section, we discuss security risk caused by Firefox extensions. Here are a few ways that an attacker can still cause damage to more secure web browsers:

1. A compromised Ethernet driver can send sensitive HTTP data (e.g., passwords or login cookies) to any remote host or change the HTTP response data before routing it to the network stack.
2. A compromised storage module can modify or steal any browser related persistent data.

3. A compromised network stack can tamper with any network connection or send sensitive HTTP data to an attacker.

4. A compromised process manager can launch any arbitrary process and script in background with user notice.

To show the utility of our proposed sandFOX, we implement security policies to protect three major OS resources.

1. We develop a flexible security policies that allow us to apply restriction in accessing the entire host file system by the browser and extensions. Our policies deny an extension to execute if it tries to access system configuration files and directories, for example, */etc, /boot, /proc*, etc. Our policies give extensions the flexibility to access host file system while still maintaining the confidentiality and integrity of critical files, even if an attacker compromises the host file system.

2. We protect processes running on host OS. Our policies deny if an extension executes an arbitrary process on the host OS. In particular, we restrict browser process in accessing other OS processes or executing the new process from running browser application.

3. We restrict network access by the browser and extensions. In particular, our policies restrict an extension to tamper with any network connection or send/receive sensitive HTTP data to/from a remote attacker location. For instance, we restrict cross-origin network access and Unsecured port from being accessed by the browser applications (extensions). However, our policy also gives extensions the flexibility to establish network connection in same origin, i.e., extensions can access, send or receive information over the network in the same domain.

The sandFOX policies does not address attacks that operate within modern browser security policy, such as cross-site scripting [108], clickjacking [100], phishing [109] and request forgery [67]. Our goal is to prevent system-level attacks in the web browser that degrades the overall security of the browser and OS. For example, executing arbitrary process through extensions may launch malware in the host OS. SandFOX prevents a browser extension in executing new process and killing of running process. We assume that the techniques that secure web applications and provide protection against click-jacking, phishing, cross-site scripting and other web application attacks are resistive and trusted.

5.2.2 SELINUX Overview

The SELinux [110] mandatory access control system is designed to prevent an intruder who has gained a foothold in the system from escalating privileges. SELinux provide the principle of least privilege to protect the system services such as BIND [111], APACHE [112] and desktop applications such as web browsers. SELinux prevents the compromise of entire system services and applications due to the compromise of a single application. With the use of SELinux, the applications programs can be placed into individual sandboxes, isolating them from one another and from the underlying operating system. Furthermore, SELinux protects the confidentiality and integrity of data that are shared among applications. SELinux restricts users over how data may be manipulated. The sensitive data can be restricted to policies from inadvertent sharing, modification, or deletion.

SELinux provides two security mechanism to achieve a flexible security policy model: type enforcement and role-based access control. In type enforcement model, the domains are used to label running programs (processes), and types are used to label files and other resources. This model protects the system from vulnerable applications that can damage or destroy the system. Type enforcement operates by tagging each entity (such as processes) and object (such as files and packets) in the system. SELinux defines all the access and transition rights of every user, application, process, and file on the system. For example, an application, attempts to access an object (for example, a file), the policy enforcement server in the kernel checks the access vector cache (AVC). AVC is where subject and object permissions are cached. If the decision cannot be made based on data in the AVC, the request continues to the security server, which looks up the security context of the application and the file in a matrix. Permission is then granted or denied, with an avc: denied message detailed in `/var/log/messages` if permission is denied.

To further explore the SELinux policy model, let us consider an example of the password management program (i.e, passwd). In Linux, the password program is trusted to read and modify the shadow password file (`/etc/shadow`) where encrypted passwords are stored. The password programs implement its internal security policies that allow any Linux user except root to change only their password, whereas root can change any password. In general, the program running as the root user (which has all access to all files) has the potential to modify `/etc/shadow`. SELinux provides the strict policies rules that enables only the password program (or similar trusted program) can access the shadow file, regardless of the user running the program as root.

Under SELinux, all file systems, files, directories, devices, and processes have an associated security context. For files, SELinux stores a context label in the extended attributes of the file system. The context contains additional information about a system object: the SELinux user, their role, their type, and the security level. SELinux uses this context information to control access to processes, Linux users, and files.

SELinux's tighter access control can confine user and system programs to the minimum amount of privilege they require to do their jobs. First, it can restrict privileged daemons. So, if such a daemon is exploited, the attacker can perform only the operations that the daemon is supposed to perform, thereby limiting his or her playing field. Second, an unprivileged process, such as a browser, can only invoke calls that its supposed to, thereby protecting the system from potential vulnerabilities in unused calls.

5.2.3 SandFOX Architecture

We present several scenarios demonstrating SELinux's usefulness in the web browsers. In particular, we show that the existing Firefox architecture is immune to extension-based attacks and no security policy exists to protect the browser from these attacks. sandFOX policies are designed using SELinux would reduce the damage it could cause when exploited. Figure 5.1 illustrates the proposed sandFOX architecture. SELinux policies are applied on both Linux OS and browser to create a secure and isolated layer between the browser and operating system. The presence of this layer prevents an attacker from unrestricted access of critical OS resources.

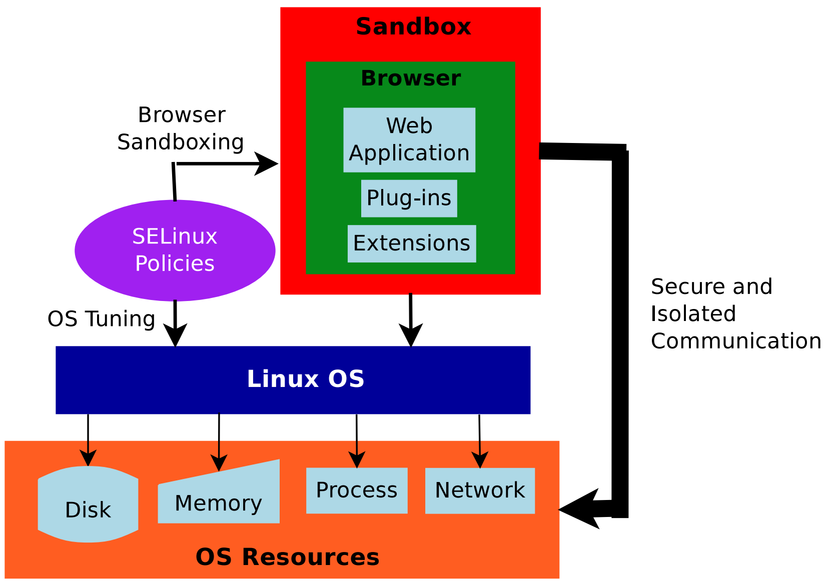


FIGURE 5.1: The execution environment of a sandbox browser.

In this section, first, we discuss three OS resources that are exploitable by extensions in current Firefox design; then we discuss the policies we implement with a focus on how each policy improves the security of the browser.

1. *Protecting Processes.* Firefox browser contains several Cross-platform Component Object Model (XPCOM) [12] interfaces that attackers might exploit to gain additional privileges. Firefox provides the `nsIProcess` interface to run other application from the third-party extension running in the browser. The `nsIProcess` interface is cross-platform, which means it is capable of executing an application in windows and Linux OS. If an attacker creates daemon to execute a malicious script on a Linux shell, the script would mis-configure Linux services. For example, let us consider a malicious shell script that modifies *iptables* [113] rules to disable host machine firewall. The browser extension downloads the script in `/boot` folder and then executes as a process. Once a malformed script exploits *iptables* rules, it can disable existing firewall rules and append new rules, which might result a security beach in victim OS.

```
<script>
var lFile = Components.classes["@mozilla.org/file/local;1"].
    createInstance(Components.interfaces.nsILocalFile);
var lPath = "\boot\malware.sh";
lFile.initWithPath(lPath);
```

```
5 var process = Components.classes["@mozilla.org/process/util;1"].
      createInstance(Components.interfaces.nsIProcess);
6 process.init(lFile);
7 process.run(false,['\boot\malware.sh'],1);
8 </script>
```

Listing 5.1: JavaScript Code Snippet for remote code execution exploit.

Listing 5.1 illustrates the browser extension code snippet for remote code execution by executing *malware.sh* script. The extension codes uses `nsILocalFile` interface to set path of *malware.sh* script stored on host OS file system (lines 2-4). The script will spawn a BASH shell on the victim's desktop (Line 7). This way an extension can access file system and invoke process on host OS.

2. *Protecting File System.* To preserve system integrity, we must prevent extensions from writing to critical executables or altering configuration files. But even read-only access should sometimes be prevented because some files might contain confidential information (such as `/etc/passwd` file). Linux uses file permissions to restrict access to files. However, a browser daemon running with root permissions allows an extension to override the permissions of the files. Listing 5.2 illustrates the browser extension code snippet for reading `/etc/passwd` file. The extension uses `nsIFile` interface to create an instance of host OS file system (lines 1-3). A file (`/etc/passwd`) is read using `nsIFileInputStream` interface (lines 5-8) .

```
1 var file = Components.classes["@mozilla.org/file/
      directory_service;1"].
2                                 getService(Components.interfaces.
      nsIProperties).
3                                 get("ProfD", Components.interfaces.
      nsIFile);
4 file.initWithPath("\\\etc\passwd");
5 var inputStream = Components.classes["@mozilla.org/network/file-
      output-stream;1"].
6 createInstance(Components.interfaces.nsIFileInputStream);
7 inputStream.init(file1, -1, 0666, 0);
8 inputStream.read(result);
```

Listing 5.2: JavaScript Code Snippet for reading Linux file.

The browser running under SELinux sandbox can limit a process's access to files even if it runs with the root permissions. For example, the *init* process does not need to write to disk, and SELinux can ensure that it never will, even though it runs as root. Applying access control on files in Firefox browser is particularly necessary. We use the SELinux policies to control critical Linux files and allow restrictive access to file system through the browser extension. The restricted policies protect system functionalities that Linux handles using files. Consequently, policies that protect files can therefore also protect critical system resources and services such as drivers, sockets, character devices, block devices, and directories.

3. *Protecting Network Service.* For every web browser to use Internet, the network service is the most required and critical. However, browser extensions provide APIs that allows an extension to make a cross-domain requests, and this violates SOP. The extensions can bypass SOP to include external content from different domains which leads to an attack. Extensions can use opened network ports, network sockets, send information through port 25, perform redirection or permit access to external resources.

```
//Change Firefox preferences
var prefs = Components.classes['@mozilla.org/preferences-service
    ;1'].
getService(Components.interfaces.nsIPrefBranch);
var headers = prefs.getCharPref('cors.headers');
// Cross domain calls
var xhr = new XMLHttpRequest();
var url = 'http://attacker.com/malscript/hooks/';
function callOtherDomain() {
  if(xhr) {
    xhr.open('GET', url, true);
    xhr.onreadystatechange = handler;
    xhr.send();
  }  }
```

Listing 5.3: JavaScript Code Snippet for making cross-domain calls.

Listing 5.3 illustrates the browser extension code snippet for making cross-domain calls and violating SOP. In this extension, first we add cross-origin resource sharing(CORS) [114] `Access-Control-Allow-Origin & Access-Control-Allow-Method`

HTTP-headers to all responses before they are processed by the browser (Line 4). This can be done by exploiting `nsIPrefBrach` XPCOM interface used to change the Firefox preferences. The preferences setting allows us to bypass the SOP and then using `XMLHTTPRequest` method (Line 6), the extension can make cross-domain calls. For instance, an attacker can access information from the web page and then send this information to attacker domain (lines 7-12).

The sandFOX enabled browser can limit network access initiated by the browser components. Our policies allow only web access on *http* and *https*. The sandFOX denies access to critical ports such as port 21, 25, which an attacker can use to leak information from the web browser. In particular, with our proposed policies, the browser extensions can not initiate any direct or indirect request on critical ports.

5.2.4 Tuning OS

The first layer of security that we propose is tuning the operating system using SELinux policies. The SELinux policies provides highly restrictive environment to the applications. In the restrictive environment applications like web browser loses some functionalities and completely disallows to interact with OS resources, even *root* has limited access. We configure the OS using SELinux policies so that the browser can achieve its functionalities in restrictive environment. In particular, we define some system level policies to limit the OS resources accessed by the web browser. For example, a policy defines a domain `user_mozilla_t` and `admin_mozilla_t` for ordinary users and administration users respectively. In this way, *user* and *admin* would have different domains for web browsing that is protected from each other.

We define role-based access control policies in which a role is assigned to a user, which is an abstraction designed to make policy rules more concise. Roles are used to determine which domains can be used. For example, `user_mozilla_t` would be associated with role `user_t`. To complete the separation, we would create separate file types for each web browser domain and only allow the domains types "write" access to their respective file types. The result would be that the web browser runs in a different domain with different privileges.

We enforce access controls to a limited number of processes that are believed to be most likely to be the target of an attacker. The targeted processes run in their SELinux domain, known as a confined domain, which restricts access to networks that an attacker could exploit. If SELinux detects that a targeted process is trying to access resources outside the confined domain, it denies access to those resources and logs the denial. In particular, we apply policies on services that allows access to the remote location using network service. For example, *httpd, sendmail* and ftp service, and processes that run as root to perform tasks on behalf of users, such as *passwd*. These services, if not controlled may be risky to web browser. For example, FFSniff extension can send web page information to remote attacker using mail service on port 25. Enforcing our policies restrict such dangerous extensions in executing privacy leakage attacks.

5.2.5 Using SELinux Sandbox

SELinux allows administrators or users to lock down tightly untrusted applications in a sandbox where they use the OS resources in a restrictive environment. This is second layer of defense that we provide to secure OS resources from browser attacks. This sandbox restricts OS resources such as network and file system operations to avoid threats. In particular, the sandbox provides an isolated environment that can be used to protect a system while allowing it to run some untrusted binary. Our goal in this chapter is to provide isolated sandbox to let the Firefox browser execute untrusted code in a more secure manner. However, SELinux policies is not directly applicable to the browser. A browser running in a sandbox is very restricted, and can not read or write to any files and directories that are not explicitly allowed, and they also have no network access. The browser is pretty useless without these services.

This chapter rewrites the SELinux policies and present virtual cage (sandFOX) in which we can lock up Firefox browser. The browser running in sandFOX environment is able to achieve required functionalities but in an isolated environment. However, our policies allow Firefox to use OS resources in highly secured and restrictive environment. The resulting solution is both intelligent and elegant.

sandFOX Policies. The default SELinux sandbox is highly restrictive, and is irrelevant for Firefox browser. To create the policies for the Firefox browser, we customize some of SELinux policies.

1. **File System Policies.** Instead of allowing the application direct access to all directories in a file system, the sandFOX allows access to `tmp` and `home` directory. Each directory is assigned its own SELinux context, making it impossible for other application to access browser information and browser application to access files in other directories.

   $\$sandbox - M - H\ home\backslash\ -T\ tmp\backslash\ firefox$

2. **Network Policies.** Firefox is pretty useless without network access. Using `sandbox_web_t` allows for web browsing.

   $\$sandbox - t\ sandbox\_web\_t\ -i/home/anil/.mozilla - Xfirefox$

   The `-t` option in sandbox tells browser which SELinux context to use. The `-i /home/anil/.mozilla` tells the sandbox to copy the contents of `.mozilla` directory into the sandbox. This should not have any sensitive information in it such as stored passwords and cookies. The `-X` tells sandbox to launch an X sandbox, and the `firefox` is the command to run. After this command, the Firefox would no longer be able to connect to any other port except port 80, because `sandbox_web_t` is only allowed to connect to `http_port_t`.

3. **Process Policies.** SELinux provides a finer-grained level of control over processes in the Linux operating system. SELinux allow to define a security policy that provides granular permissions for all processes executed and loaded by a local user or root. We use these policies in our sandFOX environment to restrict browser in handling Linux processes. Our policies enforce access controls to a processes that are believed to be most likely to be the targets of an attack on the system. The targeted processes run in their SELinux domain, known as a confined domain, which restricts access to sensitive files such as *passwd* by an attacker process. If SELinux detects that a targeted process is trying to access resources outside the confined domain, it denies access to those resources and logs the denial.

Our proposed SELinux policies for Firefox browser restricts only specific services and run them in confined domains. Without our policy, a browser extension will run under the context of the user that started it. So if the extension has malicious intent accesses a applications that is running under the root user, the extension can do whatever it wants root has all-encompassing rights on every file. For example, the browser extension that listens to a network service is capable of sending information to the remote location can send critical files to attackers location. However, our policy restricts such extension in accessing critical files such as *passwd*. In particular, if an attack compromises network process that is running in network domain, the sandFOX policy prevents an attacker in accessing resources that are not in the network domain.

5.3 Evaluation

This section presents the evaluation of sandFOX in the context of threats possessed by the Firefox browser extensions. We also benchmark our sandFOX enabled browser on web page load latency and memory footprint criterias.

All experiments were carried out on a Intel Core i5 CPU 650 @ 3.20GHz with 4 GB of memory and a 250 GB serial ATA hard drive.The OS is Fedora 20, running the 64-bit version of Linux Kernel 3.11.10. We use $Firefox25$ version to evaluate our proposed policies. Section 5.3.1 evaluates the effectiveness of our proposed sandFOX using vulnerable and malicious browser extensions. Section 5.3.2 evaluates the performance overhead of sandFOX.

5.3.1 Security Analysis

To evaluate the sandFOX resilience to browser attacks, we execute several attacks on Firefox browser. We have developed several extension with a malicious functionality of accessing critical OS resources. Table 5.1 illustrates some example extension along with their functionalities. Our proposed sandbox and isolation policies make many attacks considerably more difficult and lessen the impact of the exploits. In particular, we exploit major OS resources to show how sandFOX is effectively preventing compromise of these resources from an attack. Our goal, when performing this experiment is only to verify that the browser does not allow a third-party extension to compromise OS resources. We ensure that if such attack happens in a browser, the sandFOX policies deny access to the OS resources and log an error.

Table 5.1: Extensions used in evaluating effectiveness of sandFOX.

Extension Name	**Resources Accessed**	**Description**
FFSniff	Network, Web page information	Access information from web page and send it over remote location on port 25.
Keylogger	Web page information, File system	This extension first access the information and create a file on host file system.
Password Stealer	Network, Web page information	Access information from web page and send it over remote location using network service.
FireFTP	Network	FireFTP is a cross-platform FTP/SFTP client for Mozilla Firefox which provides easy and intuitive access to FTP/SFTP servers.
Myprocess	Processes, File system	This extenstion first read file from host file system and then execute it as a background process.
Launcher	Process	Launcher can launch an arbitrary host process.
Facebook Like	Network, Web page information	Facebook Like accesses logged in user Facebook page information and send it some arbitrary location.
Tweet Me	Network, Web page information	Tweet Me can send any fake tweet or modifies the current tweet send by logged in twitter user.
BackProcess	Process	BackProcess can launch an arbitrary host process in background with user notice.

1. *File System Attacks.* In general, a web browser has an ability to access a host OS file system to steal or manipulate critical information. The sandFOX enabled browser reduces the severity of this class of attack significantly. Our sandFOX enforce policies on host file system so that it can access limited files and directories on the host file system and hence, there is less surface for an attacker to exploit critical files. We implemented *Password Stealer* extension to test the attack on host file system. Using password stealer, a user can read */etc/passwd* file if the extension runs without sandFOX browser. However, executing the same extension in sandFOX enabled browser disallows a user to read *passwd* file. The attack fails because our policies only allows user *home* and *tmp* directory to be accessible by the browser and extensions.

2. *Process Attacks.* The sandFOX policies prohibit an extension or web application to launch an arbitrary process on a host OS. In particular, the browser is enforcing sandFOX policies that restrict browser to execute and create a process, and

execute a shell script on a host OS. We develop a proof-of-concept extension that allows the browser to execute a process, and run a script on Linux *bash* shell. We find that the sandFOX enabled browser restricts browser to execute new processes and scripts. This evaluation provides assurance that sandFOX policies are effective in restricting processes to be invoked by the browser on a host OS.

3. *Network Attacks.* The network is the major service that a browser and extensions require. However, open network service (without restrictions) allows an extension to leak private information to outside world. Our proposed sandFOX enabled browser enforce policies that restrict browser in accessing all network services. Our policies restrict cross-domain request initiated by the extensions and web applications. We test our policies using known *FFSniff* malware analyzed by various researchers [56, 57, 115]. FFSniff sends the web page information to remote location on port 25. On executing *FFSniff* on the sandFOX enabled browser, we find that the browser is enforcing sandFOX policies, which denies access on port 25 by an extension. This evaluation provides assurance that sandFOX policies are effective in restricting network services used by browser and its components.

4. *Memory Attacks.* Buffer overflow and code execution is one class of vulnerabilities common to browsers and plugins. The sandFOX enabled browser reduces the severity of this class of attack significantly. Our sandFOX policies limit or isolate a compromised browser so that it can only interact with other browser components, and prevent interaction with the OS resources. Protecting the OS resources from an exploited browser is critical to providing security for the system and browser. The modern OS provides sandboxing environment using a variety of techniques. We use SELinux policies to restricts the access of OS resources. For example, if the attack results in malicious code execution inside the browser, the browser enforce sandFOX policies that restrict direct or indirect access to the OS resources.

5. *Web Application Attacks.* Our sandFOX policies do not provide protection for clients from web application bugs, which could result in attacks such as cross-site scripting or cross-site request forgery, clickjacking. Instead, our policies can prevent the malicious script from making access to the local file system as well as preventing network access outside of the sandFOX enabled the browser and cross-domain network access. These policies can prevent the effects of some types of

cross-site scripting attacks but does not prevent the vulnerable script from being exploited.

5.3.2 Benchmarking of sandFOX

To evaluate the performance of web browser running with our proposed sandFOX policies, we measure page load times and memory footprint. Our goal when performing these evaluations is only to verify that the browser does not introduce unreasonable delays that are noticeable by the user.

1. *Latency.* To measure the latency introduced by sandFOX browser, we compare the load times of a few common pages opened in sandFOX enabled browser with that of without sandFOX browser. We divide the web pages on the basis of content they load in web browser. We use five categories illustrated in Table 5.2. The web sites considered in this experiment are popular web sites ranked by *google.co.in*. The web websites consist of several components such as CSS, JavaScript, applets, flash videos. We measure the load time of each web site using Firebug [116] Firefox extension. Each page is loaded ten times in the browser, and the loading times are averaged. The latency is dependent on the various parameters such as system memory, CPU clock cycle, network load, and bandwidth.

Table 5.2: Averaged Load latencies for 50 web pages categories on the basis on content. The latency illustrates the browser load with and without sandFOX environment.

Web Page Content	**Web Sites Tested**	**Latency With sandFOX (Seconds)**	**Latency Without sandFOX (Seconds)**
Flash Videos	50	3.55	2.86
Scripts	50	2.44	2.54
Frames	50	2.71	2.10
CSS	50	1.59	1.52
Only HTML	50	1.35	1.35

Our proposed sandFOX policies incur a negligible performance overhead on the browser. We do not notice any major slowdowns in the browsing experience during the evaluation. We observe that most of the pages tested, we see little difference in the page load times; however, for the web pages without frames and flash videos loads noticeable faster than with frames and flash videos. We observe some overhead in sandFOX enabled browser. However, despite these overheads, the performance of the browser was not noticeably slower during normal web browsing, even with JavaScript heavy web pages, such as Google maps and street views.

2. *Memory Usage.* To measure the memory footprint of the browser running with sandFOX policies, we load the browser and navigate to a single web page with no plug-ins and extensions. We use the Gnome system monitor to measure memory usage. Table 5.3 displays the comparable results when executing browser with and without our proposed sandFOX.

   We observed that our proposed sandFOX policies do not take much memory in all categories of tested web pages. Loading normal web page such as *google.co.in* in Firefox consumes around $108.8MB$ of memory without a sandFOX environment, whereas Firefox consumes around $109.8MB$ of memory when a web page is loaded with sandFOX environment. These observations clearly suggests that our sandFOX policies do not consume more memory.

Table 5.3: Memory Footprint of Firefox browser with and without sandFOX.

Web Page Content	**Web Sites Tested**	**Memory With sandFOX (MB)**	**Memory Without sandFOX (MB)**
Flash Videos	50	154.9	153.1
Scripts	50	116.5	112.4
Frames	50	113.2	113.1
CSS	50	110.5	110.5
Only HTML	50	109.8	108.8

Limitations. Our sandFOX policies do not provide protection for clients from web application bugs, which could result in attacks such as cross-site scripting or cross-site request forgery, clickjacking. Instead, our policies can prevent the malicious script from making access to the local file system as well as preventing network access outside of the sandFOX enabled the browser and cross-domain network access. These policies can prevent the effects of some types of cross-site scripting attacks but does not prevent the vulnerable script from being exploited. Furthermore, the SELinux sandbox is a part of Linux based operating system and hence our proposed sandFOX environment does not support Windows operating system.

5.4 Summary

This chapter presented sandFOX, a secure sandbox and isolated environment for Firefox browser. Our sandFOX policies restrict an attacker in executing critical attacks on operating system resources such as file system, network, process. Our proposed solution does not modify existing Firefox browser and its components. Instead, it uses Security-Enhanced Linux (SELinux) to build a sandbox that helps in reducing potential damage from extension-based attacks on OS resources. Our proposed policies let a browser application such as extensions and plug-in to access limited OS resources in the restrictive environment, and hence do not affect the functionalities and user browsing experience. We show the practicality of sandFOX in a range of settings, we compute the effectiveness of sandFOX for various browser attacks. We also show that sandFOX enabled browser imposes low overhead on loading web pages and utilizes negligible memory when running with sandbox environment.

Chapter 6

Detection of Click-Hijacking in Browser

Click Hijacking (Clickjacking) is emerging as a potential web-based threat on the Internet. The prime objective of clickjacking is to steal information on user clicks. It can be achieved by tricking the victim into clicking an element that is barely visible or completely hidden. By stealing the victim's clicks, an attacker could entice the victim to perform an unintended action from which the attacker can benefit. These actions include online money transactions, sharing malicious website links, initiate social networking links, etc.

This chapter presents an anatomy of advanced clickjacking attacks not yet reported in the literature. We demonstrate that current defense techniques are ineffective to deal with these sophisticated clickjacking attacks. Furthermore, these attacks are browser agnostics. Subsequently, we develop a novel detection method for such attacks based on the behavior (response) of a website active content against the user clicks (request). In our experiments, we found that our method can detect advanced Scalable Vector Graphics (SVG)-based attacks where most of the contemporary tools fail. We explore and utilize various common and distinguishing characteristics of malicious and legitimate web pages to build a behavioral model based on Finite State Automaton (FSA). We evaluate our proposal with a sample set of 78000 web pages from various sources, and 1000 web pages involving clickjacking. Our results demonstrate that proposed solution enjoys the good accuracy and a negligible percentage of false positives of 0.28% and zero false negatives in distinguishing clickjacking and legitimate websites.

The contributions of this work are manifold:

1. ***Novel Clickjacking Attacks.*** We presents an anatomy of novel advanced clickjacking attacks using visual effects caused by SVG images and filters. Furthermore, we present some alternative methods for achieving properties of clickjacking attacks. Our proposed attacks defeat existing clickjacking attack detection tools.

2. ***Unified Behavioral Model.*** We present a behavioral clickjacking detection approach based on the behavior (response) of websites against the user clicks (request). We describe our model using the notion of Finite State Automaton (FSA). The FSA is constructed for various request and response pairs to represent various states belong to either clickjacking attack or legitimate scenario.

3. ***Implementation Framework.*** We present a prototype model consisting of four modules: Query String Formation (QSF) module; Signature Generation module; C-CHECK parser module; and Click Inspector module.

4. ***Experimental validation.*** We present a detailed experimental evaluation of our system on real-world web pages. In our evaluation, we implemented different variants of clickjacking attacks, which includes basic as well as new advanced attacks. We tested our proposed solution on dataset of 78000 web pages taken from different sources.

6.1 Problem

Clickjacking attack is an attack against users of web application in which a malicious page is created by an attacker. The infected web page is designed in such a way that it tricks a user into clicking on a page element. This click targets an action against victim page that is hidden from the user. Sometimes, the clicked element overlaps or stacks with an element on the victim page. With this technique, an attacker can “steal” user click to target any website including authenticated website to perform malicious activities. For example, an attacker can target social networking website, such as *Facebook* to trigger *"like"* button without user's notice, *Twitter* by posting unwanted messages, unsecure banking websites by executing an online money transaction.

Clickjacking attack was first addressed by Robert Hansen and Jeremiah Grossman in a talk at OWASP AppSec 2008 [117]. Some proof-of-concept for clickjacking examples that have been made public were posted by security researchers, and by hackers [63, 117–119]. In particular, all these attacks have focused on clickjacking attacks that are caused from transparent *iframes* and overlapped web elements. The Hyper Text Markup Language (HTML) *iframe* element represents a nested browsing context, effectively embedding another HTML page into the current page. There are other advanced methods that can be used to steal user clicks from the browser. This chapter considers not only iframe based clickjacking attacks, but also new methods of achieving clickjacking attacks. For example, we consider Event bubbling, SVG-based attacks that achieve transparency and overlapping using SVG filters.

Clickjacking attacks are fundamentally attacks on limitations of human perception. In other words, any user interaction with web page elements should succeed only if the user perceived, understood and made a conscious decision to take a particular action. In this work, we develop a novel approach to detect clickjacking attack based on the behavior (response) of websites against the user clicks (request). The key idea to observe the essence of clickjacking attack is that the web page opened in a browser must have one or more suspicious properties such an overlapping, transparency, stacking, DIV nesting of elements. In particular, we propose several features to gather clickjacking symptoms, which cannot be identified by the human eyes or perceived by the human mind, from a website.

We examined a set of common and distinguishing functionalities that legitimate, and malicious web pages can perform. We then identify static as well as dynamic features from the web pages. The static features are taken when a page is downloaded or opened in the web browser window. The dynamic features are taken on the basis of the observed response of user clicks. These web page features are comprehensive and can be used to derive a behavior model for a website and classify if it is susceptible to clickjacking.

6.2 Clickjacking Attacks

In this section, we briefly present a detailed description of various existing clickjacking attack classes and the motivation behind our clickjacking attack detection approach. Table 6.1 illustrates the broad categories of attacks considered in this chapter. These attacks are further explained in Section 6.2.1 and Section 6.2.2.

Table 6.1: Illustration of existing and newly identified clickjacking attacks.

Attack ID	**Attack Type**	**Source**
a_1	Click Stealing through Visual Perception	Existing attack [65]
a_2	Click Stealing through Keystrokes	Existing attack [120]
a_3	Click Stealing through Hidden Pointer	Existing attack [65, 121]
a_4	Click Stealing through CSS (Stacking Elements)	Existing attack [69]
a_5	Click Stealing through CSS (Pointer-event)	Existing attack [69]
a_6	Click Stealing through Element Movement	Existing attack [65, 69]
a_7	Violating Display Integrity Using SVG Filters	Newly identified attack
a_8	Clickjacking with SVG Clipping and Masking	Newly identified attack
a_9	Modifying User Interface using SVG filters	Newly identified attack
a_{10}	Enforcing `Pointer-event` Property through SVG	Newly identified attack
a_{11}	Enforcing Script Injection using SVG Filters	Newly identified attack

6.2.1 Existing Clickjacking Attacks

We classify existing attack classes according to the behavior and appearance of the browser window. The attack classes are as follows.

Click Stealing through Visual Perception (a_1)**.** A visual perception is how a user sees a web page, before and after clicking an element. An attacker can compromise a user's visual perception to execute clickjacking attacks. Using frames/iframes, an attacker can embed a cross-domain web page into the current web page. An attacker entices the user to click on the malicious page, which has hidden target elements embedded in iframe placed underneath a malicious page. When the user clicks on an element on the upper page, the click will be routed to an element present on the underneath page [65].

For example, let us consider the scenario illustrated in Figure 6.1. A Facebook page is embedded into a malicious web page. A target element *"Click Here"* on the malicious page is positioned exactly above the Facebook *"like"* button in such way that user can not notice the presence of the latter. Once the user performs a click on a button labeled *"Click Here"*, the click will also route to underneath page and also trigger Facebook *"like"* button without user's notice. Using this attack scenario, an attacker can like any Facebook page on user's behalf.

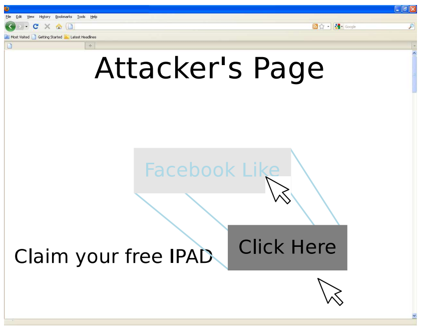


FIGURE 6.1: Illustration of clickjacking attack using frame overlays. In this attack, a Facebook page is embed on a transparent iframe. Facebook *"like"* button is shown with light colour in a figure to represent invisibility.

Click Stealing through Keystrokes (a_2)**.** This attack is called *Strokejacking*, as in this case the keystrokes can be, in addition to clicks, also hijacked. This can be achieved using social engineering tricks to capture keyboard events from text box placed on attacker page, and at the same time applying these keystrokes to the underneath text box on the victim page. In this way, an attacker can capture user input keystrokes. An attacker can, then, inject a malicious script using user input keystrokes [120].

Click Stealing through Mouse Pointer Hiding (a_3)**.** In this case, the attacker creates, and re-positions a fake cursor while hiding the original cursor. The fake cursor is positioned in such a way, that when user points to a fake cursor on a link displayed on the attacker page, the original cursor will point to a link on the victim page, which an attacker wants to trigger [65, 121].

Click Stealing through CSS (Stacking Elements) (a_4)**.** Most browsers support HTML/CSS styling attributes that allow an attacker to visually hide the target element through an overlapping element. When a user clicks on the upper layer of overlapped element, a click is also routed to lower layer element. For example, an attacker can make the target element transparent by wrapping it in a `DIV` container [122] with a CSS opacity value set to zero. This creates a stack of overlapped elements above the target element by using `z-index` [123] property. When the victim user clicks on an upper element, this click is actually routed to the lower element. In this way, an attacker can trigger a malicious link placed with the lower element.

Click Stealing through CSS (Pointer-event) (a_5)**.** In this attack class, an attacker bypasses the click event from an upper element and then routes the same click event to lower invisible target element positioned underneath it. The element can be made unclickable by setting the property `pointer-events:none` [123]. A victim's click would then fall on the decoy and land on the (invisible) target element.

Click Stealing through Element Movement (a_6)**.** In "Frame Overlay" class, an element is positioned exactly above the framed page element. This can be circumvented with element randomization [69], and thus a new class of attack is required to successfully deploy frame overlay attack. In this class, a hidden element will move along with the mouse movement. So wherever the user clicks, it triggers an event that may route to the victim page.

Motivation. Clickjacking is a serious threat in Internet domain [63, 117–119]. Our work is motivated by the fact that new variants of clickjacking attacks are not addressed in the literature. The previous published work has only focused on detecting the clickjacking attacks caused by hidden iframes or hidden mouse cursor. In this chapter, we present new attack variants using SVG filters, for which detection technique has not been developed. These attacks are an alternative way of achieving clickjacking without getting caught by existing defense mechanism. These limitations in existing clickjacking attack detection techniques motivate us to develop a new detection approach. Our novel detection approach can efficiently and effectively detect advanced clickjacking attacks in websites.

6.2.2 New SVG-based Clickjacking Attacks

In the attacks described in Section 6.2.1, the attacker has mainly adopted CSS style for hiding and overlapping iframes and other web page elements to hijack user clicks. This section presents some novel methods to create clickjacking attacks (a_7-a_{11}) that are beyond using iframes and CSS styles. We will discuss some novel attacks based on visual effects produced using SVG [124]. To demonstrate our attack, we use an attack scenario illustrated in a Figure 6.2. The attack scenario that we have taken is based on a fake pop-up window, which entices the user to perform some set of clicks (or events), and at the same time these clicks are hijacked by an attacker.

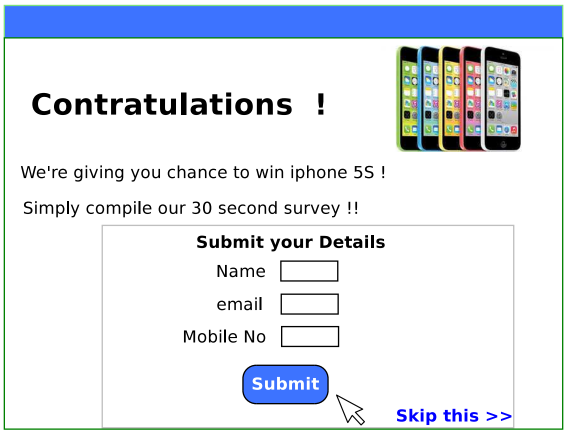

Contratulations !
We're giving you chance to win iphone 5S !
Simply compile our 30 second survey !!
Submit your Details
Name
email
Mobile No
Submit
Skip this >>

FIGURE 6.2: Attack Scenario for SVG based clickjacking Attacks.

Violating Display Integrity using SVG Filters (a_7)**.** SVG filters provide a way to make an object transparent through `<opacity-value>` attribute whose value ranges from 0.0 to 1.0. The object with opacity-value set to unity makes transparent and with the value set to zero makes a fully opaque object. Another way to make SVG drawing objects as well HTML content transparent is using CSS properties. CSS provides `opacity` attribute value, ranging from 0 to 1 to make HTML object transparent.

We find the new alternative ways to make HTML elements transparent. This can be achieved by applying SVG filter primitives with special values or by applying a series of filter effects. This technique can be used to bypass current clickjacking detection methods where only CSS `opacity` value of of HTML object is checked, i.e., `opacity=0`.

SVG filters are like image processing filters used to apply effects to SVG images like a blur, dilation, erosion, etc. A filter effect consists of a series of graphics operations that are applied to a given source to produce a modified graphical result. The result of a filter effect is rendered to the target browser instead of the original source graphic.

Table 6.2 illustrates the list of filter primitives that can be used to produce transparent objects when applied on HTML document or SVG objects. The list contains set of filter primitives. Any combination of those filter primitive result into a transparent object. Table 6.2 consists of four columns, filter name, value, image, and iframe. The Filter name is a filter primitive to be used in filter effect; value is a set of special attribute-value pair that makes object fully transparent when the filter effect is applied to it. Image and iframe are the objects on which filter effects are applied to test transparency and result is quoted as *Yes* or *No*.

Table 6.2: Transparency alternatives using SVG filters.

Filter Name	**Filter Effect**	**Image**	**iframe**
feFlood	Flood-opacity=0	Yes	No
feTurbulance	baseFrequency <= 0.0009	Yes	No
feGaussianBlur	stdDeviation >= 200	Yes	Yes
feConvolveMatrix	kernelMatrix=0 preserveAlpha="false"	Yes	Yes
feColorMatrix	Type=luminaceToAlpha	Yes	No
	Style=color-interpolation-Filters:sRGB		
	Type=matrix values=0	Yes	Yes
feComponentTransfer	<feFuncA Type=Table TableValues=0 / >	Yes	Yes
	<feFuncA Type=Discrete TableValues=0 / >	Yes	Yes
	<feFuncA Type=Linear slope=0 intercept=0 / >	Yes	Yes
	<feFuncA Type=Gamma Offset=0 exponent=0 / >	Yes	Yes

Understanding the Attack. Figure 6.3 depicts a frame in our attack, which uses alternative transparency technique. In this attack scenario, a malicious page conducts a survey of current government having two input buttons. To conduct this survey, a user must click on *Yes* or *No* button and then user submits the survey. This malicious page may also contain a hidden and transparent Facebook *"like"* button placed exactly underneath the two buttons. To perform clickjacking attack, an attacker entices a user to click either *Yes* or *No* button on the page.

When the user clicks on either of the buttons, a malicious page will steal this click and transfer it to Facebook *"like"* button placed underneath the clicked button. The complete attack steps are illustrated in Figure 6.3.

Embedding link into SVG image (a_8)**.** The SVG `<a>` element can be used to embed links into SVG images. SVG links work just like HTML links. A URL link can be embedded into any image or shape such as circle, rectangle. To achieve this feature, the attacker puts the SVG shape that is to be used as link between the `<a>` and `</a>` tags. A clickjacking page may contain such link to bypass detection technique that takes HTML `<a>` tag as a clickable element for the analysis. An example code snippet for created SVG image as a link is illustrated in Listing 6.1. Here, a Facebook *"like"* button link is embedded into the SVG image.

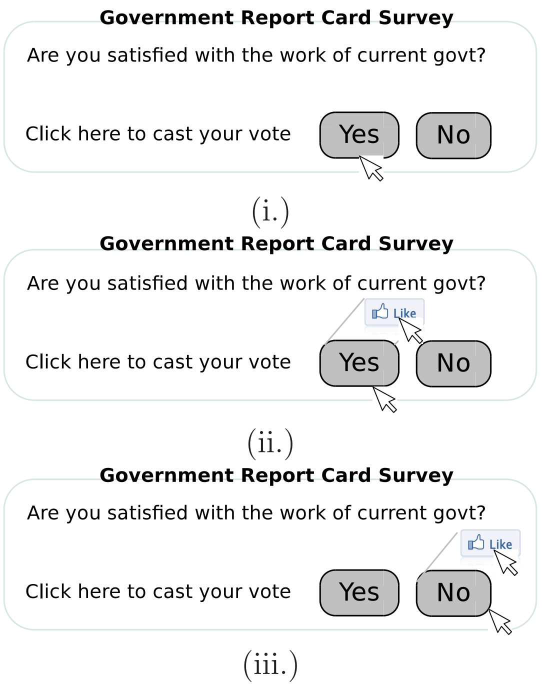

Government Report Card Survey
Are you satisfied with the work of current govt?
Click here to cast your vote
Yes
No
(i.)
Government Report Card Survey
Are you satisfied with the work of current govt?
Like
Click here to cast your vote
Yes
No
(ii.)
Government Report Card Survey
Are you satisfied with the work of current govt?
Like
Click here to cast your vote
Yes
No
(iii.)

FIGURE 6.3: Workflow for our advanced transparency attack using SVG filters. At first, (i) displays the normal working when user clicks *"Yes"* button, (ii) When the user clicks on *Yes* button a Facebook *"like"* button is clicked, (iii) When the user clicks on *"No"* button, still Facebook *"like"* button is clicked.

```
<svg>
<a xlink:href="http://www.facebook.com/plugins/like.php?href=https%3A
    %2F%2Fdevelopers.facebook.com%2Fdocs%2Fplugins%2F&width&layout=
    standard&action=like&show_faces=true&share=true" &height=80 target
    ="_top">
// Embed Facebook like URL into SVG image
</a> </svg>
```

LISTING 6.1: SVG code to embed Facebook *"like"* URL into an SVG image.

Understanding the Attack. SVG `<a>` elements greatly simplify clickjacking attacks as they can be applied on cross domain content like iframe. Let us consider the attack scenario illustrated in Figure 6.1. In this scenario, an iframe is placed underneath *"Click Here"* button. These elements are rendered in such a way that *"Click Here"* button is positioned exactly above Facebook *"like"* button opened in iframe. An attacker now entices the victim user to click on *"Click Here"* button to achieve clickjacking.

The current scenario page can be developed using basic CSS and HTML features. Here, we develop this page using SVG clipping [125, 126]. Listing 6.2, shows the code in which Facebook *"like"* URL is embedded into SVG image. This SVG image is placed below *"Click Here"* button. Furthermore, this SVG image can be embedded into website using iframe (shown in Listing 2).

```
<iframe src="facebook.svg" width="200" height="200" >
```

Listing 6.2: SVG image embed using iframe.

In this attack scenario, an attacker have used SVG image, which entices user to initiate click on Facebook *"like"* button. Since clickjacking prevention techniques check for invisibility or transparency in a website, this attack involving SVG images is difficult to detect as they can use any shape that is visible to user.

Modifying User Interface using SVG Filters (a_9)**.** SVG filters are used to give visual effects to SVG images. We observe that certain SVG filters when applied on object changes it spatially. It may either increase or decrease the size of the object. Here object refers to either SVG image object or HTML element as SVG effects can be applied on either. This property is very crucial in terms of clickjacking attack in case if newly increased area is still a part of regular user interface but does not responds to mouse events (user clicks). This increased and inactive area can be used to hide malicious link. When user clicks on increased area of element, the click is routed to malicious link.

We applied SVG effect on *"submit"* button of Figure 6.2. The shadow rectangle is a newly formed object as a result of filter effects that modify web User Interface (UI) as shown in Figure 6.4. From a user's perspective, the *"submit"* button object with shadow effect is a single object but as it is produced by filter effect, it may or may not be a single object, which solely depends on filter effect.

We have observed two scenarios with SVG filter effect: (i) in the first scenario, the shadow portion of button does not respond to user clicks, i.e., mouse events but still is part of the object; (ii) in the second scenario, it responds to mouse events, i.e., the dummy shadow button works like original button. This technique is used to hide malicious links on the dummy object created by filter effect. This can be made more sophisticated by switching control between two objects (original and dummy).

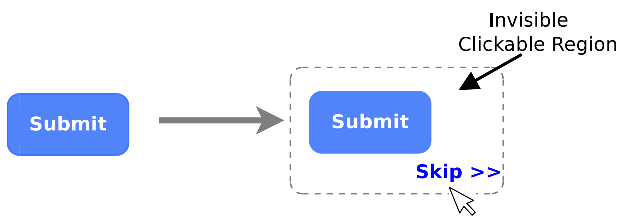


FIGURE 6.4: SVG filter effects modifies web User Interface.

Understanding the Attack. Figure 6.2 depicts an attack scenario in which a fake pop-up is being displayed to user. An attacker wants the user to fill all the details and then click on *"submit"* button or click on *"skip this"* link. Using this scenario and SVG filter effects, we have created two clickjacking attacks, which steal user click when user click *"submit"* button or even when user does not click *"submit"* button.

Figure 6.4 illustrates the more expanded version of our first attack scenario shown in Figure 6.2. In this scenario, the attack is successful only when user clicks desired portion of *"submit"* button. Here, we applied *morphological* filters dilation effect on *"submit"* button, which increases size of object spatially and modifies current web user interface. The extra space added by filter dilation extends all visual boundaries of the original object except response to user events. Hence, the user can not differentiate the object as two different entities. The region enclosed in dotted rectangle is original object whereas the solid line rectangle is an enlarged version. Underneath this extra space, we place a Facebook *"like"* button, which is not visible to the user.

When user clicks on *"submit"* button, it initiates a Facebook *"like"* button placed underneath enlarged portion of *"submit"* button. This way an attacker can initiate clickjacking without completely overlapping an element beneath *"submit"* button. This attack is not detectable by the earlier detection techniques as attacker page does not use any CSS property to either hide or overlap elements.

Enforcing `Pointer-event` Property through SVG (a_{10}). `Pointer-event` property allows control of the behavior of graphic element before it becomes the target of mouse events. A victim's click would then fall through the decoy and land on the (invisible) target element.

CSS defines eleven `pointer-events` attributes applicable to HTML and SVG objects, out of which only two are applicable to regular HTML content and other are for SVG objects. The `pointer-events` properties applied on HTML object includes `auto, none`.

The value `auto` is referred to when the pointer-event value is not specified and implies a regular behavior of clickable element on mouse events. Pointer-event property `none` is used to disable the target element from responding to mouse events like click, state and cursor actions, etc. In past, the authors have presented the use of `pointer-events` applied on HTML elements to invoke clickjacking attacks. Here, we present new methods of using `pointer-events` by applying its properties on SVG graphics objects.

SVG graphics objects can also use eight `pointer-events` properties. For example, SVG defines each object with `pointer-events` property as `fill` and `stroke` where fill refers to interior part of object and stroke refers to edges. Pointer events on the SVG objects can be handled separately using fill or stroke property. For example, if we create SVG based button through rectangle, the button can only be the target of a mouse event when the pointer is over the interior (i.e., fill) of the button. In case of stroke property, the button can only be the target of a mouse event when the pointer is over the perimeter.

Figure 6.5 illustrates the fill and stroke property applied on SVG based *"submit"* button. We have tested the ways to make SVG objects either partially or fully transparent to facilitate clickjacking attacks. We have extended this approach of clickjacking attack technique where attacker have many options to disable clickable graphics element to facilitate clickjacking attack through SVG. Attackers need to use SVG based web pages in order to steal user clicks. The attack technique bypasses traditional clickjacking defense technique where pointer-event attribute is statically analyzed against value `none` and hence, the new ways of disabling clickable elements using SVG effective clickjacking attack.

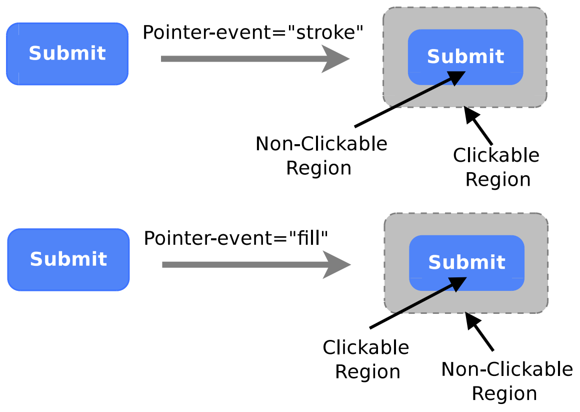


FIGURE 6.5: Illustration of the fill and stroke property applied on SVG based *"submit"* button.

Understanding the Attack. Attackers use pointer-events properties on SVG based objects to form clickjacking attacks. Figure 6.2 shows the attack scenario in which attacker expects user to click on *"skip this"* link but the click is still transferred to *"submit"* button. The attackers page includes two layers created using nested `DIV` tag one overlapped on another. A top layer consists of *"skip this"* link while inner layer consists of *"submit"* button.

The specified clickable (i.e., *"skip this"*) link is disabled by using pointer events as discussed above. When user clicks on this link the click will actually respond to its descendant inner layer, i.e., *"submit"* button. In this way attacker can steal user click using pointer-event property of CSS.

6.3 Proposed Clickjacking Detection

Our proposed approach is shown in Figure 6.6. We generate signatures of known clickjacking attacks using 1000 web pages susceptible to clickjacking attack. The pages are based on knowledge gained from the literature, basic clickjacking attacks described in the literature, attack classes describe in Section 6.2.1, and newly identified attack techniques describe in Section 6.2.2. We improve the state of the art in clickjacking attack by adding new attack classes to attack signature. Our dataset of 1000 legitimate pages consists of popular web pages that do not contain any advertisement, pop-ups, hidden iframes, and transparent elements.

We explore and utilize various common and distinguishing characteristics (features) from the dataset of known attack, and legitimate web pages. These characteristics are, then, used to build a behavioral model based on Finite State Automaton (FSA). We generate three alerts for a web page, legitimate, clickjacking warning and clickjacking attack alert. We use several heuristics to assist a test to decide, based on their behavior, whether the web page is *clickjacking warning* or *Clickjacking attack.* We define three alerts as follows.

1. **Clickjacking Attack Alert.** If the sequence of states (representing behavior) matches with the attack signature, it is labeled as clickjacking attack. Labeling a web page as *malicious* indicates we identified harmful behavior such as clicking

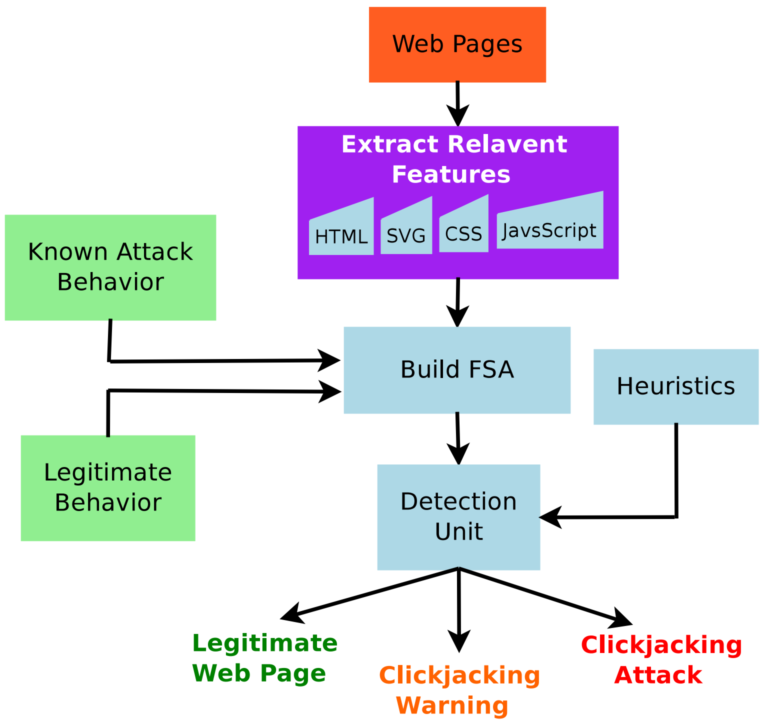


FIGURE 6.6: Proposed Approach.

of unknown source lead to cross-domain communication, execution of arbitrary hidden scripts, redirection of hidden mouse cursor etc.

2. **Clickjacking Warning Alert.** Our model also generates clickjacking warning, which indicates the presence of potentially harmful actions or exposing the user to new risks, but these risks may or may not represent clickjacking actions.

3. **Legitimate Behavior.** If we do not find any suspicious behavior, we label the web page as *legitimate*.

The experimental evaluation shows that our approach is feasible in practice. Also, our solution enjoys good accuracy and a negligible percentage of false positives (0.2%) in distinguishing clickjacking and legitimate websites. Moreover, our approach can detect novel advanced SVG-based attacks that many contemporary tools currently fail to detect.

6.3.1 Extracting Relevant Clickjacking Features

We have discussed sophisticated clickjacking attacks in Section 6.2.2 that are difficult to detect and analyze using existing approaches. Thus, we need a new robust approach to detect advance clickjacking attacks. Furthermore, new approach must handle accurately the dynamic features extracted from a web page, and should not require reconfiguration when new HTML tag is exploited for Click hijacking. Our approach relies on comprehensive dynamic and static analysis of the web pages.

Our model extracts only relevant features from a web page opened in a browser. A web page consists of HTML tags with their attributes and values associated with them. In particular, our relevant features consists of the values assigned to Relevant HTML Tags (RHT), such as, `iframes`, `frame`, `div`, `span`, `a`, `input`, `form`, `p`, `button`, `img`, `SVG`, and other clickable HTML elements for discovering the symptoms of clickjacking attack in a website. In addition to that, we are interested in text or objects that generate a click event when clicked.

To measure the significance of a feature in detecting clickjacking, we use dataset of 1000 web pages susceptible to known clickjacking attack and 1000 legitimate web pages. We perform an analysis to differentiate the feature values extracted from these two datasets. After this analysis, we come out with thirteen relevant features. In particular, the relevant features extracted from a web page characterizes the normal or hijacked click event (e.g., the instantiation of an hidden and overlapped link, the redirection of hidden mouse cursor, or the activation of malicious script with hijacked click, etc.) occurring during the interpretation of JavaScript and HTML code of a web page. In the following, we describe the relevant features used by our model.

Features we have use in our proposed approach can be categorised as:

1. Visual context of the web page.
2. Overlays in the web page.
3. Mouse pointer based features.
4. HTML and JavaScript based features.
5. Domain and Redirection Features.

In following subsections, we shall be discussing each categories in detail.

Visual context of the web page We include five features that characterize this kind of activity.

Feature 1: *Visibility of the web page elements.* We record the visibility of all HTML tags and elements present in a web page. The feature value can be obtained by checking CSS `visibility` property that is set to `hidden`.

Feature 2: *Opacity of the web page elements.* We record the `opacity` value of every page elements present on a web page. A web page can use CSS `opacity` value to `0` to hide page elements. Moreover, smart attacker sometimes partially hide the elements to avoid detection. This can be achieved by setting `opacity` value in range of `.2` to `.1`. Aim is that an element is barely visible and not visually perceptible by the user.

Feature 3: *Clipping with SVG.* We record the SVG tags along with their values used for clipping a region of the web page. The clickjacking page may be created by joining the clipped region of web pages that are taken from either same or different domain (discussed in Section 6.2.2, Attack a_8). In contrast, legitimate website do not contain the clipped elements from different domains. We identify the `clippath` [124] to record the clipping region in a web page.

Feature 4: *Manipulation of UI Elements.* We record parameters involved in the UI manipulation. This can be done by increasing the clickable area and hiding the link below increased clickable region (discussed in Section 6.2.2, Attack a_9).

Feature 5: *Moving web page elements.* We record the web page elements that move with a mouse cursor. This may cause the timing attacks. For example, an attacker could move the target element (via CSS position properties) on top of a decoy button shortly after the victim hovers the cursor over the decoy, in anticipation of the click. We identify the moving elements in a web page with two step as follow: (i) first we record the coordinates of all the clickable elements present on the web page; (ii) then, we find the elements whose position changes after mouse movement and user click.

Overlays features in Web Pages We extract a feature that is indicative of the overlay and overlapping web page elements.

Feature 6: *overlapping.* The overlapped elements can be used by an attacker to confuse victim user, and circumvent detection tool. Some detection tools check only hidden/-transparent properties as their primary features in detecting clickjacking attack. But an attacker can use overlays to overlap web page elements underneath other elements. We record the overlapped elements that are clickable.

Mouse Pointer based features We extract the following feature to record the cursor and pointer characteristics.
Feature 7: *Hidden mouse pointer.* We monitor the CSS `cursor:none` property after loading of web page to discover a hidden mouse cursor. In addition, we also monitor the APIs used in JavaScript code of web page that programmatically hide genuine cursor and draw a fake cursor on a web page. Another variant of cursor manipulation involves the duplicating the cursor. The attacker takes following two steps to execute clickjacking attack:

1. First, the attacker does not hide the genuine cursor; instead, he positions a transparent cursor look alike image on top of a genuine cursor, which a victim user does not notice.
2. In second step, the attacker draws a fake cursor on a web page, which when points to the button, the genuine cursor points to target element.

We record this attack activity by monitoring the JavaScript code that defines the image movement with the mouse movement.

HTML and JavaScript based features A click might result in an execution of JavaScript code embedded into a web page. Such scripts can result in clickjacking and/or serious attacks such as XSS attacks [108]. We include four features that characterize HTML and JavaScript properties.
Feature 8: *Hidden links with Pointer-events.* We monitor CSS `pointer-events` property to explore visually hidden target elements (discussed in sections 6.2.1 and 6.2.2, attacks a_5 and a_{10}). We extract this feature in a 2-step fashion: (i) first, we extract all the overlapped elements (fully or partially) from web page through x and y coordinates of `x-axis`, `y-axis` and checking CSS `z-index` value [123]; (ii) in next step, we check if CSS `pointer-event` property is set to `auto` and `none` value.
Feature 9: *Event bubbling and capturing.* We monitor the web page elements that are nested within each other. For example, `DIV` tags can be nested to initiate event bubbling and event capturing [127] on target element. An attacker can make target element

transparent by wrapping it in one of nested `DIV` container by setting CSS opacity value to zero, and keeping other `DIV` contents visible. In this, the handler of the parent (top `DIV`) works even if the child (nested `DIV`) is clicked and vice versa.

A hidden link wrapped into nested `DIV` may send some forge request to server leading to CSRF attack [67]. This new set of clickjacking attack is called *bubblejacking* attack. We record this activity by collecting information on nesting of `DIV` tags. In addition, we also record parent and child `DIV` tags from the nested `DIV` tags.

Feature 10: *Stacking elements with z-index.* Every Browser support HTML/CSS styling attributes that not only allows an attacker to visually hide the target element, but also allows it to route mouse events to it. For example, an attacker can make the target element transparent by wrapping it in a DIV container, and set CSS opacity value to zero. It then creates a stacking of overlapped elements under the target element by using a lower CSS z-index [123], and lure victim user to click on stacked elements. When victim user clicks on an upper element, the click also routes to the lower element(s), which may initiate hidden malicious event or click on link hidden underneath the lower elements.

Feature 11: *Script injection with SVG.* SVG tags allows an alternative way to inject script into website (discussed in Section 6.2.2). In our analysis, we record all these alternative script tags.

Domain and Redirection Sometimes, an attacker master page is not vulnerable to clickjacking attack but a page may contain a link that redirects master page to new page in (same or different domain), which indeed is vulnerable to XSS [108], phishing [128], CSRF [67], etc. For example, open redirects found on attacker master page is liable to be exploited by phishers to create a link to their site. We extract the following two features related to the page redirection.

Feature 12: *Redirection to same domain.* This feature indicates that on applying click event on a web page, a page is redirected to another page in the same domain. The new page there has no hyperlink to visit to the previous suspected domain.

Feature 13: *Redirection to other domain.* This feature indicates that on applying click event on a web page, a page is redirected to new page in different domain (or third party domain). The new page contains different SSL signature attributes compared to the previous page.

TABLE 6.3: List of behavior expressed from extracted relevant features.

Notation	**Features**	**Description**
ϕ_1	Features 1 & 2	Describes the visibility of web page elements.
ϕ_2	Features 6	Check overlapping of elements on a web page.
ϕ_3	Feature 7	Check if a web page contains hidden/duplicate mouse cursor.
ϕ_4	Feature 8	Check if elements on web page is using `pointer-event` property.
ϕ_5	Features 4 & 5	Check if any modification in user interface on moving mouse cursor or clicking.
ϕ_6	Feature 3	Check for clipped web page elements incorporate by SVG filters.
ϕ_7	Feature 11	Check if `<script>` tag (including alternative SVG script tags) is used with any hidden element.
ϕ_8	Features 9 & 10	Check for nested and stacked web page elements.
ϕ_9	Features 12 & 13	Check if clickable element on click redirects page to same or other domain.

Discussion. We use the thirteen features that are introduced in this section to characterize the properties of a web page. This gives us a comprehensive picture of the clickjacking behavior exhibited in a web page. We observe that our features can be classified into two categories: (i) static features, and (ii) dynamic features.

The values for static features can be obtained by analyzing the source code of the web page. These include following features.

1. The features from visual context category that characterizing the *visibility*, *opacity* of a website elements and clipped content of a web page (features 1, 2, and 3).
2. One feature from the mouse pointer category, which characterize the mouse cursor on a website (Feature 7).
3. The features from HTML and JavaScript based category (features 8, 9, 10, and 11).

For dynamic features whose values are obtained after a web page is rendered in browser window. Once the page is opened in the browser window, based on the position and behavior of the rendered elements, the values of dynamic features are obtained. These include following features.

1. Two features from visual context category features that characterizes the movement and user interface manipulation in web sites (features 4 and 5).
2. The overlay feature (Feature 6) that characterizes the overlapping of elements.
3. The domain and redirection features (features 12 and 13).

In our model, we define nine behavioral properties to encapsulate characteristics of a web page. We group some of the features to represent one behavioral property. For example, features 1 and 2 define the visibility of elements in website. This can be achieved using different HTML and CSS properties. So we combine these two features to create one description, which represent the visual context of an element (denoted as ϕ_1). Table 6.3 shows nine behavioral properties derived from the feature set used in our model. For example, in the fifth row, the features are combined to represent, a behavior, which is user interface modification behavior. However, in the fourth row, we use feature itself as a behavioral property because this is the only way to achieve the corresponding functionality.

6.4 Behavior Model and Testing

In this section, we first introduce the proposed behavior model in terms of response the user clicks will get from the web-based programs (or websites) in Section 6.4.1. We then define some heuristics criteria to verify clickjacking and legitimate sites in Section 6.4.2 and Section 6.4.3. Section 6.4.4 shows a relationship between a number of clickjacking attack types and heuristics.

6.4.1 Web Pages Behavior Model

We use Finite State Automaton (FSA) [129] notion to describe a program's behavior. FSA is developed on the basis of known symptoms of clickjacking and legitimate websites with respect to request (user click) and the response after clicking on suspected clickable elements present on a website. In particular, we observe the static and dynamic features to determine the characteristics of a response page.

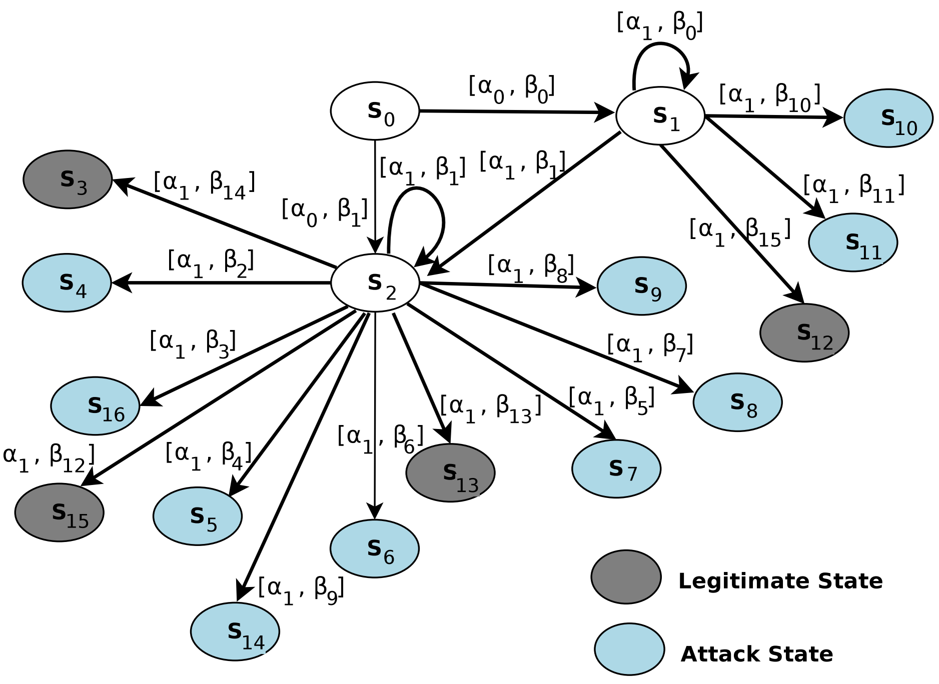


Figure 6.7: State diagram representing behaviors of clickjacking and legitimate websites.

A FSA is denoted by $\langle \Sigma, S, S_0, \delta, F \rangle$, where Σ is a finite set of inputs, S is a non-empty but finite set of states, $S_0 \subset S$ is the initial state, δ is the state transition function, and F is a set of final states. Figure 6.7 shows the state transition diagram of the FSA model that contains seventeen states from S_0 to S_{16}, where S_0 being the initial state. We consider request and response that are of interest in respect of our model. Table 6.4 enumerates the requests. If a page open in browser from initial URL does not contain any hidden clickable element (request $\alpha 0$), then the next state is considered as S_1. However, if the opened page contains non-hidden clickable elements, and on applying click on such clickable element downloads a new page that contains hidden clickable elements (request $\alpha 1$), then the next state is considered as S_2. F is the final state which belongs to $S_3, S_4, \cdots, S_{16}$. Here, a state implies a web page rendered by a browser. To avoid the state explosion problem, we consider the behavior observed from the content of a web page as a single state.

Table 6.4: Relevant requests applied on the websites.

Request ID	**Description**
α_0	Web page open from initial URL in browser window.
α_1	Clicking of suspected clickable element on website.

We denote inputs of the FSA as a interesting requests (denoted as α_0 and α_1) and corresponding responses (denoted as β_0 to β_{15}), which are discussed in detail in Table 6.5. A website is clickjacking or legitimate, if it can reach from an initial state to one of the final states. Some of the final states are legitimate ($S_3, S_{12}, S_{13}, S_{15}$), whereas others are producing clickjacking attacks ($S_4, S_5, S_6, S_7, S_8, S_9, S_{10}$ $, S_{11}, S_{14}, S_{16}$). Figure 6.7 presents the state diagram of our FSA after removal of infeasible states.

A state transition occurs for a given request and the corresponding response. A transition is labeled as request, response pair in the figure. For example, $[\alpha_1, \beta_1]$ implies that given the request α_1, the response is β_1. We summarize interesting responses in Table 6.5.

We observed 61 possible responses with respect to the nine behavioral features discussed in Table 6.3. However, in Figure 6.7, we only use sixteen interesting responses (denoted as $\beta_0 - \beta_{15}$). The rest other combinations are either infeasible or not related to attack cases, and we do not include these in the FSA. Table 6.5 illustrated sixteen interesting response states, the symbol ! represents that a feature is not present in a web page. For example, the first row (β_0 response state) represents that no relevant feature is present on a web page. The third row (β_2 response state) represents that following features are present in a web page (i.e., hidden elements are present, overlapping elements present, hidden/duplicate mouse cursor is not present, pointer-event property is associated with elements, moving elements are not present, no clipping using SVG filters, no script embedded into hidden element, no nested elements, and their is response redirection to other domain). So, the β_2 state represents a behavior, which may results into the attacks such as a_5, a_6 (discussed in Section 6.2.1).

The model provides us the flexibility to detect clickjacking websites that might steal user clicks. A clickjacking website might follow only a subset of the FSA. Moreover, the model differentiates a clickjacking and legitimate website. To test the effectiveness of our FSA, we define several heuristics based on the related work, and proof-of-concept for clickjacking attack to identify whether a website is clickjacking or legitimate. We develop request and response heuristics in the next section.

Table 6.5: Relevant responses gathered from the dataset websites.

Response ID	**Relevant Response States**
β_0	$!(\phi_1\phi_2\phi_3\phi_4\phi_5\phi_6\phi_7\phi_8\phi_9)$
β_1	$(\phi_1) + !(\phi_2\phi_3\phi_4\phi_5\phi_6\phi_7\phi_8\phi_9)$
β_2	$(\phi_1\phi_2\phi_4\phi_9) + !(\phi_3\phi_5\phi_6\phi_7\phi_8)$
β_3	$(\phi_1\phi_2\phi_3\phi_9) + !(\phi_4\phi_5\phi_6\phi_7\phi_8)$
β_4	$(\phi_1\phi_3\phi_9) + !(\phi_2\phi_4\phi_5\phi_6\phi_7\phi_8)$
β_5	$(\phi_1\phi_4\phi_8\phi_9) + !(\phi_2\phi_3\phi_5\phi_6\phi_7)$
β_6	$(\phi_1\phi_4\phi_7) + !(\phi_2\phi_3\phi_5\phi_6\phi_8)$
β_7	$(\phi_1\phi_6\phi_9) + !(\phi_2\phi_3\phi_4\phi_5\phi_7\phi_8)$
β_8	$(\phi_1\phi_5\phi_9) + !(\phi_2\phi_3\phi_4\phi_6\phi_7\phi_8)$
β_9	$(\phi_1\phi_7) + !(\phi_2\phi_3\phi_4\phi_5\phi_6\phi_8\phi_9)$
β_{10}	$(\phi_2\phi_4\phi_9) + !(\phi_1\phi_3\phi_5\phi_6\phi_7\phi_8)$
β_{11}	$(\phi_2\phi_3\phi_4\phi_9) + !(\phi_1\phi_5\phi_6\phi_7\phi_8)$
β_{12}	$(\phi_1\phi_8) + !(\phi_2\phi_3\phi_4\phi_5\phi_6\phi_7\phi_9)$
β_{13}	$(\phi_1) + !(\phi_2\phi_3\phi_4\phi_5\phi_6\phi_7\phi_8\phi_9)$
β_{14}	$(\phi_8) + !(\phi_1\phi_2\phi_3\phi_4\phi_5\phi_6\phi_7\phi_9)$
β_{15}	$(\phi_2) + !(\phi_1\phi_3\phi_4\phi_5\phi_6\phi_7\phi_8\phi_9)$

6.4.2 Request Heuristics

We developed the set of heuristics based on related work, primarily [69, 72, 118, 130]. Our approach utilizes ad-hoc heuristics to determine the class of an attack when certain types of click stealing events are detected. The selection of which heuristics to apply as well as how each is applied is influenced by the type and parameters of the event detected. Our system currently incorporates heuristics for existing, and SVG-based advanced clickjacking attack classes defined in Section 6.2.2. Our heuristics are as follows:

1. **Hidden iframes/DIVs (H1).** This heuristic criterion checks whether a web page contains hidden clickable elements or overlapped on hidden iframes. We use features 1, 2, and 7 to implement this heuristics. This heuristic returns clickjacking if it finds the clickable element position exactly above the target element on hidden iframe. Many clickjacking attacks use this characteristics to steal user clicks. A legitimate website may have hidden elements but these are generally not overlapped with other elements. This observation motivates us to define a heuristic based on the presence of overlapped and hidden elements.

2. **Hidden pointers (H2).** This heuristic criterion is satisfied, if a web page has duplicate mouse pointer, and a hidden clickable elements. The duplicate mouse

cursor is aligned with original one either hidden or transparent in such a way that when user points duplicate cursor on clickable element on attacker page, the original cursor points to target element on hidden iframe. A legitimate website is not likely contain hidden mouse cursor pointing to hidden element. In contrast, a clickjacking website may contain a duplicate mouse cursor pointing to hidden elements. This heuristic can be applied by obtaining values from features 1, 2, and 6.

3. **Pointer-events (H3).** This heuristic criterion checks whether the clickable elements on a web page are responding to mouse/touch events and whether or not the cursor is visible. A clickjacking page often use this feature to execute malicious scripts on a victim browser. This heuristic requires obtaining values from features 1, 2 and 8.

4. **Nested Divs (H4).** This heuristic criterion is satisfied, if a web page contains nested `DIV` tags either overlapping on each other or hidden. A malicious web page may use nested `DIV` tags to initiate malicious link through event bubbling and capturing [127]. Application of this heuristic requires obtaining values from features 1, 2, 9 and 10.

5. **Visible but overlapped elements (H5).** Clickjacking websites sometimes do not hide iframes or clickable elements, instead the element overlaps on a target element in an unnoticeable manner. We developed a heuristics that checks overlapping of clickable elements as well as click transfer that is achieved using `pointer-event`. This will create an attack scenario similar to the one discussing in heuristic H3. The only difference is that in this scenario the elements are visible. In addition, heuristic also checks the response of a click, i.e., on clicking the element the resultant page is in same domain or different domain. This heuristic can be determined by obtaining values from features 6, 7, and 10.

6. **Moving elements (H6).** Clickjacking websites sometimes contain hidden elements that move with a mouse cursor. Using this functionality, wherever an user clicks, an attacker is able to capture it. We developed a heuristics that for any hidden moving `button` inside `iframe`, or `DIV` containers present on a web page. A clickjacking website may use these container to hide a target `button`. Values obtained from features 1, 2, and 5 are needed for this heuristics.

7. **Other hidden elements (H7).** Clickjacking websites rarely contain hidden elements other than iframes or `DIV`. On the other hand, a legitimate website may contains other hidden elements for website functionality. This heuristics checks whether a hidden element present on website is iframe/DIV or any other element. Values obtained from features 1 and 2 are needed for this heuristics.

6.4.3 Response Heuristics

1. **Hidden Script (H8).** This heuristic criterion checks whether a clickable element invokes any script on clicking. A legitimate page may contain scripts but these may not hide behind clickable element. This heuristic returns clickjacking if it finds the clickable element hiding any script. Many clickjacking attacks use this characteristics to initiate XSS or CSFR attacks on websites. We use features 1, 2, and 11 to implement this heuristics.

2. **Domain redirection (H9).** This heuristic criterion checks whether a web page, on clicking a clickable element, generates any traffic from other domain. We use features 12 and 13 to implement this heuristics. This heuristic is useful to detect clickjacking in websites that results in response from domain other that the current working domain. However, this feature may also be present in legitimate websites, so we apply this heuristic in conjunction with other heuristics to detect clickjacking.

TABLE 6.6: Clickjacking attack type and corresponding heuristics applied to detect an attack.

Attack ID	Attack Type	Heuristic
a_1	Click Stealing through Visual Perception	H1 $\vee$ H2 $\vee$ H3 $\vee$ H9
a_2	Click Stealing through Keystrokes	H1 $\vee$ H5 $\vee$ H9
a_3	Click Stealing through Pointer	H1 $\vee$ H2 $\vee$ H9
a_4	Click Stealing through CSS (Stacking Elements)	H4 $\vee$ H5
a_5	Click Stealing through CSS (Pointer-event)	H1 $\vee$ H3 $\vee$ H5 $\vee$ H9
a_6	Click Stealing through Element Movement	H1$\vee$ H6 $\vee$ H7 $\vee$ H9
a_7	Violating Display Integrity Using SVG Filters	H1 $\vee$ H2 $\vee$ H3 $\vee$ H9
a_8	Clickjacking with SVG Clipping and Masking	H1$\vee$ H3 $\vee$ H5 $\vee$ H9
a_9	Modifying User Interface using SVG filters	H1 $\vee$ H3 $\vee$ H5 $\vee$ H9
a_{10}	Enforcing `Pointer-event` Property through SVG	H3 $\vee$ H5 $\vee$ H9
a_{11}	Enforcing Script Injection using SVG Filters	H1 $\vee$ H8

Relation between attacks and heuristics. In this section, we describe how request and response based heuristics can be applied to discover clickjacking. A summary of some example attack types and corresponding heuristics (request and response) is shown in Table 6.6. We have given a detailed description of these attack types in Section 6.2.2. Our heuristics can detect some advanced clickjacking attacks that are achieved using SVG images and filters. For example, SVG uses an alternative way to embed the script into websites, which may inject malicious script on the victim machine. This attack may result in XSS attack through clickjacking.

The detection techniques for checking malicious script only check scripts that use HTML `script` tag. This limitation led our new SVG-based advanced attacks to bypass detection techniques. We have denoted this attack type as a_{11}. Our proposed model can detect SVG based scripting tags to restrict any script injection on a victim machine. The request heuristic $H1$ and response heuristic $H8$ allow discovering the attack.

Comparing our approach with other clickjacking detection techniques. Table 6.7 shows a mapping between clickjacking attack types and defense techniques discussed in the literature. We compare our approach with other clickjacking defense techniques with respect to attack types (a_1 to a_{11}) discussed in Section 6.2.1 and Section 6.2.2. It should be noted that making an iframe nearly or completely invisible is the basic attack type for clickjacking (denoted in Table 6.7 as a_1). It is obvious that disabling JavaScript can solve most of clickjacking attack types [69, 130], although it negatively affects the access to available functionalities. Also, the basic frame busting, HTTP Header or HEAD-based solutions are not adequate when dealing with clickjacking attacks. The new advanced and alternate methods of producing clickjacking attacks are discussed in Section 6.2.2 using SVG filter are not detectable or addressed by any of the previous methods. In contrast, the proposed approach can detect advanced attack types without affecting user experience.

6.5 Implementation and Evaluation

Our detection model, is implemented to defend websites against clickjacking attacks. The implementation details of our approach is as follows.

Table 6.7: Comparison of clickjacking attacks and prevention techniques.

	Attack Types										
Detection Techniques	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
Frame busting [73]	✓	×	×	×	×	×	×	×	×	×	×
HTTP Header [131]	✓	×	×	×	×	×	×	×	×	×	×
Proclick [132]	✓	✓	✓	✓	✓	✓	×	×	×	×	×
HEAD Element [133]	✓	×	×	×	×	×	×	×	×	×	×
Confirmation/randomization [69]	✓	✓	×	×	×	×	×	×	×	×	×
Clicksafe [134]	✓	×	✓	×	×	×	×	×	×	×	×
Blocking of mouse click [135]	✓	✓	×	×	×	×	×	×	×	×	×
Detection of overlapping clickable element [72]	✓	✓	×	×	×	×	×	×	×	×	×
Incontext [69]	✓	✓	✓	✓	✓	✓	×	×	×	×	×
Disabling JavaScript [136]	×	✓	✓	×	×	✓	×	×	×	×	✓
Nepomnyashy et al. [131]	✓	×	×	×	×	×	×	×	×	×	×
NoScript (ClearClick) [70]	✓	✓	✓	✓	✓	✓	×	×	×	×	×
ClickIDS [72]	✓	✓	✓	✓	✓	✓	×	×	×	×	×
Our approach	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

1. The first module is called Query Pattern (QP) module. It consists of two submodules: (i) feature extraction module called EXTRACTOR module that is implemented as a browser plug-in to extract relevant features from a web page and; (ii) QP module generates query pattern from the relevant features.

2. The second module is a signature generation unit, which generates the attack signatures for clickjacking attacks.

3. The third module is C-CHECK parser, which parses query pattern to check symptoms of clickjacking attacks.

4. The fourth module is Click inspector, which categorizes the websites under consideration into clickjacking or legitimate websites based on the heuristics defined in Sections 6.4.2 and 6.4.2 .

We have implemented an extension, which is installed on a Firefox browser (any version). For a web page, EXTRACTOR module fetches all the features (discussed in Section 6.3.1) with the attribute values. QP module then generates the query pattern, which is, then, input to C-CHECK parser module. C-CHECK parses all the query patterns by checking them with attack signatures. The final output is given to click inspector, which generates an appropriate alert for a website based on heuristic rules. Following is a description of each module.

The QP Module. The QP module has an EXTRACTOR plug-in that extracts the relevant features with values from the website. The web pages that are to be scanned for analysis consists of HTML tags, but we are only interested in the RHTs. We identify the values associated with them these RHT using EXTRACTOR (plug-in) installed on a browser. The reason of using browser extension is that it can get access to browser internals, such as, Document Object Model (DOM) [13], which stores entire information of a website opened in the browser.

1. *Attributes Extraction Technique.* Every element in a web page is represented in the form of a DOM tree, which can be read or captured using JavaScript APIs. The DOM tree stores all elements, such as input fields, images, paragraphs, frames, links of a web page. In a DOM tree, the element and attribute nodes are represented as HTML tag and parameter values respectively. RHT values are extracted directly using JavaScript, or JQuery [137] APIs from the source code of a website. We encode RHT as a bit vector in which a it represents a given feature value 0(1) representing absence (presence) of the feature.

   The values for features 1, 2, 7, 8 and 10 are extracted from the respective CSS styles associated with a given RHT. To check for the presence of values Feature 5, we generate automatic clicks at different positions on the page and notice changes in invisible or barely visible elements coordinates after every click. This is done using JavaScript APIs. Feature 6 value is determined by extracting coordinates (top, left, right, bottom) for each element positioned on a web page, and comparing it with the coordinates of all other elements. In this way, we can determine two overlapping elements on a website. The values for features 3 and 11 is taken from source code enclosed within SVG tag.

2. *QP Module.* The QP module contains the implementation of the logic and description of symptoms of clickjacking attack that need to be checked for matching with attack signatures. The obtained behavior (defined in Table 6.3) derived from feature values are processed to build QP, $i.e., QP \rightarrow \{\phi_1, \phi_2, \cdots, \phi_{12}\}$, , where $\phi_1 \cdots \phi_{12}$ represents set of behavior discussed in Table 6.3. This QP is given as input to a parser unit where it is checked against the attack signatures for detecting clickjacking attacks.

Signature Generation Unit. This unit comprises of a set of known attack signatures derived from websites susceptible to clickjacking attack. More precisely, these signatures are built from the relevant features that check if the conditions required for a successful clickjacking attack are met. In our experience, the information collected with relevant features in our clickjacking attack classes are often sufficient to generate automatically high-quality signatures for our detection model.

Signature Database. In this model, the attack signatures are created using relevant feature values extracted from various attack classes. We use dataset of 1000 known attack web pages containing symptoms of clickjacking attack. Our signature database consists of the feature values obtained from these test sample web pages. Our model checks every new web page against the signatures for inspecting characteristics of clickjacking attack. In particular, the attack signature provides the description for all RHTs that we have selected for our model and represents the possibility of an attack. Each attack class has a separate signature, which contains the suitable discriminating attribute values for separating the suspicious and attack web pages from the legitimate web pages.

6.5.1 Experimental Setup

To test the effectiveness of our solution, we first build the effective signatures from various instances of clickjacking attacks. Thus, the base of our solution lies in building accurate attack signatures for RHTs. Our method raises an alert if properties of HTML tags present in a web page resembles with the properties of clickjacking web page.

We conducted four experiments to assess the performance of our proposed method. In the first experiment, we examined the symptoms of basic clickjacking attack features in web pages. In the second experiment, we examined advanced clickjacking attack features in web pages. In the third experiment, we examined the effectiveness of our proposed features in the adaptation of detecting clickjacking attacks on websites. In the fourth experiment, we evaluated the impact of clickjacking attack in different categories of web domains. We used two metrics to evaluate each approach. *True positives* (correctly labeling a clickjacking site as clickjacking and *False positives* (incorrectly labeling a legitimate site as clickjacking..

Dataset Preparation. We first created 1000 web pages susceptible to clickjacking attack. The pages are based on knowledge gained from the literature, basic clickjacking attacks described in the literature, attack classes described in Section 6.2.1, and newly identified attack techniques described in Section 6.2.2. In all cases, our detection system correctly raises clickjacking attacks. We also prepared a dataset of 1000 legitimate pages consists of popular web pages that do not contain any advertisement, pop-ups, hidden iframes, and transparent elements.

Furthermore, we have collect thousands of real-world websites. We have combined different sources to obtained list of URLs that a normal user experiences in his everyday web browsing. We choose 78000 legitimate websites that are representative of what an average user may encounter in his/her everyday web browsing experience. In particular, we included the top 20000 most popular websites published by Alexa [138], and 40000 websites results from the ad-hoc queries on popular search engines. In particular, we queried Google and Yahoo with various combinations of terms such as *porn, advertisement, free download, free iphone/ipod, torrent, warez, online game, free music and free movies*. We ran each query in different languages including English, Chinese, Urdu, German, Russian, and Turkish. We downloaded top 500 URL names from each query to collect around 40000 URL lists. Moreover, to increase the chances of finding attacks, we also included sources that were more likely to contain malicious content. We take down 10000 websites from *malwaredomains.com* [139], and 8000 websites of phishing URLs published by PhishTank [140].

We executed our experiments simultaneously on five Windows virtual machines for 20 days. We have visited 78000 unique domain web pages, out of which around 11.64% of pages are unreachable or not found. The remaining 68920 web pages were scanned with our proposed system. We use Chrome (versions v18.0.1025.168 and v30.0.1599.66) and Firefox (versions 14.0, 20.0 and 24.0) browsers on five virtual machines. The reason of using different browser and versions is to show that these attacks are browser agnostic.

6.5.2 Evaluation of clickjacking features

In this experiment, we examined the symptoms of clickjacking attacks by evaluating proposed prominent features in the websites collected by our data set. The goal of this experiment was to understand the impact or nature of clickjacking attack in these categories. We used an EXTRACTOR (browser extension) that we developed to gather values associated with prominent features. EXTRACTOR takes a list of URLs, loads each URL into a web browser and store feature values in MYSQL database for analysis.

Table 6.8 illustrates the presence of relevant features or attributes in the visited websites. We can see that only 33% of Alexa top 20000 are protected by framebusting, and only 9.8% of web pages are protected in other categories. Thus, server-side protection is low, and an attacker can frame these pages to execute the clickjacking attack. We found that 34.63% hidden clickable elements, 7.9% hidden iframes/frames, and 2.1% hidden DIVs elements were present in the total visited pages including Alexa top 20000. These results show that the web pages from different web domain categories are vulnerable to clickjacking attack.

Table 6.8: Relavant features in visited web pages.

Page Properties	**Reachable Web Pages (48920)**	**Alexa Top (20000)**
X-Frame Protection	9.8%	33%
iframe Usage	65%	47.93%
Hidden iframes	9.94%	4.4%
Overlapped Elements	43.71%	39.99%
Movable Elements	4.03%	1.22%
Nested DIVs	7.09%	4.78%
Hidden DIVs	2.5%	1.76%
Hidden Textbox	3.49%	0.61%
Hidden Cursor	0.02%	1.62%
Pointer-event Usage	7.09%	2.05%
z-index Usage	9.01%	6.71%
Hidden Cross-Domain Links	18.39%	12.73%
Hidden Clickable Links	34.71%	34.49%

6.5.3 Evaluation of advanced clickjacking features

In this experiment, we examined the symptoms of advanced clickjacking attacks (discussed in Section 6.2.2) by evaluating proposed prominent features in the websites collected by our data set. The goal of this experiment was to understand the impact or nature of novel clickjacking attack in these categories. We use the same method as of Experiment 1 to gather feature values and analyze them for detecting clickjacking.

Table 6.9 illustrates the presence of novel features in the visited websites. The results show that 6% of total visited websites uses SVG based images and filters, whereas only 1% of Alexa top 20000 websites uses this features. We have also observed that websites in both the categories uses an alternative method to hide web page elements, use SVG based method to embed scripts, embed links in SVG images, etc. The detail results are illustrated in Table 6.9.

Table 6.9: Newly identified advance (SVG-based) relevant features in visited web pages.

Page Properties using SVG	**Reachable Web Pages (48920)**	**Alexa Top (20000)**
SVG usage in wild	6%	1%
Transparency using SVG	1.8%	0.6%
Link embedding using SVG	0.5%	0%
Pointer-event using SVG	0.8%	0.2%
Using script using SVG	0.3 %	0.7%

6.5.4 Evaluation of clickjacking attack using proposed features

In this experiment, we evaluated how effective our adaptation of clickjacking features was in detecting clickjacking attack sites. Here, we assessed four different conditions:

1. **Basic features:** These features consisted of hidden/transparent iframes and hidden/duplicate mouse pointer. These features can detect only basic clickjacking attacks.

2. **Proposed static features:** These features are discussed in Section 6.3.1. Here, we show how effective are these features in detecting clickjacking attacks other than basic clickjacking.

3. **Proposed static and dynamic features:** Dynamic features are also discussed in Section 6.3.1. We combine static and dynamic features to detect more advanced clickjacking attacks. Besides, combining both features reduces false positives.

4. **Combining static, dynamic features and Heuristics:** We combine static and dynamic features with heuristics to detect more advanced clickjacking attacks.

We tested these features for detecting clickjacking by visiting 1000 websites that are tainted with clickjacking attack and thousands of legitimate URLs from different categories. To test these three feature sets, we collected 1000 clickjacking websites from different sources. We combined different sources to obtain an initial list of URLs, such as, proof-of-concept examples published on the Internet; our own implementation of different variants of web pages that contained clickjacking attack; and few URLs from *malwaredomains.com.*

We used a Firefox extension that we developed to gather our results. Our extension takes a list of URLs from the data set we prepared, loads each URL into a web browser pre-installed as a browser extension. The browser was customized for being suitable for running in the background, enabling automated grabbing of URLs from the list.

Figure 6.8 illustrates the detection results. In comparing basic features with our proposed static and dynamic features, we can see that basic clickjacking features are not able to detect all clickjacking attacks types discussed in Section 6.2.2. The percentage of true positives with basic features is low (90%). The percentage of false positives with basic features are very high (29.22%). The basic features label a website as clickjacking on the basis of hidden elements or hidden mouse cursor that may be used in legitimate websites also. On the other hand if we use our proposed static features, the true positives are 91% and false positives are 14%. Because high false positives with static features is because these features do not monitor the response to clicks. It only checks whether a web page contains symptoms of clickjacking on the basis of feature values obtained from website source code. The percentage true positive and false positive when static features are used with dynamic features is 92.22% and 7% respectively. Since dynamic features analyze the response of a click, the percentage of false positives on combining static and dynamic features is reduced to 7%.

To further reduce the false positives, we developed a suite of heuristics and ran another study to determine the best way of combining these heuristics to reduce false positives while not significantly impacting true positives. The heuristics are described in Sections 6.4.2 and 6.4.3. In comparing static + dynamic features to static + dynamic features + heuristics, we can see that using the heuristics with our proposed features can significantly reduce the false positives percentage (from 7% to 0.28%). Moreover, there is an improvement of true positive percentage (from 92.22% to 98.78%). This improvement of

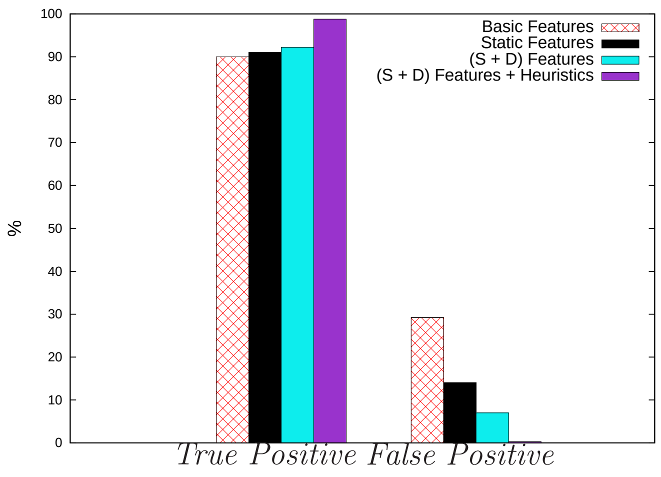


FIGURE 6.8: True positive and false positive metrics on applying basic, static + dynamic (S + D) features, static + dynamic (S + D) features + heuristics on websites.

accuracy is due to adding heuristics derived from clickjacking attack examples presented in published work as proof-of-concept. Thus, the static + dynamic features + heuristics seems to be the best method for detecting clickjacking websites.

6.5.5 Clickjacking impact on web ecosystem

In our efforts to understand how clickjacking attack impacts various categories of websites, we conducted this experiment in three phases.

1. **Phase-I.** In this phase, we first apply our detection model to separate suspicious websites from legitimate working (or reachable) websites. The legitimate websites do not contain any symptoms of clickjacking attack, whereas suspicious websites are further explored for clickjacking warning and attack alerts. Table 6.10 illustrates, for each dataset category, the number of legitimate and suspicious websites.

TABLE 6.10: Impact of clickjacking attack for each dataset.

Web Pages	**Visited URLs (40000)**	**Alexa URLs (20000)**	**Malicious URLs (10000)**	**Phishing URLs (8000)**
Reachable	35192	20000	9529	4199
Legitimate	33114	19468	6747	3177
Suspicious	2078	532	2782	989

2. **Phase-II.** In this phase, we further explore suspicious websites for clickjacking attacks. If a suspicious website does not satisfy any heuristics, our detection model generates a warning alert. In case, suspicious website satisfies one of the heuristics, our detection model generates an attack alert. Figure 6.9 illustrates, for suspicious URLs from all datasets, the percentage of warning and attack alerts generated by our model. It has been found that for suspicious URLs present in four datasets, our model generated 90.86% of warning alerts. The reason for such response is because the fact that the web pages designed for promoting advertisement contains hidden iframes, which targets social networking websites to promote their brands on social networks. It proves that the web page developers use hidden, or transparent content to confuse or to steal user clicks. Furthermore, we found that 9.1% of websites are prone to clickjacking attacks. These websites include both traditional, and advanced clickjacking attacks.

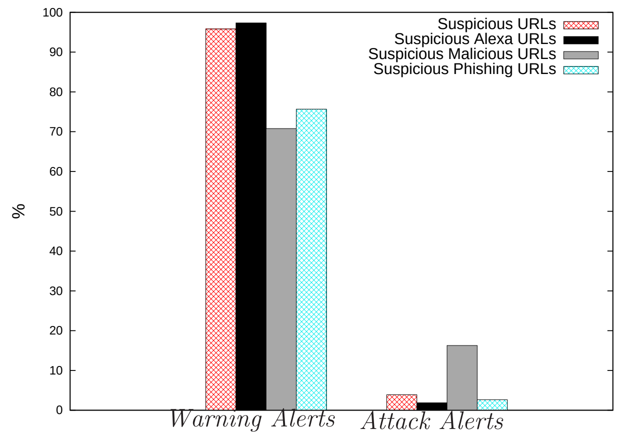


FIGURE 6.9: Breakdown of the warning and attack percentage for warning and Attack alerts % for suspicious web pages.

3. **Phase-III.** In this phase, we compute the number of false positives produced by our detection model. We reported the percentage of false positives for every dataset. The Experiment 4 shows, on applying heuristics, how our detection model reduces the number of false positives. Table 6.11 illustrates, for each dataset category, the number of false positives produces by our detection model. The results indicate that our approach not only detects new advanced clickjacking attacks, but also results in negligible false positives.

Discussion. Our study has found that even a legitimate looking website would sometimes handle compromising the browser with the clickjacking attack. The interesting

Table 6.11: Illustration of False Positives (FP).

Metric	**Visited URLs (40000)**	**Alexa URLs (20000)**	**Malicious URLs (10000)**	**Phishing URLs (8000)**
FP (%)	0.29	0.19	0.18	0.58

point in this analysis is that even a single click on the malicious page can cause a serious privacy breach. The victim is completely unaware of the click thief sitting in their browsers and do not know that their clicks are at risk of conducting unwanted business. We have seen the evidence that this attack mostly target social networking, mailing websites, and sometimes unsecured bank transactions.

1. *False Positives.* Our results show that around 0.28% of the alerts raised by our experiments are the false alarms. The false alarms generated by our solution is because few legitimate web pages often use hidden iframes or divs. In particular, most of the false alarms were generated by pop-ups that dynamically appear in response to particular events, or by advertisement banners that are placed on top of a scrollable page. In both cases, the content of the advertisement was visible to the user, but it confuses our detection method. The advertisement banner can contain clickable elements either overlapped with other clickable elements or enclosed within nested DIVs.

   Nevertheless, note that by combining the static and dynamic features along with heuristics, greatly reduces the number of false positives. For example, if some legitimate web page contains hidden iframe it cannot be called a clickjacking attack page. But, if the same page is having hidden elements wrapped into anchor `<a>` tag, and overlapped with iframe then it might result in a clickjacking attack.

2. *False Negatives.* To estimate the false negative reported by our detection model, we analyzed 1000 malicious pages dataset build from various source. In particular, we have designed a set malicious websites on our own to mimic clickjacking attacks. We developed websites containing attack payloads $a_0 - a_{11}$. We also collected malicious web pages that reported clickjacking attacks. We apply our detection model on malicious page dataset. After analysis, we found that our detection model successfully detected clickjacking attacks in all malicious web pages and had nil false negative.

6.6 Summary

In this chapter, we have identified new variants of clickjacking attacks using SVG-based filters and images. We demonstrated that current clickjacking detection techniques fail to discover these newly identified variants of clickjacking attacks. We analyze to emphasize that there is a requirement for improved detection within the browser to defend against emerging threats originated from CSS, SVG and iframes based clickjacking attacks.

On the basis of our experimental analysis, we proposed a novel approach for the detection and analysis of clickjacking attacks including advanced SVG-based attacks. The approach has been illustrated using the behavioral model in term of Finite State Automaton (FSA) to analyze different behaviors (responses) with respect to user clicks (requests) on the web pages. We evaluate our proposed method with a sample set of 78000 web pages including both malicious and legitimate web pages. The results of the evaluation illustrate that our approach not only detect novel SVG-based clickjacking attacks, but also results in negligible false positives of 0.28% and nil false negative. Finally, we show that current clickjacking attack detection tools and techniques are not able to provide a complete solution against newly identified variants of clickjacking attacks. Also, our detection model provides an improved solution against all types of clickjacking attacks. In the following chapter, we conclude this thesis by summarizing the goals and findings and by providing directions for future work.

Chapter 7

Conclusions and Future work

Browser attacks over the years have stormed the Internet world with so many malicious activities. It provides an unauthorized access, damage or disruption of the user information within or outside the browser. The appearance of various browser attacks executed on web browser causes real challenges to Internet user in protecting their information from an attacker. This thesis is aimed at addressing the issues raised with browser extensions and misjudged user clicks (Click-Hijacking).

7.1 Browser Extensions

Currently available browser extensions provide array of features enhance the look and feel of the browser, and web applications. For this reason, browser extensions enjoy widespread popularity among the users. Moreover, Browser renders extensions to run with full browser privileges, including access to browser components such as browser DOM, cookie manager, password manager and elements or information present in a web page. In addition, browser extensions can access OS resources such as file system, network services, and process system. However, as we have discussed in Chapter 3, the high privilege of browser extension scripts can be a pitfall, this may place the browser under risk of information breach, privilege escalation attacks, etc. We have shown how the current JavaScript security policy, i.e., the Same Origin Policy is not adequate for mitigating this problem.

To address issue arises from the malicious information flow among critical operating resources, we propose a static analysis model for analyzing suspicious flows in JavaScript-based browser extension (JSE). Our model detects potentially insecure information flows within the sensitive source to sensitive sink resources. In particular, it helps to investigate the suspicious flow in JSEs before installing them onto the browser. Our observations found that the suspicious flows in JSEs can be due to the unprotected and privileged access to critical resources. The experiments demonstrate that BEAM can detect critical flows even in some benign extensions, which closely resemble malicious flow and can be critical to the browser security. In the next chapter, we discuss novel attack techniques using colluding browser extensions.

To alert the browser research community, we identified new browser attacks that circumvent all detection mechanisms proposed yet for browser extensions. We demonstrated the extension collusion attacks with respect to extension communication over covert and overt channels. In particular, we showed how two legitimate extensions can collude to achieve malicious goals and how an individual malicious extension can mis-configure browser and another extensions configurations. Our attacks are undetectable by existing known client side methods used for detecting malicious flow and vulnerability in extensions. We have demonstrated our finding by targeting a malicious goal using two legitimate extensions on three critical web domains; banking, online shopping, and websites that allow users to buy download credits. We also provided a proof-of-concept explaining how multiple extensions can collude with each other for compromising the browser for data leakage.

We observed that detecting a malicious flow in an extension is a partial protection against extension-based attacks. Consequently, we propose a sandbox and isolated environment to protect operating system resources from such attacks. Our sandFOX policies restrict an attacker in executing critical attacks on operating system resources such as file system, network, process. Our proposed solution does not modify existing Firefox browser and its components. Instead, it uses Security-Enhanced Linux (SELinux) to build a sandbox that helps in reducing potential damage from extension-based attacks on OS resources. Our proposed policies let a browser application such as extensions and plug-in to access limited OS resources in the restrictive environment, and hence do not affect the functionalities and user browsing experience. We show the practicality of sandFOX in a range of settings, we compute the effectiveness of sandFOX for various browser attacks. We also show that sandFOX enabled browser imposes low overhead on loading web pages and utilizes negligible memory when running with sandbox environment.

7.2 Click-Hijacking in Browser

To understand the attacks targeting clicks of a user, we study new classes of clickjacking attacks in web browser. We find that most of the attacks against users of web application are caused by exploiting the fact that human visual system may not perceive minor changes caused due to blurring or filters used in image processing. We have identified new variants of clickjacking attacks using SVG-based filters and images. We demonstrated that current clickjacking detection techniques fail to discover these newly identified variants of clickjacking attacks. We analyze to emphasize that there is a requirement for improved detection within the browser to defend against emerging threats originated from CSS, SVG and iframes based clickjacking attacks.

On the basis of our experimental analysis, we proposed a novel approach for the detection and analysis of clickjacking attacks including advanced SVG-based attacks. The approach has been illustrated using the behavioral model in term of Finite State Automaton (FSA) to analyze different behaviors (responses) with respect to user clicks (requests) on the web pages. We evaluate our proposed method with a sample set of 78000 web pages including both malicious and legitimate web pages. The results of the evaluation illustrate that our approach not only detect novel SVG-based clickjacking attacks, but also results in negligible false positives of 0.28% and nil false negative. Finally, we show that current clickjacking attack detection tools and techniques are not able to provide a complete solution against newly identified variants of clickjacking attacks. Also, our detection model provides an improved solution against all types of clickjacking attacks.

7.3 Future Work

The work of this thesis has raised more queries and opened vistas for the future. We would like to expand our defense system for browser extensions by incorporating to measure the attack surface or attack opportunities. Intention of this study would be based on identifying over privileged browser extension and providing a solution that can reduce the attack surface of the browser extension.

In the present state of these we have only suggested possible mitigation techniques for newly identify collusion extension-based attacks. Thus, we would like to implement these techniques and improve the browser against colluding attacks. Also, we hope that Firefox and other browsers will consider these issues and come with a secured policy to provide users a secure browsing environment.

For misjudged user click in browser, we would like to extend the proposed technique to improve the detection of other variants of clickjacking attacks. In addition, we plan to implement a browser extension that is able to use the characterization learned by our approach to proactively block clickjacking attacks in real-time.

Publications

A. Journal Publications

[J-1] Anil Saini, Manoj Singh Gaur and Vijay Laxmi., "Review of Man-in-the-Browser Attacks using Security Attack Scenarios", in International Journal of Advances in Computer Networks and Its Security, ISSN (Online): 2250-3757, 2013.

[J-2] Anil Saini, Manoj Singh Gaur, Vijay Laxmi, Mauro Conti., "You Click, I Steal: Analyzing and Detecting Click Hijacking Attacks in Web Pages", in Springer Journal of Information Security. ***Communicated***

[J-3] Anil Saini, Manoj Singh Gaur, Vijay Laxmi, Mauro Conti, Muttukrishnan Rajarajan., "Detecting Threats in Browser Extensions via Semantic Analysis", in Springer Journal of Information Security. ***Communicated***

[J-4] Anil Saini, Manoj Singh Gaur, Vijay Laxmi, Mauro Conti., "Colluding Browser Extension Attack on User Privacy and its Implication for Web Browsers", in Elsevier Computer and Security. ***Communicated***

B. Conference Publications

[C-1] Anil Saini, Manoj Singh Gaur and Vijay Laxmi., "Review of Man-in-the-Browser Attacks using Security Attack Scenarios", in proceeding of ICSEE, 2013, UACEE publications, New-Delhi.

[C-2] Anil Saini, Manoj Singh Gaur and Vijay Laxmi., "Modeling Clickjacking Attacks", Poster- SIS-SNDA-2013 at BITS-Pilani, Hyderabad Campus.

[C-3] Anil Saini, Manoj Singh Gaur, Vijay Laxmi., “The Darker Side of Firefox Extensions”, Proceedings of the 6th International Conference on Security of Information and Networks. ACM, 2013.

[C-4] Anil Saini, Manoj Singh Gaur, Vijay Laxmi, Mauro Conti., “Privacy Leakage Attacks in Browsers by Colluding Extensions.” ICISS-2014, Springer International Publishing, 2014. 257-276.

[C-5] Anil Saini, Manoj Singh Gaur, Vijay Laxmi., “Colluding Browser Extensions: A Threat to Banking Domains”, in 4th Doctoral Colloquium (IDC), IDRBT, Hyderabad, 2014.

[C-6] Anil Saini, Manoj Singh Gaur, Vijay Laxmi., “sandFOX: Secure Sandboxed and Isolated Environment for Firefox Browser”, International Conference on Security of Information and Networks. ACM, 2015. ***Communicated***

C. Book Chapter

[B-1] Anil Saini, Manoj Singh Gaur, Vijay Laxmi., “A Taxonomy of Browser Attacks.”, Book-Chapter: Handbook of Research on Digital Crime, Cyberspace Security, and Information Assurance, IGI-Global (2014), 291-313.

Bibliography

[1] Van Lam Le, Ian Welch, Xiaoying Gao, and Peter Komisarczuk. Anatomy of drive-by download attack. In *Proceedings of the Eleventh Australasian Information Security Conference - Volume 138*, AISC '13, pages 49–58, 2013.

[2] Eric L Howes. The anatomy of a 'drive-by-download', 2004.

[3] Philippe Beaucamps, Daniel Reynaud, and France Loria-Nancy. Malicious firefox extensions. In *Symp. Sur La Securite Des Technologies De LInformation Et Des Communications*, 2008.

[4] Jiangang Wang, Xiaohong Li, Xuhui Liu, Xinshu Dong, Junjie Wang, Zhenkai Liang, and Zhiyong Feng. An empirical study of dangerous behaviors in firefox extensions. In *Information Security*, pages 188–203. Springer, 2012.

[5] Alan Grosskurth and Michael W Godfrey. A reference architecture for web browsers. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 661–664. IEEE, 2005.

[6] Marin Silić, Jakov Krolo, and Goran Delač. Security vulnerabilities in modern web browser architecture. In *MIPRO, 2010 Proceedings of the 33rd International Convention*, pages 1240–1245. IEEE, 2010.

[7] Adam Barth, Collin Jackson, Charles Reis, TGC Team, et al. The security architecture of the chromium browser, 2008.

[8] Andy Zeigler. Ie8 and loosely-coupled ie (lcie), 2011.

[9] Ahmed Obied and Reda Alhajj. Fraudulent and malicious sites on the web. *Applied intelligence*, 30(2):112–120, 2009.

[10] R. S. Liverani and N. Freeman. Abusing firefox extensions. *In Defcon17*, 2009.

[11] Mike Ter Louw, Jin Soon Lim, and VN Venkatakrishnan. Extensible web browser security. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–19. Springer, 2007.

[12] XPCOM. Xpcom interface, 2014. `https://developer.mozilla.org/en-US/docs/XUL/Tutorial`

[13] Document object model. `https://developer.mozilla.org/en/docs/DOM`.

[14] Chris Grier, Shuo Tang, and Samuel T King. Designing and implementing the op and op2 web browsers. *ACM Transactions on the Web (TWEB)*, 5(2):11, 2011.

[15] Elias Athanasopoulos, Vasilis Pappas, and Evangelos P Markatos. Code-injection attacks in browsers supporting policies. In *Proceedings of the 2nd Workshop on Web 2.0 Security & Privacy (W2SP)*, 2009.

[16] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)*, 44 (3):11, 2012.

[17] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security*, volume 3, 2003.

[18] Mozilla developer network-extensions. `https://developer.mozilla.org/en/docs/Extensions`.

[19] Michal Zalewski. Browser security handbook. *Google Code*, 2010.

[20] Extensions. Technologies used in developing extensions, 2014. `https://developer.mozilla.org/en-US/docs/Firefox_addons_developer_guide/`.

[21] Sara Williams and Charlie Kindel. The component object model: A technical overview. Technical report, Microsoft Technical Report, 1994.

[22] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. Technical report, 2009.

[23] Deg Caraig. Firefox add-on spies on google search results, 2009.

[24] Security issue on amo, 2014 . `http://blog.mozilla.com/addons/2010/02/04/please-read-security-issue-on-amo/`.

[25] XPConnect, 2014. https://developer.mozilla.org/en/docs/XPConnect.

[26] XUL. Xul overlays, 2014. https://developer.mozilla.org/en-US/docs/XUL_Overlays.

[27] Anil Saini, Manoj Singh Gaur, and Vijay Laxmi. The darker side of firefox extension. In *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN '13, pages 316–320. ACM, 2013.

[28] A. Barth. Severity guidelines for security issues. *The Chromium Project.* http://dev.chromium.org/developers/severity-guidelines.

[29] L. Adamski. Security severity ratings. *MozillaWiki.* URL https://wiki.mozilla.org/Security_Severity_Ratings.

[30] N. Freeman and R. S. Liverani. Exploiting cross context scripting vulnerabilities in firefox. *Security-assesment.com*, 2010. http://www.security-assessment.com/files/documents/whitepapers/Exploiting_Cross_Context_Scripting_vulnerabilities_in_Firefox.pdf.

[31] R. S. Liverani. Cross context scripting with firefox. *Security-assesment.com*, 2010. http://www.security-assessment.com/files/whitepapers/bfCross_Context_Scripting_with_Firefox.pdf.

[32] David M Martin Jr, Richard M Smith, Michael Brittain, Ivan Fetch, and Hailin Wu. The privacy practices of web browser extensions. *Communications of the ACM*, 44(2):45–50, 2001.

[33] Adrienne Porter Felt. A survey of firefox extension api use. 2009.

[34] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[35] Seyed Hossein Ahmadinejad and Philip WL Fong. Unintended disclosure of information: Inference attacks by third-party extensions to social network systems. *Computers & Security*, 44:75–91, 2014.

[36] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, Vern Paxson, Dhilung Kirat, Giancarlo De Maio, Yan Shoshitaishvili,

Gianluca Stringhini, et al. Hulk: eliciting malicious behavior in browser extensions. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 641–654. USENIX Association, 2014.

[37] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 51–60, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4.

[38] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[39] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Information Security*, pages 346–360. Springer, 2011.

[40] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.

[41] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 2–11. ACM, 2007.

[42] Chris Karlof, Umesh Shankar, J Doug Tygar, and David Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 58–71. ACM, 2007.

[43] Collin Jackson, Andrew Bortz, Dan Boneh, and John C Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web*, pages 737–744. ACM, 2006.

[44] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 525–530. ACM, 2014.

[45] Chris Grier, Shuo Tang, and Samuel T King. Designing and implementing the op and op2 web browsers. *ACM Transactions on the Web (TWEB)*, 5(2):11, 2011.

[46] Richard S Cox, Jacob Gorm Hansen, Steven D Gribble, and Henry M Levy. A safety-oriented platform for web applications. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[47] Sotiris Ioannidis and Steven Michael Bellovin. Building a secure web browser. 2001.

[48] Helen J Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in mashupos. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 1–16. ACM, 2007.

[49] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *Proceedings of the 17th international conference on World Wide Web*, pages 535–544. ACM, 2008.

[50] Shun-Wen Hsiao, Yeali S. Sun, and Meng Chang Chen. A secure proxy-based cross-domain communication for web mashups. *J. Web Eng.*, 12(3-4):291–316, July 2013.

[51] VMware. Vmware, inc. browser appliance virtual machine., 2005. `http://www.vmware.com/vmtn/vm/browserapp.html`.

[52] GreenBorder. Green border technologies. greenborder desktop dmz solutions., 2005. `http://www.greenborder.com`.

[53] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[54] Zhuowei Li, XiaoFeng Wang, and Jong Youl Choi. Spyshield: Preserving privacy from spy add-ons. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID'07, pages 296–316. Springer-Verlag, 2007.

[55] D. Esposito. Browser helper objects: The browser the way you want it. *Microsoft Corporation.*

[56] Sruthi Bandhakavi, Nandit Tiku, Wyatt Pittman, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM*, 54(9):91–99, September 2011.

[57] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 382–391, 2009.

[58] K. Onarlioglu, M. Battal, W. Robertson, and E. Kirda. Securing legacy firefox extensions with sentinel. In *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 122–138. Springer, 2013.

[59] Hossain Shahriar, Komminist Weldemariam, Mohammad Zulkernine, and Thibaud Lutellier. Effective detection of vulnerable and malicious browser extensions. *Computers & Security*, 47:66–84, 2014.

[60] Vladan Djeric and Ashvin Goel. *Securing script-based extensibility in web browsers.* University of Toronto, 2010.

[61] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 115–130. IEEE, 2011.

[62] Mozilla foundation, 2013. `https://bugzilla.mozilla.org/show_bug.cgi?id=154957`.

[63] Robert Hansen and Jeremiah Grossman. Clickjacking, 2008.

[64] Adam Barth, Collin Jackson, and John C Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[65] Mark Zalewski. Dealing with ui redress vulnerabilities inherent to the current web, 2009.

[66] Collin Jackson, Andrew Bortz, Dan Boneh, and John C Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web*, pages 737–744. ACM, 2006.

[67] Philippe De Ryck, Lieven Desmet, Frank Piessens, and Martin Johns. Attacks on the browsers requests. In *Primer on Client-Side Web Security*, pages 57–68. Springer, 2014.

[68] Helen Meyer. Want security? see what hacker does with a cookie ". *Computers & Security*, 16(2), 1997.

[69] Lin-Shung Huang, Alexander Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, pages 413–428, 2012.

[70] G Maone. Noscript. *A Firefox Extension*, 2009.

[71] Krzysztof Kotowicz. Cursorjacking again, 2012. `http://blog.kotowicz.net/2012/01/cursorjacking-again.html`.

[72] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 135–144. ACM, 2010.

[73] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web*, 2:6, 2010.

[74] David Ross and Tobias Gondrom. Http header field x-frame-options, 2013.

[75] XSS Filter Evasion Cheat Sheet. Retrieved june 20, 2013 from the open web application security project: https://www. owasp. org/index. php, 2013.

[76] Antonio Ruiz-Martínez. A survey on solutions and main free tools for privacy enhancing web communications. *Journal of Network and Computer Applications*, 35(5):1473–1492, 2012.

[77] Philipp Ghring. Concepts against man-in-the-browser attacks, 2006.

[78] Gaya K Jayasinghe, J Shane Culpepper, and Peter Bertok. Efficient and effective realtime prediction of drive-by download attacks. *Journal of Network and Computer Applications*, 38:135–149, 2014.

[79] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290. ACM, 2010.

[80] Common Vulnerability (CVE-2007-3743) Exposures. Mitre corporation, 2007.

[81] Common Vulnerability (CVE-2008-3360) Exposures. Mitre corporation, 2008.

[82] Anil Saini, Manoj Singh Gaur, Vijay Laxmi, Tuahar Singhal, and Mauro Conti. Privacy leakage attack in browser using colluding extensions. In *Information Systems Security - 10th International Conference, ICISS, Hyderabad, India. Proceedings*. Springer, 2014.

[83] Using data tainting for security. *Netscape Navigator 3.0.* `http://www.aisystech.com/resources/adv-topic.htm`.

[84] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *Usenix Security*, volume 6, 2006.

[85] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Xiaodong Song. Dynamic spyware analysis. In *USENIX annual technical conference*, pages 233–246, 2007.

[86] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *NDSS*, 2012.

[87] N Hoque, Monowar H Bhuyan, Ram Charan Baishya, DK Bhattacharyya, and Jugal K Kalita. Network attacks: Taxonomy, tools and systems. *Journal of Network and Computer Applications*, 40:307–324, 2014.

[88] T. Parr. *The definitive ANTLR reference: building domain-specific languages*. Pragmatic.

[89] Ecmascript language specification. `http://www.antlr3.org/grammar/list.html`.

[90] Helen J Wang, Chris Grier, Alexander Moshchuk, Samuel T King, Piali Choudhury, and Herman Venter. The multi-principal os construction of the gazelle web browser. In *USENIX Security Symposium*, volume 28, 2009.

[91] Mauro Conti, Arbnor Hasani, and Bruno Crispo. Virtual private social networks and a facebook implementation. *ACM Transactions on the Web (TWEB)*, 7(3): 14, 2013.

[92] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. Jsflow: Tracking information flow in javascript and its apis. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1663–1671, New York, NY, USA, 2014. ACM.

[93] Anton Barua, Mohammad Zulkernine, and Komminist Weldemariam. Protecting web browser extensions from javascript injection attacks. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 188–197. IEEE, 2013.

[94] Hossain Shahriar, Komminist Weldemariam, Mohammad Zulkernine, and Thibaud Lutellier. Effective detection of vulnerable and malicious browser extensions. *Computers & Security*, 47:66–84, 2014.

[95] Swarup Chandra, Zhiqiang Lin, Ashish Kundu, and Latifur Khan. Towards a systematic study of the covert channel attacks in smartphones. Technical report, Tech. Report, University of Texas at Dallas, 2014.

[96] Claudio Marforio, Aurélien Francillon, Srdjan Capkun, Srdjan Capkun, and Srdjan Capkun. *Application collusion attack on the permission-based security model and its implications for modern smartphone systems.* Department of Computer Science, ETH Zurich, 2011.

[97] Google Chrome. sendmessage. *[Online]. Available: https://developer.chrome.com/extensions/messaging*, .

[98] Google Chrome. Message passing. *[Online]. Available: https://developer.chrome.com/extensions/runtime*, .

[99] Jeff Walden. Implement html5s crossdocument messaging api (postmessage), 2007–2008.

[100] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking revisited a perceptual view of ui security. *BlackHat USA, August*, 2013.

[101] Steven B Lipner. A comment on the confinement problem. In *ACM SIGOPS Operating Systems Review*, volume 9, pages 192–196. ACM, 1975.

[102] Adam Barth, Collin Jackson, Charles Reis, TGC Team, et al. The security architecture of the chromium browser, 2008.

[103] Chester Rebeiro, Debdeep Mukhopadhyay, and Sarani Bhattacharya. An introduction to timing attacks. In *Timing Channels in Cryptography*, pages 1–11. Springer, 2015.

[104] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*. Citeseer, 2010.

[105] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, Vern Paxson, Dhilung Kirat, Giancarlo De Maio, Yan Shoshitaishvili, Gianluca Stringhini, et al. Hulk: eliciting malicious behavior in browser extensions. In *Proceedings of the 23rd USENIX Security Symposium*, pages 641–654, 2014.

[106] Adam Barth, Collin Jackson, Charles Reis, TGC Team, et al. The security architecture of the chromium browser, 2008.

[107] Helen J Wang, Chris Grier, Alexander Moshchuk, Samuel T King, Piali Choudhury, and Herman Venter. The multi-principal os construction of the gazelle web browser. In *USENIX security symposium*, volume 28, 2009.

[108] Ulfar Erlingsson, V Benjamin Livshits, and Yinglian Xie. End-to-end web application security. In *HotOS*, 2007.

[109] Jason Hong. The state of phishing attacks. *Communications of the ACM*, 55(1): 74–81, 2012.

[110] Sven Vermeulen. *SELinux Cookbook*. Packt Publishing Ltd, 2014.

[111] Douglas Brian Terry, Mark Painter, David W Riggle, and Songian Zhou. *The berkeley internet name domain server*. University of California, 1984.

[112] Mohammed J Kabir. *Apache Server Bible*. IDG Books Worldwide, Inc., 1998.

[113] Oskar Andreasson. Iptables tutorial, 2006, 2011.

[114] Anne van Kesteren et al. Cross-origin resource sharing. *W3C Working Draft WD-cors-20100727*, 2010.

[115] Hao Zhang, William Banick, Danfeng Yao, and Naren Ramakrishnan. User intention-based traffic dependence analysis for anomaly detection. In *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, pages 104–112. IEEE, 2012.

[116] Chandan Luthra and Deepak Mittal. *Firebug 1.5: Editing, Debugging, and Monitoring Web Pages*. Packt Publishing Ltd, 2010.

[117] Jeremiah Grossman. Clickjacking-owasp appsec talk, 2008.

[118] Marcus Niemietz. Ui redressing: Attacks and countermeasures revisited. *in CONFidence*, 2011, 2011.

[119] Paul Stone. Next generation clickjacking. *BlackHat Europe*, 2010.

[120] Michal Zalewski. Strokejacking, 2010. `http://seclists.org/fulldisclosure/2010/Mar/232`.

[121] E Bordi. Proof of concept-cursorjacking, 2010.

[122] Refsnes Data. Html5 introduction, 2013.

[123] Patrick Lynch and Sarah Horton. Yale c/aim web style guide. *Yale Center for Advanced Instructional Media.[Online]*, 1997.

[124] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. *Scalable vector graphics (SVG) 1.0 specification*. iuniverse, 2000.

[125] J David Eisenberg. *SVG Essentials: Producing Scalable Vector Graphics with XML*. " O'Reilly Media, Inc.", 2002.

[126] Andrew Watt. *SVG unleashed*. Pearson Education, 2002.

[127] Benjamin S Lerner, Matthew J Carroll, Dan P Kimmel, Hannah Quay-De La Vallee, and Shriram Krishnamurthi. Modeling and reasoning about dom events.

In *Proceedings of the 3rd USENIX conference on Web Application Development*, pages 1–1. USENIX Association, 2012.

[128] SH Kim, SH Lee, and SH Jin. Active phishing attack and its countermeasures. *Electronics and Telecommunications Trends*, 28(3), 2013.

[129] Martin Kay and Ron Kaplan. Finite-state automata.

[130] Sebastian Lekies, Mario Heiderich, Dennis Appelt, Thorsten Holz, and Martin Johns. On the fragility and limitations of current browser-provided clickjacking protection schemes. In *WOOT*, pages 53–63, 2012.

[131] Michael Nepomnyashy. Protecting applications against clickjacking with f5 ltm. *SANS Institute InfoSec Reading Room*, 2013.

[132] Hossain Shahriar, Vamshee Krishna Devendran, and Hisham Haddad. Proclick: a framework for testing clickjacking attacks in web applications. In *Proceedings of the 6th International Conference on Security of Information and Networks*, pages 144–151. ACM, 2013.

[133] G Aharonovsky. Malicious camera spying using clickjacking, 2008.

[134] Jawwad A Shamsi, Sufian Hameed, Waleed Rahman, Farooq Zuberi, Kaiser Altaf, and Ammar Amjad. Clicksafe: Providing security against clickjacking attacks. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 206–210. IEEE, 2014.

[135] Clickjacking defense cheatsheet, 2014. `https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet`.

[136] F Aboukhadijeh. How to: Spy on the webcams of your website visitors, 2011.

[137] Bear Bibeault and Yehuda Kats. *jQuery in Action*. Dreamtech Press, 2008.

[138] Alexa internet, inc. alexa - top sites by category, 2014. `http://www.alexa.com/topsites/category/Top/`.

[139] Malware domain list, 2014. `http://www.malwaredomainlist.com/`.

[140] Phishtank domain list, 2014. `http://www.phishtank.com/`.