Ensuring Session Integrity in the Browser Environment

PATIL KAILAS RAVSAHEB
(M.E Computer Engg), University of Pune, India

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2013

Abstract

Over the past decade, web applications have undergone a transformation from a collection of static HTML web pages to complex applications containing dynamic code and rich user interfaces. As the supporting platform for such applications, web browsers execute and manage dynamic and potentially malicious code. However, lack of protection mechanisms in the execution environment provided by web browsers has made various attacks possible that can compromise the integrity of web applications.

Various existing solutions are proposed to secure web applications, but they fail to regulate the behaviors of JavaScript code, such as manipulations of the UI elements or communications with web servers. However, such *behaviors* are key indicators of attacks against web applications. By capturing malicious behaviors exposed by such attacks, we can robustly defeat them. Thus, in this thesis, we focus on fundamental ways to control the behaviors of untrusted code. We develop a line of novel solutions to bring necessary behavior control mechanisms into web browsers, which effectively combat threats to the integrity of web applications.

We first analyze the mediation requirements inside a web origin and propose a technique to regulate behaviors of untrusted code inside an origin using fine-grained access control. We further develop a solution that protects the integrity of web sessions from malicious requests. In a complex browser environment, the attacker may find different ways to inject malicious requests in a victim users active web application. Our solution extracts the client-side dependency of a request and enforces the integrity checks on the request dependency. In addition, we propose an approach to address the problem of insecure extensibility allowed by web browsers that pose threats to the integrity of web applications. Our approach extracts the behaviors of browser extensions to detect integrity violations from the execution of untrusted browser extensions and selectively apply extracted behaviors in a web session.

This thesis proposes new solutions for extracting and controlling the behaviors of untrusted code in the execution environment. They provide an effective way to combat integrity problems in web sessions. As shown by evaluation results on detecting and preventing malicious behaviors in web sessions, this thesis shows that the behaviors of untrusted code play an important role in the development of security solutions for ensuring integrity of web sessions. Our evaluation with real-world web applications also demonstrate the practicality, effectiveness, and low-performance overhead of the proposed solutions.

Declaration

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

PATIL KAILAS RAVSAHEB

21 January 2013

Acknowledgements

First, I would like to thank my adviser, Dr. Liang Zhenkai, to express my profound gratitude and deep regards for his constant encouragement, monitoring and exemplary guidance throughout the course of this thesis. This thesis would have been inconceivable without the blessing, help and guidance given by him time to time on my research works and academic writing. His creativity, dedications and infinite energy are inspiring and motivating for me.

I would also like to thank professors Roland Yap, Chang Ee-Chien, and Prateek Saxena for helpful feedback and constant support on my research works. I sincerely thanks to all my coauthors over the years for all of the hard work, and late nights. I would especially like to thank Professor Xuxian Jiang, for his dedication to left our research greatly enriched.

Many friends have brightened my life in Singapore and provided much needed help and entertainment. I would particularly like to thank Sai, Xinshu, Xiaolei, Meingwei, Bodhi, and DaiTing.

Finally, I would like to give my special thanks to my wife, Shital, and my parents. I could not have made it without their constant love and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Web applications such as online banking, web-based email, shopping, etc., have become ubiquitous in people's everyday lives. For example, a survey by the American Bankers Association shows 62% of respondents prefer Internet banking to in-person banking [89]. To respond to the demand for dynamic services, web browsers have evolved from an application that displayed simple static web pages to a complex environment that executes a myriad of content-rich web applications. Unfortunately, because of the increased popularity, web applications have also attracted the attention of attackers as a new venue for malicious actions. A few examples of web attack techniques are Clickjacking, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF), amongst others.

To better understand these attacks and the corresponding security mechanisms, we first need to understand the basic relationship between web applications, the browser environment, and web sessions. A *web application* architecture is split between client-side and server-side components. Web requests are used to connect these two components. Client-side components are stored and executed in the environment provided by web browsers, i.e., the *browser environment*.

A typical session of a web application on the client-side can often involve execution of code from various *untrusted sources*. Examples of untrusted sources might include third-party widgets, mashups, or external advertisements. Code from these untrusted sources provide new content functionality to users, but they can also introduce malicious code such as malicious browser extensions [15, 54, 78] or intrusive advertisements into a web session [116]. Moreover, bugs in the code from these sources may also introduce vulnerabilities into web applications such as code injection or redirection to malicious sites [129], even if the original web applications contain no vulnerabilities. Simply put, the code from untrusted sources poses a significant risk to the integrity of web sessions, including that of all code and data enclosed in the sessions.

To prevent data of one web application from being accessed by other web applications in the execution environment, web browsers use the same-origin policy (SOP) [99]. For each web page, browsers associate *origins*[1] with web page objects. In effect, SOP partitions the execution environment of web applications, based on these origins. Under SOP, all contents included in one web page are associated with the origin of the hosting web page. Hence, when third-party code such as libraries or advertisement code is embedded in a web page, it runs with the privileges of the web application's origin, even though it only needs limited access to the resources in the origin. Moreover, SOP policy does not control behaviors of browser extensions in active web sessions. This raises a challenge for web applications and web users, such as *how to ensure that the integrity of a web application's logic is not violated by untrusted third-party code running in the execution environment*.

Several research efforts [10, 30, 65, 72, 76, 80, 81, 86, 146] have aimed to isolate resources in different origins or restrict the functionality of JavaScript, but these solutions are coarse-grained and impose an all-or-nothing restriction. To constrain potential threats from malicious web requests, several solutions have been proposed by industry and research community [11, 58, 67, 68, 85, 120, 121, 135] suggesting that web applications should send additional information to the web server in the request. However, these solutions lack the knowledge of client-side behaviors. Thus, attackers constantly find new ways to evade existing solutions, such as self-XSS attack [17], anti-CSRF token social engineering attack [41], and other circumventions. One class of research solutions [12, 69] proposed permission-based frameworks to mandate the declaration of permissions requested by extensions. Under the current permission-based system, an extension code can perform much more than what users expect them to do. Recent attack examples [104, 139, 164] and academic research efforts [69, 78] reveal that browser extension frameworks failed to achieve their design goals.

A study by Dasient [32] on one million websites reveals that 75% of websites use third-party JavaScript widgets on their websites, 42% websites use third-party ad-related resources, and 91% websites use outdated third-party applications. Each of these practices greatly increases their potential exposure to malicious code. According to study by Kirda et al. [73], many malware programs have implementations based on browser extension. Another study by Chia [114] of 5,943 Chrome extensions shows that 35% of Chrome extensions request permissions to access users data on all websites. The broad popularity of third-party code usage and unrestricted access to web applications data raises a major challenge to the integrity of web sessions. For example, the extension *Turn Off the Lights* provides a more pleasant video viewing experience, by inserting a lamp

[1]An origin is defined as the triple of `protocol`, `host`, and `port`.

button in the browser menu bar which makes an active web page dark when clicked. To achieve its functionality, the *Turn Off the Lights* extension requests permissions to inject code into web sessions and access to random network requests. With the granted permissions, the extension can perform dangerous activities such as execute arbitrary JavaScript code in web sessions. As an another example, to earn more revenue web applications could sell space on their web pages to an advertising network. The advertising network would then take advertisements from its clients and display them on the publisher's web pages. An attacker may subscribe as a client to the advertisement network and submit malicious code as an advertisement. When such malicious code is loaded into the publisher's web pages, it will compromise the integrity of the application, and can spawn pop-ups, download and execute code, or forge malicious requests to web servers. Here, one key intuition is that instead of speculating whether an advertisement is malicious or not before it is executed, it will be much more accurate to detect attacks by monitoring the behaviors of the advertisement code during its execution.

We observe that the runtime *behaviors* are key indicators of malicious code. However, existing solutions fail to provide proper control on the behaviors of the untrusted code executed from various sources in the web session, such as browser extensions or embedded third-party code in web pages (for e.g. third-party widgets, mashups, or external advertisements). As a result, they are unable to regulate the malicious behaviors of untrusted code, exposing web applications to attacks against integrity.

Our observation entails that we should build our defense against threats to web session integrity based on the behaviors of untrusted code. Such behavior-based techniques can transparently regulate the behaviors of untrusted code, without restricting its functionality. This thesis focuses on the fundamental ways to use behaviors of untrusted code to protect integrity of web applications in the browser environment. We believe that *client-side behaviors are key aspects to protect integrity of web sessions on the Web platform.*

This thesis supports the above thesis statement and develops a line of solutions to detect and prevent malicious behaviors in web sessions from untrusted JavaScript libraries, user intention interference, malicious requests, and malicious browser extensions.

1.1 Thesis Overview

The goal of this thesis is to protect integrity of web sessions in the execution environment. We develop solutions that control the behaviors of untrusted code to protect web session data in the execution environment.

We examined the execution environment provided by web browsers and identified major components and their interactions [39, 110]. We used this analysis to identify behavior control mechanisms required within the execution environment to protect integrity of web sessions, and developed a line of novel solutions: 1) a behavior control approach to regulate behaviors of untrusted JavaScript code embedded within a web page, 2) a user inference monitor to prevent clickjacking behaviors, 3) a client-side request dependency extractor to validate web requests by using browser behavior dependencies in the execution environment, and, 4) a behavior control approach to confine browser extensions without restricting JavaScript capabilities.

We start from the assumption that the browser itself and the underlying operating system are trusted and they are not under the control of attackers. However, there exists untrusted JavaScript codes that can run in the execution environment provided by web browsers in the form of browser extensions, advertisements, and external JavaScript libraries. We addressed the challenges that arise when controlling behaviors of untrusted JavaScript code running in the execution environment.

Securing web applications from untrusted JavaScript included within an origin Within an origin, needed are security mechanisms that monitor and control the behavior of untrusted code. This is because web applications commonly use JavaScript from third-parties, and these scripts from untrusted sources are executed in the JavaScript execution environment with the full privileges of the web application's origin.

To address these challenges, Chapter 3 presents a reference monitor called JERMonitor, which enables behavior control of untrusted code by using fine-grained access control in the JavaScript execution environment.

Preventing clickjacking by user intention inference We propose a mechanism to combat Clickjacking attacks, which can trick users by exploiting layout features provided by browsers [111], by which attackers can create web page objects that hijack user clicks. Such objects look like normal web page objects, but user clicks on these objects can lead to unexpected browser behaviors, such as visiting different URLs or sending out malicious requests. Although the actual techniques involved in these attacks may vary, they generally aim to make users clicks trigger browser actions that users do not expect.

We observed that user intentions play an important role in detecting clickjacking behaviors, but it is a challenging task to correctly infer user intentions. In Chapter 4, we propose a technique to automatically infer a user's intentions in the browser environment.

Securing web session from malicious requests Web requests are the cornerstones of modern web applications. As the browser environment evolves with increasing complexity, attackers have continued to develop varied ways to trigger malicious requests to the server. Traditional security solutions, such as HTTP cookies and session IDs, are now insufficient in helping the server to distinguish benign web requests from malicious ones.

By design, a web application only expects requests to be generated by specific *browser behavior sequences*. Browser behaviors of a web request play a key role in validating requests. In Chapter 5, we propose a technique that collects browser behaviors by extracting the dependency of web requests from the browser environment, representing them in a request dependency graph (RDG). RDG allows web servers to detect malicious requests through checking request dependency.

Confine behaviors of malicious browser extensions Web browsers allow themselves to be extended by third-party code, such as browser extensions, to provide enhanced functionality and customization features to their users. Browser extensions can have high privileges to access web page contents, thus recent browsers, such as Chrome, control extensions' capabilities with permissions. Browsers associate permissions requested by an extension during the extension installation time. However, malicious extensions can usually perform more dangerous actions than what they are expected to do with the permissions given to them. For example, once injected by the extension into victim web sessions, malicious content scripts can send malicious requests in the web application without the user's knowledge. To address this problem, we propose an approach, SessionGuard, which confines malicious extensions based on the behaviors of extensions as described in Chapter 6.

1.2 Summary of Contributions

In this thesis, we examine the security of web sessions in the execution environment and show that the capability to control behaviors of code plays an important role to protect the integrity of web sessions in the execution environment. This thesis makes the following contributions:

1. We develop an approach that regulates behaviors of untrusted code in the JavaScript execution environment. It uses privilege separation and access rules enforcement techniques to control behaviors of untrusted JavaScript code. Compared to existing solutions, our approach provides fine-grained access control to untrusted JavaScripts within an origin.

2. We develop a new approach that infers intent to ensure that browser behaviors match user intentions. Our approach prevents clickjacking attack behaviors in web sessions.

3. We analyzed various attack vectors that can be leveraged to launch client-side session misuse attacks and summarize how *browser behavior dependencies* of malicious requests from these attacks differ from legitimate ones in the client-side execution environment. We identified the *request dependency* as the inherent characteristic to validate web requests. We proposed a new approach that uses request dependency to validate web requests.

4. We analyzed behaviors of browser extensions and propose a protection mechanism to control behaviors of browser extensions. Our approach effectively extracts the net effects of browser extensions and only applies benign ones to the original web sessions. This way, our approach does not restrict JavaScript capabilities of browser extensions.

Chapter 2

Background and Related Work

We introduce the background and related work of this thesis in this chapter. First, an overview of core web application technologies is given in Section 2.1. It serves as the basis for understanding behaviors in web applications. Then, we elaborate on the internals of the execution environment provided by web browsers in Section 2.2. In Section 2.3, we summarize existing attacks against the integrity of web sessions. We discuss existing security solutions against these attacks in Section 2.4.

2.1 Core Web Application Technologies

The modern web application uses three essential technologies: HTML, CSS, and JavaScript. HyperText Markup Language (HTML) [154] is the primary language for creating web pages and displaying information within web browsers. HTML consists of HTML elements (such as <html>) that are enclosed in angle brackets within a web page's contents. A web browser uses HTML tags to mark web page contents. The Cascading Style Sheets (CSS) [150] are used to describe the look and formatting of a web page within web browsers. The CSS are used to isolate the content of a web page from its presentation to improve content accessibility, provide more flexibility, and enable multiple web pages to share style formatting. The JavaScript [155] is the scripting language which makes web applications responsive and dynamic. JavaScript interacts with the user, controls the browser, communicates asynchronously with web servers, and dynamically changes web page contents.

Web applications are coded in browser-supported languages such as HTML and JavaScript, among others, and accessed over a network using HTTP or HTTPS protocol. Web applications are preferred over dedicated client-server software programs because they do not need to be installed on client computers and allow cross-platform compatibility.

A web application contains server components and client components. To connect server-side components and client-side ones, web applications expose HTTP-based interfaces on web servers to web clients, so that the web clients can request operations on web servers using the HTTP protocol. An HTTP session involves a sequence of request-response transactions between the web browser and the server. However, HTTP is a stateless protocol, meaning that web servers do not maintain states to relate one request to another. To support session-based web applications, such as, email or e-commerce websites, the session-related states must be stored by the browser. Requests should carry the session information to identify themselves to the server. To track sessions among multiple HTTP requests, web applications use different mechanisms such as session cookies. Cookies are widely used as a mechanism to "remember" the current user and the ongoing session of a web application. The web application server sets the cookie in the response to the browser after a user successfully logs in, and the browser stores the cookie for a certain amount of time. For *every* subsequent HTTP request sent to the web application server, the browser automatically attaches the cookie to identify the session.

Web applications use a browser-supported language such as JavaScript or HTML to develop the client-side part, and commonly embed resources from third-parties such as JavaScript libraries, advertisement scripts, mashups, or widgets. Web sessions are vulnerable when there are untrusted contents embedded in web applications.

2.2 The Execution Environment

To better understand the attacks and existing solutions, we need to understand the client-side execution environment that is provided by web browsers. Figure 2.1 shows a browser environment that contains a set of components to process and render web contents and communicate with web servers.

- *The network library* The network library of the execution environment is responsible for handling various network communication protocols such as HTTP and HTTPS.

- *Host objects* These are objects provided by the hosting application (for example, the web browser) of the JavaScript engine for accessing peripheral resources outside the JavaScript engine. Host objects include the Document Object Model (DOM) objects, XMLHttpRequest (XHR) object, cookies, and others. The *DOM* objects are a cross-platform way to represent and interact with HTML tags. The DOM is a platform and language-independent interface to

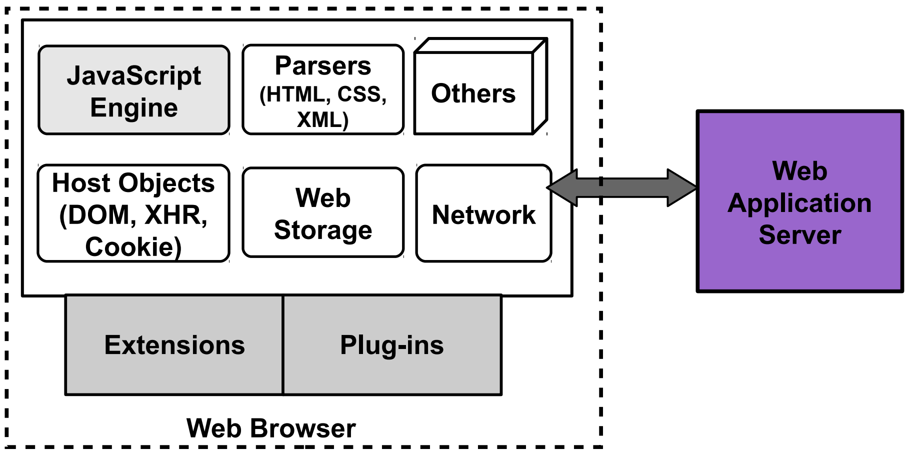


Figure 2.1: Components in the web application's execution environment

describe HTML and well-formed XML documents. It allows scripts to dynamically access and update these documents. Web browsers parse the markups such as HTML, XML, and build an in-memory DOM, which is an organized tree structure of the document. The DOM tree has a node for every element in the document [28]. Web pages may store sensitive information in HTML tags such as security tokens in hidden input field or administrator email. The *XMLHttpRequest (XHR)* is a DOM API that allows JavaScript to access data from a web server without refreshing the web page content. *Cookie* has been widely used as a mechanism by web applications to remember the current user and the ongoing web session of the web application. The cookies allow a web application to track a user's session between multiple requests and are attached by the client web browser with each HTTP request to the server in order to authenticate the user of the session.

- *Parsers* Web browsers implement parsers to parse various types of content, including HTML, XML and CSS. The HTML/XML *parser* components parse the raw content of HTML and XML and build the document object model (DOM) tree. Each node in the DOM tree also has the associated CSS data, including any web application-defined event handlers.

- *The JavaScript engine* The JavaScript engine is a key component for script execution. It consists of *JavaScript contexts* and *host objects*. The JavaScript context includes JavaScript objects representing variables and functions. The set of host objects gives access to local objects in the host environment. For example, host objects in a web page include the page's Document Object Model (DOM), while host objects of a Firefox extension environment provide interfaces to native services such as local file access.

- *Web storage* This allows the storing of structured data on the local file system of a client machine. There are two main types of web storage: *sessionStorage* and *localStorage* [29]. Because cookies are included with every request, it becomes inefficient for web applications to store megabytes of user data at the client side. The web storage mechanism is used to address the weakness of cookie. In addition, the web storage mechanism allows multiple transactions to be conducted on the same web application in different windows at the same time. For instance, if a user of an airline ticket-purchasing web application is buying tickets from that application in two different windows, then cookies could possibly leak the ticket being purchased from one window to another and potentially lead the user to buy two tickets for the same flight. The sessionStorage mechanism allows the web application to store the client side data that spans multiple windows, and available in the current session. The localStorage mechanism not only allows client-side data that span multiple windows, but also allows client-side data to be persistent and last beyond the current session.

- *Browser extensions* To enhance user experience and provide new functionality, user-installed browser extensions have been developed by browser vendors and third-party developers. Browser extensions are written in JavaScript, but they can also contain binary components. Extensions written in JavaScript access resources using APIs exposed to extensions by web browsers, while extensions with binary components, also known as plug-ins, can directly use APIs provided by an operating system to access the user file system or resources.

- *Plug-ins* Web browsers do not natively support all media types used in web applications, such as PDF or Flash. Plug-ins extend the web browsers' support to these media types, by processing and rendering these media types for web browsers. Web browsers provide mechanisms to allow the content rendered by plug-ins to inject scripts into a web page.

- *Others* This includes components such as the rendering engine, the event manager, and Visual Basic script interpreter. The rendering engine uses the DOM tree to compute the position and style of each element, and renders web application's cross-platform user interface. The *Visual Basic script*, interpreter is only supported by Internet Explorer. Other web browsers such as Opera, Firefox, and Google Chrome do not provide built-in support for it. The *event manager* in web browsers is responsible for handling events including registering event handlers, invoking appropriate event handlers and listening to user interface events.

The execution environment provided by web browsers runs client-side code for web applications. Web applications expect that the behavior of its code is not tampered within the execution environment by malicious code of unverified origin. To achieve this, modern web browsers embodied same-origin policy (SOP) mechanism.

The primary security model in browsers: the same-origin policy (SOP) Modern browsers partition the execution environment of one web application from another using the same-origin policy (SOP). The same-origin policy prevents JavaScript from accessing DOM properties and methods across partitions. However, there are a few weaknesses of SOP as discussed below.

- *The inconsistent enforcement of SOP under different scenarios.* SOP is enforced differently for different resources in the execution environment [163]. For example, web pages can set `document.domain` property to a fully-qualified fragment in the right-hand part of its current domain name. For instance, `abc.example.com` can set it to `example.com`. If two web pages set their respective `document.domain` property to the same value, then they are granted to access each other's web session.

- *Third-party resources included in a web page.* When scripts are on a completely different domain and `document.domain` cannot be used, then web publishers often include the contents within their web pages, such as third-party widgets or external advertisements, as a means to earn revenue. By hosting malicious widgets or malicious advertisements, attackers often find a way to execute malicious code into web sessions of an application. Under such scenarios, origin-based security mechanism is too coarse-grained and inadequate to protect the integrity of web sessions from attacks that occur from within the origin of the victim's web application.

- Limited to web applications. SOP is only enforced on web applications. Browser extensions are not subject to SOP, even though they have full access to web sessions running in the execution environment.

Even though SOP lays a useful security foundation inside the execution environment to protect integrity of web sessions, it is far from providing comprehensive protection to web sessions. The next section describes various examples of possible threats to the integrity of web sessions.

2.3 Attacks in the Execution Environment

In this section, we focus on the possible threats from untrusted JavaScript included in web sessions which run in the execution environment provided by browsers. In this thesis, we focus on web-based attacks against web applications.

Threats within an origin It includes threats from JavaScript either included in web applications by design (such as mashups, JS libraries, etc.) or by exploiting vulnerabilities in web applications (such as XSS attack).

Malicious mashup: JavaScript allows functions to be defined and assigned in the execution environment during runtime. For example, the ability to dynamically override functions is commonly used in web applications to mask the differences in various browser implementations, or temporarily fix browser implementation bugs [156]. There are JavaScript libraries to override JavaScript functions provided by the browser, so that web applications can be coded in a browser-independent manner. However, attackers can exploit this feature to inject code into benign JavaScript, and carry out malicious activities with escalated privilege.

Let's look at an example of how a malicious gadget integrated into a web page can affect the behavior of other benign gadgets integrated within the same web page, thereby severely threatening the integrity of the entire mashup web page. Suppose the integrator page `http://public.com` includes two gadgets from two different domains as follows:

```
<div id='Gadget1'>
  <script src='http://example1.com/gadget1.js'></script>
</div>

<div id='Gadget2'>
  <script src='http://example2.com/gadget2.js'></script>
</div>
```

Suppose `Gadget2` checks the current web page's location, by calling the JavaScript function `location.href.toString`. In this case, the location is used to construct a request for a piece of external JavaScript from `example2.com` that can contain sensitive information, if the integrator page is `https://private.com`.

If `Gadget1` is malicious, it can override the native `toString` function of the `String` object with its own function:

```
String.prototype.toString = function() {
  return "https://private.com";
}
```

Consequently, when `Gadget2` calls the `location.href.toString()` function to retrieve the location of the integrator page, its call to `location.href.toString()` is "answered" by the function defined by `Gadget1`, which then returns `https://private.com`, instead of the location `http://public.com`. This way, attackers can retrieve the sensitive information supposed to be only displayed at `https://private.com`.

Another attack example is dynamic code injection by malicious embedded Flash applications into the execution environment. Flash contents embedded in a web page can inject scripts through interfaces exposed to plug-ins by web browsers for interactions with the integrator's web page DOM. Although the permissions of a flash object to inject scripts is controlled by web applications, some web applications erroneously give full access to all flash contents. For example, to track user clicks on advertisements, the Yahoo! web application allows Flash contents to inject JavaScript code into the web page. However, malicious Flash contents can exploit this feature to inject scripts into the web page and compromise the integrity of the entire web application session, or generate malicious requests to web servers by fraudulently using the credential of legitimate users.

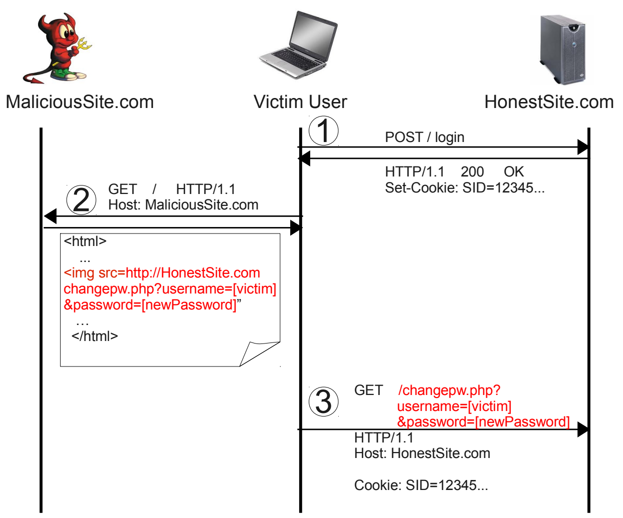


Figure 2.2: Illustration of cross-site request forgery (CSRF)

Injecting malicious web requests Web browsers enforce the same-origin policy (SOP) to prevent a website from accessing cookies that belong to another site. However, web browsers do not impose any restriction on outgoing requests to external resources. For example, if a user visits a malicious website (*MaliciousSite.com*) while the cookie-based session on the website *HonestSite.com* is still active, the malicious site can initiate cross-site HTTP requests to the *HonestSite.com*.

As shown in Figure 2.2, a user authenticated to *HonestSite.com* (Step 1) visits *MaliciousSite.com* (Step 2). The malicious site triggers a cross-site request to *HonestSite.com*, using an `<img>` tag to request a sensitive service (Step 3). Since the browser automatically attached the user's cookie, *HonestSite.com* processes the request. This attack example is constructed from a real world attack example CVE-2009-3759 [3, 107].

Malicious extensions Under the permission-based system, an extension can perform much more than what users expect them to do such as executing arbitrary JavaScript code in web pages.

JS/Febipos.A a JavaScript trojan: This malicious Firefox and Chrome extension gives an attacker remote control of the victim user's Facebook profile [19]. Figure 2.3 shows the high-level mechanism of the trojan JS/Febipos.A. Once installed in a user's browser (either Firefox or Chrome), this trojan extension monitors victim users to see if they are currently logged in to Facebook, as shown in the Step 1. It then attempts to get a configuration file from its master server, as shown in Step 2. This file is one of the following:

- $avbr. < removed > /sqlvarbr.php$
- $frupsv. < removed > /sqlvarfr.php$
- $le - chinatown. < removed > /sqlvarch.php$
- $leferrie. < removed > /sqlvarbr.php$
- $lesmecz. < removed > /sqlvarbr.php$
- $supbr. < removed > /sqlvarbr.php$

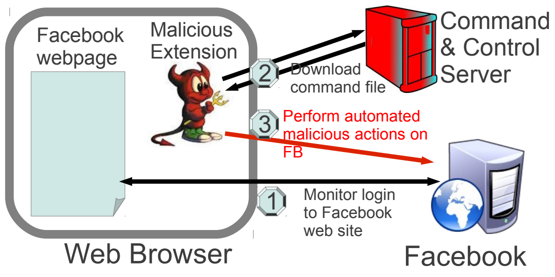


Figure 2.3: A sample demonstration of download-and-execute botnet using the JS/Febipos.A trojan

The file contains a list of commands that the browser extension will execute. In Step 3 shown in Figure 2.3, this trojan can perform any of the following operations on the infected user's profile: add posts to a profile, like pages, join groups or invite others to join groups, chat to friends and comment on posts. The message posted may vary depending on the configuration file. The message it posts also entices other users to infect themselves.

Malicious extensions can perform dangerous activities in web sessions than what are expected with the permissions given to them. When malicious extensions tamper with web application logic, it cannot be stopped by permission-based mechanisms.

2.4 Existing Defense Solutions

In this section, we discuss existing solutions to protect web sessions within the execution environment.

Defense solutions against threats within an origin Several solutions have been suggested to increase the granularity of access control within the browser execution environment. A class of research efforts [9, 13, 14, 53, 66, 74, 82, 101, 118, 122, 125, 130, 137, 141, 146] prevents cross-site scripting (XSS) attacks by identifying injected JavaScript and preventing it from executing in the execution environment. Other approaches [2,45,48,52,62,66,83,84,88,115,115,161] either isolate the JavaScripts, or limit accesses from certain JavaScript by disallowing dangerous features. These approaches target injected or untrusted JavaScript within a confined portion of a web page, but it is not generally applicable to other types of environments, such as web page and browser extensions, where full JavaScript functionality is expected. Escudo provides fine-grained access control to host objects within the context of web pages [65]. Ter Louw et al. [81] provides a policy enforcement framework for the JavaScript in Firefox extensions. It controls extensions' access to XPCOM services, network services, or a browser's password manager, among other restrictions. These two approaches provide fine-grained access control to the *host objects* of their JavaScript environment. Similarly, research efforts [12,60,80] regulate DOM objects. Nevertheless, a malicious JavaScript code can compromise the integrity of web applications either by injecting a malicious JavaScript code or by injecting a forged HTTP request in live sessions of web applications.

Several research proposals develop new primitives for web applications to adapt to mashup applications [30, 64, 72, 146]. Extensions [63, 70, 79] to the same-origin policy have also been proposed to allow better separation of JavaScripts even if they are from the same origin. However, these approaches do not support dynamic access control policies and provide coarse-grained access control.

Flax [124] proposed dynamic analysis techniques to systematically discover any client-side validation vulnerabilities. Flax used the features of dynamic taint analysis combined with automated random fuzzing to remedy any the limitations. Adsentry [40] proposed an in-browser isolation mechanism to combat attacks from malicious scripts in advertisements. It provides isolated, separate environments for trusted and untrusted scripts that previously were executed in the same environment. In addition, it enables flexible policy enforcement to control the behaviors of untrusted JavaScripts.

To mitigate threats of XSS attacks, Mozilla proposed Content Security Policy (CSP) [134] a more in-depth defense. CSP has become a part of W3C specification, and CSP 1.0 is in the state of Candidate Recommendation [117]. It aims to combat XSS attacks by providing a declarative content restriction policy in an HTTP header that the browser can enforce. CSP defines directives associated with various types of content that allow developers to create whitelists of trusted content sources and instruct client browsers to only load, execute, or render content from those sources. However, writing an effective and comprehensive CSP policy for websites is laborious. A policy can break website functionality if legitimate content is overlooked during policy generation. Moreover, web developers at large technology companies may not have direct access or authorization to change the CSP header on web servers, making it difficult to iterate over policies. This is hindering the adoption of CSP by real-world web applications. We performed a usage study of CSP policy on real-world desktop and mobile websites. Our results show that out of the top 104,500 Alexa websites [6], only twenty seven unique websites are using CSP policies. In particular, only twenty desktop websites actually enforced CSP policies, and the remaining seven websites use CSP policy in report-only mode. Similarly, we analyzed 289 mobile websites [7]. In our experiments we noticed that only one mobile website, `mobile.twitter.com`, uses a CSP policy. UserCSP [112] is a Firefox extension that allows security-savvy users to specify their own CSP policies on websites.

In addition, Firefox's implementation of CSP lets a protected website specify a set of whitelisted websites that can embed it through the *frame-ancestors* directive. The *X-Frame-Options* HTTP header provides a similar but more restrictive functionality, as it only allows the website to be embedded in its own origin *or* a single trusted third-party, but not both [16]. With *frame-ancestors*, websites can specify a list of trusted origins that can embed the page, while still protecting the website from being embedded in arbitrary and potentially malicious websites. The *frame-ancestors* directive will enforce that a website's parent and all its ancestors are in the list of whitelisted resources before it loads the website in an iframe, while *X-Frame-Options* will only enforce restrictions on an iframe's direct parent. Hence, the *X-Frame-Options* header doesn't provide the same level of security as *frame-ancestors*.

Defense solutions against malicious requests To detect malicious requests, several research solutions propose to send additional information to the web server in the requests. For example, some validate requests at the server-side by using authentication tokens such as NoForge [68], the work by Florian [71], and web application development frameworks such as Django [38], or Struts 2 [26]. The `Origin` header [11] validation is an effective mechanism that prevents CSRF attacks, but it is inadequate to serve web applications in judging the legitimacy of a request when that request gets generated by injected script in a web page. Scuta [135] proposed an approach to enable web servers to implement a ring-based access control that would detect privilege escalation of a request within a web session by assigning different privileges to requests within the same session.

NoScript is a plugin for the Firefox web browser that allows users to disable JavaScript on a per domain basis to mitigate XSS attacks [106]. However, the major problem with disabling the JavaScript on a website substantially reduces the website functionality and responsiveness.

BEAP [85], implemented as a browser extension for Firefox, mitigates the CSRF attack on the client side. It can infer whether a request reflects the user's intention, by using the heuristics derived from analyzing real-world web applications. If a sensitive request does not appear to reflect a users' intention, BEAP strips the authentication token from it. RequestRodeo [67] is also a client-side solution to mitigate the CSRF attack. It works as a local HTTP proxy, and identifies HTTP requests that are suspicious as CSRF attacks by monitoring the URLs of all incoming requests and outgoing responses. It removes the authentication token, such as the `Cookie` header, from outgoing HTTP requests for which the origin and target do not match.

These solutions target specific characteristics of particular attacks, but they do not prevent a more general class of attacks that send malicious requests to honest websites.

Defense solutions against malicious browser extensions Ter Louw et al. [81] developed a solution to monitor XPCOM calls by extensions to a subset of Firefox's privileged APIs, in order to secure the extension installation process. However, primary extension APIs remain unprotected, and extensions still have full privileges to access web sessions. SABRE [36] enhances a browser with JavaScript information flow analysis to prevent malicious extensions from accessing sensitive information in web sessions. The proposed approach attaches security labels with each JavaScript object, and tracks the propagation of these labels to prevent a sensitive object being sent to an external domain. It is an effective mechanism against privacy leakage attacks but it doesn't control the behaviors of malicious extensions to protect integrity of web sessions. Another line of research [46, 51, 109, 136, 147] proposes new browser architectures to improve security. They use process-level isolation for different components of a browser. However, threats from malicious extensions to web sessions were not considered in these architectures.

The Google Chrome web browser's least-privilege principle and multi-process architecture for browser extensions improve the overall security of the web browser [12]. However, it only focuses on threats from malicious websites and does not protect web sessions against malicious browser extensions. Google Chrome adopts a permission-based system to control what an extension can do. For various sensitive operations, or access to sensitive resources, such as network capabilities, cookies, geographic location, or browsing history, it enforces extensions to declare and request corresponding permissions [25]. When users install an extension, the permissions requested by the extension are shown to the users for approval, and users may be warned about the security implications of granting dangerous permissions. However, users may not be able to make a sensible or informed judgment on whether to allow or deny such permissions, especially when it is subjective. Furthermore, as users want to use such extensions, they may become habituated to permission prompts and click through them routinely, without actually checking them [23]. Browser extension permissions can only loosely restrict the behaviors that extensions can perform at run time. A commonly requested and approved permission by an extension is to inject scripts (also known as content scripts) into web pages, which is needed for functionalities including page touch-up, page content translation, user input capturing, etc. Once scripts are injected into web pages, they have full privileges to access web application's resources, such as issuing unauthorized transactions. On the other hand, the existing security mechanism to regulate content injection in a web application is Content Security Policy (CSP) and it also does not provide sufficient protection against malicious content scripts injected by browser extensions.

Next, in Chapter 3 we discuss in-depth the lack of privilege separation and fine-grained access control within the partition of an origin to confine behaviors of untrusted JavaScripts embedded in a web page.

Chapter 3

Securing Web Applications from Untrusted JavaScript Included within an Origin

JavaScript enables a new generation of dynamic and interactive web applications, which commonly use JavaScript from various sources, such as third-party JavaScript libraries, advertisement scripts, etc. Some of these third-party scripts are untrusted and can become malicious. To ensure web applications session security, it is critical to properly regulate behaviors by untrusted JavaScript to web application resources. Under SOP all scripts in a web page share JavaScript context and host objects. Some of these included scripts in a web page are untrusted and can become malicious in the future. Therefore, untrusted JavaScript in a web page must run in a controlled environment to regulate its behaviors with trusted content on the web page.

3.1 Motivating Example

In this section, we show the lack of privilege separation and fine-grained access control techniques in the JavaScript environment with examples and define the problem addressed by our solution.

The bookmarklet attack A bookmarklet is a piece of JavaScript saved as a bookmark entry in the browser's bookmark menu. When activated by a user click, the JavaScript code in the bookmarklet executes in the execution context of the current web page. MashedLife [77] is a bookmarklet-based password manager, providing an on-line password management service that helps users remember passwords of websites. Users need to first store their passwords on the MashedLife's server. If users want to log into the website, say `www.example.com`, they need to invoke MashedLife's bookmarklet, whose JavaScript retrieves the password stored in

MashedLife's server and fills in the log-in information to the web page.

More specifically, MashedLife's bookmarklet first checks the current web page's location by calling the JavaScript function `location.href.toString()`. The location is used to construct a request for a piece of external JavaScript from the MashedLife server, which fills the user's log-in information (user-name and password) encoded in the script.

An attack on the bookmarklet has been reported in Adida et al. [1]. In the attack, a malicious web page at `http://www.malicious.com` overrides the native `toString` function of the `String` object with its own function:

```
String.prototype.toString = function() {
  return "https://www.example.com";
}
```

When MashedLife's bookmarklet checks page location of the malicious page, its call to the function `location.href.toString()` is answered by the function defined on the malicious page, which returns the string `https://www.example.com`, instead of the location string of `http://www.malicious.com`. In this way, attackers can pretend to be any victim web page and retrieve its password stored in MashedLife's server.

```
<script
  src='http://untrusted.com/ulib.js'>
</script>
...
<script>
  result = function_in_ulib();
  ...
  if (location.href.toString() == "http://public.com") {
    ...}
  ...
</script>
```

Figure 3.1: An example of using a JavaScript library in web applications

Untrusted Third-Party Content Attack Third-party JavaScript libraries are commonly used in web applications. Figure 3.1 is an example of using JavaScript library in a web application, which shows scripts on a web page from `http://public.com`. It uses a JavaScript library `ulib.js` hosted on `untrusted.com`. On the page, there is another piece of JavaScript, which calls `function_in_ulib` in the JavaScript library `ulib.js`. This piece of JavaScript is shared among several pages and behaves differently on different pages. It checks its location through `location.href.toString()`.

According to the same-origin policy, the library `ulib.js` runs in the origin of the web page at `public.com`. Therefore, the script gets access to all resources on the page. The threat from the untrusted JavaScript can be partially mitigated by existing solutions to provide fine-grained access control to host objects. For example, it can disallow the untrusted script to modify the body of the web page. However, in practice, without fine-grained access control to the JavaScript context, either the JavaScript library gets isolated from other scripts on the page, which breaks the page functionality, or full access to the JavaScript context is allowed, which is vulnerable to the following attack through the JavaScript context.

If access to JavaScript context is fully allowed, the code in the untrusted JavaScript library `ulib.js` can override the native `toString` function of the `String` object with its own function:

```
String.prototype.toString=function(){
  // code to add the malicious AD content
}
```

When the trusted script calls the `location.href.toString()` function to retrieve the location of the web page, its call to `location.href.toString()` is answered by the function defined by the untrusted JavaScript library, which for example, can add malicious contents to the page.

Summary From the above examples, we can see the underlying problem behind them is insufficient behavior control inside the JavaScript environment. Existing solutions are either coarse-grained or only mediate access to host objects, whereas objects in JavaScript context are still unprotected. We analyze the JavaScript environment and propose a general framework in the JavaScript environment that allows a web application to separate the privileges of included content and enforce fine-grained access control decisions on web application resources.

3.2 Background

To better understand the attacks and existing solutions, we need to understand the JavaScript environment in which JavaScript gets executed in web browsers. Figure 3.2 illustrates the components of a JavaScript environment. We define JavaScript environment to mean *JavaScript engine* and *host objects*. The *JavaScript engine* contains an execution module and provides *JavaScript context* for JavaScript. JavaScript runs in the execution module of the JavaScript engine, which has access to the JavaScript context and the host objects [61]. The JavaScript engine executes JavaScript. The

JavaScript context contains two types of objects, *native objects* and *custom objects*. Native objects (a.k.a., built-in objects) are defined by the JavaScript standard, such as `Date`, `String`, and etc. Custom objects are defined by a JavaScript code, such as local variables and functions. The host objects are provided by the hosting application (for example, the web browser) of the JavaScript engine for accessing peripheral resources outside the JavaScript engine. Host objects includes Document Object Model (DOM) objects, DOM APIs, XMLHttpRequest (XHR) objects, cookies, etc.

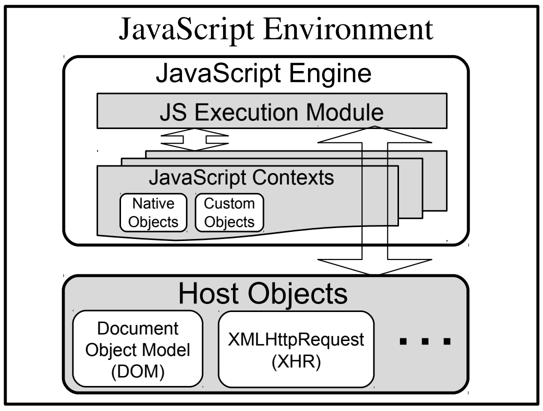


Figure 3.2: Components in a JavaScript Environment

Under SOP, all scripts in a web page share the JavaScript context and host objects. Some of these include scripts in a web page that are untrusted and can become malicious in the future. Therefore, untrusted JavaScript in a web page must run in a controlled environment to regulate its interaction with trusted contents on the web page. For example, in targeting advertisements, advertisement scripts analyze publisher's web page content to display relevant ads to the user. Therefore, to allow targeting advertisement entire web page contents are open to advertisement scripts. Some portion of web page may contain sensitive information such as user credentials or CSRF (cross-site request forgery) token. Advertisement scripts could read sensitive information on a web page, which should be protected while allowing ad scripts to read other contents to serve targeting ads. Therefore, host objects need a fine-grained access control mechanism to protect the integrity of a web application while allowing legitimate functionality.

Similarly, the overly restrictive policy that blocks a certain JavaScript feature affects normal functionality of legitimate web applications. For example, JavaScript allows functions to be defined and assigned during execution. This ability to dynamically override functions is commonly used in web applications to change the JavaScript's environment. For example, there are JavaScript libraries to mask differences in various browser implementations by overriding JavaScript functions provided by the browser, so that web applications can be coded in a browser-independent manner. JavaScript function overriding can also be used to temporarily fix browser implementation bugs [156].

However, attackers can exploit this feature to inject code into benign JavaScript and carry out malicious activities with escalated privilege [1, 22, 113]. Finer-grained access control to host objects in web pages [65] and browser extensions [81] prevents untrusted JavaScript code from performing dangerous operations. However, host objects are only part of a JavaScript environment. If the access to JavaScript context is not controlled, malicious JavaScript can dynamically inject code into benign JavaScript to perform dangerous operations. This weakness is illustrated by a recent attack on bookmarklet-based password managers [1]. A class of research efforts [9, 53, 66, 74, 82, 101, 118, 130, 141]. prevents malicious JavaScript from accessing objects from an origin by excluding unwanted JavaScript from the JavaScript environment. Other approaches [2, 45, 62, 88] allow third-party JavaScript to be included in a JavaScript environment, but the functionality of third-party JavaScript is limited. However, mediation to object accesses in JavaScript involves fine-grained control in both JavaScript context and host objects. Unfortunately, none of the existing approach provides fine-grained control in the JavaScript context.

As our solution, we present JavaScript Environment Reference Monitor (JERMonitor), a reference monitor in the JavaScript environment that provides the necessary fine-grained access control mechanism for host objects and JavaScript context. It allows privilege separation of scripts within an origin according to their trustworthiness and enforces access control rules.

3.3 Our Approach

The goal of our approach is *to provide fine-grained access control in JavaScript environment.* More specifically, our approach protects the execution integrity of trusted JavaScript, i.e., to prevent untrusted JavaScript from changing the behavior of trusted JavaScript by updating the JavaScript execution context. To achieve this goal, our approach mediates all accesses to host objects as well as to objects in the JavaScript context, and prevents malicious JavaScript from affecting the integrity of other JavaScripts running in the same JavaScript environment.

Figure 3.3 shows the design of JERMonitor. It extends the JavaScript engine to mediate access to host objects and objects in the JavaScript context. Privilege separation and enforcement of fine-grained access controls on host objects are provided by a host object access monitor and JavaScript context objects are protected by JavaScript context shadow monitor (JCShadow).

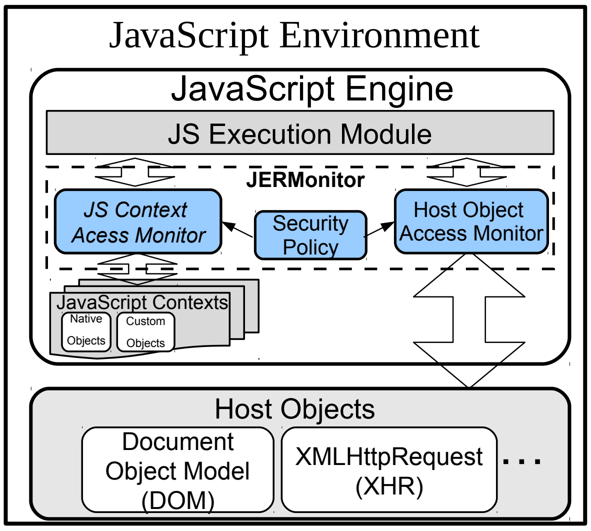


Figure 3.3: Overview of JERMonitor. It extends the JavaScript engine to support privilege separation within an origin.

3.3.1 Mediating Host Objects

JERMonitor provides a fine-grained access control to host objects. More specifically, the *Host Object Access Monitor (HOAM)* component of JERMonitor intercepts all access to host objects by JavaScripts and regulates access with the help of security policy. The HOAM enforced by JERMonitor is based on a ring based protection mechanism, where web publishers or users are allowed to specify rings or groups according to their trust towards JavaScripts included in a web page. Web publishers also need to provide a ring or group based assignment for all elements in a web page. In JERMonitor, when a script asks for access to host objects, the HOAM takes allow or deny access decisions based on the ring or group assignments and the security policy. In essence, using rings or groups containing JavaScripts carrying different levels of trustworthiness, JERMonitor effectively isolates the privileges of JavaScripts embedded in a web page, and enforces a security policy over JavaScript's ability to perform operations (such as read, write, and invoke) on host objects. Section 3.4 describes security policies in detail.

An example of a group assignment HTML `div` tag is used to specify visual style information for a group of HTML tags. We use a `div` tag to separate the privileges of web contents and to facilitate web applications in specifying the trustworthiness of web contents. We introduced a new attribute `group` to the `div` tag. The `group` attribute specifies a level label to all HTML elements within the scope of a `div` tag. Groups can be labeled from 0,1,...,N, where N is depends on the number of groups the web application wants to create. The lower number in the `group` attribute value indicates a higher privilege and a larger number in the `group` attribute value indicates a lower privilege.

```
<html><body>

  <!-- REGION A -->
  <div id='div_A' group=0>
    <script>
      ...
    <script>
  </div>

  <!-- REGION B -->
  <div id='div_B' group=3>
    <script>
      ...
    </script>
  </div>

</body></html>
```

Figure 3.4: An example of host object privilege separation

Figure 3.4 shows an example of host object privilege separation. Web publishers specify regions into a web page according to the trustworthiness they have towards the web content. The host object access monitor honors the groups created by the web publisher in a web page according to the trustworthiness of the web content and enforces security policy on each object access.

However, control of access to host object alone is not enough. An attacker can bypass the host object access control by modifying variables or functions of more trusted code during runtime. Figure 3.5 shows an example of malicious scripts breaking the host object protection by overwriting the JavaScript function of the higher privilege code. The example web page *P* is divided into two regions, *A* and *B*. The region *A* is assigned to level 0, and the region *B* is assigned to level 3, so the JavaScript in region *B* cannot access the DOM elements from region *A*. For example, the function `bar` will not succeed. However, the JavaScript in *A* has full access to A. To bypass the access control enforced in [65], *B* may include JavaScript that overwrites the function `foo` defined inside *A* to `bar`. Then whenever the function `foo` is invoked, it is actually the code of `bar` that gets executed with the group number of 0.

From the above example, we can see that regulating the access only to host objects is not enough, as untrusted JavaScripts can escalate their privileges and access host objects by overriding the logic of trusted JavaScript during run time. Hence, we also need to ensure execution integrity in JavaScript contexts. However, web applications have several resources that require sharing, such as server-side data, JavaScript libraries, etc. Similar to host object access control, simply allowing or disallowing access to JavaScript objects in JavaScript context affects the normal functionality of legitimate web applications. For example, JavaScript libraries are commonly used in web ap-

```
<html><body>
  <!-- REGION A -->
  <div id='div_A' group=0>
    <script>
      function foo(a, b) { return a + b; }
    </script>  </div>


  <!-- REGION B -->
  <div id='div_B' group=3>
    <script>  function bar() {
        document.getElementById('div_A').
          innerHTML = "..some..content..";
      }
      ...
      foo = bar;
    </script>  </div>


  <!-- REGION A -->
    <div id='div_A2' group=0>
    <script>
      var c = foo(5, 10);
      ...
     </script>  </div>
</body></html>
```

Figure 3.5: An Example of a host object access control bypass

plications to smooth out browser differences or fix browser bugs by a runtime function overriding. Similarly, a web application that shows real estate prices on the map needs sharing of JavaScript objects between the web application and third-party mapping service. Therefore, to support less restricted JavaScript functionality as well as to allow fine-grained regulated sharing of JavaScript objects, our approach creates shadow contexts in a JavaScript context.

3.3.2 Mediating Objects in a JavaScript Context

The *JavaScript Context Access Monitor (JCAM)* component of the JERMonitor mediates access to objects in the JavaScript context. Rather than simply allowing or denying access to JavaScript objects in a JavaScript context, JCAM creates *shadow contexts* to support less restricted JavaScript functionality. A shadow context is an isolated copy of the JavaScript context. Each shadow context contains a separate copy of native JavaScript objects such as Date, String, Math, etc. The idea of the shadow context is similar to Codejail [157] which isolates untrusted libraries to control the privileges of libraries. Shadow contexts are created according the rings or groups specified by web publishers for untrusted scripts in a web page. By default, the JavaScript associated with one shadow context is only allowed to get access to objects in its own shadow context. However, JERMonitor allows web publishers to specify permissive security policy to allow JavaScript to have regulated access to objects in other shadow contexts in the same origin. The JCAM integrates shadow contexts according to user-specified security policies and presents a single view of JavaScript context to JavaScript in the execution module.

In browsers, a unique JavaScript context is typically assigned for each web page. JavaScript contexts are isolated from each other to enforce the all-or-none same-origin policy: JavaScript code running in one context is unable to get access to objects defined in other contexts. However, once a piece of JavaScript is allowed to execute in a context, it has full control of all objects in that context. To provide finer granularity of access control to JavaScript contexts, JERMonitor creates *shadow contexts* to further isolate JavaScript in the same JavaScript context. Each shadow context is conceptually a dedicated subset of the original JavaScript context with all the objects created or updated in the corresponding JavaScript group. As a result, objects in the original JavaScript context are now separated into different shadow contexts according to the way JavaScript is grouped. Native objects accessible in the original JavaScript context are provided for each derived shadow context. Custom objects created by the JavaScript are created only in the shadow context of the JavaScript to which it belongs.

JERMonitor allocates a shadow context for each group and runs scripts in its own dedicated shadow context. When scripts access JavaScript object, if the requested object exists in the requester's own shadow context, the JCAM directly allows the access. Otherwise, the JCAM uses security policies to choose objects in the shadow contexts. When the JavaScript in a shadow context P requests to get access to custom objects Q defined in another shadow context R, the JCAM can abort the access if it is denied by security policy or creates a copy of the object in shadow context P. The security policy specifies, when more than one instance of the object Q is found in another shadow context, which one to select for current access. When JavaScript is running in a shadow context i request to access an object not available in its own shadow context, the JCAM looks at shadow contexts in sequence from high to low. If it finds the object in the shadow context k, it consults the security policies and takes action according to the security policies. For example, it either disallows access to the object or creates a copy of the object in the shadow context of the script that requested access to the object. If JCAM finds the object in shadow context k, but the object is disallowed to be accessed by the script running in shadow context i, then the context integrator keeps looking for an object in next shadow context, until it cycles through all the shadow contexts.

Next, we present sample security policies to show how JERMonitor prevents different attacks.

3.4 Security Policies

Security policies focus on protecting the execution integrity of trusted JavaScript against various threat scenarios and separating the code privileges. JERMonitor is configured and hardcoded with the following two security policies to protect execution integrity.

Sample security policy 1, Horizontal isolation

When multiple JavaScript libraries are included in the same web page, they may end up conflicting with each other, if they override JavaScript objects shared by other scripts on the page. With this policy, JERMonitor ensures that the scripts cannot interfere with one another.

JERMonitor prevents the JavaScript from one region accessing or modifying the host object belonging to another region. It only allows JavaScript to gain access to host objects belonging to its own region. Moreover, it also controls access to XHR, cookie objects by JavaScripts.

Under this policy, the JCAM component of JERMonitor allocates a shadow context to each JavaScript library. When the JavaScript in one shadow context attempts to add new properties to native or custom JavaScript objects or to override certain objects, the modification occurs locally in its own shadow context and is not visible to other shadow contexts. Therefore, each script running in a particular shadow context uses its own native and custom objects. This security policy isolates the execution of different JavaScript libraries and does not allow any sharing of native or custom

Access Type	**Source Trust**	**Dest. Trust**	**Action**
Read, Invoke	Low	High	Disallow
Read	High	Low	Allow
Invoke	High	Low	Degrade trust
Write	Low	High	Allow-In-Isolation
Write	High	Low	Allow

Table 3.1: The security policy 2 for host and custom objects

JavaScript objects between them.

To maintain compatibility with current web applications where scripts on the web page carrying different trust levels need to collaborate with each other, we discuss the Security Policy 2 below.

Sample security policy 2, Ring-based access control in JavaScript context

This security policy of JERMonitor provides fine-grained access control to the Host objects similar to ESCUDO.

JERMonitor uses a ring-based access control framework to regulate access to host objects. By default, all scripts in one web page run in the same JavaScript context and have access to all host objects on that page. To prevent untrusted JavaScript included in a web page from arbitrarily accessing the host objects, JERMonitor divides the web page into groups according to the trust levels on page elements. JavaScript in one region can only access regions with higher group numbers, i.e., lower trust.

The policy is shown in Table 3.1. The JavaScript on a web page is assigned to groups based on its trustworthiness. The policy decides the action by the type of operation, the trust of the source JavaScript, and the trust of the destination object. Generally speaking, this policy only allows JavaScript with higher trust to access objects with lower trust. JavaScript with lower trust is not allowed to read objects with higher trust or invoke functions with higher trust. However, if the JavaScript with lower trust overwrites objects with higher trust, JCAM allows the write operation in the JavaScript's own shadow context ("allow-in-isolation"). Read and write operations from scripts with higher trust to objects with lower trust are allowed. A special treatment is provided for invoking lower trust functions. To prevent lower trust functions from accessing the JavaScript context with higher trust, the invoked function runs with temporarily degraded trust, i.e., in the functions own shadow JavaScript context. The parameters of the function are explicitly copied into the function's shadow context. By allowing function invocations with degraded privilege, it allows legitimate use cases in the mashup scenario where JavaScripts with higher trust may need to invoke the functions of lower trust. For example, the integrator (i.e. hosting web page) may invoke function of a third-party gadget integrated in the web page to initialize it. With privilege degradation for function invocations, this security policy supports legitimate functionality.

With this policy, JavaScript with lower trust is not allowed to modify the functions defined in JavaScript with higher trust. Unless JavaScript with higher trust intentionally reads lower trust data and turns them into code via `eval`, etc., this policy guarantees the integrity of JavaScript with higher trust and enforces the ring-based policy as well. For this reason, this security policy currently disallows `eval`.

3.5 Implementation

We implemented JERMonitor in Mozilla Firefox 3.5 and Chromium 16.0.884.0 (Developer Build 101476 Linux).

3.5.1 Mediating Access to Objects in JavaScript Context

Firefox Implementation In the SpiderMoneky JavaScript engine in Firefox, `JSRuntime` is the top-level object that represents an instance of the JavaScript engine. The JavaScript context, `JSContext`, is a child of the `JSRuntime`. In Firefox, JavaScript from the same origin runs in the same `JSContext`. Each `JSContext` contains a global object that holds pointers to all variables and functions accessible from that context.

For example, the `toLowercase()` function is a property of the `String` object, which is in turn a property of the global object. As a script runs within a `JSContext`, all the newly created global functions and variables will be added as properties of the global object of that `JSContext`.

JERMonitor extends SpiderMonkey with three main components: assigning shadow contexts to JavaScript, tracking JavaScript in shadow contexts, and enforcing the access control policy. JERMonitor relies on web application developers to divide the web page into regions and assign each region with a shadow context ID. JERMonitor keeps track of the JavaScript shadow context ID of the script running in the JavaScript engine and uses it to enforce policies. The JavaScript interpreter in Firefox 3.5 has 234 bytecodes for different JavaScript operations. For example, bytecode `JSOP_DEFFUN` for function object creation and bytecode `JSOP_DEFVAR` for variable creation. JERMonitor intercepts access to objects in the SpiderMonkey's interpreter by intercepting object creation, function invocation, and object property `setter`/`getter` operations.

When an object gets created by JavaScript, JERMonitor marks it with the shadow context ID of the running script. In Firefox, all global functions and variables are properties of the *global object*. If a script modifies an object, it actually modifies the property of the object's parent object that eventually triggers the set property operation in the JavaScript engine. In the property `setter` or `getter`, if the operation is to get access to an object that belongs to a different shadow context, the context integrator consults security policies to decide the appropriate operation. JERMonitor also instruments the function invocation in the JavaScript engine. Function invocations are handled in a similar way as the object property `getter` by the context integrator.

Script Execution Isolation in Chromium We also implemented an alternative prototype in the Chromium browser. JERMonitor extends Barth et al.'s isolated worlds implementation to isolate extensions' scripts for each web page. Isolated worlds isolate extension scripts into different JavaScript contexts by providing a separate copy of custom and native JavaScript objects to the scripts, while allowing host objects to be shared among different JavaScript contexts. Implemented with isolated worlds mechanisms, JERMonitor allocates shadow contexts to JavaScript based on regions and maintains the mapping between each script and its corresponding JavaScript context where it gets executed.

Sharing between JavaScript contexts are controlled by security policies. If the script from context 1 needs to get access to a custom object from context 2, and if the security policy allows the script in context 1 to access an object from context 2, then we temporarily inject the object into context 1 for the script to access it. Such temporarily injected objects are removed after the current script finishes execution.

Dynamic Script Introduction Assigning the correct shadow context IDs to dynamically generated JavaScript is critical in JERMonitor. Otherwise, JavaScript in web pages may easily escalate their privileges by injecting code into DOM elements. Hence, JERMonitor assigns dynamically generated JavaScript the shadow context ID of the script that creates it. In practice, third-party JavaScript makes use of DOM interface functions such as `setAttribute`, `createElement`, `addEventListener`, etc. to attach a piece of JavaScript code into an HTML element event handler (for example, `<a onclick='maliciousfun()'/>`) or to create a new script element dynamically using `document.write`, `document.writeln`.

We implement attribution to the original JavaScript by simulating the call stack functionality. We intercept the JavaScript engine before it invokes DOM functions, and record the current shadow context ID. After the DOM function call returns, the shadow context ID gets cleared. We also intercept functions that may bring in new JavaScript, and check whether the JavaScript shadow context ID is set. If so, and if the DOM function is a call from user JavaScript, we assign the newly generated JavaScript with the shadow context ID stored.

More specifically, we intercepted three DOM interface functions namely, `setAttribute`, `createElement`, `addEventListener` to propagate the shadow context ID of the script that invoked it into the modified or newly created DOM element. We also intercepted 22 standard DOM events and executed them in the correct shadow context, according to DOM element group ID.[1] Moreover, we also intercepted `document.write`, `document.writeln` DOM interface function calls and propagated the shadow context ID of the script that invoked these functions into newly created script nodes.

Mediating Flash Script Injection Mediating access to web application resources by injected JavaScript using flash application is crucial in JERMonitor. Otherwise, malicious JavaScript could escalate their privileges by injecting it through flash applications. Script injection by embedded flash applications in a web page is controlled by using a property named `allowScriptAccess`, which either allows the flash application to inject script into the web page's execution environment or prohibits script injection into the web page's execution environment. If script injection is allowed then the injected code is by-default granted with the full privileges to access host objects and objects in the web page's JS context.

To run the JavaScript code injected by a flash application with the privileges assigned by web developers to the flash application, JERMonitor intercepts the flash and web page communication. More specifically, we intercepted functions in web browsers that take data from flash. Flash applications can send data to the web page using ActionScript functions such as *getURL*, *navigateToURL*, and *ExternalInterface.call.* These three ActionScript functions are used by flash applications to inject scripts into a web page. The JERMonitor intercepted at the `GetURL` function of `nsPluginInstanceOwner` class that takes data from the flash sent using *getURL* and *navigateToURL* ActionScript methods and `_evaluate` function of the *nsNPAPIPlugin.cpp* file that executes data injected by flash in the web page using *ExternalInterface.call* ActionScript method. JERMonitor retrieves the group ID of the flash application and executes the injected script with the privileges of the flash application group.

In addition, the *fscommand* function of ActionScript lets a flash application to communicate with a JavaScript in a web page. In a web browser, fscommand function invokes the JavaScript function `moviename_DOFSCommand()`, which resides in the web page that contains the flash application. The *moviename_DOFSCommand* function resides in the web page JavaScript which is already marked by web developers with a group id, JERMonitor executes the *moviename_DOFSCommand* function in the shadow context of JavaScript in which it is defined.

[1]Standard DOM events are click, mouseup, mousedown, mouseover, mouseout, mousemove, dblclick, keydown, keyup, keypress, focus, blur, select, change, submit, reset, load, unload, abort, error, resize, scroll.

3.5.2 Mediating Access to Host Objects

JERMonitor regulates access to host objects using fine-grained security policies. Web developers partition a web page's contents into several groups according to their trustworthiness. Script from one group by-default can get access to host objects in its own group, however, it can only access objects from other groups if the security policy permits it. JERMonitor intercepts method calls to host objects. To decide whether to allow access to host objects or not, JERMonitor, makes an extra call to the target host object to retrieve its group ID. JERMonitor allows access to objects if the ID of the script requesting access is same as the target host object, otherwise, it checks the security policy. If the security policy permits access then JERMonitor returns a host object node, otherwise it returns null to the requesting script.

3.5.3 Configuration Files of Shadow Contexts and Security Policies

JERMonitor uses XML-based configuration files to specify the grouping of JavaScript on web pages, assign shadow context IDs to them, and specify security policies. We also implemented a GUI tool that is integrated into web browser preferences' dialog to help users specify configurations. The configuration files can also be specified by web developers. In contrast to using markups in a web page, the external configuration files make it flexible to adapt the context ID assignment to JavaScript environments beyond web applications, such as browser extensions.

A sample XML-based security policy configuration file is shown in figure 3.6. Lines 2-22 specify the security policy for the JavaScript library `ulib.js` loaded from `http://untrusted.com` domain. The shadow context ID for the script is defined on Line 3. The access control rules that are in force when low trust JavaScript tries to gain access to high trust JavaScript are specified on Lines 5-9, and Lines 11-15 specify access control rules for regulating high trust JavaScript access to low trust JavaScript. Lines 6-8 and Lines 12-14 specify operations such as `read/write/invoke` and action to be taken on those operations such as, `Allow/Allow-In-Isolati` `trust/Disallow`. Lines 16-21 specify restriction to the properties of the JavaScript library `ulib.js`, i.e. it can not modify or access `cookie` and `domain` properties of the `Document` object.

We implemented an interpreter to analyze the XML-based security policy file. We used *libxml2* XML parser APIs written in C to interpret the security policy file. JERMonitor reads the security policy file only once and stores the information globally in its internal state. To prevent tampering of the security policy file from scripts in a web page, JERMonitor does not expose it to scripts in a web page.

```
<secPolicy>
  <host src="http://untrusted.com/ulib.js>
    <sctxid>1</sctxid>
    <!-- Low trust JS access High Trust -->
    <LowToHigh>
      <read>Disallow</read>
      <invoke>Disallow</invoke>
      <write>Allow-In-Isolation</write>
    </LowToHigh>
    <!-- High trust JS access Low Trust -->
    <HighToLow>
      <read>Allow</read>
      <invoke>Degrade trust</invoke>
      <write>Allow</write>
    </HighToLow>
    <ObjectBlackList>
      <object name="Document">
        <property>domain</property>
        <property>cookie</property>
      </object>
    </ObjectBlackList>
  </host>
</secPolicy>
```

Figure 3.6: A sample XML-based security policy

```
SecurityPolicy: <Policy>
                <secPolicy>.....</secPolicy>
```

Figure 3.7: An example of a policy sent with the SecurityPolicy HTTP response header

JERMonitor uses the `SecurityPolicy` custom HTTP response header in the HTTP response to transfer the security policy file to the web browser. The `SecurityPolicy` header contains a tuple (<type, data>), where "type" indicates whether the security policy is embedded in the header or available as an external resource. Figure 3.7 exemplifies the header where the security policy file is embedded in the HTTP header, while Figure 3.8 illustrates the header when the security policy file is an external resource, where its URL is given. If the security policy file is available as an external resource then JERMonitor first fetches the security policy file and halts the parsing of the web page until the security policy file gets downloaded.

3.6 Evaluation

We evaluated JERMonitor with real-world examples to ensure that it is able to protect the JavaScript context. We also measured the runtime overhead of JERMonitor, and tested its compatibility with

```
SecurityPolicy:
    <url>
     http://www.example.com/policyfilename.xml
```

Figure 3.8: An example of the SecurityPolicy HTTP response header to indicate that the policy is available as an external resource

Operation	Unmodified Browser (ms)	Time With JERMonitor (ms)	Time With Web Sandbox (ms)	JERMonitor Overhead (%)	Web Sandbox Overhead (%)
Function Invocation	8.4808	11.0522	95.153	30.32%	1021.99%
Get Object Member	5.4962	7.9173	94.6788	44.05%	1622.62%
Set Object Member	7.128	10.0705	95.1626	41.28%	1235.05%
Invoke Object Member	9.3994	12.6262	95.488	34.32%	915.9%

Table 3.2: Performance of our solution for basic operations. Time in first three columns above is measured in millisec.

real-world websites. Our experiments were performed on a computer with an Intel Core 2 Duo 2.33GHz and 4GB RAM, running Ubuntu 9.10.

3.6.1 Effectiveness

We tested JERMonitor using a web mashup example, as well as attacks to bookmarklets where native JavaScript functions were overwritten by malicious web pages.

`Bookmarklet Attack:` A bookmarklet is a piece of JavaScript saved as a bookmark entry in the browser's bookmark menu. When activated by a user click, the JavaScript code in the bookmarklet executes in the JavaScript context of the current web page. MashedLife [77] is a bookmarklet-based password manager, providing an online password management service that helps users remember passwords of websites. Users need to first store their passwords on the MashedLife's server. If they want to log into some website, say `www.example.com`, they invoke MashedLife's bookmarklet, whose JavaScript retrieves the password stored in MashedLife's server and fills in the log-in information to the web page.

More specifically, MashedLife's bookmarklet first checks the current web page's location by calling the JavaScript function `location.href.toString()`. The location is used to construct a request for a piece of external JavaScript from the MashedLife server, which fills the user's log-in information (user name and password) encoded in the script.

An attack on the bookmarklet has been reported in Adida et al. [1]. In the attack, a malicious web page at `http://www.malicious.com` overrides the native `toString` function of the `String` object with its own function:

```
String.prototype.toString = function() {
  return "https://www.example.com";
}
```

When MashedLife's bookmarklet checks the location of the malicious page, its call to `location.href.toString()` is answered by the function defined on the malicious page, which returns `https://www.example.com` instead of the actual location `http://www.malicious.com`. The bookmarklet then retrieves the user's password of `example.com` and fills it into the malicious page, which is accessible by the attacker.

With JERMonitor enabled, using the security policy 2, the web page and bookmarklet runs in different shadow contexts, say `WPC` and `BMC` respectively. We assign higher trust to the JavaScript in the bookmarklet's shadow context. Scripts on the web page are allowed to override native JavaScript functions in the shadow context `WPC`. Specifically, the web page script with a shadow context ID of `WPC` is allowed to replace the `toString` function object with its custom function in `WPC`. When the bookmarklet invokes the `toString` function to retrieve the location string of the web page, JERMonitor intercepts the function invocation. Because the call is issued from the more trusted BMC, JERMonitor invokes the native function object provided by the JavaScript environment to `BMC`.

Similarly, we use this security policy to successfully prevent malicious web mashup attacks described in Section 3.1.

3.6.2 Compatibility

To evaluate the compatibility of our sample policies, we tested JERMonitor on 25 top websites listed by Alexa [138] with our security policies. We locally cached the copies of these websites and assigned different shadow context IDs for web page scripts and third-party JavaScript libraries. We whitelisted the content distribution networks (CDN) and sub-domains of websites to run scripts from sub-domains and CDNs in the same shadow context as the original websites.

The security policy 1 is compatible with 16 websites out of 25 websites evaluated. The broken functionality was because these web pages have interactions between scripts on the page and scripts in the libraries. For example, Google Analytics was included in web applications in the forms of both inline page JavaScript and external JavaScript. As a policy which supports sharing across shadow JavaScript contexts, the security policy 2 of the JERMonitor is compatible with 19 of the

Benchmark Name	Original Browser (runs/sec)	JERMonitor (runs/sec)	% Overhead
Dromaeo	9.386	9.06	3.60%
SunSpider	15.938	15.15	5.20%
V8 Test Suite	2.496	2.358	5.85%

Table 3.3: Overhead incurred by JERMonitor on industry-standard JavaScript benchmark

25 websites in our experiments. In this test, we assigned scripts from the web page with high trust, and assigned third-party scripts with low trust. The broken functionality is caused by the use of `eval`, which was disabled by security policy 2. `Eval` were used to deobfuscate JavaScript. If `eval` is permitted for this usage, the sample security policy 2 was compatible with all 25 websites.

3.6.3 Performance Overhead

Overhead incurred on real-world websites We measured the performance overhead of JERMonitor on the 25 different websites top-listed by Alexa. We locally cached the copies of these websites and assigned shadow context IDs for third-party scripts. In the JavaScript engine we measured the total time taken by the execution of all JavaScript in a web page (averaged on five runs). We noted that the overhead ranged from 0.44% to 13.45%, averaged at 3.65%.

JavaScript Runtime Overhead JERMonitor performs access control check on objects in the JavaScript engine. We measured the performance overhead incurred by JERMonitor in the JavaScript engine with basic operation tests and industry-standard benchmarks. We used Mozilla Firefox to measure the performance overhead.

Basic Operation Overhead In JERMonitor, we interposed on object property `getter` and `setter` functions, as well as function object creation and invocation points in the JavaScript engine. In the experiment, we measured the performance overhead incurred by these basic operations. Table 3.2 shows the performance overhead (average computed over five runs) for basic operations against user-defined objects used in a loop for 10,000 iterations. We did this experiment to estimate the worst-case overhead of JERMonitor and compared it with the translation-based WebSandbox [62]. The WebSandbox is a technology for securing web content through isolation. JERMonitor has a much lower overhead of 30 to 40 percent, while WebSandbox's overhead ranges from 900 to 1600 percent.

JavaScript Industry Standard Benchmark We also measured the overhead of JERMonitor on industry-standard benchmarks. Table 3.3 shows the results. On the Mozilla Dromaeo JavaScript benchmark, we observed a 3.60% performance overhead; On SunSpider JavaScript benchmark, we observed a 5.20% performance overhead; And on the V8-test suite, we observed a 5.85% slowdown. As these numbers indicate, JERMonitor incurs low performance overhead on industry-standard benchmarks. The current prototype of JERMonitor uses a suboptimal search operation when searching for variables in shadow contexts, which has room for further improvement.

3.7 Related Work

Access Control in JavaScript Environment Recent work [12, 81] has studied the JavaScript security problem in Firefox extensions, and Ter Louw et al. [81] developed a sandbox solution to secure Firefox extensions. ESCUDO [65] is a fine-grained model to address the similar problem in web applications.

The main focus of these approaches is to prevent untrusted JavaScript from accessing certain *host objects*, while our approach focuses on supporting complete fine-grained access control to the whole JavaScript execution environment.

Secure subset of JavaScript Various object-capability solutions exist that try to restrict the excessive power of the JavaScript language with a secure subset of the original language.

FBJS [45] does not require external JavaScript to be embedded into iframes, and appends any identifiers fed to it with the application's ID. This creates a separate virtual scope for every application running in Facebook, where interfaces similar to those in JavaScript are provided with FBJS DOM objects. As these objects only provide limited capabilities, FBJS may mitigate JavaScript security issues, especially those with user privacy concerns.

ADsafe [2] is a static verifier that defines a subset of JavaScript whose safety can be statically verified by tools like JSLint [31]. It removes blacklisted JavaScript features such as global variables, `this` pointer, `eval`, `with`, and provides the safe versions of these excluded functionalities in an `ADSAFE` object. Finifter et al. [48] proposed statically verified JavaScript subset that whitelists known-safe properties using namespaces to mitigate threats from malicious advertisements.

There are also dynamic enforcement solutions, such as Caja [88] and Web Sandbox [62]. Developed by Google, Caja translates the original web content (JavaScript, HTML, and CSS) into a *cajoled* version. During the rewriting process, static analysis is performed to verify security properties, and runtime checks are also added when necessary. A mechanism called *virtual frames* is used to provide tamed DOM APIs that virtualize portions of DOM to allow embedded web applications interaction. Web applications decide which capabilities are exposed to embedded applications.

Web Sandbox [62] from Microsoft provides similar web application transformation to code that would run in the virtual machine that provides an isolated running environment. The virtual machine interacts with the JavaScript and DOM modules of the browser, and uses the policy rules to mediate runtime checks.

These solutions mainly aim at preventing embedded third-party malicious JavaScripts from compromising other portions of web applications. Although Caja and Web Sandbox provide automatic source-to-source transformation, they still restrict the features of JavaScript, which makes them less applicable in the scenario of bookmarklet and Firefox extensions.

JavaScript injection attacks and defenses Cross-site scripting (XSS) attacks are the most serious threats to web applications, which are caused by malicious JavaScript executing in the victim's JavaScript context. This problem has been extensively researched [9, 14, 53, 74, 82, 101, 118, 130, 137, 141, 146]. However, due to the lack of granularity in existing JavaScript context, the main focus of these solutions is to identify untrusted JavaScript and prevent it from being executed. Our approach aims to improve the control granularity inside a JavaScript context, which can be combined with some of the above solutions to enforce more permissive rules for untrusted JavaScript.

Adida et al. [1] reports the bookmarklet attack introduced in Section 3, where the malicious JavaScript code can silently change the execution context of bookmarklets, similar to the rootkit in an operating system environment. It introduces several types of attack techniques, which are different cases of dynamic JavaScript code injection via function overriding. It also provides solutions that have been adopted by bookmarklet developers. Compared to our approach, the solution in [1] tries to detect whether the context has been changed and to reliably find the objects unaffected by attackers, while our approach targets a more general class of JavaScript attacks and focuses on preventing the JavaScript execution environment from being modified.

3.8 Summary

The growing ubiquity of untrusted third-party JavaScript in web applications offers attackers an opportunity to compromise the integrity of web applications and subvert the behavior of web applications. In this chapter, we show that the root cause of this problem is the lack of behavior control in the *JavaScript environment*. By offering the desired fine-grained access control mechanism in the *JavaScript environment* for host objects and JavaScript context, our approach provides separated privilege and least privilege principle support within an origin. Our approach allows privilege separation of scripts within an origin according to their trustworthiness and enforces access control rules.

Chapter 4

Preventing Click Event Hijacking by User Intention Inference

By exploiting the complex features of web applications, attackers can create web page objects to hijack users' clicks. Hijacked clicks may lead to unexpected browser actions, such as visiting phishing websites or sending malicious requests from web applications. The Clickjacking [57,105] attack, also known as UI-redressing, is an example of such an attack. In this attack, attackers carefully craft overlapped layers on web pages to trick users into clicking web page objects without their consent. For example, in the Facebook attack [57], the malicious web page includes an invisible layer, loaded with Facebook's page on top of a game web page. In this way, attackers trick users into clicking objects in the game, but the clicks actually occur on a button in the Facebook page, whose event handler in turn sends requests to share the malicious page with the victim's friends.

As another example, attackers may also use the `onClick` event of a hyperlink to redirect the browser to arbitrary pages. This type of attack is often used to launch phishing attacks, or to open annoying pop-up advertisements. For example, a malicious web page can include a link to a bank website A, but use the `onClick` event handler to redirect the browser to visit phishing site B after users click on the link. By exploiting the cross-site scripting (XSS) [27] vulnerability, attackers can attach `onClick` event handlers to links within trusted websites to redirect users to phishing pages, which helps the phishing site to gain additional trust. If a user hovers the mouse pointer over the hyperlink to the site A, the browser's status bar still shows the URL to the site A. Although not as dangerous as Clickjacking, this type of JavaScript-based click redirection can annoy users, or expose them to malicious sites hosting phishing pages or malware. These attacks generally aim to make users' clicks trigger browser actions that users do not expect. We named this type of attacks *click event hijacking* attacks.

At the time our work was published, few solutions were proposed to detect the Clickjacking attack [111]. The NoScript [106] Mozilla Firefox extension uses a module called ClearClick to protect users against Clickjacking attacks. When a click happens on a web page with embedded elements that are partially obstructed or transparent, ClearClick suspends further actions triggered by the click, and reveals the real click target to users. Balduzzi et al. [8] has developed a Clickjacking detection system and has conducted a large-scale study of Clickjacking. Such solutions specifically target the characteristics of Clickjacking.

We observed a common behavior in the general click event hijacking attack: users are lured to click on page objects that they would not normally click if they knew the resulting browser behaviors. That is, based on the information presented, users, believe the browser will perform certain actions when they click, but the actual browser behaviors are different. In other words, the actual browser behaviors do not match the users' original intentions.

With this observation, we present a novel solution, ClickGuard, to mitigate click event hijacking attacks. Its goal is to ensure that *the web browser's behavior resulting from a click matches the intention of the user*. When users make clicks, ClickGuard infers users' intentions. It then tracks the web browser's behaviors, and ensures the resulting activities match the inferred intentions.

4.1 Motivating Examples

In this section, we discuss examples of different types of click event hijacking attacks.

```
<-- Page from www.Websitename.com -->
<html>
...
<iframe id="victim" src="http://example.com" scrolling="no"
    width="600px" height="600px" style="opacity:0;
    position:absolute; left:10px; top:10px;">
</iframe>
...
<div style = "position:absolute; top:Ypx;left:Xpx;">
    <a href= "http://example.com">Click Here</a>
</div>
...
</html>
```

Figure 4.1: Clickjacking using transparent iFrame and overlay objects.

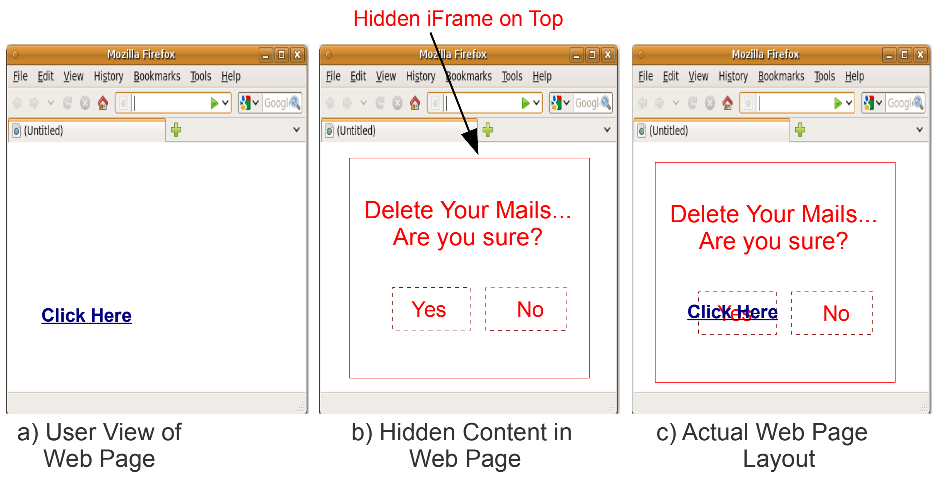


Figure 4.2: Illustration of Clickjacking using transparent iFrame and overlay objects

Clickjacking In Clickjacking attacks [56, 105], attackers exploit the layout feature introduced by iFrames. Specifically, they load a victim web page into an iFrame on the top layer, and make it transparent. Then they load a deceptive page in another iFrame at the bottom layer, to attract users to click. Figure 4.1 shows an example of a Clickjacking attack. The front page of `http://example.com` is loaded inside a transparent iFrame, set to zero opacity value to make it transparent. To lure users to click at a particular location of the page loaded inside the transparent iFrame, the attacker creates a link in the visible bottom layer, which is located exactly at the same position where the attacker wants users to click in the top layer. As shown in Figure 4.1, the attacker specifies the location of a link by setting its X and Y coordinates. When users try to click on the link, they actually click on the transparent layer of the iFrame that is loaded with the page from `example.com`. An illustration of such a Clickjacking attack is presented in Figure 4.2.

Floating objects Alternatively, attackers can put malicious code inside a floating object, and automatically bring that object under the mouse pointer when users hover the mouse pointer over a particular link. Clicking triggers the malicious code in the floating layer. Figure 4.3 shows such an example, where `float_layer` is a JavaScript class defined by the website that is hidden and floating around on a web page. The scenario of a floating object attack is illustrated in Figure 4.4.

The example shown here uses the `onMouseover` event. After `onMouseover` is triggered when users move the mouse cursor over the link, the `float_layer` object is moved to the point of the current mouse cursor, and when users click on the link, they click on the `float_layer` object. Therefore, the `onClick` event of the `float_layer` object is triggered.

Attackers are not limited to hijack `onMouseover`. They can also exploit `onKeyup`, `onKeydown` or other JavaScript event handlers to launch attacks. For a more complete list of exploitable

JavaScript event handlers, we refer readers to [119]. Attackers can also use the floating object concept to make the cursor follow the iFrame on which they want users to click [105].

Pop-up on click By default, web browsers only allow pop-ups triggered by certain user interactions, such as click or double click. Therefore, attackers cannot create web pages that pop up within windows *automatically*. Instead, they have to generate pop-ups when users click on the page. Figure 4.5 shows such an example. When users click on the link that displays its destination as `http://www.example.com`, a pop-up window will be opened. This can also be modified to open pop-up windows only when users click inside a particular area or on a particular object on the web page. In these scenarios, users' original intentions are subverted into opening a pop-up window.

Our observation From the examples above, we can see that the intrinsic property of click event hijacking is the mismatch of user intentions and actions taken by the browser, resulting from client-side scripting and complex page layouts. When users click somewhere on a web page, they believe the browser will perform common and expected actions, such as going to a target web page, or submitting form data to the server. Careful users should check the information provided by the browser before they click, such as the destination URL shown in the status bar when the mouse pointer is over the link. The status bar information is partially protected by web browsers that prevent JavaScript code from changing the status bar information. However, this protection does not prevent the attacks discussed in this chapter. Therefore, even if users check the destinations of the elements they click on, and believe the click will result in expected browser actions, the browser may actually carry out a different action controlled by attackers.

4.2 Background

In the past decade, web technology has evolved rapidly from simple static web pages to complex dynamic web applications. These web applications often use a complex layout, consisting of components or pages from different sources. They also often rely on dynamic features such as JavaScript, a client-side scripting language designed to add *interactivity* to web pages.

Although web pages are more dynamic, their functionality is controlled, to protect websites from being affected by others. Using the same-origin policy [99] and JavaScript sandboxes, browsers only allow JavaScript on a web page to get access to the objects and services of its originating website. Browsers also use user actions, such as clicks, as an indicator of user inten-

```
<-- Page from www.Websitename.com -->
   <float_layer id="layerk"
       onclick="document.location='http://www.malicious.com';"
       style="position:absolute;width:2px;height:2px">
   </float_layer>
   <script type="text/javascript">
       function clickjack(evt) {
         mouseX=evt.pageX?evt.pageX:evt.clientX;
         mouseY=evt.pageY?evt.pageY:evt.clientY;
         document.getElementById('layerk').style.left=mouseX;
         document.getElementById('layerk').style.top=mouseY;
       }
   </script>
   <a href="http://www.example.com"
       onmouseover="clickjack(event)"> Click here </a>
```

Figure 4.3: Floating object example

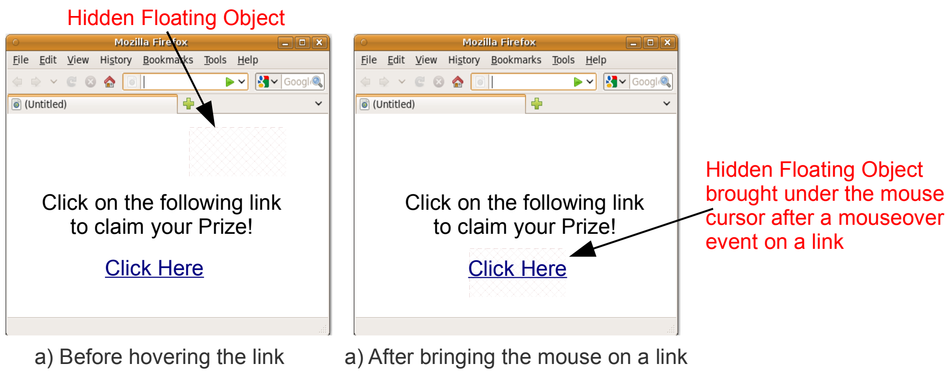

Hidden Floating Object
Mozilla Firefox
File Edit View History Bookmarks Tools Help
(Untitled)
Click on the following link to claim your Prize!
Click Here
Hidden Floating Object brought under the mouse cursor after a mouseover event on a link
a) Before hovering the link
a) After bringing the mouse on a link

Figure 4.4: Illustration of a floating object in a web page

```
  <script Language="JavaScript">
    function popUp() {
       window.open ("http://www.malicious.com","malwin");
    }
  </script>
  <a href="http://www.example.com"
    onclick="javascript:popUp()" > Click here </a>
```

Figure 4.5: An example of Pop-up on click

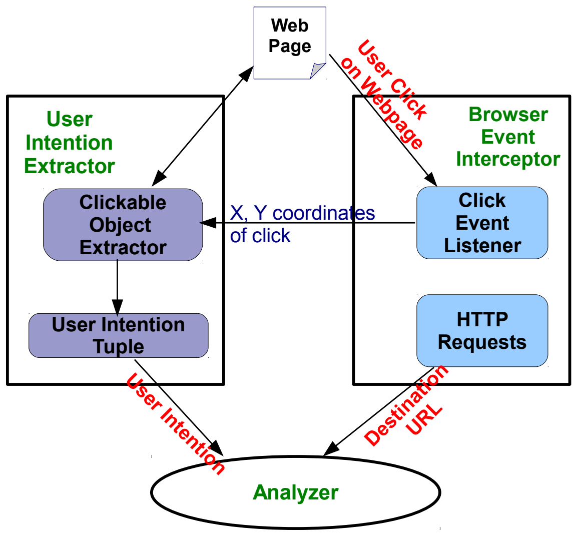


Figure 4.6: Component overview of ClickGuard

tions. For example, most of the web browsers have implemented pop-up blockers, which only allow a window to be opened if it is associated with a user action. As another example, settings of Adobe Flash objects on a web page can only be changed by user clicks.

By exploiting the complex features of web applications, attackers can create web page objects to hijack users' clicks. Hijacked clicks may lead to unexpected browser actions, such as visiting phishing websites or sending malicious requests within web applications. Attackers trick users to click on page objects that they would not normally click if they knew the resulting browser behaviors.

4.3 Design of ClickGuard

The main idea of our solution is to ensure *the browser's action after a click matches the user's original intention*. To achieve this goal, we first need to know what users intend to do. Secondly, we need to know browser's behaviors resulting from the click, which can be intercepted as browser events. With user intentions and the corresponding browser behaviors, we can check whether they match, and report unmatched results to warn users of the potential threat.

4.3.1 Overview of Our Approach

Figure 4.6 presents the component overview of ClickGuard. It has three main components: *browser event interceptor*, *user intention extractor*, and *analyzer*. The *browser event interceptor* intercepts important browser events, such as JavaScript events and HTTP request events. When a click event is intercepted, the *user intention extractor* infers the user intention of the click from web page objects in the browser window, and associates the inferred user intention with the click event. When the browser is about to perform important actions, the browser event interceptor finds the corresponding click event and its associated user intention, and asks the *analyzer* to match the extracted user intention with the suspicious browser event. If they do not match, the analyzer reports an attack and triggers actions specified by users, such as displaying an alert.

4.3.2 Intercepting Browser Events

This module intercepts two main types of browser events: input events and output events. Input events are those corresponding to user actions in web applications, such as user clicks, which reflect user intentions. Output events are events that can modify web applications' states or transfer data to external parties, such as HTTP requests.

The browser event interceptor intercepts input events by registering listeners for JavaScript events related to user actions. For example, when a user click happens on a web page, the `onClick` JavaScript event occurs, and notifies the corresponding event listener. Then, the event listener may invoke user-defined *event handlers*, which are JavaScript functions that have the access to event properties, such as, *type, target, pageX, pageY, screenX,* and *screenY*, etc. *Type* indicates the type of the event; *target* indicates the object to which the event is originally sent; *pageX, pageY, screenX,* and *screenY* represent the cursor location at the time the event occurs. *Event handlers* can perform tasks such as modifying the content of their web pages or sending out HTTP requests.

The output events are HTTP requests, the main interface for web applications to communicate with the outside world. ClickGuard intercepts HTTP requests just before they are sent out.

When the intercepted output event is triggered, ClickGuard starts the detection process. It finds the input event that triggered this output event and invokes the analyzer. Next we will discuss how to infer user intentions from input events, and the details of correlating input events to output events are explained in Section 4.3.4.

4.3.3 Inferring User Intentions

After an input event is intercepted, ClickGuard infers the user intention for this event before the browser starts processing the event. This task must be performed immediately because the browser environment may change during the processing of the event, thus making users' original intentions harder to determine.

Intuitively, if users click on a multi-layered region on a web page, the object on the visible layer is what they want to click. For example, if they click in a single-layered region, the object clicked is what they want to click. Next, we use the object on which users want to click and its attributes to infer users' intentions.

In ClickGuard, the user intention is defined as a tuple <Target Object, Destination URL>, explained as follows:

- "Target Object" is a clickable object found under the mouse pointer when a click happens, and so is considered the target object of the click. Clickable objects are either hyperlinks or call-to-action buttons. Hyperlinks include HTML elements `<a>` and `<link>`. The appearance of hyperlinks depends on the HTML content embedded in these hyperlinks, such as text, images, etc. Other HTML elements, such as `<div>` or, `<b>` are considered non-clickable objects, and are not considered as user intentions, because they do not reveal any destination information to the user. We believe that it is not a good practice to use non-clickable objects as links, since this would confuse users in their understanding of the potential behaviors of web page objects.
- "Destination URL" is the URL of the inferred destination of the Target Object.

When a user clicks on the web page, ClickGuard stores the user intention tuple that is later fed into the analyzer component for matching the user intention with the actual browser behavior. To compose the user intention tuple, we need to obtain the Target Object as well as the Destination URL. The main challenge of this step is to deal with the complexity of page layout. Next we explain how ClickGuard finds the Target Object when there are multiple layers, and how the Destination URL is inferred from the Target Object.

Finding the Target Object from multiple layers If there are multiple layers on a web page formed using Frames or iFrames and there are multiple objects under the mouse pointer when the click happens, we find the Target Object based on the visibility of the objects under the mouse pointer. A naive approach to prevent attackers from exploiting users using complex layout is to disallow clicks on transparent layers. However, that approach would prevent legitimate web applications from using complex layouts. The aim of ClickGuard to match browser actions with the user intention, rather than simply disallowing clicks on the transparent layers. Our approach determines the visibility of layers and objects based on their opacity values. The *opacity* attribute [142] is used by the Cascading Style Sheets (CSS) to adjust the visibility of objects on a web page, where

0.0 denotes fully transparent and 1.0 indicates fully opaque. We consider an object as more related to the user intention if its opacity value is higher. Our approach works as follows:

1. **Collect all objects under the clicked position** We traverse the DOM (Document Object Model) [152] tree to examine the main web page and its iFrame and Frame elements to retrieve all objects under the mouse pointer at the time of the user click.

2. **Check opacity values** If there are multiple layers in the web page and the layer clicked on is transparent, we check the opacity values of all objects collected in the previous step.

3. **Choose the object representing user intention** Here we filter out objects with lower opacity values than our threshold. If none of the objects have an opacity value above our threshold, we record *nil* as the Target Object. If multiple clickable objects have the same opacity value higher than our threshold, we examine their vertical layer order and pick the object on the top as the Target Object.

Obtaining the Destination URL from the Target Object Now we describe the way we infer the Destination URL information from the Target Object. If the Target Object is a link created by an HTML [28] anchor element (`<a>`), then it is of type HTMLAnchorElement, and has a HREF attribute. In this case, the value of the HREF attribute is taken as the Destination URL. For example, in Figure 4.1 and Figure 4.3, when the user clicks on the link, our interceptor stores the user intention information as <Target Object, `http://example.com`> and <Target Object, `http://www.example.com`>, respectively. If the Target Object is a submit button in an HTMLFormElement, it does not have the HREF attribute, but its enclosing HTMLFormElement has an ACTION attribute. We take the value of the ACTION attribute as the Destination URL in the user intention tuple. If users click on blank space or a non-clickable object, then the Target Object is *nil*. We believe that there is no user intention to send a new HTTP request or open a pop-up window. In this case, the Destination URL is also *nil*.

4.3.4 Correlating Output Events to Input Events

To match browser behaviors with user intentions, ClickGuard needs to correlate an output event to the input event (as described in Section 4.3.2) that triggers it. For example, a single click on a web page may be followed by multiple HTTP requests: some of the requests result from the click, while others may be generated by other asynchronous events in the browser. So for each browser action intercepted, we need to figure out whether it is triggered by an input event or not.

If the HTTP request is not triggered by an input event, we should allow those requests, as web pages often send requests automatically to same origins or other domains to load resources (such as images, flash, etc.). If the HTTP request is triggered by an input event, we need to find the user intention information associated with it. In our approach, we leverage the JavaScript call stack to find the association.

In Firefox, JavaScript-to-JavaScript function calls are implemented using a JavaScript call stack [97]. When a JavaScript event occurs, the JavaScript event handler is pushed onto the top of the JavaScript call stack, and the event handler executes the pre-defined JavaScript code inside that event handler. When the execution terminates, its frame is popped off from the JavaScript call stack. If a JavaScript event handler initiates an HTTP request, it does not terminate until the HTTP request is sent to the network. As a result, if there is a JavaScript event handler in the JavaScript stack, it means this HTTP request results from that event handler. If the current JavaScript stack does not have any JavaScript event handler, then this HTTP request is not from client-side scripts and is allowed by ClickGuard. However, one exception is indirectly generated HTTP requests. For example, the click event handler can use the JavaScript `setTimeout()` or `setInterval()` functions to generate timed or deferred execution of specified code. In these cases, when the HTTP request is finally initialized, it is the timeout or interval event handler that is on the JavaScript call stack, not the original click event handler. Our current version of ClickGuard does not handle these cases, and we will discuss more on this issue in Section 4.7.

4.3.5 Detecting and Responding to Attacks

Once ClickGuard correlates an output event to its corresponding input event, it retrieves the user intention associated with the input event and activates the analyzer. The analyzer detects the click event hijacking attacks by two inputs: the user intention and the intercepted output event.

We match user-intended destination against the target in the intercepted output event. We perform two checks between the user intention and the HTTP request output event, using a policy similar to the same-origin policy (SOP) [99].

The first check is between the Destination URL value in the user intention and the destination URL of the HTTP request. The second check is between the URL of the enclosing web page of the Target Object in the user intention tuple and the URL of the web page that initiates the HTTP request. As we mentioned in Section 4.3.3, the URL of the web page enclosing the Target Object is stored as one of its attributes. If either check fails, ClickGuard shows a security warning.

The first check ensures there is no JavaScript-based click event hijacking attacks and the second check is used to prevent layout-based attacks. As seen in the example of Figure 4.1, the attacker creates a visible link on the bottom layer to trick users into clicking on it. However, what is actually clicked is the invisible object in the layer above. In this case, the link in the bottom layer is inferred as the user intention by ClickGuard. An HTTP request is prepared for the object in the top layer, and its destination is matched against the URL extracted from the user intention. Since the link at the bottom layer will never be touched, the attacker can freely set its destination to that of the invisible object above it to bypass our first check. But this attack will be detected in our second check, as the attacker's web page has a different URL than the trusted one embedded in the iFrame.

For HTTP request output events, we use the following check on origins. We define the origin as a combination of the protocol, host name, and port number of a URL, the same as that in the same-origin-policy (SOP) enforced by browsers. We require the two URLs to have the same origin, because naturally the actual destination should not have significant difference from the user intention.

4.3.6 Inferring Host Relationships by Cookie Policy

The criterion above is strict, as it prevents requests from being sent to hosts in different sub-domains of the same domain, which is common in big web applications. For example, a website hosting videos may store videos on its sub-domains to reduce the load on the main web server. To prevent such unnecessary restrictions, we propose a method to automatically infer relationships among hosts within one domain.

HTTP, a stateless protocol, uses *cookies* to track user sessions, to authenticate users to web applications, or to remember custom preferences about users. The contents of cookies are `name=value` pairs [75]. To allow cookies to be sent to sub-domains, a web application sets `domain=<domain-name>` pair in its cookie. When the cookie is going to be sent to a website, the URL of that website is compared with the `domain` attribute of the cookie. If there is a tail match[1], then the cookie can be sent to that sub-domain. For example, if the `domain` attribute has the value `.example.com`, then it is allowed to send cookies to sub-domains such as `first.example.com` or, `second.first.example.com`. However, it is not allowed to send cookies to `example.first.com`. The presence of a leading dot (.) in `.example.com` indicates it is a domain cookie, otherwise it is treated as a host cookie, which is sent back, during subsequent visits, only to the server that sets it. A domain cookie is sent back to any site in the same domain as the site that sets it.

The sub-domain policy in cookies indicates the trust among web servers within one domain.

[1]The value of the `domain` attribute in the Set-cookie header is matched with the fully qualified domain name of the host.

We leverage this information to handle sub-domain communications of JavaScript on legitimate websites to avoid false positives. If the origin check we perform fails, we then check the cookie policy. Specifically, we check the `domain` attribute in the cookie. If it exists, we perform the tail matching between the fully qualified domain name of the HTTP request and the `domain` attribute value in the cookie. If it fails, we report failure of the check; otherwise, the check passes. If the `domain` attribute is not set by the web application in its cookie, we also report failure of the check.

In summary, we relaxed our same origin checking criteria, and allowed access to sub-domains if the cookie policy allows cookies to be sent to them.

4.4 Implementation

To demonstrate a feasibility and effectiveness of our approach, we developed the prototype ClickGuard as an extension [98] of the Mozilla Firefox browser. We chose Firefox as the platform, because it is the most popular open-source web browser. Our approach can be implemented by either modifying Firefox's source code, or extending Firefox's functionality by its extension interfaces. We decided to implement ClickGuard as an extension, as extensions are much easier to deploy and distribute. Meanwhile, the event interception and object access functionality needed by our approach are mostly supported by Firefox's extension interfaces. More on this issue is discussed in Section 4.7.

To intercept input events, ClickGuard intercepts JavaScript events such as `click` and `keypress`. By calling the function `targetObject.addEventListener( eventType, listener, capt)` in Mozilla Firefox, we can add an event listener on `targetObject` which can be an HTML document, window, etc [91]. The first parameter, `eventType`, can be `click`, `keypress`, etc. The second parameter, `listener`, is a JavaScript event listener function that will be invoked when the event occurs. The third parameter is important to us. If the third parameter is *true*, when the specified event occurs, our registered listener will be notified first, before it is dispatched to other event targets. To get the genuine user intention, we need to execute our listener before the execution of other event handlers. Therefore, when we register the event using the `addEventListener` function, the third parameter must be set to *true*.

To intercept HTTP-related output events, ClickGuard intercepts the HTTP request event via the *http-on-modify-request* notification in Firefox. Firefox sends an *http-on-modify-request* notification to extensions after an HTTP request is prepared and before it is sent out to the network. Notifications are just like *events* or *signals* in other programming languages and frameworks.

We use the JavaScript call stack to look for correlations between input and output events. To get the JavaScript call stack, we use a Mozilla specific property `stack` of the `Error` object. It shows the functions called, their order, and the arguments to them. Alternatively, in Mozilla Firefox extension we can also use the `Components.stack` property of the `nsIStackFrame` interface.

The prototype of ClickGuard registers event listener functions for the `click`, `mouseover`, and `keypress` events. It is straightforward to include other JavaScript events, such as `mousedown`, `keyup`, `keydown`, etc [143].

In this implementation, we use 0.2 as our threshold value of opacity, since during our experiments, layers with opacity values less than 0.2 were hardly visible. To decide the vertical layer order of objects, we use the CSS *z-index* attribute [144] of the layers to identify their vertical layer order. The z-index attribute specifies the stack level of the generated box in HTML rendering, and the layer at the top has the largest z-index value.

4.5 Evaluation

We evaluated the prototype of ClickGuard on a computer with an Intel Core 2 Duo 2.33GHz CPU and 4GB RAM, running Ubuntu 9.10. We tested our solution in Firefox 3.0 and 3.5.7.

4.5.1 Effectiveness

We evaluated the effectiveness of ClickGuard using several types of click event hijacking attacks. We created attack examples for each type of attacks discussed in Section 4.1.

Clickjacking Similar to most malicious websites, a Clickjacking web page is usually available for only a small period of time, and thus, hard to archive without actively crawling the web. Instead of using a real page, we created a Clickjacking attack modeled after the recent Facebook attack. The attack is similar to the Clickjacking example in Section 4.1, but the HREF value of the `anchor` tag in Figure 4.1 is changed to `www.facebook.com`. Instead of using `example.com` as a source of iFrame in demo example 4.1, we used a Facebook URL:

```
http://www.facebook.com/tos.php?api_key=b2b2967f51e927af4b
  4b8e07e8ba03d4&next=http%3A%2F%2Fl1.slashkey.com%2Ffaceb
  ook%2Ffarm%2F%3Fref%3Dbm%26cid%3Dbm%26_fb_fromhash%3D189
  b80a34bd2444c4537513a3ec37691&v=1.0&canvas
```

This link makes a user a fan of the "Farm Town" application, if he or she has already logged into Facebook. The Facebook page was loaded into a hidden iFrame at the top layer, and a deceptive `anchor` element at the bottom layer was placed exactly at the same position where Facebook's "Allow" button appears inside the hidden iFrame. When users click on the deceptive link, they actually click on the "Allow" button of Facebook inside the top-layer iFrame. In our experiment, when the button was clicked, our event handler registered with the `click` event was triggered, we collected all objects under the mouse pointer, and in this case, they were the "Allow" button in Facebook and the `anchor` element. The opacity value of the "Allow" button in Facebook is below our threshold 0.2. Therefore, we inferred the `anchor` element as the user intention (`www.facebook.com`). In the HTTP request output event handler, `http-on-modify-request`, we examined the Destination URL in the user intention (`www.facebook.com`), with the URL of the HTTP request (`http://www.facebook.com/...&canvas`). In this case the first origin check passed. Hence, we performed the second origin check between the enclosing web page URL of the clickable object (`www.Websitename.com`), and the URL of the web page generating the HTTP request (`www.facebook.com`). The second check failed, so we showed a security warning to the user.

Floating objects Figure 4.3 is an example that floats an object and brings it under the mouse pointer on the `onMouseover` event, which subsequently triggers the `onClick` event handler on the floating object. It coordinates two events to achieve the goal of changing the DOM location on the fly. To detect this kind of attacks, we collected all clickable objects under the mouse pointer and recorded the Destination URL value as `http://www.example.com` in the user intention tuple. On the HTTP output event, we performed the first check between the URL of the HTTP request (`malicious.com`) and the Destination URL in the user intention (`http://www.example.com`). It failed because the host part in the Destination URL and the HTTP request were different. Therefore, we showed a security warning to the user.

Pop-up on click On some carefully crafted websites, even a click in a white-space area triggers a pop-up window, which is both annoying and potentially insecure to users. Some of the most annoying pop-up window actions include windows that continually reopen themselves whenever users attempt to close them [87]. Figure 4.5 is an example of a pop-up window opening when a user click event occurs on a web page. Our prototype ClickGuard recorded the X and Y coordinates of the mouse click, and retrieved all objects at that position. However, if a user click occurs in blank space or on a non-clickable object, such as a piece of text on the web page, the user intention extractor returns *nil*. In the HTTP request event interceptor, we correlated HTTP requests with user clicks by examining the presence of click event listener in the JavaScript call stack. The first check failed in this case, because the user intention extractor returned *nil*. We showed a security warning to the user.

4.5.2 False Positive and Performance

In our system, a false positive is a normal page that is detected as malicious. To evaluate false positives generated by our system, we did arbitrary surfing among the top 180 sites from Alexa [6]. During the experiment, we did not observe any false positives. After we turned off the cookie policy check for relaxation, the origin check failed for 37 sites out of the 180 websites we arbitrarily surfed. The reason the origin check failed for those 37 sites was that those websites were sending requests to their sub-domains and changing the HTTP addresses dynamically. However, our origin check passed for all those websites when the cookie policy check was enabled.

We also measured the performance overhead of ClickGuard, incurred from intercepting at two types of events, JavaScript events (such as `onClick`, `onMouseover`, etc), and the *http-on-modify-request* notification event generated by Mozilla Firefox. We measured the wall clock time of a typical browsing session. The average overhead was around 3ms per session. Our solution does not cause noticeable slowdown to the interactivity of web applications.

4.6 Related Work

Pop-up window blocking Using user clicks as indicators of user intentions, web browsers implemented pop-up blockers to suppress unwanted pop-up windows. Pop-up blockers block pop-up windows that are created during page loading by JavaScript functions such as `window.showHelp`, `window.showModalDialog`,
`window.showModelessDialog`, `window.external.NavigateAndFind` [94, 95], etc.

Web browsers block calls to `window.open()` if one of the following conditions are met: 1) global scripts executed during document loading request to open pop-up windows; 2) scripts executed as part of an `onload` event handler ask to open pop-up windows, or 3) scripts executed in `setTimeout()` or `setInterval()` try to open pop-up windows [95].

Clickjacking defense ClearClick, the module in NoScript [106], protects users against Clickjacking. It performs same origin check between the URL of the web page loaded in the iFrame and the URL of the top-level document.

With ClearClick, NoScript is a good prevention for Clickjacking attacks. However, it does not provide a general solution for other attacks discussed in this paper, such as click redirection. Instead, users are expected to specify for each domain whether to allow JavaScript, which lacks the fine granularity to selectively allow/deny specific type or source of JavaScript code.

ClickIDS [8] introduces a solution to automatically detect Clickjacking attacks. ClickIDS registers for the `onClick` event listener, and retrieves the coordinates of the mouse pointer when clicks happen. Then it searches for objects at the same position. If there exists more than one object under the mouse pointer, ClickIDS generates an alarm. ClickIDS detects overlay-based attacks effectively. Compared to ClickIDS, our approach focuses more on studying the effect of user intentions on a broader class of attacks.

To defend Clickjacking, a web application can use a technique called "framekiller" to prevent itself from being loaded in an iFrame [153]. One piece of example code is shown below in Figure 4.7:

```
if ((top.location != self.location)) {
   top.location = self.location.href;
}
```

Figure 4.7: An example of a Framekiller code

On the server-side, web application developers can protect their users against Clickjacking attacks by including similar framekiller JavaScript code in those web pages they do not want to be embedded inside frames by others websites. However, this solution suffers from obvious disadvantages. First, it affects the functionality of websites that use overlays or frames. Second, it is limited to Clickjacking attack, and does not prevent click redirection attacks. Third, it only works when JavaScript is not disabled by users.

Another direction to prevent Clickjacking attacks is to enhance web browsers. Microsoft released a Clickjacking prevention solution in Internet Explorer 8 (IE8), which detects and prevents overlays or frame-based attacks [16, 59, 100, 127]. IE8 introduces a new HTTP response header `X-Frame-Options`, which can be set to either `Deny` or `SameOrigin` [63, 99] by websites. If it is set to `Deny` then IE8 prevents pages of that website from being embedded into frames. If `X-Frame-Options` is set to `SameOrigin`, then IE8 will prevent pages of that website from being embedded into frames in web pages from a different origin. The solution implemented in IE8 only mitigates overlay or frame-based attacks, and requires an extra header in HTTP responses.

Content Security Policy (CSP) [134] is another mechanism intending to mitigate web application vulnerabilities, but its primary focus is Cross-Site Scripting (XSS). To mitigate Clickjacking attacks, CSP enables the site to specify which sources are valid for Frame and iFrame elements.

It maintains a `frame-ancestor` list which indicates valid sources for Frame and iFrame tags. However, CSP addresses only overlay Clickjacking attacks. It cannot prevent click redirection attacks. In addition, each website needs to maintain a `frame-ancestor` list in order to activate Clickjacking protection, whose size may grow rapidly as the number of sites allowed increases.

Chen et al. [21] describe "self-exfilteration" attack technique that allows attackers to steal users' information through either the victim website or another whitelisted website. Self-exfilteration can make use of XSS vulnerabilities to steal users credentials or exploit complex layout structure to trick users into leaking sensitive information.

4.7 Limitations

Browser extension vs. browser instrumentation Our ClickGuard prototype is an extension to the Mozilla Firefox web browser, which facilitates convenient deployment and distribution. However, malicious Firefox extensions can affect the ClickGuard prototype. Therefore, ClickGuard assumes the browser environment (including extensions) is not affected by malicious programs. If implemented via browser modification, ClickGuard will be immune to threats from malicious extensions.

Moreover, as previously mentioned, the current prototype of ClickGuard cannot handle indirectly generated HTTP requests by JavaScript functions like `setTimeout()` and `setInterval()`. Default settings in web browsers disable the opening of a pop-up window by built-in JavaScript functions such as `setTimeout()` or `setInterval()`, but HTTP requests from these functions are still allowed.

The reason our prototype lacks support in these scenarios is that the current Firefox extension interfaces [20] do not provide notifications for the setting of timeouts and intervals. These limitations are solved in our next work through browser modification. Through timer object reference inside the Firefox browser, we can correlate the JavaScript code activated by a timer with the JavaScript code that sets the timer.

Obstructed UIs Although ClickGuard is preventing overlay attacks, attackers can embed legitimate page inside an iFrame with opacity value set to 1, and obstruct part of the page using other objects. This may change the appearance of the victim application, such as swapping the labels of the "Yes" and "No" radio buttons, leading to incorrect actions by users. This is a limitation of the current approach of ClickGuard. Although obstructed victim pages can be identified by users familiar with the interface, this remains a potential issue for first visits to websites

4.8 Summary

Clickjacking is an attack that encompasses multiple techniques to trick web users into clicking on web elements that lead to harmful actions. It is an example of a broader range of attacks of hijacking click events. In this chapter, we study using user intentions to detect this type of attacks. Based on intercepting browser events, we proposed an approach to detect the mismatch between user intentions and browser actions. For events related to user actions, our approach infers the associated users' intentions; for events related to suspicious browser behavior, our approach finds the corresponding user action and matches the browser behavior with the user's original intention. If they do not match, our approach reports an alert to the user.

Chapter 5

Securing Web Sessions from Malicious Requests

In Chapter 4, we used a user intention inference to combat clickjacking attacks that trick web users into clicking on web elements that lead to harmful actions. However, as web technology advances and produces more dynamic and complex web applications, the threat to web application integrity is also increasing. Web applications with a wide menu of choices and features for the user also give attackers more choices and flexibility as to where they can hide their malicious requests. Attackers can trick a victim user or direct his/her browser to send requests to the web server, and then steal or destroy user's data by riding or piggybacking on existing sessions. The server will accept this request, because the request will include the user's valid session information.

5.1 Motivating Examples

In the versatile browser environment, there are many ways to trigger web requests sent to the web server, such as URL visits, image loading, and XMLHttpRequest of JavaScript. Attackers exploit many techniques to automatically generate malicious requests, such as cross-site scripting (XSS) [27] and cross-site request forgery (CSRF) [11]. Next, we will describe attack examples that send malicious requests to web servers.

Cross-Site Scripting (XSS) Attack If a user visits a web page of a trusted, but vulnerable, website, for example, *HonestSite.com*, in which an attacker has injected inline malicious script, for example, M_s. Malicious script M_s, is executed within the context of the website and can send a state modifying request to *HonestSite.com*

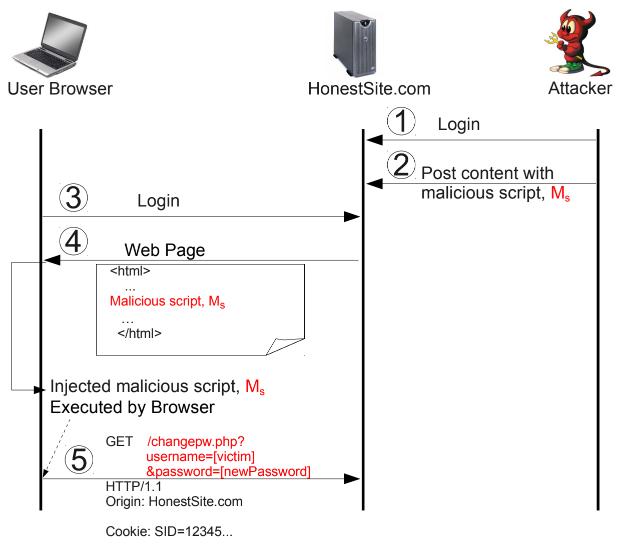


Figure 5.1: Illustration of cross-site scripting (XSS)

As shown in Figure 5.1, the attacker first logs in to the website, *HonestSite.com* (Step 1). On the vulnerable website page, the attacker then submits the malicious script, M_s (Step 2). When a user logs in to the website (Step 3), and visits the affected web page (Step 4), the injected malicious script M_s, is executed within the context of website. This compromises the integrity of website by sending a malicious request to modify the state of the user on the website (Step 5). The request is generated by the injected script, but carries a valid authentication token. Therefore, the website processes the malicious request.

Cross-Site Request Forgery (CSRF) Attack If a user visits `MaliciousSite.com`, possibly because of an enticing email, a web page of `MaliciousSite.com` can send a request to `bank.com`. In addition, if web browsers hold active session cookies of `bank.com`, they automatically attach those cookies to the outgoing request. The `bank.com` web server will authenticate the request based on the session information in the attached cookie, rather than verifying how the request the generated on the client side.

Dynamic CSRF Attack HTTP `Referer` header is widely used to detect CSRF, which sends information about the page where a web request is initiated. To defeat this defense mechanism, a new technique launches CSRF attacks dynamically [55]. For example, figure 5.2 shows that on a website forum that allows html tags, the attacker embeds images using the <`img`> HTML element that points to the server under the attacker's control. Using session information leaked through the `Referer` header, attackers can dynamically create CSRF requests, redirecting user browsers via HTTP redirect (302) to the web forum with a users' valid credential and session information.

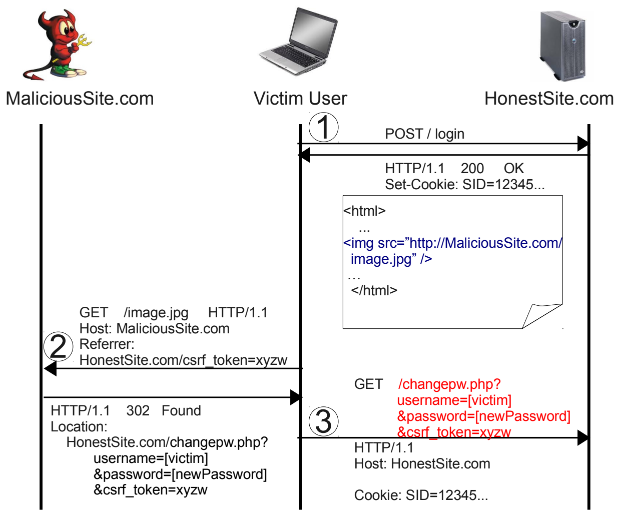


Figure 5.2: Illustration of dynamic cross-site request forgery

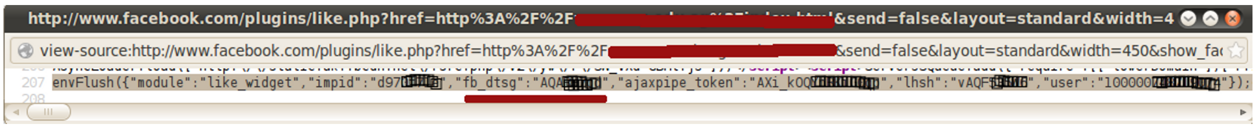
```
javascript:
var list=document.getElementsByName('like');
for(var i=0;i<list.length;i++){
 // invoke click event on Like button
 list[i].click();
 // introduces delay
 for(var j=0;j<100000;j++);
}
```

Figure 5.3: An example of self-XSS code

5.2 Development of Session-Misuse Attacks

As effective defense techniques for attacks (such as CSRF, XSS, Clickjacking) have been widely adopted, attackers have turned to various types of social engineering to lure users into participating in an attack. For example, attackers explicitly trick the victim user to copy and paste code segments into the user's session. We introduce examples from this new attack trend that involves users to assist session misuse attacks on a target web application (e.g., Facebook).

Self-XSS Attack In this type of attack, the malicious JavaScript is injected by users into their web sessions, by users who were tricked by attackers into "copy-and-pasting" the malicious code in browser's *URL* bar [17]. The code entered in the *URL* bar of the web browser runs within the context of the web application in the active tab of the browser. To combat this technique, modern browsers will disallow execution of JavaScript in the *URL* bar if it is copy-pasted. However, they still allow user-typed JavaScript to be executed within the context of the current web page. Moreover, modern web browsers allow JavaScript in browser bookmarks (*bookmarklet*) to be executed



```
http://www.facebook.com/plugins/like.php?href=http%3A%2F%2F[redacted]&send=false&layout=standard&width=4
view-source:http://www.facebook.com/plugins/like.php?href=http%3A%2F%2F[redacted]&send=false&layout=standard&width=450&show_fa
envFlush({"module":"like_widget","impid":"d97[redacted]","fb_dtsg":"AQA[redacted]","ajaxpipe_token":"AXi_kOQ[redacted]","lhsh":"vAQF[redacted]","user":"100000[redacted]"});
```

Figure 5.4: An example of user-assisted attacks

within the context of current web application. A bookmarklet is a piece of JavaScript saved as a bookmark entry in the browser's bookmark menu. To launch the attack, attackers will trick users into saving a piece of malicious script as a bookmark entry into the web browser's bookmark menu. When it is activated by the user, the malicious script sends requests to a target server.

Figure 5.3 shows sample code of a self-XSS attack on Facebook to "Like" all posts on the victim user's wall. The malicious script is stored as an entry in the browser's bookmark menu. To trick users to store the malicious script in the web browser's bookmark menu, attackers can use various tricks, such as fake security dialog boxes and fake "link to giveaway" coupons, which ask users to save malicious JavaScript as a bookmark entry. When the saved bookmark entry is invoked by the user from the bookmark menu, it retrieves all Like buttons from the victim user's wall and invokes the *Click* event on each *Like* button to like all posts displayed on the victim user's wall.

Anti-CSRF Token Social Engineering Attack To protect against CSRF and dynamic CSRF attacks [55], a web application authenticates web forms using a secret token (also known as an anti-CSRF token). The anti-CSRF token is a one-time token, generated randomly by web applications, which is submitted as a hidden input parameter in a web form. When form data is submitted by the user, the web application checks whether the form data contains the correct anti-CSRF token. If no such token is found, which means the form data has been generated by a forged request, the web application will reject the request. For example, Facebook uses a unique anti-CSRF token each session to defend against CSRF attacks that post scam message on a user's wall. To make web applications (such as, Facebook) to accept their malicious request, attackers need to guess the anti-CSRF token. To overcome this hurdle, a recent attack technique uses social engineering to retrieve the anti-CSRF token [41].

To trick users into revealing their Facebook anti-CSRF token, a malicious website, for example, *M*, might entice users into clicking a malicious link that shows a false verification message, such as *Before you proceed, you must complete a quick Security Check*. When the user clicks to continue, it opens a Facebook page in a pop-up window using `view-source` protocol, and asks the user to copy and paste some JavaScript code from the Facebook pop-up window on a text field in malicious website *M*. As shown in Figure 5.4, the JavaScript code contains the anti-CSRF token `(fb_dtsg)` for the victim user's session. When users copy and paste code into the attacker-controlled pages, attackers can extract the value of anti-CSRF token of the victim user's session. After retrieving the anti-CSRF token, attackers can then launch CSRF attacks to post malicious links on the victim user's Facebook wall.

Summary The above examples demonstrate examples of various ways to bypass the existing defenses that guard against session-misuse attacks. The root cause of session-misuse attacks is the mismatch between the complex client environment and the simple server validation mechanisms.

5.3 Background

Web application logic is split between the web server and the web browser. To connect server-side components and client-side ones, web applications use the HTTP protocol. Moreover, web applications need to expose HTTP-based interfaces on the web servers to the web clients, so that the web clients can request operations independently on the web application servers using the HTTP protocol. However, HTTP is a stateless protocol, where web servers do not maintain states to relate one request to another. To support session-based web applications, such as e-commerce or email websites, the session-related states must be stored by the browser. Requests carry the session information to identify themselves to the server. To track users' authentication information between multiple HTTP requests, web applications use different mechanisms, such as session cookies or client SSL certificates. This kind of authentication information, if present in the browser, is automatically attached by the client web browser to each HTTP request sent to the web application. HTTP cookies have been widely used as a mechanism to "remember" the current user and the ongoing session of a web application. The web application server sets the cookie in response to the browser after a user successfully logs in, and the browser stores the cookie for a period of time. For *every* subsequent HTTP request sent to that web application server, the browser automatically attaches the cookie to identify the session to the server. By exploiting the complex layout and JavaScript features of a web application, attackers can force browsers to send requests to web applications carrying a victim user's session identifier. From the web server's point of view, it's difficult to identify malicious requests injected by an attacker within an active user's session.

Although the client-side logic of web applications is more dynamic, their functionalities are controlled, to protect a website from being affected by others, using the same-origin policy [99]. The same-origin policy allows session identifiers, for instance, session cookies, from different domains to coexist on the user's browser, without interfering with each other. To perform misdeeds on behalf of a victim, an attacker needs the session cookies of the victim. However, same-origin policy prohibits the web page from one origin to access the cookies or DOM (Document Object Model) properties of a web page from another origin. Hence, to achieve their goals, attackers find ways to trick either the web browser or the user into sending malicious requests to exploit their active sessions. The requests carry the user's valid session information, and the validation mechanism on web application servers are limited in deciding whether a request is generated legitimately or triggered by attackers. This limitation is actively exploited by attackers to generate malicious requests. When these malicious requests are sent to web servers where the victim user has already established a session, the web server cannot distinguish malicious requests from benign ones. We call such attacks *session-misuse* attacks. For example, attackers can trigger malicious requests through exploiting cross-site scripting (XSS) vulnerabilities or through cross-site request forgery (CSRF) attacks.

To help the web server distinguish benign requests from malicious requests, the common defense technique is to include more information about the request sender. For example, the HTTP `Origin` header [11] is used to indicate from which origin the current HTTP request is generated. It is used by web applications to prevent CSRF attack. Web applications only accept requests with HTTP `Origin` from its own origin. However, the `Origin` header is not attached to all HTTP requests, such as HTTP requests loading images. Another line of solutions is to make forging malicious requests more difficult, such as including an authentication token in each request [26, 38]. The authentication token is randomly generated and recorded in web pages, which provides additional authentication to cookies. Although effective in detecting attacks, these solutions are limited to specific attack types, such as CSRF.

Despite the many solutions proposed by industry and the research community to address session-misuse attacks, this threat has continued to gain momentum. Although solutions to prevent XSS and CSRF have been actively developed and deployed, attackers keep finding new ways to evade them. A recent attack technique uses social engineering to trick users to assist session-misuse attacks [17, 41]. In such user-assisted attacks, attackers lure users to participate in the attack process, such as by injecting a piece of malicious JavaScript into their active web sessions through the browser interface, which generates requests to the server on behalf of the user.

The root cause of session-misuse attacks is that *the server-side web request validation mechanism lacks the knowledge of client-side behaviors*. Given the complexity of the browser environment, it is important to know *how* the request is generated from it. Such details of the environment from which the request is generated can help web servers to validate web requests. For example, from which web page, which region of the page, and by which event the request is generated all can be used for validation purposes. Unless we bridge this gap between the browser environment complexity and the power of server-side request validation, attackers will constantly find new ways to trigger browsers to generate malicious requests that can be accepted by the server. The user-assisted attack is just an example.

By design, a web application expects a particular request to be generated by specific browser behavior sequences, such as executing JavaScript and processing user events, and from certain browser components, such as nodes in Document Object Model (DOM) tree. In other words, the request's dependency on the browser behaviors and the browser environment states should form an invariant, which we call *request dependency integrity*. This is a fundamental security property that is violated by a wide-range of session-misuse attacks.

To protect from session-misuse attacks, our main idea is to enforce the request dependency integrity of web requests. We define a Request Dependency Graph (RDG) to represent the client-side dependency. The RDG captures information on *how a request is generated from the browser environment*. Such execution environment information includes events that trigger the generation of a request and the elements on which the event occurs. The dependency relationship among web page elements, events, and HTTP requests is crucial for the web server to identify malicious requests. Based on RDG, we developed an end-to-end solution called ClearRequest, which extracts RDGs at the browser side and enforces the request dependency integrity on the server side. We design techniques to compute the *slice* of RDG, which only maintains the information related to the specified request. The corresponding slice of RDG is used by web servers to validate the incoming request accurately.

5.4 Request Dependency Graph

The key technique of our approach is to extract request-related dependency information in a browser environment. We define the Request Dependency Graph to represent the dependency between events in the web browser runtime environment that is relevant to HTTP requests.

Definition 1 Request Dependency Graph (RDG) *is a directed, rooted graph, defined by* $RDG =< V, E >$. V *denotes the sets of nodes of an RDG and each* $v \in V$ *is an* `Element` *or a* `Request`*;* E *represents the set of edges in an RDG and each edge* $< v_1, v_2 > \in E$ *denotes the relationship that the node* v_2 *is dependent on the node* v_1.

- *An* `Element` = *<Name, DevId, SerId, Origin, Opacity, Hash> represents an element in a web application. Name defines the type of the element, such as form, frame, address bar, etc.*

DevId is the ID attribute of a HTML element in the web page and it is specified by developers in the HTML tag. SerId is the unique identifier for the element by DOM serialization. Origin denotes the origin from which the resource is loaded. Opacity is the transparency value of the element. Hash is the hash value of the attributes or data of the element.

- *A* `Request`$=< id, URL >$ *is a pair that contains information about the destination URL, where id is the unique identifier for the request assigned by our approach, and URL is the URL of the request's destination.*

- *An edge* $< v_1, v_2 >$ *represents the dependency between them, and is labeled by an* `Event` *that generates the dependency. For example, a request can be generated by a click on an HTML hyperlink. In this case, the request is dependent on the link element, which is labeled by a click event.*

- *An* `Event`$=< EventType >$ *is a singleton that contains information about the type of the event, where EventType denotes the type of the event, such as Click, Parse, DOM Mutation, Load, Timer, Execute, AutoSubmit, etc. An EvenType describes the action performed by a web browser, whereas an Element represents HTML tags in a web application.*

- *The node* $r \in V$ *is the root of the RDG, which represents the top window element.*

The RDG includes information on the evolving structure, as well as run-time events of the web applications at the client side, such as the origin of the request, resource loading events, user actions, DOM element information, etc.

In the rest of this section, we will first introduce in detail how to extract RDG from the browser environment. Then, we will present the slicing algorithm to find the dependencies related to a request.

5.4.1 Extracting RDG from Browser Environment

Web browsers send requests to web servers, and receive responses from them. When they receive a response, they parse it and build the Document Object Model (DOM), which is a tree structure that consists of page elements as its nodes. Other information, such as rendering properties and JavaScript event listener functions, is stored as the attributes of the nodes. In addition, modern web browsers use an event-driven model. Hence, lots of events are generated and processed by the browser, such as loading a resource when encountering an image or iframe element, and triggering JavaScript event handler routine on the element by a user action. This browser environment

Ways of generating HTTP Request	Relevant Information
User enters URL either in address bar or selects from bookmarks	Address bar or bookmark
Extensions or plugin	Extension name or plugin name
Fetch resources while parsing	Element where the request is issued
User action	Action name & element that triggered the action
JavaScript sends request directly	The <SCRIPT> element where the request is generated
JavaScript sends request indirectly	A sequence of elements & events that affect the request
Redirection	The Origin where the request redirects

Table 5.1: Methods to generate HTTP request and relevant information to record

provides information about the request relationship, for example, whether a request is triggered by user action on a web page element, or by client-side JavaScript code in web application.

When one request is generated from a web page, it is important to know how it is generated. For example, when untrusted content is injected in web pages, different regions of web pages may indicate different trustworthiness.

On the other hand, requests can be generated in web browsers in different ways. Table 5.1 describes ways to generate HTTP requests in a browser and the relevant information about how the requests are generated.

Next, we will explain how to build RDG from the browser environment.

- **DOM Element Identification.** The first problem we need to solve is to uniquely identify DOM elements in the RDG. This is difficult, because web pages usually contain multiple elements of the same type, but not each of them has an `id` attribute set by web developers. Therefore, we need to leverage additional information to identify DOM elements. In our approach, to allow web applications to identify an element uniquely when the element does not contains an ID, we calculate the hash of the attributes of DOM elements. In addition, we assign unique integer IDs to DOM elements by serializing DOM.

- **Timer Events.** Web applications use JavaScript to create timers to execute certain scripts after a period of time, or at certain intervals. We record the mapping between timers and the origins of their creator JavaScript, when the timer is created. When a timer event is triggered, we look up its creator JavaScript in our record. We add a node for timer handler script and connect it with its creator node by an edge label as *Timer*.

- **User Actions.** User actions are also important in deciding the legitimacy of requests. We intercept event handling in the browser, such as click, keypress, mouseover, etc. The target element in the event handler routine is the DOM element on which the user interaction occurred. We add this target element as a node in our request dependency graph, and we maintain an attribution table that records the target element and any events that occurs on it. When an HTTP request is generated, we look up its initiator, the target element that we have already added in the graph. We draw an edge from the target element found towards a new node created for the HTTP request. The edge is labeled with the event that triggers it.

- **JavaScript Modifications.** JavaScript can dynamically create new DOM elements or modify existing ones, and those DOM elements may in turn generate HTTP requests. Modifications on DOM elements trigger DOM Mutation events. For example, the modification of an attribute of a DOM element fires a `DOMAttrModified` event. We draw an edge from the JavaScript node that represents the one that directly generates the new request to the target DOM element node with label `DOM Mutation`.

We construct the RDG based on the above cases, representing dependency between web application elements and events that affect requests. When an element is parsed and constructed by the browser, if it can generate or affect subsequent requests, we add it into RDG. Such elements include script, image, and iframe elements. We do not add them into RDG immediately after they are created by the browser. Instead, we maintain dependency relationship involving them separately, and add them, along with their dependency information, into RDG whenever a request affected by them is generated. Constructed this way, the RDG contains all relevant dependency information of elements and nodes that affect HTTP requests.

Let's consider a sample web page snippet from *www.example.com* that initially contains a link, an image, an iframe, and a script element. The script modifies the attribute of the link and creates another piece of script on the fly, which in-turn sends an XMLHttpRequest request. The iframe loads a page from a third-party server, which in turn loads an image and a script element. This is shown in Figure 5.5.

Figure 5.6 shows the RDG built for this page. As shown in the figure, *www.example.com* is a web domain that generates various HTTP requests. First, it sends HttpRequest0 for its front page. Then during parsing, it sends HttpRequest1 and HttpRequest2 to load an iframe and an image resource, respectively. The iframe further requests HttpRequest4 and HttpRequest5 for external image and script resources, and the script in the iframe also generates a new request HttpRequest6 during its execution. Back to the main page, some inline script creates a new script element by mutating the DOM, which in turn sends out HttpRequest7. The inline script Script0 also modifies the HTML hyperlink. The link is clicked, so it sends HttpRequest7 to navigate to another page.

```
//A sample web page from www.example.com
<img src="http://www.example.com/img1.jpg"
     id='img1' />
<iframe
  src="http://www.thirdparty.com/page.html" />
<a href="#" id='link1'>click here</a>
<script>
// modify link
var lk=document.getElementById('link1');
lk.href="www.example.com/page2.htm";
//Create new script element that sends XHR
//request to example.com
....
</script>
```

Figure 5.5: A sample web page snippet.

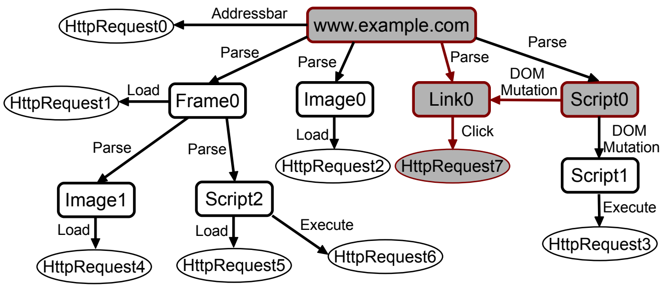


Figure 5.6: A request dependency graph for the web page snippet

5.4.2 RDG Slicing

In a long-lived user session, RDG often becomes huge in size. Hence, we only maintained the RDG for the current web page, rather than storing the entire execution history of web pages visited in the session. However, it will be prohibitively inefficient to directly pass the entire graph to the web servers for processing, and when multiple pages are integrated into the same page, the RDG also contains information of different pages, and should be filtered with great care before being sent to the web server. To solve the above two practical concerns, we also develop an RDG Slicer to perform backward dynamic slicing, according to different needs on the RDG we obtain earlier. RDG Slicer is used to obtain a slice for a particular request.

Our slicing algorithm takes an HTTP request node and the RDG as inputs. It outputs a *slice*, which is a sub-graph that contains only nodes and edges that the request node is dependent on. Starting with the sub-graph containing only the HTTP request node, we iteratively add to the sub-graph nodes that the nodes in the sub-graph depend on, as well as the edges of such dependencies, until there are no more nodes and edges to add. As illustrated in Figure 5.6, the slice extracted for HttpRequest7 is shown in gray color in the figure. The slice contains the dependency from the root node to the HTTP request. To further reduce slice sizes, the algorithm only includes the *iFrame* elements that are outside the request-generating *iFrame* and the Root node in the RDG slice, excluding other elements that are outside the request generating *iFrame*.

The primary objective of ClearRequest is to detect malicious requests. RDG includes the client-side environment information of web applications, such as the domain of the request initiator, region of the page, and browser and user events. To protect privacy, the RDG slicer strips sensitive information from the RDG slice. In particular, the request dependency graph built by our approach contains information about DOM elements that sometimes store sensitive information, such as CSRF security token or users' private information. To prevent leakage of privacy information, ClearRequest does not store any sensitive information in RDG nodes. It only stores non-sensitive information, such as the DOM element type, its ID, the hash of its attributes, opacity value, etc. In addition, web applications may also append security tokens to the URLs. Hence, we only maintain origins, defined as $< protocol, domain, port >$, in RDG, rather than the complete URLs.

Furthermore, ClearRequest handles the cases of same-origin and cross-origin requests differently. For same-origin requests, the slice with full element information is sent; for cross-origin requests, it only sends the *name*, *origin*, and *opacity* of elements to servers. Cross-origin sites can use user-action events in the RDG slice to track user's behavior. Therefore, ClearRequest strips user actions information from the slice for cross-origin requests.

5.5 Design of ClearRequest

In this section, we introduce the design of ClearRequest. For a high-level view, ClearRequest extracts RDG from browsers to represent how a request is generated, and sends it to web application servers to help them identify malicious requests.

We focus on enabling web servers to validate the legitimacy of HTTP requests to prevent attacks through malicious requests, regardless of their attacking techniques, such as CSRF, XSS, and malicious mashups. There are a number of threats we do not consider in this chapter, such as information leakage, cookie-stealing-based session hijacking, and XSS attacks that send victim's sensitive information directly to third-party servers, self-exfiltration [21], or phishing [162, 165]. Attacks targeting browser vulnerabilities and active network attacks are also out of the scope.

To facilitate web application in authenticating web requests using browser environment information, we propose ClearRequest. Figure 5.7 illustrates the architecture of ClearRequest. We

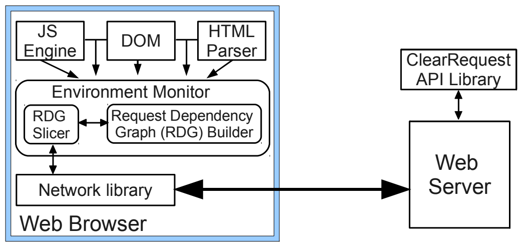


Figure 5.7: Architecture of ClearRequest

monitor the runtime environment of web applications by monitoring the web browser internal behaviors, using a new component called Environment Monitor (EM). EM logs all important events, including JavaScript execution, Document Object Model mutations, HTML parsing, and HTTP requests. Moreover, EM also records the dependency relations between different events. Such events, as well as their interdependency are used by the RDG builder to generate RDG. The RDG contains the rich set of browser environment information that can be leveraged by web servers to better authenticate incoming requests.

Before sending a request, ClearRequest prepares a slice of request dependency from the RDG using our slicing algorithm. The slice is sent with the request to the server. When the server receives a request, it first authenticates the request using its session tracking mechanisms, such as valid authentication token or session cookie, etc. Then it checks the legitimacy of the request by examining the RDG slice field, checking whether the properties in RDG matches the web application's expectation. ClearRequest provides a library to web application servers to examine the RDG Slice and ease the task of web developers. Web developers can perform various types of property checks on the RDG slice, such as matching user action, matching initiator element, matching origin, etc. Table 5.2 shows API's provided by ClearRequest for the web application to examine the RDG slice of a request for its properties. For example, the `checkUserAction` API of ClearRequest takes an RDG slice in the request and the expected user action provided by web developers as inputs, and examines whether an element (i.e immediate parent of the request node in the RDG slice) initiated request from matches the expected user action. This API is useful to defend against XSS, because it usually initiates request using injected scripts.

5.6 Implementation

In this section, we will describe the implementation of ClearRequest in detail.

ClearRequest, consists of a client-side component and a server-side component. The client-side component records all events in the web browser, and generates and sends a slice of the *RDG*

ClearRequest API for Web Application	**ClearRequest API Description**
checkDependencySlice ($RDG_Slice,$expectedSlice)	This API returns TRUE if $expectedSlice matches with the $RDG_Slice; otherwise returns FALSE.
checkInitiatorOrigin ($RDG_Slice,$expectedOrigin)	This API returns TRUE if $expectedOrigin value matches with the origin value in the $RDG_Slice; otherwise returns FALSE.
checkInitiatorElementType ($RDG_Slice,$expectedType)	This API returns TRUE if $expectedType value matches with the parent element type value of the request in the $RDG_Slice; otherwise returns FALSE.
checkInitiatorElementId ($RDG_Slice,$expectedId)	This API returns TRUE if $expectedId value matches with the parent element ID value of the request in the $RDG_Slice; otherwise returns FALSE.
checkUserAction ($RDG_Slice,$expectedAction)	This API returns TRUE if $expectedAction value matches with the parent event value of the request in the $RDG_Slice; otherwise returns FALSE.
checkPageStructure($RDG_Slice)	This API checks page structure and returns FALSE if request is generated by an element in frame, or an initiator element of the request is found transparent; otherwise returns TRUE.

Table 5.2: ClearRequest APIs for a web application to examine RDG slice of a request

for the HTTP request to the web application. In ClearRequest, we labeled client-side component as Environment Monitor. The server-side component is the API library, written in PHP to allow easy integration with PHP web applications.

5.6.1 Integration of the ClearRequest with Web Browser

We prototyped the client-side component of ClearRequest in Mozilla Firefox 8.0 open source web browser. In the Firefox web browser, we instrumented a networking component, which is also known as Necko, JavaScript Engine, DOM Mutation Events and HTML parser in the browser engine. The entire system consists of 3200 lines of C++ code.

Browser Environment Monitor hooked browser events like resource loading (such as IMG, CSS, etc.), new page creation (such as iFrame or Frame loading), user actions (such as click, mouseover, etc.), and DOM modifications. In Firefox, when an HTTP request is generated, the necko (network library) does not contain information about the node that generated the request. Therefore, ClearRequest maintained a mapping table of the elements generating the request.

The `nsIContentPolicy` interface provided by Firefox controls the loading of various resources and provides mapping of the element that generated a request [92]. The current implementation of ClearRequest uses an nsIContentPolicy interface and records nine different types of nodes [1] and eleven different types of edges[2] in RDG.

To track the dynamic modification correctly inside a web browser, we used DOM mutation

[1]List of Nodes: script, image, stylesheet, object, iframe, frame, font, media and request

[2]List of Edges:timer, xhr, parse, load, execute, addressbar, bookmarklet, DOM mutation, click, dbclick, keypress

events such as `DOMNodeInserted` and `DOMAttrModified`. According to DOM Level 3 event specification [143], the mutation events are designed to notify listeners of any changes to the structure of a document. The *MutationEvent* interface provides contextual information associated with DOM mutation events. We use the contextual information provided by the *MutationEvent* interface to check the original and current target elements associated with that event, and recorded the source and target elements in the RDG graph by adding an edge from source to target node labeled with DOM Mutation event.

We added a new HTTP header to the each request to send an RDG slice of the request. For each HTTP request sent to the web application, the web browser attaches the slice of the RDG of an HTTP request in the `RequestDependency` HTTP header.

5.6.2 Integration of the ClearRequest with Web Applications

Integration with PHP applications We implemented a prototype of the server-side ClearRequest component as a PHP class. This class provides interfaces to web developers for examining the RDG slice in the request. Our implementation of ClearRequest class is 135 lines of code. It is easy to be integrated in web application, by including it as a library. We tested ClearRequest library in an open source web forum phpBB. We inserted checks, using the interfaces provided by ClearRequest API before the important operations were performed in the phpBB web forum, such as posting a new post or deleting a post, etc.

Integration with CodeIgniter Recently, web application development frameworks became more popular as they free web developers from low-level implementation details and allow them to focus on high-level application logic. We also integrate ClearRequest with a popular web application development framework CodeIgniter [44], to enable adoption of ClearRequest in framework-generated web applications.

CodeIgniter is an open source web application development framework for building websites using PHP. It allows rapid development of web applications, by providing a rich set of libraries. This removes the burden of implementation of commonly-needed tasks from web developers. We extended the *Security* core class of CodeIgniter v2.1.3 with 142 lines of code to support RDG checks. Our extension provides the web developers API to perform checks, such as originator element, user action that triggered request, domain that initiated request, and RDG slice. ClearRequest doesn't accept state-modifying requests that do not contain any RDG. This prevents an attacker from bypassing the mechanism, by suppressing the RDG header. Our prototype is tested with CodeIgniter-developed web applications fule-cms and ukulima.

Synthesizing RDG checking rules To help web developers specify the rules to check RDG slices, we develop an RDG slice synthesizing mode. During the synthesizing mode, our instrumented browser records the RDGs for requests sent to web applications. For each particular request, our prototype merges different RDG slices, only retaining the common entities in the RDGs, such as the origin, the triggering user action and the elements during throughout the testing process. These merged RDG slices represent the expected ones for corresponding requests. To derive RDG checking rules from these expected RDG slices, our prototype provides a few heuristics to web developers, such as matching user action, matching initiator element, and matching origin. Web developers can choose among these heuristics, and our prototype automatically generates rules that check whether the specified user action, initiator element, and/or origin matches the expected values in the expected RDG slices. We use this mechanism to obtain synthesized RDG checking rules for expected request dependency in our experiments.

5.7 Evaluation

We evaluated the effectiveness of the request dependency graph (RDG) technique to enforce request dependency integrity. Our technique was tested against session-misuse attacks, through several experiments. The experiments were performed on a desktop computer with an Intel Core 2 Duo 2.33 GHz CPU and 4GB RAM, running Ubuntu Linux 10.04. To measure the performance overhead of our technique, we integrated our technique as a prototype called ClearRequest in Firefox open source web browser, and `phpBB` open source web forum.

5.7.1 Session-Misuse Attack Detection

We analyzed our technique using various session-misuse attacks we discussed in sub section 5.2.

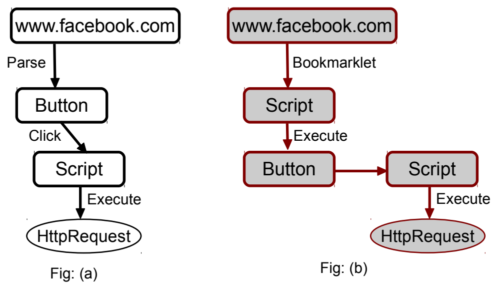


Figure 5.8: (a):An RDG slice of benign request to Like a link (b):An RDG slice of a request generated using self-xss

Self-XSS Attack In this social engineering attack, users were tricked into bookmarking a piece of malicious JavaScript in the browser's bookmark menu. If the user activated a malicious JavaScript code in bookmark menu while the Facebook tab was active in the browser, the malicious JavaScript executed in the context of Facebook, and by default, acquired all privileges of Facebook. It retrieved all Like *buttons* from the Facebook page and triggered a `Click` event on each Like `Button` element, to automatically like all posts on victim user's wall. The benign request scenario is illustrated in Figure 5.8 (a), in which Facebook expects a request to Like a friend's post from user's wall should come from a Like `Button`, when the user *Clicks* on that button. In the Self-XSS attack scenario, shown in Figure 5.8 (b), the attacker injected malicious script using a bookmarklet that initiated the `Click` event on the *Like* `Button` element, which in turn initiated a request to like a post. Hence, using RDG, we differentiated a benign request invariant from a malicious request.

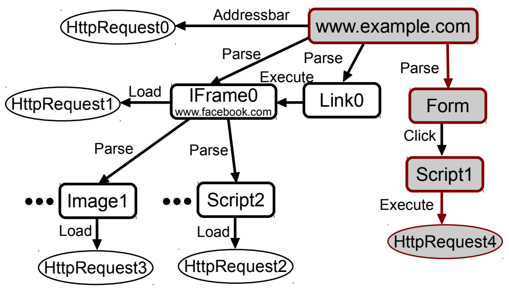


Figure 5.9: An RDG slice of a malicious request

Anti-CSRF Token Social Engineering Attack In this social engineering attack, the goal of attackers is to bypass anti-csrf protection used by web applications (such as Facebook), to force the web application into processing malicious requests. Figure 5.9 shows a RDG slice for a user-assisted attack constructed to automatically *Like* the malicious link. To ease understanding, we have shown only a few nodes from RDG that are relevant to this attack. The attacker tricked the user into copying and pasting the anti-csrf token into the attacker's control field, so that attacker can gain control of the anti-csrf token of the user's session. After stealing the anti-csrf token, the attacker generated requests to Facebook to post messages or updates on user's wall, without the user's consent. In the attack scenario, a request was prepared and sent by a script from `www.example.com`, as shown in the figure as gray color nodes with red border. The benign request was expected to originate from a Like button inside the Facebook iFrame, whereas the malicious request was triggered from `www.example.com`. Thus, RDG provides vital information about how the request is generated at client-side, and this helps the web applications ensure the

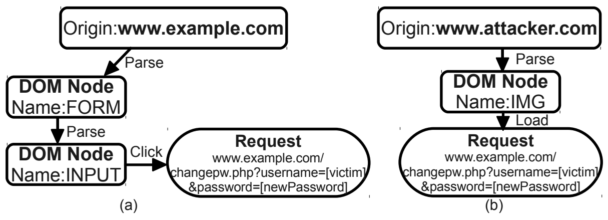

Origin:www.example.com
Parse
DOM Node Name:FORM
Parse
DOM Node Name:INPUT
Click
Request www.example.com/ changepw.php?username=[victim] &password=[newPassword]
(a)
Origin:www.attacker.com
Parse
DOM Node Name:IMG
Load
Request www.example.com/ changepw.php?username=[victim] &password=[newPassword]
(b)

Figure 5.10: (a):An RDG slice of benign request (b):An RDG slice of a CSRF attack

request dependency integrity.

5.7.2 Effectiveness on other session-misuse attacks

To evaluate the effectiveness of RDG against session-misuse attacks, we also conducted detailed case studies of our RDG against various real-world attack examples. We used attack samples that generated malicious requests to vulnerable hosts using various attack vectors, such as CSRF and XSS. We simulated real-world vulnerabilities and attacks, using virtual hosts and virtual machines. In our experiments we simulated total eight number of real-world attack examples, in which five were CSRF attack examples (CVE-2009-3759, CVE-2008-7248, CVE-2009-1339, CVE-2009-4898, and CVE-2010-3449), three XSS-worm examples (WHID 2010-175, Orkut XSS [159], and MySpace XSS worm [158]).

CSRF Attack Figure 5.10 shows the RDG slice for both the benign request and the CSRF attack constructed from real world attacks against vulnerabilities in CVE-2009-3759 [3, 107]. For simplicity, we have omitted details of DOM nodes in the figure, and only show slice of the RDG for the benign and malicious requests.

This attack involved a victim user, an honest site *www.example.com* and a malicious site *www.attacker.com*. The victim user visited the malicious site while he/she had an active session with the honest site. The malicious site then crafted an HTTP request to the honest site that injected a request into the victim user's current session. We simulated this CSRF attack using virtual hosts, and set up the two sites in a virtual machine. In normal scenarios, as illustrated in Figure 5.10 (a), the honest site used an HTML *FORM* to submit a request to change the password when the user *clicked* on the Submit *INPUT* button in its enclosing *FORM*. In the CSRF attack, shown in Figure 5.10 (b), the malicious site *www.attacker.com* sent an HTTP request to change the user's password, with an *IMG* tag requesting for an image resource. This image load request carried the victim user's session token, hence it performed the password change action on behalf of the victim user.

The slice of the malicious request in the CSRF attack contained an *IMG* tag that was generated when parsing the document, and the *IMG* tag in turn generated the request when loading the image resource. This was different from the expected behavior, where the request to change the password was generated by a user click on a Submit *INPUT* button whose parent node was a *FORM* node. Hence, using RDG, we detected the attempt of this CSRF attack, and prevented the processing of the request.

In addition to above attack, we also simulated four real-world attacks namely, CVE-2008-7248, CVE-2009-1339, CVE-2009-4898, and CVE-2010-3449 [90], to evaluate the effectiveness of the RDG in the detection of malicious requests sent using CSRF. Using RDG, ClearRequest successfully prevented these malicious requests in our experiments.

XSS Attack Figure 5.11 shows an RDG slice of both a benign request and an XSS attack constructed from real world attacks against vulnerabilities in WHID 2010-175 [33].

This simulated stored XSS attack injected malicious script into the vulnerable web forum site *www.example.com*, so when a victim user visited it, the malicious script sent a request to *www.example.com* to post a message on the web forum on behalf of the victim user.

Figure 5.11 (a), shows a normal scenario of posting a message on the vulnerable site. The request came from an HTML *FORM* element when the user *clicked* on the Submit *INPUT* button in the *FORM*. Figure 5.11 (b), shows an RDG slice of a request generated using XSS attack. Malicious script was injected into the site *www.example.com*, which automatically sent two requests to post two messages on behalf of the victim user when the user hovered a mouse over the link. One request posted the malicious link itself on victim user's account, which then attacked others if they hovered their mouse on the link. Another request posted an embarrassing tweet on the victim user's account. The slice of RDG for these requests contained an edge from the `SCRIPT` node to the request with the labeled `Execute` rather than `Click` as in the expected behavior. Moreover, the immediate parent node of the request was the *SCRIPT* node, rather than the *INPUT* element node in the `FORM`. Furthermore, this XSS request slice contained a `LINK` node as the parent node of the *SCRIPT* node with an edge labeled as `MouseOver`. This indicated that the request was sent by JavaScript. As shown in the Figure 5.11 (b), the slice of the RDG for the request provided contextual information of the generation of the request. In the RDG slice of XSS, the element nodes and events triggered that request differed from those of the benign request. Therefore, using RDG slice, ClearRequest detected this malicious request and prevented the modification of user state on the web server.

In addition to the above attack, we also simulated two real-world XSS worms to evaluate the

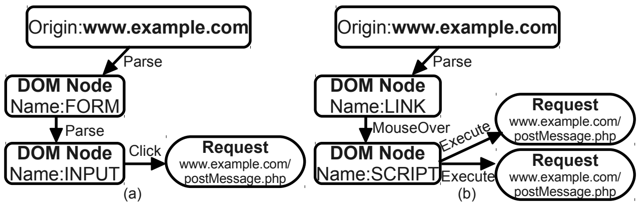


Figure 5.11: (a):An RDG slice of benign request (b):An RDG of an XSS attack

effectiveness of the RDG against detection of malicious request sent using XSS [159] and [158]. In our experiments using RDG, ClearRequest successfully protected user's state on the web server from malicious requests.

Clickjacking Attack Figure 5.12 shows an RDG slice of both a benign request and a Clickjacking attack, constructed from real world Clickjacking vulnerability WHID 2010-109 [33, 133]. In this real-world attack, users were tricked into posting a Clickjacking worm to their status updates. Victim users who clicked on the links in these malicious posts redirected to a Clickjacking web page hosted on an attacker-controlled domain. The Clickjacking web page loaded a Facebook page in an invisible *iFrame*, on top of another visible page that attracts user clicks. When victim users clicked anywhere on the page, the click fell on the invisible *iFrame*, which in turn posted the same malicious post on the victim user's status update, to further infect his/her friends, and so on.

We simulated this Clickjacking worm using virtual websites, where *www.example.com* was an honest site like Facebook, and *www.attacker.com* simulated the domain controlled by an attacker. Figure 5.12(a) shows a normal request to post a message, and 5.12(b) shows an RDG slice for the request in a Clickjacking scenario that embedded the honest site within an invisible *iFrame*. In both (a) and (b), the honest site *www.example.com* generated the request in the same way, through a click on an *INPUT* element whose parent node was a *FORM* element.

Therefore, both slices were identical, except their root nodes that indicated the page that generated the request. In (b), the root node indicated that it was loaded in an *iFrame* with an opacity value zero, i.e., being invisible to users. Such contextual information provided by the RDG enabled the detection of this Clickjacking attack in our experiments.

We tested the effectiveness of RDG with two other simulated real-world clickjacking attacks WHID 2011-96 [128] and [132]. Using RDG, ClearRequest successfully prevented these attacks in our experiments.

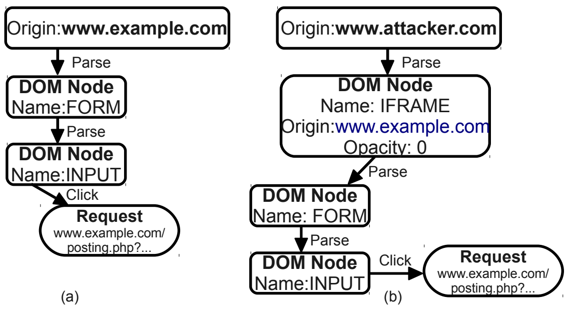


Figure 5.12: (a):An RDG slice of benign request (b):An RDG slice of a Clickjacking attack

5.7.3 Study of recent incidents of session-misuse attacks

Table 5.3 summarizes the results of our analysis of recent session-misuse attacks. It summarizes session-misuse attacks reported in the Web Hacking Incident Database (WHID) [33] and also the Common Vulnerabilities and Exposures (CVE) [90] database, both of which maintain web application vulnerabilities and web hacking incident reports. We include those attacks from the past three years that sent malicious requests from within the victim user's established web sessions. We manually analyzed the mechanism used by various attacks to trigger malicious requests. and the effectiveness of RDG against those malicious requests which resulted from different attack vectors. In total, we analyzed 385 attacks that used different attack vectors to trigger malicious requests to vulnerable hosts, and our approach was able to detect all of them.

Attack Methods	Number of Attacks	ClearRequest Detected	ClearRequest Missed
XSS	17	17	0
CSRF	368	368	0
Total	**385**	**385**	**0**

Table 5.3: Summary of real-world attacks

5.7.4 Protecting Web Applications

To make the adaptation of ClearRequest easier, we integrated it into the CodeIgniter framework. We used three open-source applications namely phpBB web forum, FUEL-CMS [50], and Ukulima [140] as a testbed. We found seven, eleven, and six RDG check-point locations for phpBB, FUEL-CMS and ukulima, respectively. Those RDG check points were the locations in the server-side code where web requests cause database interactivity. Out of the eleven security check-points in FUEL-CMS, nine were post-form submissions and the remaining two were XHR requests. In phpBB and ukulima all requests were post-form submissions. Furthermore, we automated verification of RDG checks for form submission, we modified the Form_builder class in web applications to extract form ID, submit button ID, and use *click* as the user action to trigger form submissions.

We conducted testing of the applications in the RDG slice synthesizing mode, and specify the following to derive RDG checking rules from the collected and merged RDG slices. We require matching user action, so the checking rule requires that the triggering user action must be clicks, as in those RDG slices in the synthesizing mode. Similarly, we require matching initiator element and origin. Thus, the generated checking rules also enforce that the initiator element must be a *Form* element, and the initiator origin be the root node, as seen in the synthesizing mode. With such derived RDG checking rules, the web applications detected malicious requests from session-misuse attacks in our evaluation, which attempted to post new threads or update user profile.

5.7.5 Performance Overhead

To evaluate the performance of ClearRequest, we measured the execution overhead of ClearRequest at client-side and server-side with industry-standard benchmarks. All results presented here are on an average of five runs.

Client-side overhead At the client side we measured the overhead incurred by ClearRequest on JavaScript and DOM performance and page load time.

JavaScript and DOM We measured JavaScript and DOM performance overhead introduced by ClearRequest in the web browser with the Mozilla Dromaeo benchmark. This benchmark contains JavaScript and DOM tests of different categories, ranging from text processing to object manipulation. Table 5.4 shows the result of the test. We observed a 1.43% performance overhead of ClearRequest, as compared to an unmodified Firefox browser. We also measured the performance overhead with the SunSpider JavaScript benchmark and the V8 Benchmark Suite. These benchmarks are used to measure the performance overhead incurred by our approach in JavaScript execution. In the SunSpider JavaScript benchmark tests we observed a 2.09% performance overhead, and in the V8 Benchmark Suite we observed a 2.29% slowdown. As these numbers indicate, ClearRequest incurred low performance overhead on industry-standard benchmarks.

Overhead on page loading We measured the impact of our approach on the page-load performance using the Mozilla page-loader tool, which is commonly used by Mozilla as a benchmark of Mozilla Firefox performance [93]. It takes a text file as an input that contains a list of URLs specified by the user and loads each URL ten times, reporting statistics about the time each page

Benchmark Name	Original Browser (runs/sec)	ClearRequest (runs/sec)	% Overhead
V8 Test Suite	2.496	2.44	2.29%
SunSpider	16.382	16.046	2.09%
Dromaeo	9.668	9.532	1.43%

Table 5.4: Overhead incurred by ClearRequest on industry-standard JavaScript benchmark.

Website Name	Min Slice Size (bytes)	Max Slice Size (bytes)	Avg Slice Size (bytes)
Gmail	96	391	124
Facebook	102	1490	167
Msn	85	416	129
Yahoo	95	390	122
Youtube	93	256	146

Table 5.5: Overhead incurred by ClearRequest in network traffic.

takes to load in the web browser. For each page load, we measured the time taken by the web browser. ClearRequest incurred on average 2.35% performance overhead in page loading.

Server-side performance overhead To measure server-side performance overhead, we configured phpBB forum and integrated server-side RDG check for posting the message, deleting message operations, etc. To calculate the performance overhead incurred by ClearRequest at the server-side to examine RDG slice before authorizing the request to modify server-side state, we measured the execution time with and without ClearRequest API library. In our experiments ClearRequest incurred only 3% of overhead on RDG slice check operations.

Network traffic overhead To measure the network overhead incurred by our technique RDG to enforce request dependency integrity, we measured the sizes of RDG slices for requests by logging into five real-world websites and randomly visiting several web pages. Table 5.5 shows the results of the test. We also measured the number of nodes in an RDG slice[3] .

To measure network delay introduced by RDG we used a client machine and a web server from different countries within the Asia region. The client machine in our lab measured the RTT (round-trip time) of HTTP request with and without RDG. The web server responded to each request with an empty HTML page. The two RTT estimates (with RDG and without RDG) are compared to the median of the five HTTP request measurements. The earlier procedure was repeated on three different slots in a day, such as morning, afternoon, and evening. It is important to note that although

[3]Gmail 1-8(avg:3 nodes), Facebook:1-22(avg:8 nodes), MSN:1-16bytes(avg:6 nodes), Yahoo:1-9(avg:6 nodes), Youtube: 1-13 (avg:7 nodes)

the trace was collected at one of the two connection end-hosts, the RTT in an HTTP request/response pair is not constant due to queuing delay variation. For this reason we use the median of the HTTP request/response measurements as a basis for comparison. We noticed network delay for RDG size of 200 bytes ranging from 7-18%, but averaging a modest 11.19%. On RDG size of 500 bytes we observed a delay ranging from 15-36%, but averaging to 21%.

5.8 Related Work

In this section, we compare our approach with other related research efforts. BEAP [85], implemented as a browser extension for Firefox, mitigates the CSRF attack on the client side. It can infer whether a request reflects user's intention, by using the heuristics derived from analyzing real-world web applications. If a sensitive request does not appear to reflect a users' intention, BEAP strips the authentication token from it.

RequestRodeo [67] is also a client-side solution to mitigate the CSRF attack. It works as a local HTTP proxy, and identifies HTTP requests that are suspicious as CSRF attacks by monitoring the URLs of all incoming requests and outgoing responses. It removes authentication token, such as the `Cookie` header, from outgoing HTTP requests for which the origin and target do not match.

Content Security Policy helps prevent both XSS and Clickjacking attacks. CSP is supported and implemented in the Firefox, Chrome, and Safari browsers. Content Security Policies contain a large set of directives that are set on a per-page basis. These directives indicate which servers the website trusts to deliver JavaScript, images, stylesheets, and a number of other content types embedded on the page. Through per-directive whitelists, a developer has control over the source of each piece of data included in their webpage. By default CSP does not allow inline JavaScript or calls to the *eval()* function on a website. The inline JavaScript restriction prevents execution of scripts within <*script*> blocks as well as inline event handlers within the webpage such as <*a ... onclick=“JAVASCRIPT CODE”*>. Restricting inline JavaScript substantially mitigates the risk of XSS attacks. With this restriction, attackers must compromise a whitelisted source in order to perform a successful attack. When a CSP [134] policy is applied on a website, the web browser will only execute script in source files from the whitelisted origins and will disregard everything else, including inline scripts. Hence, website administrators can reduce or even eliminate their XSS attack surface by implementing CSP. Firefox's implementation of CSP lets a protected website specify a set of whitelisted websites that can embed it through the *frame-ancestors* directive. The *X-Frame-Options* HTTP header provides a similar but more restrictive functionality, as it only allows the website to be embedded in its own origin *or* a single trusted third-party, but not both.

With *frame-ancestors*, websites can specify a list of trusted origins that can embed the page, while still protecting the website from being embedded in arbitrary and potentially malicious websites. The *frame-ancestors* directive will enforce that a website's parent and all its ancestors are in the list of whitelisted resources before it loads the website in an iframe, while *X-Frame-Options* will only enforce restrictions on an iframe's direct parent. Hence, the *X-Frame-Options* header doesn't provide the same level of security as *frame-ancestors*.

Florian proposed a gateway at the server to mitigate reflected XSS and CSRF attacks [71]. The website `S` is divided into entry pages and regular pages. Gateway does not allow entry pages to accept any input, thus preventing the reflected XSS attack. To prevent the CSRF attack, access to regular pages must contain authentication cookie and a referrer string from site `S`; otherwise the request is discarded.

NoForge [68] is a proxy-based approach deployed at the server side. It inspects and modifies client requests as well as web application responses to automatically append CSRF tokens to all hyperlinks and form submissions. NoForge is unable to handle dynamically generated HTML at the client side. Moreover, it does not differentiate between hyperlinks to itself and hyperlinks to other domains. NoForge is also susceptible to dynamic CSRF attacks [55], as its CSRF tokens could be leaked to other domains. Furthermore, NoForge can not differentiate between user intention and forged requests. For example, it is unable to handle a CSRF variant know as clickjacking, in which an honest website is embedded into a transparent frame and the user is tricked into clicking on a button on the honest website without the consent of the user. Because the request is generated by the honest website embedded into a frame that includes a correct token, it passes the check of NoForge.

Compared to the above approaches, our solution focuses more on how requests can be authenticated accurately using browser execution environment information. Instead of preventing or mitigating a specific type of attack, our solution provides client-side behavior information that helps web applications understand where and how the request is generated. We believe that the new request validation approach we proposed is more resilient to future attack techniques, as it solves the root cause of the request validation problem.

According to the HTTP standard [47], the POST method is used to mutate the state of the user on the web server, whereas the GET method is used to retrieve information. To protect against CSRF attack and prevent mutation of server-side state of a user, a web application authenticates web forms using a secret token. To achieve this goal the web application validates the web form using a hidden form element with a one-time random token. When form data is submitted by the user, the web application checks whether the form data contains a hidden one-time random token.

If no such token is found, that means the form data has been generated by a forged request and the web application thus rejects the request. Web application development frameworks such as django [38] (python), Ruby-on-Rails [108] (Ruby), Struts 2 [26] (Java) provide a mechanism to web application to add a hidden pseudo-random token to web forms and validate that token when the web form is submitted. Since data in forms are not automatically attached in a request, this solution makes it much harder to ride an existing web session by CSRF. However, injected script in a web page can read authentication tokens and send malicious HTTP requests with authentication of the valid token. Therefore, the authentication token is effective in preventing CSRF but not a general solution to help web applications judge authenticity of requests.

Moreover, to prevent CSRF attacks from mutating the server-side state of users, web applications can use the multi-step transaction technique [151]. In this technique, a web application gathers data from web forms via multiple requests and processes the final request on behalf of the user only when it receives requests from all steps. However, an attacker can simulate multi-step requests using iFrames and automatically submit the requests using JavaScript.

Another line of work is to provide ad-hoc information to a web server in HTTP Header with each request. When a web browser issues a request, it includes an `Origin` header in the request that indicates the origin of the initiator of the request [11]. Origin is expressed as a triplet consisting of protocol, host, and port number. If a web browser cannot determine the origin, it sends the value `null`. For example, for requests generated from bookmarks. Origin header validation is an effective mechanism to prevent CSRF attacks, but it is inadequate to help web applications in judging the authenticity of a request when the request is generated by injected script in a web page.

To ensure the confidentiality of sensitive data (such as cookies), Noxes [74] and NoMoXSS [141] are client-side XSS mitigation mechanisms. However, attacks that do not violate same-origin policy (SOP) [99] and modify user state on the server are not detected by these solutions.

Even though Web 2.0 and Ajax websites favor the use of JavaScript, users could opt to disable JavaScript in their browsers to mitigate XSS attacks. NoScript is a plugin for the Firefox web browser that allows users to disable JavaScript on a domain basis [106]. However, the major problem with disabling the JavaScript on a website is that it substantially reduces the website's functionality and responsiveness, as client-side script is faster than server-side scripting and use of Ajax reduces the need for a page to be reloaded. Furthermore, many sites do not work without enabling JavaScript, hence forcing users to disable protection for that site.

To protect its users from injected scripts, web applications use content filtering. Content filtering, also known as sanitization [122, 149], is a technique used by web applications to identify and remove potentially malicious content from user input. However, sanitization is difficult to implement correctly by web applications that allow HTML in user input such as blogs, web forums, etc. Furthermore, there exist parsing quirks in browser to render legacy web applications as documented in XSS-CheatSheet [119]. Attacks such as Samy worm [123] offer an example of exploiting weakness in content filtering.

5.9 Summary

Session-based authentication mechanisms are threatened by the complex environment of browsers where the client-side logic of web applications is running. In such an environment, attackers have different ways to generate a malicious request, where the server is unable to decide whether the request has been generated legitimately or generated by client-side attacks that exploit users' active sessions. Such attacks either use JavaScript to directly generate malicious web requests, or exert UI spoofing techniques to trick users into initiating them. As these malicious requests are sent to websites where the victim user has already logged in, they will be automatically authenticated and thus processed by the web server. As a result, web servers often fall victim to forged web requests. In this Chapter, we proposed a novel approach ClearRequest, which extracts the request-related dependency information in a browser environment into the *request dependency graph (RDG)*. ClearRequest sends the slices of RDG to web application servers to help them detect malicious requests in the web session.

Chapter 6

A Behavior-based Approach to Confine Malicious Browser Extensions

While the solution from Chapter 3 allows privilege separation and fine-grained access control to untrusted JavaScript within an origin to confine its behavior, today's popular web browsers run untrusted code from various sources in web sessions such as browser extensions. To enhance functionality and customization features, web browsers allow themselves to be extended by third-party code such as browser extensions. A *browser extension* extends the functionality of a web browser. Web browser extensions allow for modification of the behaviors of existing features of the browser or addition of new features. Examples of new features an extension may add include bookmark managers, password managers, IRC clients, etc. In addition, extensions can modify user views of the web pages, such as removing distracting page components such as advertisements.

Our Observations Google Chrome uses a permission-based system to control what an extension can do. Under the permission-based system, an extension can perform much more than what users expect it to do. It is the *behavior*, instead of requested permissions, that makes an extension malicious. Browser extension permissions and CSP policies can only loosely restrict the behaviors that extensions can perform at run time. Therefore, we need a solution that controls an extension based on its behaviors to prevent them from tampering with web applications.

Our Solution We propose a permissive and flexible security mechanism, SessionGuard, which protects web sessions from malicious behaviors of extensions. Instead of enforcing permission checks, SessionGuard extracts behaviors of extensions and monitors high-level actions performed by malicious extensions such as "keystroke logging", "proxying", and "runtime code download and execution". We used low-level events such as modification to DOM, use of extension APIs to access web session, network request, etc., to define high-level behavior specifications. Such a design allows a full spectrum of JavaScript functionality to untrusted extensions and regulates their behaviors in a transparent manner. The key of our approach is that high-level malicious behaviors can be efficiently matched against runtime behaviors of an unknown extension to detect malicious extensions. The key benefit of behavior-based detection is that behaviors are invariants in malicious extensions and allow detection of previously unseen malicious extensions.

SessionGuard isolates the execution of content scripts injected by extensions into web pages in a controlled environment, called the *shadow DOM*. In the shadow DOM, SessionGuard extracts content script behaviors without sacrificing their functionality and integrity of web application data. Thus, SessionGuard ensures that injected content scripts by extensions will not affect the website contents without control, protecting the integrity of web applications. The isolated environment provided by SessionGuard is different from isolated worlds [12]. In isolated worlds, content scripts get separate copies of JavaScript objects while they share DOM objects with the web page; in the shadow DOM, content scripts run on shadow copies of DOM to operate on web page data. In addition, the shadow DOM allows network access but prevents extensions from sending HTTP requests to the origin of web sessions. To reflect changes done by content scripts into the actual web page DOM, SessionGuard takes net effects of DOM changes from the shadow DOM at regular intervals and applies them to the real DOM. When applying net effects, SessionGuard allows new changes if they are benign changes, consisting of changes to style property of DOM elements and HTML tags that do not introduce third-party origins or scripts.

6.1 Background

In this section we briefly give an overview of browser extensions and describe the Google Chrome extension security model and limitation of existing defense solutions.

6.1.1 Overview

The popularity of browser extensions is further increased with extension stores, such as the Chrome Web Store [24], which offers convenient ways for users to browse, download, and install browser extensions. However, as shown in prior studies [12, 18, 78, 148], browser extensions can run with high privileges in the browser, which poses risks to web applications and users' host systems. To constrain potential threats from malicious extensions, modern browsers propose permission-based frameworks to mandate the declaration of permissions requested by extensions [12, 69]. Such permissions are presented to users during installation time, where users have to approve them if they want to install and use the extension. For instance, without permissions for access to the browser history and cookie storage, extensions cannot read users' browsing history and cookies stored (for specific domains) in the browser. A commonly requested and approved permission by an extension is to inject scripts (also known as content scripts) into web pages, which is needed for functionalities including page touch-up, page content translation, user input capturing, etc. Once scripts are injected into web pages, they have full privileges to access a web application's resources, such as issuing unauthorized transactions.

The wide popularity of browser extensions has attracted the attention of malware writers. According to the study by Kirda et al. [73], many malware programs have implementations based on browser extension. The malicious extension is one of the main threats to the security of web users [19, 131]. Another study by Chia [114] of 5,943 Chrome extensions shows that 35% of Chrome extensions request permissions to access user data on all websites. Unrestricted access to web applications contents from extensions raises a major challenge to the integrity of web applications.

The permission-based extension framework cannot address the above mentioned challenge, as users have to approve permissions during extension installation time if they want to use the requesting browser extensions. For example, after content scripts enter web sessions, the extension permission enforcement mechanism cannot have further control. On the other hand, existing security mechanisms within a web page also have no sufficient protection against malicious content scripts injected by browser extensions. A well-known protection mechanism to regulate content injection in a web application is Content Security Policy (CSP) [134]. CSP provides a declarative content restriction policy in an HTTP header that the browser can enforce. CSP defines directives associated with various types of contents that allow developers to create whitelists of content sources and instruct client browsers to only load, execute, or render content from those trusted sources. However, CSP is an all-or-none approach and it is limited to content injected into web pages.

6.1.2 Google Chrome Extension

The Google Chrome extension security model [12] aims to protect users from buggy extensions by following three security principles: *least privilege*, *privilege separation*, and *strong isolation*. According to the Chrome extension security model, extensions are limited to a set of privileges chosen at install time. In addition, extensions are divided into three isolated components: *content scripts*, *an extension core*, and *a native binary*. The *content script* gets injected into a web page when the web page is loaded. The injected content script runs in the same process of the web page. Each content script is allowed to interact with the DOM of a web page where it is injected. However, by default it does not have permission to get access to API's provided to extensions, or send network requests, except the ability to send messages to the extension core. Each *extension core* is executed in a dedicated process from the browser process or web page's renderer process, and its accesses to system resources and browser interfaces are controlled by using a manifest file. The extensions core has all the privileges of browser extension, but to achieve the least privilege principle, the power of the extension core is limited to the permission it has explicitly requested in the `manifest.json` file, which is in the extension package. In addition, extension core is not allowed to access the user's file system. To access the host machine, an extension can optionally include a *native binary*, which has a user's full privileges. Furthermore, extensions that contain a native binary component are not permitted in the gallery unless the extension developer signs a contract with Google.

Each component of an extension runs in a different process. The content scripts are injected into a web page and run in the sandboxed process of the web page. The extension core and native binary each run in separate dedicated processes. Even though the security model of Google Chrome extensions is comprehensive, malicious Chrome extensions still exist. Threats to a web session integrity from malicious extensions are a major security concern.

It is common for extensions to inject scripts. In a manual study using fifty extensions from ten different categories on Google Chrome Extension gallery[1], we selected five unique extensions from each category of the *Recommended* section of the Google Chrome gallery. We found that 78% (39 out of 50) extensions injected content scripts into web pages.

To prevent users from installing malicious extensions, Google Chrome disables the ability to install extensions outside of its official web store of extensions. In addition, Google Chrome adopts a permission-based system to control what an extension can do. For various sensitive operations or access to sensitive resources, such as network capabilities, cookies, geographic location, browsing history, etc., it enforces extensions to declare and request corresponding permissions [25]. When users install an extension, the permissions requested by the extension are shown to the users for approval; users may also be warned for the security implications of dangerous permissions. However, users may not be able to make sensible judgment on whether to allow or deny such permissions, especially when it is subjective. Furthermore, as users want to use such extensions, they may become habituated to permission prompts and click through them without actually checking them [23].

Mozilla Firefox does not use the permission based framework like Google Chrome. Instead, Mozilla add-on gallery performs automatic validation of extension code using a set of static checks,

[1]Categories:Accessibility, Blogging, By Google, Developers Tool, Fun, Photos, Productivity, Search Tools, Shopping and Social

which aims to identify common bad practices and possible security problems in extension code. A limited set of static checks performed by an add-on validation module on the extension code includes checking the use of *eval*, *setTimeout*, *setInterval* functions to evaluate remote code, remote script injection, loading content from unsecured third-party into secured sites, <iframe> elements with no type information, insertion of remote content with *innerHTML*. This limited set of checks identifies bad practices in extension code, but they lack the ability to comprehensively identify malicious behaviors of extensions.

Even though web browsers regulate resources of one web application from another using same-origin policy (SOP) [99], browser extensions are allowed to inject content script into web pages. Once content scripts are injected into web pages they can load resources from untrusted third-parties, access web resources and send it to extension core, etc. Therefore, it is critical to regulate behaviors of browser extensions to protect the integrity of web sessions.

6.1.3 Motivating Example

In this section we describe the threats from malicious extensions to web sessions. We use an example based on the extension *Turn Off the Lights* throughout the chapter to illustrate our solution.

Turn Off the Lights This extension inserts a lamp button in the browser menu bar, which makes an active web page dark when clicked. It sets everything on the screen to dark except the Flash and HTML5 video. It injects content scripts into all websites users visit. The content script in turn injects a DIV element and STYLE elements. The style element covers the entire page with the injected DIV element by setting its *z-index* property to the value *999*. Elements on web pages are vertically layered according to the *z-index* property values with the highest *z-index* value element on top. Therefore, the DIV element injected by the extension in the web page will be on top of all existing elements in the page. It also sets the color of the DIV region to black. Furthermore, the content script also injects a JavaScript file named *injected.js* from the extension directory into web pages. The *injected.js* file checks for video element in web pages and changes its style value to make it above the DIV element that covers the entire web page. To do so, it sets the *z-index* property value to *1000* for video element.

To achieve its functionality the "Turn Off the Lights" extension requests permissions shown in the Figure 6.1. By installing the "Turn Off the Lights" extension users expect to enhance usability by minimizing the distraction for a more pleasant video viewing experience. With the granted permissions, the extension has permissions to perform dangerous activities such as execute arbitrary JavaScript code in web pages. The permissions are dangerous enough to enable a

```
"content_scripts": [ {
      "css": [ "css/light.css" ],
      "js": [ "js/content.js" ],
      "matches": [ "http://*/*", "https://*/*" ],
      "run_at": "document_end"
   } ],
"permissions": ["contextMenus", "tabs",
                "http://*/*", "https://*/*"]
```

Figure 6.1: A sample of manifest file of the *Turn Off the Lights* extension

remote-controlled web trojan, such as JS/Febipos.A, demonstrated in the Section 2.3.

Summary Malicious extension can perform dangerous activities in web sessions that are not expected with the permissions given to them. When malicious extensions tamper with web application logic, it cannot be stopped by permission-based mechanisms.

6.1.4 High-level Behaviors of Extensions

The behavior of an extension can be described as its observable effect on the execution environment. In this paper, we focus on the effect of a sequence of APIs used by extensions and modification done to web page DOM. Since extension APIs provided by a browser capture the interaction of an extension with web applications, we aim to capture integrity violations using unintended interactions with APIs and DOM. An extension behavior refers to the range of activities exhibited by the extension as a result of granted permissions to the extension.

In our running example, *Turn Off the Lights*, it injects content scripts into web pages. The injected content scripts add dynamic SCRIPT elements as well as DIV and Style elements in all web pages users visit. That is, it not only introduces new elements in web pages but also inserts dynamic code in web pages. Introducing dynamic code in web pages posses a threat to the integrity of web applications. Hence, it is necessary to control the behaviors of dynamically injected untrusted code in web sessions. Content scripts, if injected into web pages, can exhibit various behaviors. For example, behaviors of a content script can include injecting new HTML elements into the web page's DOM, introducing contents from new origins, modifying existing DOM elements' value, changing style properties of DOM elements, etc. Behaviors of content scripts can be different in different extensions even though they have acquired the same permissions. For instance, two extensions with the same *content script* permission can have different behaviors, such as one that introduces new origins in web pages whereas another changes style attributes of elements.

6.2 Design of SessionGuard

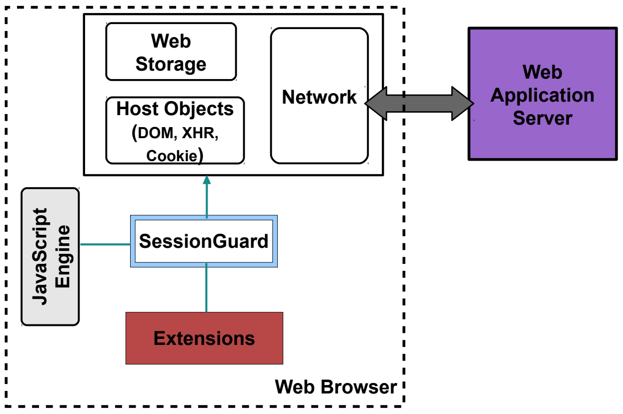


Figure 6.2: Overview of SessionGuard

Extensions that interact with arbitrary web applications on client browsers pose a wide variety of threats to the integrity of web sessions. In this work, we focus on threats from JavaScript-based malicious extensions.

In-scope threats Extensions written by untrusted third-parties can be malicious and tamper with web sessions or steal web application data. We address such threats to web sessions in this work.

Out-of-scope threats Our work does not address threats from a binary component of extension. If an extension contains the binary component then it can compromise a host system, preventing which would require mechanisms on the operating system outside the browser. Thus, we leave this type of threat out of the scope of our approach.

6.2.1 Components of SessionGuard

The primary goal of SessionGuard is to effectively protect the integrity of a web application from malicious extensions. To achieve this goal, SessionGuard provides an isolated environment to capture the behaviors of extensions, shown in Figure 6.2. SessionGuard detects integrity violations based on the observation of the execution of malicious extension. That is, SessionGuard executes and monitors an extension in a controlled analysis environment. Then it extracts the behaviors that characterize the execution of the extension from the controlled analysis environment. SessionGuard inspects observable effects of extensions and selectively applies those effects into real the web page DOM based on the security policy of the protected web session.

Figure 6.3 shows the architecture of SessionGuard that extracts behaviors of an extension. SessionGuard needs to record the interactions of content scripts with web pages and extension

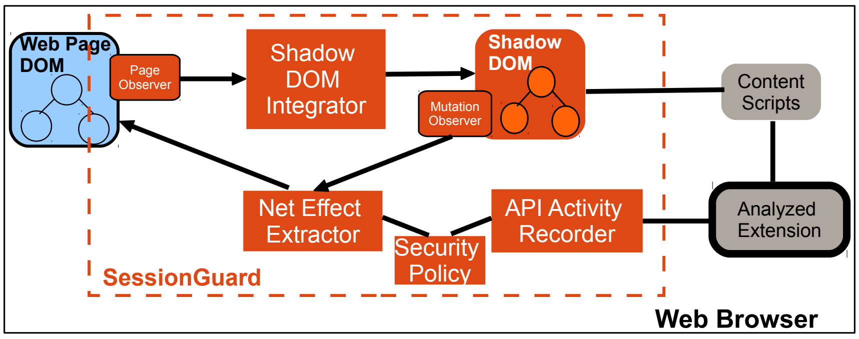


Figure 6.3: Architecture of the SessionGuard

APIs used by the extension. To monitor the behaviors of content scripts in a web page, SessionGuard interposes a layer of shadow DOM between the web page DOM and content scripts. JavaScript from a web page runs on a real DOM tree, and changes done by them occur in the real DOM tree, whereas content scripts run on their "own" shadow copy of the DOM tree and changes made by content scripts on shadow DOM objects are selectively reflected into the real DOM tree only if those behaviors are allowed by users. The *Mutation Observer* component in the shadow DOM monitors for changes performed in the shadow DOM tree by content scripts and sends those changes to the *Net Effect Extractor* module, which in-turn reflects those changes into the web page DOM, subject to the security policy constraints. Similarly, the *Page Observer* component in the real web page DOM monitors for changes made in the web page DOM and forwards them to the *Shadow DOM Integrator* module, which in-turn reflects DOM data into the shadow DOM tree.

6.2.2 The Net Effect Extractor Component

The *Net Effect Extractor* component of the SessionGuard extracts behaviors of content scripts injected by the extension under analysis. To extract behaviors of content script, SessionGuard runs it inside a controlled environment and monitors the important security-related actions performed by content script on web sessions. In particular, SessionGuard isolates content scripts of extensions and web pages' DOM objects by providing a shadow copy of DOM to content scripts. Hence, content scripts of each extension operate on a shadow copy of real DOM objects rather than accessing real DOM objects of the web page. This helps to detect behaviors of content scripts and allows recovering from unwanted changes done by malicious content scripts. The summary of changes extracted from shadow DOM defines the behaviors of the content scripts injected by the extension into web session. To extract actions from shadow DOM the difference between shadow DOM and real DOM is calculated. The changes in the shadow DOM as compared with real DOM show the

Net Effect	Description
Element newly added	The *newly added* nodes mean they are present in the shadow DOM but not in the real web page DOM.
Element removed	The *removed* DOM nodes are DOM elements that were present in the real web page DOM but now they are not present in the shadow DOM.
Element reparented	The *reparented* nodes are present in both the real DOM as well as the shadow DOM but they moved to be children of new parents.
Element reordered	The *reordered* nodes that are present in both the real DOM and the shadow DOM and are children of same parent but now moved to new location in their parentNode's childList.
Element attributeChanged	The *attributeChanged* means value of a DOM element attribute is changed.
Event listener register	A call back function registered on particular event (such as submit, click, mouseover, keypress ,etc.)
Communicate with the extension core	Send message or data to extension core from content scripts
New origin introduction	New element added to the web page DOM that loads resources from third-party origin.

Table 6.1: Net effects captured from the shadow DOM tree

actions performed by content scripts on the web page such as new DOM elements added, existing DOM elements removed, DOM elements reordered or reparented, etc. These changes allow identification of an extension's behavior on web contents. By default, SessionGuard selects clean changes and discards changes that introduce a script element and network requests into the DOM of a web page.

SessionGuard uses mutation observers [42] to extract behaviors of content scripts from the shadow DOM. It records changes done by content scripts into shadow DOM using mutation observers and reflects the net effect of those changes into the real DOM tree of the web page if allowed by the security policy. Net effects are a summary of changes done by a content script into shadow DOM after comparing with the real web page DOM. It describes how the document is different in the shadow DOM as compared to the real web page DOM. The difference includes elements that are newly added, removed, reordered, reparented, and attribute modified. Table 6.1 shows net effects of the content scripts extracted by the *Net Effect Extractor* component of the SessionGuard.

By default, SessionGuard allows all DOM tree modifications except changes that introduce HTTP requests or JavaScript elements into real web page DOM. SessionGuard prevents changes made in the shadow DOM, such as SCRIPT elements and DOM elements with *href* or *src* attributes from the changes, before reflecting them into the real web page DOM tree. The *Net Effect Extractor* module is responsible for enforcing such security policies. The communication between the *Page Observer* and the *Shadow DOM Integrator* is in the form of message passing. Similarly, message passing is used to communicate between the *Mutation Observer* in shadow DOM tree and the *Net Effect Extractor*. When components receive a message of DOM changes, they extract the message and process it.

6.2.3 The API Activity Monitor Component

The *API Activity Monitor* component of the SessionGuard monitors behaviors of extension core and records APIs used by the extension under observation. SessionGuard uses in-browser monitoring of APIs used by extensions to record APIs used by extensions to interact with web session. The set of activities an extension can perform using extension APIs in web sessions is as follows:

- *Cookie-API:* If an extension invokes APIs provided by the browser to get handle to a cookie store then it indicates the extension has a behavior of cookie access.
- *HTTP-interception:* This behavior indicates an extension invoking APIs that allow interception of HTTP requests/responses to read/modify network traffic.
- *Web-storage:* This behavior indicates that an extension invokes APIs that allow the extension to get access to the user's web history and localStorage of web applications.

To regulate the behaviors of extensions, web application developers can specify a behavior control policy, which is composed of *directives* like Content Security Policy (CSP) [134]. Each directive specifies how the behavior of an extension code is to be controlled on the protected web application. The behavior control policy is an opt-in mechanism to allow a web application to specify security policies to control the interaction of extension APIs with the web application's session using a special response header such as *X-Behavior-Control-Security-Policy*.

Examples of Security Policy to Control Extension Core Behavior

Example 1. A website wants extensions to interact with its web session cookies.

```
X-Behavior-Control-Security-Policy:
cookie-api:*;
```

The security policy in this example allows all extensions to access protected web session cookies.

Example 2. A bank site wants extensions should not access its web session cookies and client-side web storage. Also the bank site wants extensions should not intercept its HTTP requests.

```
X-Behavior-Control-Security-Policy:
cookie-api:none; http-interception:none;
web-storage:none;
```

The security policy in this example doesn't allow extensions to access cookies, send HTTP requests, or access client-side web storage of the protected web session.

Challenges The main challenge in achieving behavior confinement on content scripts injected by extensions into web sessions is from dynamic behaviors of JavaScript on DOM. For example, JavaScript can create new origin, insert new DOM elements, delete existing DOM elements, change style properties of a DOM element, etc. during its execution. SessionGuard should identify net effects of content scripts and should confine them inside the corresponding shadow DOM tree to prevent breaches on a real DOM tree. In particular, SessionGuard needs to track the creation of new script element into shadow DOM and prevent a reflection of it into the real web page DOM; otherwise, content scripts would easily bypass protection mechanisms by injecting malicious script elements into a web page's DOM. While creating a shadow DOM from the real web page DOM, we insert a web page *SCRIPT* element as a *NO-SCRIPT* element in the shadow DOM to prevent them from executing in the shadow DOM. In other words, SessionGuard provides a copy of DOM to content scripts that don't include executable SCRIPT elements of the web page. SessionGuard adds SCRIPT elements as text contents to maintain a serialized DOM tree order between the shadow DOM and the real web page DOM. Because this makes it easy to identify net effects by performing comparisons between the shadow DOM and the real DOM.

Another challenge in reflecting changes from the shadow DOM into the real web page DOM and vice versa was to avoid cycles of modifications. SessionGuard reflects changes in the real web page into the shadow DOM and changes occurred in the shadow DOM into the real web page. Hence, this process can cause cycles. To avoid cycles, we disable observers before applying collective changes and start observer after applying changes. In addition, SessionGuard first removes all reordered and reparented nodes from DOM, then inserts new nodes added to DOM and reordered and reparented nodes at new locations.

6.3 Implementation of SessionGuard

We implemented SessionGuard in the Chromium web browser version 24.0.1290.0(160599). The execution environment of web application's session contains a set of components to process and render the content and communicate with the web application servers. These components are the *network* library, *host objects*, *web storage*, and the *JavaScript engine*. The network library is responsible for sending HTTP requests, receiving HTTP responses, and provides APIs to extensions to intercept or modify requests in-flight and to observe network traffic. Host objects include the Document Object Model (DOM) objects, XMLHttpRequest (XHR) object, cookies, etc. By default the JavaScript engine executes JavaScript from web pages as well as content scripts from browser extensions in an isolated environment.

In Chromium's source code, the *binding* module glues the V8 JavaScript engine and the WebKit DOM. We intercepted JavaScript calls to the DOM in the binding module and obtained the JavaScript context of the current JavaScript engine. Content scripts injected by extension in web pages run in *IsolatedWorld*. In Chromium, JavaScript context provides information whether it is in *IsolatedWorld* or not. If it is not in *IsolatedWorld* then the JavaScript context belongs to a web page and no security check is required. Otherwise, the JavaScript context belongs to an extension, and web page DOM write/modify operations intercepted by SessionGuard. We use this information to regulate network requests initiated by content scripts to the web page origin. SessionGuard intercepted APIs available for an extension under *cookies* permission in the manifest file.

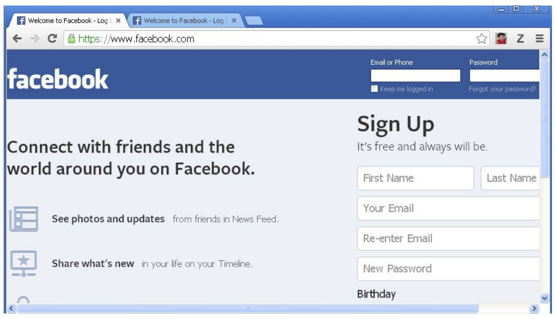
Shadow DOM Construction We mirror the real web page DOM tree to create a shadow DOM to run content scripts. We created a separate browser tab for the shadow DOM and constructed a copy of objects from the browser tab of the real web page DOM in the browser tab of the shadow DOM. We leverage a browser's SOP policy to isolate the shadow DOM in the newly created tab. In the shadow DOM, we assigned tab ID instead of the web page origin, thus the browser's same-origin policy ensures that the shadow DOM by default has no access to cookies or other data belonging to the real web page.

SessionGuard maintains a mapping from the real web page DOM node to the shadow DOM node, so that when we process the inserted, removed, reordered, reparented, and modified attributes nodes, we can efficiently locate the corresponding shadow DOM nodes by simply looking them up in the map. To avoid cycles, SessionGuard first removes all reordered, and reparented nodes before inserting all newly added, reordered and reparented DOM nodes.

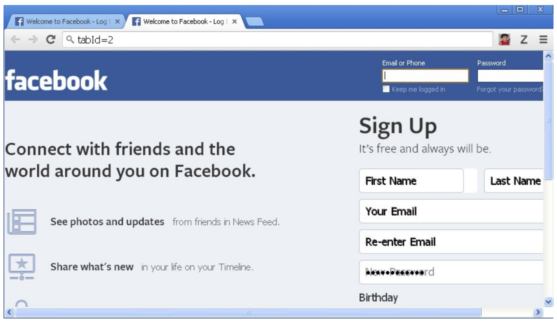
For each node added to the web page DOM, SessionGuard creates a node in the shadow DOM. In addition, SessionGuard creates *NO-SCRIPT* elements for all *SCRIPT* DOM elements in web pages rather than inserting them as *SCRIPT* elements and executing them in the shadow DOM as well. SessionGuard uses messages to communicate between the browser tab of the real web page DOM and browser tab of the shadow DOM. Ideally, each extension should get a separate copy of shadow DOM. However, our current implementation is limited to only one copy of the shadow DOM for all extensions. Therefore, we used only one extension at a time in our isolated environment to get correct behavior of content scripts injected by extensions.

Reflection of Shadow DOM Changes into Real Web Page DOM To reflect changes done in the shadow DOM into the real web page, we registered `MutationObserver` on the shadow DOM nodes rather than Mutation Events. Mutation Events are slow and they are fired too frequently in a synchronous way, whereas Mutation Observers are faster and invoked asynchronously [35]. In MutationObserver, a callback on observing nodes will not be fired until the DOM has finished changes. When the callback triggers it is supplied with a list of the changes occurred in the DOM. We registered mutation observers on all DOM nodes. When one or more nodes are added or removed from a node's childNodes collection, a "childList" record is created with the target set of nodes, which is a contiguous sequence of nodes that was inserted or removed. In addition, the position in the DOM at which the nodes were inserted or removed is recorded via the previousSibling and nextSibling attributes of a mutation observer object. In mutation observers, when an attribute is inserted, removed, or modified, an *attribute* record is created with the target set to the element owning the attribute.

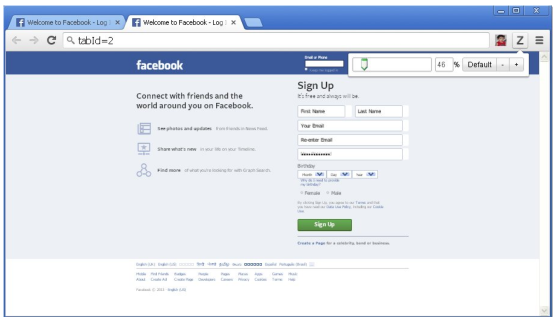
Figure 6.4 shows screen shots of the real-world web pages (Facebook in this example), the shadow DOM and reflection of changes done by content scripts in the shadow DOM into the real web page. For illustration purpose only, we show the shadow DOM contents in a separate tab in the web browser. The shadow DOM web page assigned with a `tabId` rather than original web page domain to leverage the SOP policy implemented by browsers. Figure 6.4(a) shows the home page of the Facebook web application. Figure 6.4(b) shows the shadow DOM of the Facebook's home page. We used the *Zoom* extension from the Google Chrome gallery to zoom the web page. Figure 6.4(c) shows content scripts of the `Zoom` extension modified style properties of DOM contents in the shadow DOM. It zooms out the Facebook's home web page content into shadow DOM. Figure 6.4(d) shows the net effects of changes performed in the shadow DOM by the *Zoom* extension are reflected into real web page's DOM.



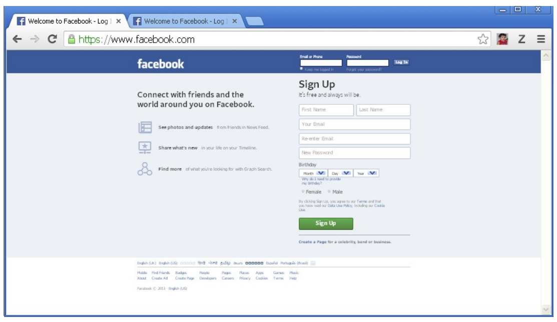
(a) Original web page DOM


facebook
tabId=2
Sign Up
It's free and always will be.
Connect with friends and the world around you on Facebook.
First Name
Last Name
Your Email
Re-enter Email
Birthday

(b) Shadow DOM provided to browser extensions



(c) Style change in the shadow DOM by ZOOM browser extension



(d) Style change reflected in the real web page DOM from the shadow DOM

Figure 6.4: Screenshots of the original web page and the shadow DOM of the web page

6.4 Evaluation

We evaluated SessionGuard to ensure that it is able to protect the integrity of web applications. We also measured the runtime overhead of SessionGuard and tested its compatibility with real-world extensions. All experiments were conducted on a computer with an Intel Core 2 Duo 2.33 GHz CPU and 4GB RAM, running Linux Mint 12.

6.4.1 Effectiveness

To measure the effectiveness of our approach we created two extensions, namely *ExtBot* and *adInjector*. The *ExtBot* download commands from our controlled server and executes them victim user's Facebook session. The *adInjector* extension injected advertisement from our domain in a website. We created a website `attacker.com` in a virtual machine to act as our botnet control server and domain to serve advertisements.

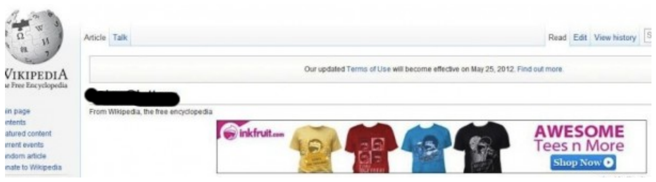

WIKIPEDIA
The Free Encyclopedia
Article
Talk
Read
Edit
View history
Our updated Terms of Use will become effective on May 25, 2012. Find out more.
[redacted]
From Wikipedia, the free encyclopedia
inkfruit.com
AWESOME
Tees n More
Shop Now

Figure 6.5: An Advertisement injection into Wikipedia by `I Want This` malicious extension

Download and execute commands

Trojan:JS/Febipos.A: This is a real world malicious botnet extension that downloads and executes commands from its controlling server as describe in section 2.3. To simulate the behavior of download and execute commands, we constructed extension *ExtBot* in our test environment. If the user is logged in to Facebook, then *ExtBot* downloaded dynamic scripts from `attacker.com` and executed within a victim user's Facebook session to like all posts on the victim user's profile.

The *ExtBot* sends dynamically downloaded code to content scripts that scans all DOM elements from Facebook's web page that contains the name *like*. Then, it automatically invokes *click* event on *Like* buttons to like all posts on the victim user's profile. SessionGuard executes dynamically downloaded scripts into the shadow DOM and prevents network requests from being initiated from the shadow DOM. Thus, SessionGuard prevented botnet behavior of the malicious extension in our test environment.

Inject Malicious Advertisements to Generate Revenue

I Want This: This is a real world example of a malicious extension that injects advertisement without the consent of the website publishers [131]. In particular, this real-world malicious extension injected ads in non-profit websites, such as `en.wikipedia.org`, that do not support advertisements, as shown in the Figure 6.5.

The Figure 6.6 shows an example of a malicious extension manifest file. The malicious extension in its manifest requests to access `en.wikipedia.org/` HTTP and HTTPS web pages to inject `injectAd.js` content scripts. This injected malicious script inserted an advertisement into ad-free web application Wikipedia. We created an extension `adInjector` that loaded static advertisements from our domain `attacker.com` and injected them into the web pages of *Wikipedia*.

The *adInjector* extension injects content scripts into Wikipedia sites that add new cross-origin elements into Wikipedia web pages. SessionGuard captured this behavior of *adInjector* at run-time using shadow DOM. To protect the integrity of web applications, the basic policy of Ses-

```
"content_scripts" :
[ { "matches":
 ["https://http://en.wikipedia.org/*",
  "http://http://en.wikipedia.org/*"],
  "js": ["injectAd.js"] } ],
"permissions":
 ["tabs", "https://http://en.wikipedia.org/*",
  "http://http://en.wikipedia.org/*" ]
```

Figure 6.6: A sample of manifest file to inject advertisements in Wikipedia

sionGuard is to prevent content scripts from introducing new origins or JavaScript code in the web page's DOM. According to this policy, before reflecting changes into the real web page DOM, SessionGuard discarded those DOM elements added by *adInjector* into the shadow DOM that introduced the new origins. Thus, SessionGuard successfully prevented content scripts from compromising the integrity of the web application by introducing new origins or dynamic code.

6.4.2 Compatibility

To evaluate the compatibility of SessionGuard, we tested it with 10 real world extensions[1]. SessionGuard is compatible with all extensions we evaluated.

Turn Off the Lights

This is a typical example on how SessionGuard supports existing extensions. SessionGuard executed content scripts injected by this extension into the shadow DOM. Content scripts inserted the *injected.js* JavaScript file into the shadow DOM. In addition, it also inserted DIV and STYLE element into the shadow DOM. These inserted DIV and STYLE elements created a black layer on all web page elements except a pause/active video element on the page. SessionGuard only accepted DIV and STYLE elements from the shadow DOM, while reflecting changes into the real web page DOM. Thus, we observed a black layer is also created in the real web page DOM for all DOM elements, except the pause/active video element in the web page. In addition, SessionGuard created a *NO-SCRIPT* element into the real web DOM, instead of a *SCRIPT* element for *injected.js* into the shadow DOM. Thus, SessionGuard prevented the injection of dynamic JavaScript into the web page, but accepted net effects of the changes done by content script in the shadow DOM.

[1]Adblock Plus, Alexa Traffic Rank, Custom Google Background, Color Changer for Facebook, Google Dictionary, Highlight Keywords for Google Search, Speed Tracer, Turn Off the Lights, Zoom, LastPass

Benchmark Name	Original Browser (runs/sec)	SessionGuard (runs/sec)	% Overhead
Dromaeo DOM Core	338.32	311.56	8.59%
V8 Test Suite	765.09	749.358	2.1%

Table 6.2: Overhead incurred by SessionGuard on industry-standard benchmark

6.4.3 Performance

We inserted security checks in five cookie APIs [2] provided by the Google Chrome extension system, and allowed access to the web application's cookies if the security policy specified by the web application allowed it. We also intercepted APIs such as executeScript, content script injection, event listener registration, and others. We created an extension that uses all five cookie APIs and we hooked and measured the performance overhead of API interception, both with and without our approach. SessionGuard incurred, on average, 1.20% performance overhead to perform access control checks.

Overhead incurred on real-world websites We measured the performance overhead of SessionGuard occurred upon page load of the 25 different top websites as listed by Alexa. To measure page load latency, we visited websites with and without SessionGuard. Each experiment was repeated 10 times, and the averages of obtained results were recorded. SessionGuard incurred, on average, 8.29% performance overhead. This overhead is introduced by SessionGuard, due to creation of the shadow DOM copy for content scripts of browser extensions.

Industry Standard Benchmark We also measured the overhead of SessionGuard on industry-standard benchmarks. Table 6.2 shows the performance overhead incurred by SessionGuard, as compared to an unmodified Google Chrome web browser. On the Mozilla Dromaeo DOM Core benchmark, we observed an 8.59% performance overhead, and on the V8-test suite, we observed a 2.1% slowdown. As these numbers indicate, SessionGuard incurs low performance overhead on industry-standard benchmarks.

6.5 Related Work

In this section, we discuss the existing research efforts that mitigate threats from browser extensions.

[2]Cookie APIs: get, getAll, set, remove, getALLCookieStores

Ter Louw et al. [81] developed a solution to monitor XPCOM calls by extensions to a subset of Firefox's privileged APIs, in order to secure the extension installation process. However, primary extension APIs remain unprotected, and extensions still have full privileged to access web sessions. Barth et al. [12] developed the multi-process security architecture used for Google Chrome extensions. It focuses on threats from a malicious web page exploiting a buggy extension, rather than a malicious extension compromising web session integrity and privacy. Therefore, the Chrome security mechanism usually grants extra permissions to content scripts. SessionGuard provides the ability to check and restrict how browser extensions interact with web sessions, for more precise control on browser extensions than the default extension manifest.

Liu et al. [78] examined Google Chrome extension security mechanism against malicious extensions. They introduced macro privileges in the Google Chrome extension permission system, to limit by-default access of extensions to sensitive information on websites. Our approach also focuses on malicious extension, but without introducing macro privileges. SessionGuard extracts behaviors of content script and selectively applies them into the web page DOM.

The JetPack [102] framework is Mozilla's extension development framework, and aims to improve security by containing any vulnerabilities in an extension module. From the security perspective, it aims to reduce interaction interfaces between extensions and browser resources. However, the current implementation of JetPack technology is fully-privileged, which allows uncontrolled access of users and web applications data.

SpyShield [76] uses an access control proxy to control communications between untrusted add-ons and their host application. It aims to protect users from spy add-ons. Some static approaches are also proposed to detect vulnerabilities in JavaScript-based widgets. GATEKEEPER [52] is a static approach for enforcing security and reliability policies for JavaScript programs. VEX [10] proposed a static information flow technique to perform an analysis of the JavaScript code of browser extensions to identify potential security errors. Similar to the Chrome browser, VEX does not aim to detect malicious extensions. As compared to the above approaches, our solution focuses on controlling dangerous behaviors of content scripts, instead of vulnerabilities.

Egele et al. [43] developed a flow-tracking technique that examines the guest system states from outside, with complete knowledge of all important data structures. It uses a dynamic taint analysis technique to analyze how sensitive information is processed by the system to monitor the behaviors of untrusted browser extensions.

Sabre [36] is a system that uses in-browser information flow-tracking and analyzes the browser extensions. It produces an alert when an extension attempts to access any sensitive information in an unsafe way.

Another line of research [51,136,147] proposed new browser architectures to improve security. They use process-level isolation for different components of a browser. However, threats from malicious extensions to web sessions were not considered in these architectures.

One class of research solutions [49,145,157,160,166] used fault isolation and system call interposition techniques to securely run native plug-in code. These techniques focus on the isolation of untrusted native code, and they are complementary to our work. Our approach focuses on threats by JavaScript of browser extensions to web sessions.

Akhawe et al. [5] proposed a privilege separation technique for HTML5 applications. It uses abstractions available in existing browsers to isolate untrusted components in web applications into an arbitrary number of temporary origins, in order to leverage SOP policy for isolation.

Akhawe et al. [4] proposed a data-confined sandbox (DCS) primitive for client-side HTML5 applications that handle sensitive data. The proposed approach by the authors executes web applications code by handling sensitive data in data-confined sandboxes, and provides complete mediation on data communication channels.

CSP [134] mechanism aims to protect the integrity of web applications from content injection. However, it is *all-or-none* approach that either allows content in web pages to execute with full privileges of web application or not at all. In addition, CSP blocks execution of inline script and inline style sheets thus enforcing content scripts to host JavaScripts/stylesheets they want to inject in web pages on at least one of the whitelisted domain of web applications. Furthermore, a recent survey revealed that only 79 out of the Alexa Top 1,000,000 websites [126] implement CSP, showing that CSP has a very low adoption rate [34].

BLUEPRINT [82] uses an alternative approach to protect the integrity of web applications from content injection. BLUEPRINT treats the HTML parsing component of a browser as untrustworthy, and instead uses web servers to parse the document and create output representing the structure of the web page (the blueprint). This is sent to the browser, which uses the blueprint to build the document exactly as intended by the web application. SessionGuard's main goal is to extract the behavior of content scripts to protect integrity of web sessions.

Code signing allows identification of the author of a piece of code. Java uses code signing to establish trust by the behavior of an executable, such as Java applet [103]. Netscape Communicator uses object signing that permits Java code to request specific kinds of access to local file-system [37]. The JavaScript security model in Netscape Communicator is based upon the Java security model for verifying the signed script of the object. JavaScript in web pages is sandboxed and not allowed to access local file system. Signed scripts certify the owner of the script, therefore only signed scripts can be granted extended privileges, such as, reading the user's file system [96].

6.6 Summary

To enhance functionality and customization features, web browsers allow themselves to be extended by third-party code such as browser extensions. In modern web browsers, code in the extension is usually granted with more privileges than code in web pages. In this paper, we proposed a novel approach, SessionGuard, to prevent malicious extensions from compromising the integrity of web applications. SessionGuard extracts net effects of browser extensions in web sessions by executing them in a controlled environment, and monitoring and recording their interactions within web sessions. We implemented SessionGuard in the Chromium web browser. Our evaluation and measurements demonstrated the effectiveness and efficiency of SessionGuard.

Chapter 7

Conclusion

This thesis proposed solutions to incorporate necessary behavior control mechanisms into web browsers to effectively battle against threats to the integrity of sessions. We have begun with the systematic analysis of the execution environment provided by web browsers and proposed techniques to control behaviors of untrusted code. First, we examined protection mechanisms available inside an origin, and the existing research efforts to protect a web application's session from malicious embedded JavaScript. We observed that there is a lack of behavior control mechanism in the JavaScript context component of the JavaScript engine. This allows attackers to compromise the integrity of web sessions. To protect web sessions and address this problem, we proposed the behavior control approach that uses privilege separation and fine-grained access control techniques inside the origin.

Although our approach provides behavior confinement for any untrusted JavaScript embedded within web pages by using privilege separation, we also extracted client-side information on how a request is generated and transported to the server. The complex browser environment provides various ways for an attacker to forge malicious requests into an active session to modify server-side state of the web users. Therefore, it is crucial for web servers to know the client-side code behavior that generated state modifying requests to the server. To help web servers distinguish benign requests from malicious ones, information on how those requests are generated at the client-side is also critical.

In the process of designing and implementing behavior control mechanisms to untrusted JavaScript embedded in a web page, we found that malicious browser extensions can also access and modify web session data. Modern browsers implicitly trust extensions and allow uncontrolled behaviors of extensions within web sessions. To protect the integrity of web sessions, we enhanced existing browsers, to confine behaviors of extensions within web sessions.

This dissertation demonstrates that behavior is the key in detecting malicious code, and it provides an effective way to combat integrity problems within web sessions. This thesis proposed a line of solutions to extract and control behaviors of untrusted code in the execution environment, which is a promising direction to build a more secure platform for web applications.

Bibliography

[1] Ben Adida, Adam Barth, and Collin Jackson. Rootkits for JavaScript Environments. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[2] ADSafe. Adsafe. http://www.adsafe.org/.

[3] Secure Network Security Research Advisory. Citrix xencenterweb multiple vulnerabilities. http://securenetwork.it/ricerca/advisory/download/SN-2009-01.txt, 2009.

[4] Devdatta Akhawe, Frank Li, Warren He, Prateek Saxena, and Dawn Song. Data-confined html5 applications. In *Proceedings of the International Conference on European Symposium on Research in Computer Security (ESORICS)*, 2013.

[5] Devdatta Akhawe, Prateek Saxena, and Dawn Song. Privilege separation in html5 applications. In *Proceedings of the Usenix Security Symposium (Usenix Security)*, 2012.

[6] Alexa. Top sites. http://www.alexa.com/topsites, 2009.

[7] Alexa Internet, Inc. Top sites. `http://www.alexa.com/topsites`, 2013.

[8] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Chrisopher Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the Symposium on Information, Computer and Communication Security (ASIACCS)*, 2010.

[9] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S & P)*, 2008.

[10] Sruthi Bandhakavi, Samuel King, P Madhusudan, and Marianne Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *Proceedings of the USENIX Security Symposium (Usenix Security)*, 2010.

[11] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the International Conference on Computer and Communications Security (CCS)*, 2008.

[12] Adam Barth, Felt Adrienne Porter, Prateek Saxena, and Boodman Aaron. Protecting browsers from extension vulnerabilities. In *Proceeding of the International Conference on Network and Distributed System Security Symposium (NDSS)*, 2010.

[13] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the International Conference on World wide web (WWW)*, 2010.

[14] Prithvi Bisht and V.N. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the International Conference on Detection of Intrusions & Malware, and Vulnerability Assesment (DIMVA)*, 2008.

[15] TrendLabs Malware Blog. Malicious firefox extensions. http://blog.trendmicro.com/malicious-firefox-extensions/.

[16] Bugzilla. bug 475530 - X-FRAME-OPTIONS header against "UI redressing" aka Clickjacking. https://bugzilla.mozilla.org/show_bug.cgi? id=475530, 2009.

[17] Commtouch Cafe. Nasty facebook picture attack based on "self-xss" - how does this work? http://blog.commtouch.com/cafe/web-security/nasty-facebook-picture-attack-based-on-self-xss/.

[18] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the USENIX Security Symposium (Usenix Security)*, 2012.

[19] Microsoft: Malware Protection Center. Trojan:js/febipos.a. `http://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Trojan%3aJS/Febipos.A#tab=2`.

[20] Mozilla Developer Center. Observer notifications. https://developer.mozilla.org/en/Observer_Notifications.

[21] Eric Yawei Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. Self-exfiltration: The dangers of browser-enforced information flow control. In *Proceedings of the Workshop of Web 2.0 Security & Privacy (W2SP)*, 2012.

[22] Brian Chess, Yekaterina T. O'Neil, and Jacob West. Javascript hijacking. Technical report, 2007.

[23] Pern Hui Chia, Andreas P. Heiner, and N. Asokan. Use of ratings from personalized communities for trustworthy application installation. In *Proceedings of the 15th Nordic conference on Information Security Technology for Applications (NordSec)*, 2012.

[24] Google Chrome. Chrome web store. `https://chrome.google.com/webstore`.

[25] Google Chrome. Declare permissions. `http://developer.chrome.com/dev/extensions/declare_permissions.html`.

[26] Nick Coblentz. Csrf prevention in struts 2. http://digg.com/news/technology/ CSRF_Prevention_in_Struts_2.

[27] The Web Application Security Consortium. Cross-site scripting. http://projects.webappsec.org/Cross-Site-Scripting, 2013.

[28] World Wide Web Consortium. Document object model (dom) level 2 core specification, w3c recommendation. http://www.w3.org/TR/DOM-Level-2-Core/introduction.html, November 2000.

[29] World Wide Web Consortium. Web storage editor's draft. http://dev.w3.org/html5/webstorage/, April 2012.

[30] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *Proceedings of the International Conference on Computer and Communications Security (CCS)*, 2008.

[31] Douglas Crockford. Jslint. http://www.jslint.com/.

[32] Dasient. Continued growth in web-based malware attacks – over 1m web sites infected in q2 2010. `http://blog.dasient.com/2010/09/continued-growth-in-web-based-malware_9357.html`, 2010.

[33] Web Hacking Incident Database. The web application security consortium. http://projects.webappsec.org/w/page/13246995/Web-Hacking-Incident-Database.

[34] Isaac Dawson. Security headers on the top 1000000 websites. `http://www.veracode.com/blog/2012/11/security-headers-report/`, 2012.

[35] Ernest Delgado. Detect dom changes with mutation observers. http://updates.html5rocks.com/2012/02/Detect-DOM-changes-with-Mutation-Observers, 2012.

[36] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the International Conference on Annual Computer Security Applications Conference (ACSAC)*, 2009.

[37] Oracle docs. Netscape object signing: Establishing trust for downloaded software. http://docs.oracle.com/cd/E19957-01/816-6171-10/.

[38] Django Documentation. Cross site request forgery protection. http://docs.djangoproject.com/en/dev/ref/contrib/csrf/.

[39] Xinshu Dong, Kailas Patil, Xuhui Liu, Jian Mao, and Zhenkai Liang. An entensible security framework in web browsers. *Technical Report TR-SEC-2012-01, Systems Security Group, School of Computing, National University of Singapore*, 2012.

[40] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. Adsentry: comprehensive and flexible confinement of javascript-based advertisements. In *Proceedings of the International Conference on Annual Computer Security Applications Conference (ACSAC)*, 2011.

[41] Nishant Doshi. Please send me your facebook anti-csrf token! http://www.symantec.com/connect/blogs/please-send-me-your-facebook-anti-csrf-token.

[42] DOM4 W3C Working Draft. Mutation observers. http://www.w3.org/TR/domcore/#mutation-observers.

[43] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceeding of the USENIX Security Symposium (Usenix Security)*, 2007.

[44] EllisLab. Codeigniter: Php web application development framework. http://ellislab.com/codeigniter.

[45] Facebook. FBJS - Facebok Developers Wiki. http://wiki.developers.facebook.com/index.php/FBJS, 2008.

[46] Adrienne Porter Felt, Helen J. Wang, Alex Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the USENIX Security Symposium (Usenix Security)*, 2011.

[47] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999.

[48] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing capability leaks in secure javascript subsets. In *Proceedings of the International Conference on Network and Distributed System Security Symposium (NDSS)*, 2010.

[49] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, 2008.

[50] FUEL-CMS. An open source codeigniter based content management system. https://github.com/daylightstudio/FUEL-CMS.

[51] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the op browser. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S & P)*, 2008.

[52] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the USENIX Security Symposium (Usenix Security)*, 2009.

[53] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceeding of the International Conference on Network and Distributed System Security Symposium (NDSS)*, 2009.

[54] ha.cakers.org Blog. Fizzle firefox extension vulnerability. http://ha.ckers.org/blog/20070324/fizzle-firefox-extension-vulnerability/, 2007.

[55] Nathan Hamiel and Shawn Moyer. Dynamic CSRF. In *Black Hat USA*, 2009.

[56] Robert Hansen and Jeremiah Grossman. Clickjacking. http://www.sectheory.com/clickjacking.htm, 2008.

[57] Kelly Jackson Higgins. Facebook hit by clickjacking attack. http://www.darkreading.com/insiderthreat/security/attacks/showArticle.jhtml?articleID=222100098, 2009.

[58] Lin-Shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schechter, and Collin Jackson. Clickjacking: Attacks and defenses. In *Proceedings of the USENIX Security Symposium (Usenix Security)*, 2012.

[59] IEBlog. IE8 security part vii: Clickjacking defenses. http://blogs.msdn.com/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking- defenses.aspx, 2009.

[60] Lon Ingram and Michael Walfish. Treehouse: Javascript sandboxes to help web developers help them-selves. In *Proceedings of the USENIX annual technical conference*, 2012.

[61] ECMA International. Standard ECMA-262. http://www.ecma-international.org/publications/standards/Ecma-262.htm, 2009.

[62] Scott Isaacs and Dragos Manolescu. WebSandbox - Microsoft Live Labs. http://websandbox.livelabs.com/, 2009.

[63] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2006.

[64] Collin Jackson and Helen J. Wang. Subspace: secure cross-domain communication for web mashups. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2007.

[65] Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve J. Chapin. Escudo: A Fine-grained Protection Model for Web Browsers. In *Proceedings of the International Conference On Distributed Computing Systems (ICDCS)*, 2010.

[66] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2007.

[67] Martin Johns and Justus Winter. RequestRodeo: Client-side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference, Refereed Papers Track, Report CW448*, 2006.

[68] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *Proceedings of the Second IEEE Conference on Security and Privacy in Communications Networks (SecureComm)*, 2006.

[69] Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-chieh Shan. An analysis of the mozilla jetpack extension framework. In *Proceedings of the 26th European conference on Object-Oriented Programming (ECOOP)*, 2012.

[70] Chris Karlof, Umesh Shankar, J. D. Tygar, and David Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings the International Conference on Computer and Communications Security (CCS)*, 2007.

[71] Florian Kerschbaum. Simple cross-site attack prevention. In *Proceedings of the International Conference on Security and Privacy in Communications Networks and the Workshops (SecureComm)*, 2007.

[72] 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 International Conference on World Wide Web (WWW)*, 2008.

[73] Engin Kirda and Christopher Kruegel. Behavior-based spyware detection. In *Proceedings of USENIX Security Symposium (Usenix Security)*, 2006.

[74] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks. In *Proceedings of the Symposium on Applied Computing (SAC)*, 2006.

[75] D. Kristol. Http state management mechanism. http://www.ietf.org/rfc/rfc2109.txt, 1997.

[76] Zhuowei Li, XiaoFeng Wang, and Jong Youl Choi. Spyshield: preserving privacy from spy add-ons. In *Proceedings of the International Conference on Recent Advances in Intrusion Detection (RAID)*, 2007.

[77] Mashed Life. Mashedlife. http://mashedlife.com.

[78] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *Proceeding of the International Conference on Network and Distributed System Security Symposium (NDSS)*, 2012.

[79] Benjamin Livshits and Úlfar Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2007.

[80] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the USENIX Security Symposium (Usenix Security)*, 2010.

[81] Mike Ter Louw, Jin Soon Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. In *Proceedings of the Journal in Computer Virology*, August 2008.

[82] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S & P)*, 2009.

[83] Sergio Maffeis, John C Mitchell, and Ankur Taly. Run-time enforcement of secure javascript subsets. In *Proceedings of the Workshop of Web 2.0 Security & Privacy (W2SP)*, 2009.

[84] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S & P)*, 2010.

[85] Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2009.

[86] Leo A. Meyerovich and Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S & P)*, 2010.

[87] Microsoft. About the pop-up blocker. http://msdn.microsoft.com/en-us/library/ms537632(VS.85).aspx.

[88] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja - Safe Active Content in Sanitized JavaScript. http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf, 2007.

[89] Travis Mitchell. Aba survey: 62% of u.s. adults now prefer online banking. http://www.fiercefinance.com/story/aba-survey-62-us-adults-now-prefer-online-banking/2011-09-08, 2011.

[90] MITRE. Common vulnerabilities and exposures. the standard for information security vulnerability names. http://cve.mitre.org/find/index.html.

[91] Mozilla. event.addeventlistener. https://developer.mozilla.org/en/DOM/element.addEventListener.

[92] Mozilla. nsicontentpolicy-mdn. https://developer.mozilla.org/en/NsIContentPolicy.

[93] Mozilla. Performance:tinderbox tests. https://wiki.mozilla.org/Performance:Tinderbox_Tests.

[94] Mozilla. Pop-up blocker. http://support.mozilla.com/en-US/kb/Pop-up+blocker.

[95] Mozilla. Pop-up window controls. https://developer.mozilla.org/en/Popup_Window_Controls.

[96] Mozilla. Signed scripts in mozilla. http://www.mozilla.org/projects/security/components/signed-scripts.html.

[97] Mozilla. Spidermonkey internals. https://developer.mozilla.org/En/SpiderMonkey/Internals.

[98] Mozilla. Extensions. https://developer.mozilla.org/En/Extensions, 2009.

[99] MDC Mozilla. Same origin policy for javascript. https://developer.mozilla.org/En/Same_origin_policy_for _JavaScript, 2009.

[100] MSDN. What's new in Internet Explorer 8. http://msdn.microsoft.com/en-us/library/ cc288472.aspx, 2009.

[101] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceeding of the International Conference on Network and Distributed System Security Symposium (NDSS)*, 2009.

[102] Mozilla Developer Network. Jetpack. https://developer.mozilla.org/en/Jetpack.

[103] Oracle Sun Developer Network. Chapter 10: Signed applets. http://java.sun.com/developer/onlineTraining/Programming/JDCBook/signed.html.

[104] BitDefender News. Trojan poses as fake google chrome extension. www.bitdefender.com/site/News/pdfDescription/1487.pdf.

[105] Nex. The clickjacking meets xss: a state of art. http://www.milw0rm.com/papers/265, 2008.

[106] NoScript. http://noscript.net, 2009.

[107] National Vulnerability Database (NVD). Cve-2009-3759. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3759, 2009.

[108] Ruby on Rails. http://rubyonrails.org/.

[109] Bryan Parno, Jonathan M. McCune, Dan Wendlandt, David G. An-dersen, and Adrian Perrig. Clamp: Practical prevention of large-scale data leaks. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S & P)*, 2009.

[110] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards fine-grained access control in javascript contexts. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[111] Kailas Patil, Xinshu Dong, and Zhenkai Liang. Clickguard: Preventing click event hijacking by user intention inference. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS) [Industrial Track]*, 2010.

[112] Kailas Patil, Tanvi Vyas, Fredrik Braun, and Mark Goodwin. Usercsp- user specified content security policies. SOUPS'13 POSTER, 2013.

[113] S. D. Paula and G. Fedon. Subverting ajax. In *Proceedings of the International Conference on Chaos Communication Congress (CCC)*, 2006.

[114] Chia P.H., Yamamoto Y., and Asokan N. Is this app safe?: a large scale study on application permissions and risk signals. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2012.

[115] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight Self-Protecting JavaScript. In *Proceedings of the Symposium on Information, Computer, and Communications Security (ASIACCS)*, 2009.

[116] Ameet Ranadive, Tufan Demir, Shariq Rizvi, and Neil Daswani. Malware distribution via widgetization of the web. In *Proceedings of the BlackHat Technical Security Conference, DC*, 2011.

[117] W3C Candidate Recommendation. Content security policy 1.0. `http://www.w3.org/TR/CSP/`, 2012.

[118] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[119] RSnake. Xss(cross site scripting) cheat sheet esp: for filter evasion. http://ha.ckers.org/xss.html.

[120] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In *Lecture Notes in Computer Science*, 2010.

[121] Justin Samuel. Requestpolicy. http://www.requestpolicy.com, 2010.

[122] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceeding of the International Conference on Computer and Communications Security (CCS)*, 2011.

[123] Samy. Technical explanation of the myspace worm. http://namb.la/popular/tech.html.

[124] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Proceedings of the International Conference on Network and Distributed System Security Symposium (NDSS)*, 2010.

[125] Prateek Saxena, David Molnar, and Benjamin Livshits. Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization. Technical Report, Microsoft Research, 2010.

[126] Scottdb56. mobi*list - a list of mobile device-friendly websites. `http://mobi.sdboyd56.com/`, 2013.

[127] Security and the Net. About IE8's Clickjacking protection. http://securityandthe.net/2009/02/01/about-ie8s-clickjacking-protection/, 2009.

[128] Websense security labs blog. A weekend of click-jacking on facebook. `http://community.websense.com/blogs/securitylabs/archive/2011/05/02/a-weekend-of-click-jacking-on-facebook.aspx`.

[129] O Segal, O Weisman, Adi Sharabani, Y Amit, and L Guy. Close encounters of the third kind. ftp://public.dhe.ibm.com/common/ssi/ecm/en/raw14252usen/RAW14252USEN.PDF, 2010.

[130] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Proceeding of the International Conference on Network and Distributed System Security Symposium (NDSS)*, 2009.

[131] IDG News Service. Browser malware is injecting ads into our pages, warns wikipedia. http://www.computerworlduk.com/news/security/3358034/browser-malware-is-injecting-ads-into-our-pages-warns-wikipedia/.

[132] Nakedsecurity Sophos. Try not to laugh xd: Worm spreads via facebook status messages. http://nakedsecurity.sophos.com/2010/05/21/laugh-xd-worm-spreads-facebook-status-messages/.

[133] Nakedsecurity Sophos. Viral clickjacking 'like' worm hits facebook users. http://nakedsecurity.sophos.com/2010/05/31/viral-clickjacking-like-worm-hits-facebook-users/.

[134] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2010.

[135] Xi Tan, Wenliang Du, Tongbo Luo, and Karthick D. Soundararaj. Scuta: A server-side access control system for web applications. Technical report, SYR-EECS-2011-09, July 14, 2011.

[136] Shuo Tang, Haohui Mai, and Simon King. Trust and protection in the illinios browser operating system. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[137] Mike Ter Louw, Prithvi Bisht, and V.N. Venkatakrishnan. Analysis of hypertext isolation techniques for XSS prevention. In *Proceedings of the workshop on Web 2.0 Security and Privacy (W2SP)*, 2008.

[138] Alexa the Web information company. Top sites by category. http://www.alexa.com/topsites/category.

[139] Amir Tinkering. Chrome extension spyware?: Smooth gestures. http://amirtinkering.com/52/chrome-extension-spyware-smooth-gestures/.

[140] Ukulima. An open source codeigniter based social networking platform and knowledge base for farmers. https://github.com/PamojaMedia/ukulima.

[141] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the International Conference Network and Distributed System Security Symposium (NDSS)*, 2007.

[142] W3C. 3.2 transparency: the opacity property - css color module level 3. http://www.w3.org/TR/css3-color/#transparency.

[143] W3C. Document object model (dom) level 3 events specification. http://www.w3.org/TR/DOM-Level-3-Events/.

[144] W3C. 'z-index' - cascading style sheets level 2 revision 1 (css 2.1) specification. http://www.w3.org/TR/CSS21/visuren.html#propdef-z-index.

[145] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 1993.

[146] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in mashupos. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2007.

[147] Helen J Wang, Chris Grier, Alex Moshchuk, and Sam King. The multi-principle os construction of the gazelle web browser. In *Proceedings of the USENIX Security Symposium (Usenix Security)*, 2009.

[148] 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 *Proceedings of the Information Security Conference (ISC)*, 2012.

[149] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A systematic analysis of xss sanitization in web application frameworks. In *Proceedings of the International Conference on European Symposium on Research in Computer Security (ESORICS)*, 2011.

[150] Wikipedia. Cascading style sheets. `http://en.wikipedia.org/wiki/Cascading_Style_Sheets`.

[151] Wikipedia. Cross-site request forgery (csrf) prevention cheat sheet. http://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

[152] Wikipedia. Document object model. http://en.wikipedia.org/wiki/Document_Object_Model.

[153] Wikipedia. Framekiller. http://en.wikipedia.org/wiki/Framekiller.

[154] Wikipedia. Html. `http://en.wikipedia.org/wiki/HTML`.

[155] Wikipedia. Javascript. `http://en.wikipedia.org/wiki/JavaScript`.

[156] J. Max Wilson. IE JavaScript bugs: Overriding internet explorer's document.getElementById() to be W3C compliant exposes an additional bug in getAttributes(), 2007. http://www.sixteensmallstones.org/ie-javascript-bugs-overriding-internet-explorers-documentgetelementbyid-to-be-w3c-compliant-exposes-an-additional-bug-in-getattributes.

[157] Yongzheng Wu, Sai Sathyanarayan, Roland H. C. Yap, and Zhenkai Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *Proceedings of the International Conference on European Symposium on Research in Computer Security (ESORICS)*, 2012.

[158] Xssed.com. Myspace.com hit by a permanent xss. `http://www.xssed.com/news/83/Myspace.com_hit_by_a_Permanent_XSS/`.

[159] Xssed.com. New orkut xss worm by brazilian web security group. `http://www.xssed.com/news/77/New_Orkut_XSS_worm_by_Brazilian_web_security_group/`.

[160] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S & P)*, 2009.

[161] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript Instrumentation for Browser Security. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[162] Chuan Yue and Haining Wang. Anti-phishing in offense and defense. In *Proceedings of the International Conference on Annual Computer Security Applications Conference (ACSAC)*, 2008.

[163] Michal Zalewski. Browser security handbook. `http://code.google.com/p/browsersec/wiki/Main`.

[164] ZDNet. Malicious chrome extensions hijack facebook accounts. http://www.zdnet.com/blog/security/malicious-chrome-extensions-hijack-facebook-accounts/11074.

[165] Yue Zhang, Jason I. Hong, and Lorrie F. Cranor. Cantina: a content-based approach to detecting phishing web sites. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2007.

[166] lfar Erlingsson, Martn Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.