Mitigating Web-borne Security Threats by Enhancing Browser Security Policies

Thesis submitted in partial fulfillment
of the requirements for the degree of

DEGREE
in
Master of Science (by Research)
in
Computer Science and Engineering

by

Krishna Chaitanya Telikicherla
201107633
KrishnaChaitanya.T@research.iiit.ac.in

Software Engineering Research Center
International Institute of Information Technology
Hyderabad - 500 032, INDIA
May 2016

Copyright © Krishna Chaitanya Telikicherla, 2016
All Rights Reserved

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Mitigating Web-borne Security Threats by Enhancing Browser Security Policies" by Krishna Chaitanya Telikicherla, has been carried out under my supervision and is not submitted elsewhere for a degree.

______________	________________________________
Date	Adviser: Dr. Venkatesh Choppella

To my mother Srimati Jaya Lakshmi, and father Sri Telikicherla Rama Mohana Rao, and my wife Sindhuri.

Acknowledgments

I would like to express my deep sense of gratitude to Dr. Venkatesh Choppella, my adviser, for his continuous support, guidance and motivation during the course of my research. It is said: "*When the student is ready, the teacher appears*". Four years back when I was in a desperate search for an adviser who could support my interest in web security and sculpt my career, I happened to meet Venkatesh at a local technical meetup. That kickstarted my research journey and since then, every meeting we had was full of energetic discussions, interesting observations and promising ideas that questioned the fundamentals. Till date, the sense of satisfaction we have when we stretch for hours, re-discover the whys and hows of certain web security mechanisms and think of better approaches, is immense and beyond words. I am indebted to Venkatesh for selecting me as one of his research students and will always be thankful for the work ethics and discipline he has inculcated in me.

I would like to express my gratitude to the faculty of IIIT-Hyderabad who helped me build the foundations required for pursuing research. The rigorous course work, the innumerable assignments, hands-on labs and exams helped me in strengthening my understanding of several concepts. The *Principles of Programming Languages* class taught by Dr. Venkatesh immensely helped me to correlate open web security problems with similar problems in programming languages that have already been solved. The *Compilers* class taught by Dr. Suresh Purini helped me understand some of the intricacies of web browsers. The *Topics in Programming Languages* class taught by Dr. Venkatesh introduced me to logic and formal modeling tools, which helped me analyze and build formal models of web browsers using Alloy. The *Semantics of Programming Languages* class taught by Dr. Viswanath Kasturi, Dr. Kesav Nori and Dr. Venkatesh gave me diverse insights about interpreters. The *Middleware* class taught by Mr. Ramesh Loganathan helped me gain a good understanding of communication protocols. The *Research in Information Security* class taught by Dr. Bruhadeshwar Bezawada greatly helped me in gaining insights about the field of Information Security. Also, the numerous discussions I had with Dr. Bruhadeshwar during our web security meetings helped me in refining my research. I will be grateful to all of them for spending their valuable time to mould my career.

I would like to express my heartfelt thanks to the faculty and members associated with Software Engineering Research Center (SERC): Dr. Raghu Reddy, Dr. Devi Prasad, Saurabh Barjatiya, Kirti Garg, Manjula P, Sridhar Chimalakonda, Naveen Kulkarni, Sai Gollapudi and Amulya Sri, for all the wonderful debates, discussions and advices. I am fortunate to be associated with such a vibrant research group

and will always cherish the precious moments we have shared, especially during R & D showcases, conferences and SERC meetings.

On a special note, I would like to thank the management of Infosys Labs, my ex-employer[1], for allowing me to pursue research at IIIT-Hyderabad on a part-time basis. Particularly, the support I have received from Dr. Srinivas Padmanabhuni, AVP and Head of Software Engineering Lab at Infosys, is unparalleled. Without his encouragement and freedom at work, this research would not have been possible. Also, I am fortunate to be associated with several researchers at Infosys Labs who always gave the apt guidance whenever required.

Lastly, I would like to thank the members of NULL and OWASP security chapters and Microsoft User Group at Hyderabad. I had the opportunity to present some of my works at these vibrant security and developer communities. This helped me in evaluating my understanding of several concepts in web security, while ensuring that my work holds relevant to the latest web application architectures.

[1] I was working at Security and Privacy Research Lab of Infosys Labs while completing the credits for M.S by Research degree. At the time of writing this thesis, I am working at Microsoft India as a Security Consultant

Abstract

The World Wide Web has evolved from a set of simple static pages connected by hyperlinks to a complex platform, to meet the demands of users and businesses. The modern web is characterised by complex operations such as online social networking, content sharing, electronic payments, single sign-on etc. The evolution of web APIs (Application Programming Interfaces) and open data initiatives encouraged developers to build Mashups, web applications that integrate data from multiple servers. Data has become the currency on the web, due to which the web has turned into a lucrative target for attackers a.k.a cyber criminals. Newer web standards such as HTML5 are evolving and newer versions of browsers are emerging to meet the needs of the modern web. However, the security policies governing the web have not evolved at the same pace. Due to this, the number of vulnerabilities and newer web based attacks are increasing rapidly.

Browsers, being the entry points to the web, are heavily targeted by attackers. One good reason is that the time and effort required to compromise a website due to a vulnerability in the application layer is much lesser compared to that of other layers. E.g., to steal certain sensitive data from a web server, it is much easier to inject a small snippet of JavaScript into a vulnerable web page loaded in a browser and exfiltrate the data, than to intercept a connection and break a fairly strong crypto system, or to bypass firewalls and break into the network.

This work attempts to understand the core security policies of web browsers that govern the security and privacy of web interactions. It closely examines a series of client side web attacks, their existing defenses and deficiencies. It observes the need for novel application-level security frameworks as well as browser security policies in mitigating them. The outcome of the work is two fold: Firstly, it presents a security abstraction layer (as an API library) called “SafeMash”, which helps developers build safe mashups over the current low-level security APIs in HTML5. Secondly and more importantly, it proposes a novel declarative browser security policy called CORP (Cross Origin Request Policy) to mitigate a set of attacks which we refer to as “Web Infiltration attacks”. CORP enables a server to control which site can access which resource on a cross-origin server, and through which browser event.

To evaluate the effectiveness of SafeMash and CORP, several experiments were conducted. The usage of SafeMash was empirically demonstrated by first building an interactive mashup using open APIs from *ProgrammableWeb* (without using state-of-the-art security mechanisms) and then rebuilding it using SafeMash without losing functionality. To clearly understand the security model of web browsers and its limitations, a corpus of web attacks was developed. The formulation, effectiveness and ease of

deployment of CORP was demonstrated based on the insights derived from examining the corpus of web attacks. The design of CORP was formally verified by building a light weight model in the Alloy model finder. An implementation of CORP was provided as a browser extension for Chrome and it is evaluated against real-world cross origin attacks on open source web applications. Our initial investigation revealed that most of the popular websites already segregate their resources in a way which makes deployment of CORP easier.

Contents

Chapter Page

1 Introduction . . . 1
1.1 Anatomy of HTTP transactions . . . 2
1.1.1 HTTP end points . . . 2
1.1.2 URL . . . 2
1.1.3 HTTP Transaction . . . 3
1.1.4 HTML elements, DOM and Origin . . . 3
1.1.5 Cascading HTTP requests . . . 4
1.2 Threat Model . . . 5
1.3 Organization of The Thesis . . . 7

2 Web-borne Security Threats . . . 8
2.1 Browser Security Model . . . 9
2.1.1 Origin . . . 9
2.1.2 Same Origin Policy (SOP) . . . 10
2.2 Limitations of the browser security model . . . 11
2.2.1 Cross Origin Content Inclusion . . . 11
2.2.2 Cross-Site Scripting (XSS) . . . 12
2.2.3 Data-Exfiltration . . . 12
2.2.4 Cross-Site Request Forgery (CSRF) . . . 12
2.2.5 Clickjacking . . . 13
2.2.6 Cross-Site Timing Attacks . . . 13

3 Related Work . . . 15
3.1 Security of Web Mashups . . . 15
3.1.1 Fragment Identifier Messaging . . . 15
3.1.2 Subspace . . . 16
3.1.3 Safe JavaScript Subsets . . . 16
3.1.4 HTML5 Enabled Privilege Separation . . . 16
3.2 Mitigating Web Infiltration Attacks . . . 17
3.2.1 Approaches to Mitigate CSRF . . . 17
3.2.2 Approaches to Mitigate Clickjacking . . . 19
3.2.3 Approaches to Mitigate Cross-Site Timing Attacks . . . 19

4 Building Secure Web Mashups 20
4.1 Evolution of Mashups 20
4.1.1 Security concerns in mashups 21
4.1.2 Security versus Interactivity 22
4.2 Newer Browser Security Model 23
4.2.1 PostMessage API 23
4.2.2 Iframe Sandbox 24
4.2.2.1 Relaxing Sandbox Restrictions 24
4.2.3 Content Security Policy 24
4.3 Insecure Usage of HTML5 APIs 25
4.3.1 Security Considerations in Sandbox 26
4.3.1.1 Sandbox Flags and Privilege Escalation 26
4.3.1.2 Disabling Frame Busting Defense 27
4.3.2 Security Considerations in PostMessage 28
4.3.2.1 Attack on Confidentiality 28
4.3.2.2 Attack on Integrity 29
4.4 Implementation of SafeMash 29
4.4.1 SafeMash API 30
4.4.2 Security Checks Built into SafeMash 31
4.4.2.1 Sandbox Related Checks 31
4.4.2.2 PostMessage Related Checks 31
4.4.3 Evaluation 32

5 Enhancing Browser Security Policies 33
5.1 Web Infiltration attacks 33
5.1.1 Observations and Inferences 34
5.2 Cross Origin Request Policy 36
5.2.1 Core Idea Behind CORP 36
5.2.2 Browser Model with CORP 37
5.2.2.1 Setting the Policy 37
5.2.2.2 Deleting the Policy 38
5.2.2.3 CORP and CSP - How They Differ 38
5.2.3 Abstract Syntax of CORP 38
5.2.3.1 Order of Precedence for CORP rules 39
5.2.3.2 Example Policies 39
5.2.4 Security Guarantees Provided by CORP 40
5.2.4.1 Fine Grained Access Control 40
5.2.4.2 Combating CSRF 40
5.2.4.3 Early Enforcement of Clickjacking Defense 41
5.2.4.4 Controlling Social Engineering Attacks 42
5.2.4.5 Defeating Cross-Site Timing Attacks 42
5.2.4.6 Mitigating Application-level DDoS Attacks 42
5.3 Validating the Soundness of CORP 44
5.3.1 A Brief Introduction to Alloy 44
5.3.1.1 Alloy specifications 45
5.3.1.2 Sample model 46

5.3.2 Design considerations of CORP Alloy model 47
5.3.2.1 Simpler Abstraction 47
5.3.2.2 Non-empty browser context 47
5.3.2.3 Single browser instance 47
5.3.3 Modelling cross-origin requests in the web platform (Pre-CORP.als) 48
5.3.3.1 HTTP Transactions 48
5.3.3.2 Origin 50
5.3.3.3 HTTP Event Initiators 51
5.3.3.4 Fact: EventInitiatorsInheritParentOrigin 52
5.3.3.5 Fact: TransactionRules 53
5.3.3.6 Fact: Disjointness 54
5.3.3.7 Pred: SameOriginTransaction 54
5.3.3.8 Pred: CrossOriginTransaction 55
5.3.4 Modelling restrictions introduced in CORP (Post-CORP.als) 56
5.3.4.1 Key idea of CORP 56
5.3.4.2 Resource Paths 58
5.3.4.3 Pred: maliciousXOriginTransaction 58
5.3.4.4 Pred: corpCompliantTransaction 59
5.3.4.5 Assert: showMaliciousTransactionWithJsCode 61
5.4 Experimentation and Analysis 62
5.4.1 Implementation 62
5.4.2 Experiments 63
5.4.2.1 Evaluating CORP Against a Corpus of Attacks 63
5.4.2.2 Configuring CORP on Open Source Web Applications 63
5.4.2.3 Analyzing Adherence of Top Websites to CORP 64

6 Conclusions and Future Work 65
6.1 Research Contributions 65
6.2 Future Work 66

Appendix A: Sample Alloy Model 69
A.1 A basic academic time table Alloy Model 69

Appendix B: CORP Alloy Models 74
B.1 Pre-CORP Alloy Model 74
B.2 Post-CORP Alloy Model 78

Bibliography 84

List of Figures

Figure Page

1.1 Cascading HTTP requests (Same Origin) 5
1.2 Cascading HTTP requests (Cross Origin) 6

2.1 *Exfiltration of data from a website due to script injection* 9

4.1 *A typical web mashup built by embedding third party JavaScript. The bidirectional arrows in the webpage indicate lack of privilege separation between content in the widgets and their parent.* 22
4.2 An interactive web mashup with privilege separation. Sandboxed iframes restrict JavaScript activity across widgets. PostMessage channel (depicted by a pipe) enables inter-widget communication. CSP enforces restrictions on HTTP traffic (e.g., blocking request to Ox.com in the figure). 26
4.3 Attacks on postMessage API. (1) *postMessage* communication with targetOrigin set to '*'. (2) Attacker redirecting a frame via descendant policy. (3)Attack on confidentiality. (4) Attack on integrity. 28

5.1 Exfiltration vs. Infiltration attacks 34
5.2 *Browser model showing exfiltration & infiltration and how they are mitigated by CSP & CORP* 37
5.3 *Browser model showing the enforcement of Clickjacking defense in CSP/XFO and CORP* 41
5.4 *Understanding an application-level DDoS attack* 43
5.5 Meta model of the Pre-CORP Alloy model 49
5.6 An instance of an HTTP transaction in the Pre-CORP model 50
5.7 An instance of the predicate *showBasicModel* 51
5.8 An instance of the predicate *sameOriginTransaction* 55
5.9 An instance of the predicate *crossOriginTransaction* 56
5.10 Meta model of the Post-CORP Alloy model 57
5.11 An instance of the predicate *maliciousXOriginTransaction* 59
5.12 An instance of the predicate *restrictJsCodeWithCorp* 61
5.13 Checking the post-CORP assertion *showMaliciousTransactionWithJsCode* 62
5.14 Bar chart showing adherence of Alexa Top 15,000 websites to CORP. 64

A.1 Model instances generated by Alloy analyzer for the predicate *showTimeTable* 72
A.2 Model instance generated by Alloy analyzer for the predicate *studentCanAttendSameClassInDifferentSlots* 72

A.3 No counterexample is found for the assertion *studentCannotAttendDifferentClassesInSameSlot* 73

List of Tables

Table Page

2.1 Understanding what *Same Origin* means. 10
2.2 Browser model with Same Origin Policy 11

5.1 Summary of open source web applications we experimented with 63

Chapter 1

Introduction

Web browsers are the entry points of the web platform and are heavily targeted by attackers. When a user types a URL in the address bar of a web browser, the browser initiates one or more HTTP requests, as triggered by the HTML content in the page. Some of these requests can load resources such as images, scripts, style sheets etc., from any remote server. The browser builds a DOM (Document Object Model) tree, a data structure containing static HTML elements and fetched resources, and finally renders the page. The basic browser model allows resources to loaded from any server and this is one of the design flaws which has made the web an uncontrolled platform, and as explained in the later sections, one of the main reasons for several web attacks.

The last decade has seen the evolution of web APIs (Application Programming Interfaces) and open data initiatives. This has encouraged developers to build Mashups, web applications that integrate data from multiple servers. As a result, the web has seen a drastic change in the way online interactions happen. E-Commerce, online social networks, single sign-on mechanisms etc. are some of the outcomes of enhanced web technologies. To meet the needs of the modern web, newer web standards such as HTML5 are evolving and newer versions of browsers are emerging. At the same time the number of vulnerabilities and newer web based attacks are increasing rapidly. Same Origin Policy (SOP), the core security policy driving today's web platform, was designed at a time when the web had static web pages connected by hyperlinks. As shown in several studies [37, 53, 64], Same Origin Policy is not sufficient to meet the security considerations of the modern web.

Researchers have proposed several interesting browser security policies [40, 37, 53, 64, 35, 52, 30, 58] to fix the loopholes of SOP and mitigate dangerous web based attacks. These policies have made very good contribution to browser security by restricting cross origin resource inclusion, script execution and data exfiltration. However, as explained in the later sections, there are a few known attacks which either escape or do not fall under the scope of these solutions.

1.1 Anatomy of HTTP transactions

Before understanding the current browser security model and its weaknesses, it is important to understand the anatomy of HTTP transactions.

1.1.1 HTTP end points

HTTP [19] is an application layer protocol used by the World Wide Web. HTTP connections are typically established between the HTTP end points - *browsers* and *web servers*. A browser is an HTTP client software installed on a user's computer. It conforms to HTTP specifications [71]. It is the means through which users interact with the World Wide Web. A web server is a computer which hosts software that can respond to HTTP requests. It typically has higher hardware configurations such as memory, RAM etc., so that it can process several concurrent requests (sometimes millions) initiated by clients.

1.1.2 URL

Resources on the web (e.g., html pages, images etc.) are accessed through a browser by means of a URL (Uniform Resource Locator) [20]. A generic URL is of the following form:
`scheme://host[:port][/path][?query][#fragment]`.
The segments in square braces denote optional segments. *Port* is an optional segment of a URL which corresponds to the port number on a web server to which a connection has to be established by an HTTP client. Typically, shemes have a default port number (e.g., the default port for http is 80). When a port is omitted from a URL, it is understood by HTTP clients that the connection has to be made to the default port on a server. *Path* is another optional segment of a URL which specifies how a specific resource on a server can be accessed. Web servers have default resources e.g., web pages such as html, php, asp etc., which will be served when path is not mentioned in a request's URL. *Query String* and *Fragment Identifier* are optional segments of a URL which help in conveying additional information about a request to a web server.

```
http://example.com
http://example.com/scripts/events.js
http://example.com/categories/product?id=1&price=100#description
```

Listing 1.1 Examples of URLs

Listing 1.1 shows a few examples of URLs. The first URL in the listing is of the simplest form having only a *scheme* (http) and a *host* (example.com). The second URL additionally has the *path* `/scripts/events.js`, which specifies how the JavaScript file *events.js* can be accessed. The third URL has the path `/categories/product?id=1&price=100#description`. The section of the path following the *?* (`id=1&price=100`) is the *query string*, while the section of the path following the *#* (`description`) is the *fragment identifier*. The query string and the fragment identifier provide

additional information to the web server about the request. Based on the values in these segments, the web server can vary its response. The triple - scheme, host and port are together called the *origin*. It is the basic unit of isolation on the web. Chapter 2 explains in detail the importance of origin in understanding the security model of a browser.

1.1.3 HTTP Transaction

When a user types a URL in the address bar of a browser, an *HTTP request* will be initiated by the browser. Firstly, the hostname in the URL will be resolved by a DNS server on the web into an IP address. Once the hostname resolution is done, the request will be sent to the web server that has the resolved IP address. The server generates an *HTTP response* that has a document with HTML content (i.e., an HTML document, also known as a web page). The browser receives the HTML document and renders it. This request-response round trip is referred to as an *HTTP Transaction*.

1.1.4 HTML elements, DOM and Origin

An HTML document contains a set of HTML elements - some of which are passive elements such as div, span, textbox etc., while the rest are active elements such as script, image, iframe etc. The passive HTML elements contribute to the presentation and formatting of the document. They are referred to as passive elements as they do not trigger HTTP transactions. The active HTML elements trigger HTTP transactions and enhance the document with additional content. When a browser renders HTTP responses, it constructs a tree data structure called DOM (Document Object Model). In the HTML DOM tree, the HTML document is the root node and is referred to as the *document* object. Rest of the HTML elements are the descendants of the HTML document. The document object has properties and methods to access/modify other nodes using JavaScript. One of the important read-only properties of the document object is the *origin*, which can be accessed as *document.origin*. When a document is rendered, the browser sets the origin property with the scheme-host-port triple, extracted from the URL from which the document has loaded. All the elements of a document inherit the origin from their parent. When a new window/tab is opened in a browser, the browser creates a *window* JavaScript object corresponding to each open browser window/tab. When a web page is loaded in the browser tab/window, the document object of the page in the tab/window can be accessed as *window.document*.
The iframe element of an HTML document has special properties. It can trigger an HTTP request to a web page and embed the HTML document in the HTTP response into the iframe's parent document. If an HTML document contains iframe elements, the browser creates a window object for the HTML document and one additional window object for each iframe's document. This means, each of the document objects (one parent document and one or more child document objects) have their respective origin properties set by the browser. Certain access control mechanisms e.g., permission to access/modify the nodes of one document by a script in another document are granted only if the origins of the two documents match. This is discussed in detail in Chapter 2.

1.1.5 Cascading HTTP requests

Let us say a user types a URL `http://example.com` in the address bar of a browser. Since there is no path in this URL, the web server responds with a default HTML document, say index.html, which is rendered by the browser. The origin of the document, extracted from the URL from which it is loaded, will be `http://example.com`. Listing 1.2 shows the source code of the HTML document loaded in the browser. It shows *script* and *link* elements, which would in-turn trigger two additional cascading HTTP requests to load a JavaScript file (logic.js) and a stylesheet (style.css) respectively. This phenonmenon of embedding content from a server into a document is known as *content inclusion*.

```
<html>
    <head>
        <script src="/logic.js"/>
        <link href="/style.css"/>
    </head>
    <body>
        ...
    </body>
</html>
```

Listing 1.2 Same Origin HTTP requests triggered by a document loaded from `example.com`

Note that the URLs referenced by script and link elements are relative URLs i.e., their paths are relative to the URL from which the document has loaded. The *origin* of the resources (extracted from the resources' URL) is same as the *origin* of the document, which is `http://example.com`. Therefore, the HTTP transactions triggered by these elements are known as *Same Origin* transactions. Figure 1.1 shows the corresponding sequence diagram.

```
<html>
    <head>
        <script src="http://cdn.com/script.js"/>
        <link href="/style.css"/>
    </head>
    <body>
        ...
    </body>
</html>
```

Listing 1.3 Cross origin HTTP requests triggered by a document loaded from `example.com`

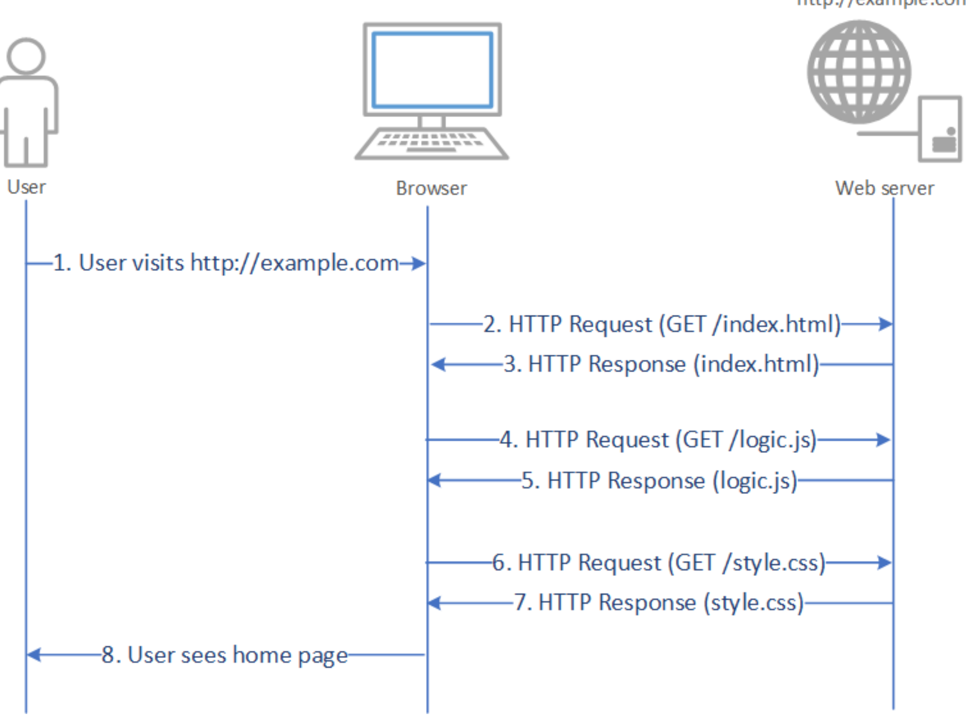

Figure 1.1 Cascading HTTP requests (Same Origin)

Listing 1.3 shows the source code of an HTML document where one of the resources (script) is loading from a third party server. The origin of the document is `http://example.com` whereas the origin extracted from the script's URL is `http://cdn.com`. Therefore, the HTTP transaction triggered by the script element is known as a *Cross Origin* (informally referred to as *cross-site*) transaction. Figure 1.2 shows the corresponding sequence diagram. It is important to note that though the script element initiates a cross origin transaction (to `http://cdn.com`), once the response (script.js) is received by the browser, the origin of the received JavaScript continues to be `http://example.com`, since it inherits the origin of the script element's document.

1.2 Threat Model

The threat model we assume is similar to the model described by Akhawe *et al.* [5]. The principals involved in our attack scenarios are Web attackers and Genuine users. For the attacks we discuss in this thesis, we do not consider the involvement of Network attackers.

Web Attacker: A web attacker is a malicious principal who owns web servers and serves malicious web pages. A web attacker can control the following:

- **Web servers:** The web attacker controls one or more web servers, which host malicious resources. Malware, browser extensions, web pages with malicious JavaScript etc., are examples of

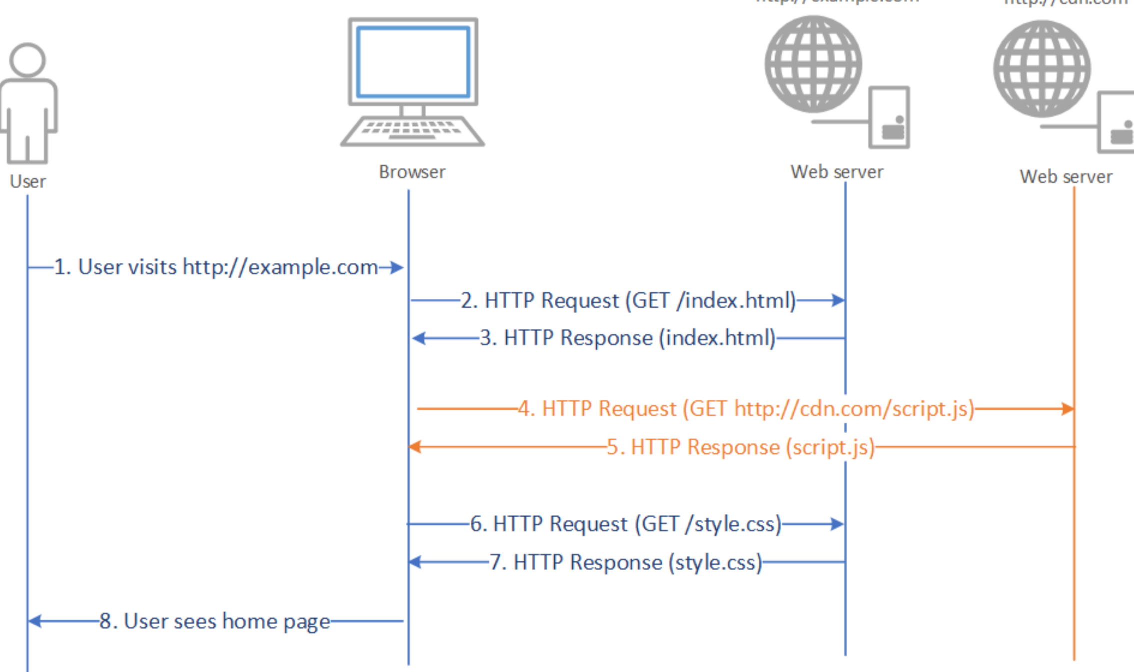

Figure 1.2 Cascading HTTP requests (Cross Origin)

malicious resources which can be served by the web server. The web attacker also owns one or more domain names which can point to any web server. In this thesis, we use the domain names "attacker.com" and "evil.com" interchangeably to designate the domain names controlled by web attacker. In addition, the web attacker can obtain HTTPS certificates for the domains controlled by him and can serve malicious pages through `https://attacker.com`.

- **Network:** The web attacker does not have special privileges to control the network. i.e., he cannot eavesdrop or forge network messages. However, by virtue of controlling web pages, he can utilize HTML elements to trigger network calls to honest or evil servers, thereby exfiltrating data. Since he can control web servers, he can control HTTP responses of the requests received by his servers. More importantly, he might opt out from all server side security mechanisms based on the context of the request. e.g., Content Security Policy requires the server to set "x-content-security-policy" response header, to prevent attacks like script injections, content inclusions etc. By denying to set this header, the web attacker is at free will to make cross origin calls via content inclusions.

- **Browsers:** When a genuine user visits a web attacker's website, the attacker can gain certain level of access on the user's browser through browser APIs. e.g., the attacker can have code like "window.open" or "window.postMessage" which can open a new popup window or can communicate with another popup using HTML5 postMessage API. However, we assume that the web attacker does not have access to file system of the user's operating system due to the sandbox restrictions

set by browsers. Also, we assume that the user uses a standard web browser which respects Same Origin Policy 2.1.2, the core browser security policy which restricts access to content based on origin of the webpage.

1.3 Organization of The Thesis

In this thesis, we analyze and demonstrate why the state-of-the-art browser security policies are not sufficient to prevent certain web-borne threats. We attempt to propose and formalize stricter policies, which mitigate such threats, and contribute towards building a safer web. The outcome of the research is in two folds:

1. **SafeMash:** In this, we present a survey of security concerns in the insecure usage of HTML5 APIs, particularly relevant to the security of mashups. We then present a high-level library called "SafeMash", which helps developers build safe mashups over the current low-level security APIs in HTML5. SafeMash allows the mashup developer to configure the degree of interaction and communication of a widget. It warns developers in case of any misconfiguration. Our initial empirical analysis shows that an interactive mashup that does not leverage state-of-the-art browser security features can be rebuilt with SafeMash, without any loss in functionality.

2. **CORP:** In this, we propose a new declarative browser security policy — "Cross Origin Request Policy" (CORP) — to mitigate a set of attacks based on cross origin interactions. CORP enables a server to have fine-grained control on the way different sites can access resources on the server. The server declares the policy using HTTP response headers. The web browser monitors cross origin HTTP requests targeting the server and blocks those which do not comply with CORP. Our initial investigation reveals that most of the popular websites already segregate their resources in a way which makes the deployment of CORP easier.

Apart from the above, other important contributions of this thesis are our simulations which explain the foundations of web security. In the process of experimenting with various web attacks, we have built several empirical simulations [63] which greatly enhanced our learning. Also, we have simplified the formal model of the web platform proposed by Akhawe *et al.*. [5] using Alloy. These simulations serve as good references for future researchers working on web security.

The rest of the thesis is organized as follows: Chapter 2 gives an overview of web-borne security threats, which are important to understand the effectiveness of browser security policies. Chapter 3 gives an overview of the related work done in preventing the aforementioned threats. Chapter 4 explains about the security of web mashups and presents our work, SafeMash. Chapter 5 explains about the deficiencies of existing browser security policies and presents our proposal, CORP, and Chapter 6 concludes.

Chapter 2

Web-borne Security Threats

When the World Wide Web was invented in 1989 [69], it only had a set of static pages interconnected via hyperlinks. With the addition of images in 1993[56], a request to a website could cascade a set of requests to multiple other sites. There is something unnerving about such *cross-origin* (or *cross-site*) HTTP requests triggered without explicit user interaction.

With the advent of forms and scripts in 1995[12], cross-site interactions became a real security threat. For example, as shown in Figure 2.1, a genuine website, say `example.com`, could now be compromised by an attacker who injects a malicious JavaScript code. This is an example of a cross-site scripting (XSS) attack. Listing 2.1 shows a concrete example of such a malicious code. In this, the JavaScript code attempts to create a new image element and crafts its URL (the *src* property) such that a cross origin HTTP request is sent to the attacker's site. The query string of the URL (*cookie="+cooky*) is crafted to contain a user's session cookie, accessed by JavaScript's *document.cookie* property. Finally, the specially crafted image element is appended to the web page's DOM. When a victim visits the infected page, the script executes in the victim's browser and steals his/her cookie. The victim could end up unwittingly participating in exfiltration, i.e., the leakage of private data to the attacker's site, say `attacker.com`. This is possible by the design of HTTP specifications since the image attempts to perform a content inclusion, which is a valid HTTP transaction.

```
var img=new Image();
var cooky=encodeURIComponent(document.cookie);
img.src="http://attacker.com/listener.php?cookie="+cooky;
document.appendChild(img);
```

Listing 2.1 Malicious JavaScript which steals and exfiltrates a user's cookie

JavaScript is the scripting language interpreted by web browsers and it is widely used in modern web applications. Two main security policies embeded in browsers restrict the access privileges of JavaScript — Sandbox and Same Origin Policy (SOP). In short, sandbox restricts JavaScript running in a web page from accessing the underlying file system, whereas, Same Origin Policy defines the conditions under which JavaScript running in one webpage can access the resources of another webpage. While the

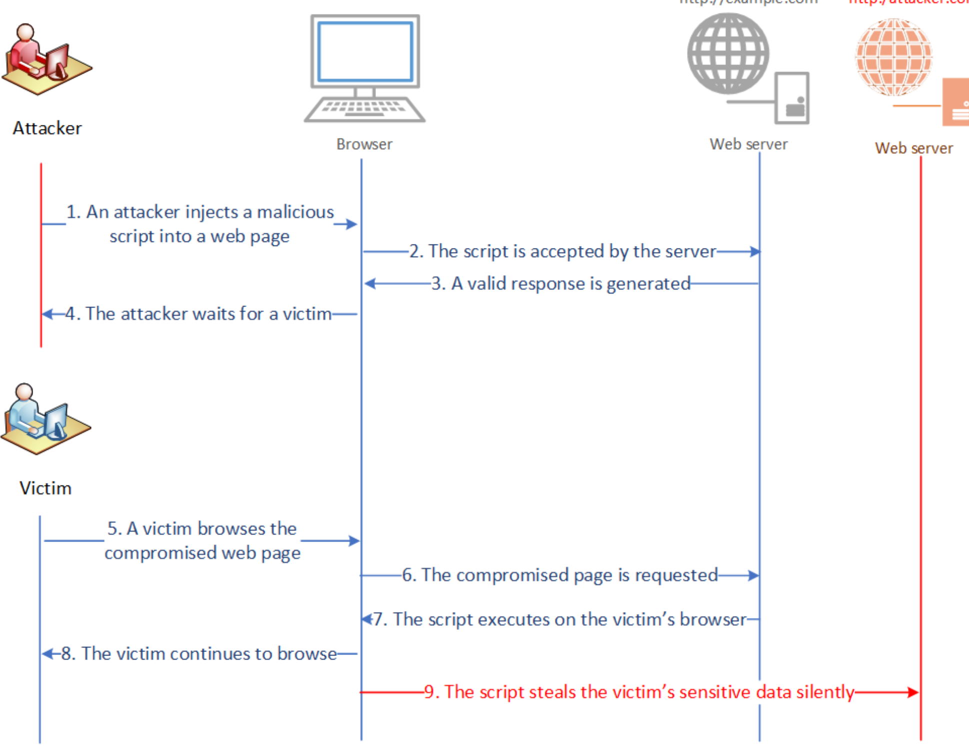

Figure 2.1 *Exfiltration of data from a website due to script injection*

sandbox restrictions are sufficiently strong, the restrictions around SOP are not sufficient by design and this is one of the main rootcauses of web-borne threats. The subsections below expand on SOP, its limitations and the need for stricter browser security policies.

2.1 Browser Security Model

To clearly appreciate the web-borne security threats, an understanding of the browser security model is required. Since the advent of JavaScript, browsers introduced a security policy called *Same Origin Policy (SOP)*, which is responsible for restricting the capabilities of JavaScript code in a web page.

2.1.1 Origin

Origin is the basic unit of isolation on the web platform, represented by the 3-tuple *(scheme, host, port)* [9].

- *Scheme* defines the protocol used for triggering a request to a web server (e.g., http, https, ftp etc.).

- *Host* defines the domain name that resolves to the IP address of the destination web server (e.g., google.com, yahoo.com).

- *Port* is the port number that is configured on the web server to accept web requests.

E.g., `http://A.com` and `https://A.com` belong to different origins since they differ in scheme (http vs https). Port 80 is the default port for HTTP connections and hence it is not explicitly mentioned in notations. e.g., In the case of `http://A.com`, the scheme is "http", the host is "A.com" and the port is 80. Table 2.1 helps in understanding the term *Same Origin* better.

Table 2.1 Understanding what *Same Origin* means.

Origin 1	Origin 2	Same Origin?
http://A.com	https://A.com	No
http://A.com	http://A.com:81	No
http://www.A.com	http://chat.A.com	No
http://A.com/user1	http://A.com/user2	Yes

Communications between different origins are generally known as cross-origin communications and the requests which trigger such communications are called cross-origin requests. Sub-domains (e.g, http:// chat.google.com and http:// mail.google.com), though have the same parent origin (http://google.com), are treated as different origins by most browsers for enhanced security.

Every document loaded in a web browser will be associated with an origin (derived from the document's URL). A document can in turn load resources such as images, scripts via content inclusion (<script src=URI> for example). Furthermore, the HTML standard allows content to be included across origins i.e., from third party servers (see Table 2.2). Any element that is embedded in a document inherits the document's origin.

2.1.2 Same Origin Policy (SOP)

SOP states that a script associated with an origin will have complete access to the DOM (Document Object Model), storage (e.g., cookies), and network (e.g., AJAX calls) within that origin. It will not be able to access any of these across origins. E.g., If a page belonging to the origin `http://A.com` opens a popup window pointing to `http://B.com`, a script in `http://A.com` will not have access to the DOM, storage, network of `http://B.com` and vice-versa.

Even if a script is loaded from a cross origin server, by virtue of SOP, the script has complete privileges on the document's data structures (Section 2.2.1). Since developers embed possibly malicious scripts from third party servers, the resultant web applications, called web mashups, will be insecure. A malicious script can deface the mashup by mutating its DOM, read sensitive information present in cookies and export it to destinations unknown to the mashup developer, through network calls. This problem is explained in detail in Chapter 4.

Table 2.2 Browser model with Same Origin Policy

Privilege	Initiator	Within origin?	Across origins?
DOM access	JavaScript	Yes	No
Storage access	JavaScript	Yes	No
Network access (AJAX)	JavaScript	Yes	No
Network access (Form Submission)	<form action=URI>	Yes	Yes
Network access (Content inclusion)	<script>,<iframe>,<audio>,<video>,<img>,<embed> etc.	Yes	Yes

Apart from restricting JavaScript access, browsers use SOP checks in features such as caching, pop-up blocking, geolocation sharing, password management, camera and microphone access etc.

2.2 Limitations of the browser security model

The sub-sections in this section provide details on why the current browser security model, specifically SOP, is not sufficient to meet the security requirements of the modern web.

2.2.1 Cross Origin Content Inclusion

As outlined in Table 2.2, SOP enforces restrictions on certain privileges requested by JavaScript, on cross origin content. However, it is quite liberal with respect to cross origin content inclusion i.e., loading content from one origin and including it in another origin. For example, a script element in orign `http://A.com` can make an HTTP GET request and load a JavaScript file from origin `http://B.com`'. This applies to all elements which make HTTP GET requests to load remote content. Few examples are shown in Listing 2.2.

```
<script src="URL"/>
<img src="URL"/>
<iframe src="URL">
<link rel="stylesheet" href="URL">
```

Listing 2.2 Examples of Cross origin content inclusion via HTML elements

In the case of loading JavaScript files across origins, the problem gets aggrevated further. Though the script file belongs to `http://B.com`, once it is loaded in the DOM of the document having the origin `http://A.com`, it inherits the origin of `http://A.com`. Due to this origin inheritence, the script will be treated as same-origin script and will have complete privileges on the origin `http://A.com`. i.e., the script will inherit all the privileges listed in Table 2.2. Unrestricted cross origin content inclusion is one of the main weaknesses of SOP. Apart from this, there are other inconsistencies, some of them being: an image tag can point to a script file, load and execute it., script from a trusted source can

dynamically and recursively load another script file from an untrusted origin. Due to this, trust cannot be verified by static analysis of web pages.

2.2.2 Cross-Site Scripting (XSS)

XSS is an attack technique in which attackers use vulnerabilities in web applications to inject malicious JavaScript code into the server. Code injection attacks happen due to insufficient validation of inputs before storing into database or lack of proper encoding before rendering the output of a web page to the browser. Based on the way XSS is triggered, it is classified into 3 types: Stored XSS, Reflected XSS and DOM XSS.

The Same Origin Policy does not differentiate between JavaScript code injected due to an XSS attack, versus third party JavaScripts embeded in a web page by the application developer. Scripts executing in a web page in either ways are treated the same and have equal privileges. They have read/write access to the DOM, full network access via XMLHTTPRequest and storage access via cookies/web storage. This is the reason why JavaScript injections a.k.a XSS attacks are considered highly dangerous.

2.2.3 Data-Exfiltration

If an attacker succeeds in injecting malicious content such as JavaScript into a web page through an XSS attack, he/she can access sensitive data from the page and send it to an attacker-controlled server. This can be achieved through the vectors listed in Table 2.2, that can trigger cross origin HTTP GET and POST requests. Listing 2.1 shows a JavaScript code snippet which, when injected into a vulnerable server's web page, steals the cookies set by that page. It creates a new image tag in the web page's document, which sends the stolen cookies to an attacker-controlled server (`http://attacker.com`). This outward flow of data across origins is called "Exfiltration" and SOP does not prevent this. Figure 2.1 gives a clear depiction of an exfiltration attack.

2.2.4 Cross-Site Request Forgery (CSRF)

CSRF is a popular web based vulnerability, listed as one of the top ten application security risks by OWASP [54]. In this attack, a malicious site instructs a victim's browser to send a request to an honest site. This malicious request is initiated on behalf of the victim using his/her network connectivity, browser state, cookies etc., thereby disrupting the integrity of victim's session. To launch a CSRF attack, an attacker crafts a malicious web page, which triggers cross origin HTTP requests to the honest site. Listing 2.3 shows a sample code snippet from such a page. It has an image tag that has a maliciously crafted URL, pointing to a banking site, `http://bank.com`.

```
<img src="http://bank.com/send.php?amount=100000&TargetBankAccount=1337"/>
```

Listing 2.3 Code snippet in a malicious web page that can trigger a CSRF attack on a banking site

Let us consider that a victim has logged into the banking site in a browser tab/window and opens the attacker's page in a different tab/window of the browser. The image tag in the attacker's web page triggers a cross origin HTTP GET request, which transfers the amount from the victim's bank account to the attacker's bank account (note the querystring *"?amount=100000&TargetBankAccount=1337"*). CSRF attacks can be triggered not only by HTTP GET requests, but also through HTTP POST requests using HTML form tags. It is important to note that the cross origin requests launched in a CSRF attack are not restricted by browsers, since SOP allows content inclusion and form submission across origins (Refer Table 2.2).

Note that in CSRF, the attacker masquerades as a genuine user and initiates requests from a malicious webpage to a vulnerable server (as opposed to data-exfiltration). Irrespective of the origin from which a request has initiated, browsers attach authentication credentials i.e., cookies, to every request made to the destination origin. Due to this, browsers do not distinguish between a request triggered by a genuine and a malicious web page. The Same Origin Policy does not include any mechanism to distinguish between a genuine and a forged request. Also, in most cases, servers do not have information about the origin that triggered the request (see Section 3.2.1).

2.2.5 Clickjacking

Clickjacking was first reported in web browsers in 2008 [27]. It is also known as UI-redressing and has gained popularity in the modern attacker community. In this, attackers lure users to visit a malicious page and trick them to click on invisible targets e.g., buttons, which belong to a cross origin web page. Typically, attackers embed target cross origin content in iframes, reduce their opacity to zero and position them above seemingly genuine buttons. End users will not have any suspicion or indication that their click is hijacked, but the attacker will be able use their click for malicious purposes. Clickjacking differs from CSRF in the fact that along with the click, user's credentials as well as CSRF tokens (Section 3.2.1) are submitted. This makes clickjacking more dangerous than CSRF.

There are many online scams/spams, especially on social networks, which use clickjacking and make money. Recently, Facebook sued an ad network [21], which stole personal information of users via clickjacking. The ad network made upto $1.2 million a month by employing this attack technique.

2.2.6 Cross-Site Timing Attacks

Bortz et al. [13] explained that the response time for HTTP requests can expose private information of a web user e.g., detecting if a user has logged in at a particular site, finding the number of items in the user's shopping cart etc. Though there are several ways to time web applications, as shown by Bortz et al., we examine a class of timing attacks called *cross-site timing attacks*, which rely on cross origin HTTP requests. In these attacks a genuine user is tricked to open a malicious page, which tries to load resources e.g., images, html pages etc. from a site being targeted. On measuring the time taken for the loading of the resources, sensitive information such as the login status of a user can be extracted.

Two recent works by Stone and Kotcher et al., showed how SVG filters [59] and CSS shaders [41] can be used as vectors for cross-site timing. Technically, cross-site timing attacks can be classified as CSRF attacks with the exception that the traditional defenses for CSRF i.e., tokens do not generally work for these. Typically, attackers target authenticated resources [36], which do not have CSRF tokens e.g., private profile pictures, script files etc. This means that a majority of websites are vulnerable to cross-site timing attacks. We have analyzed popular social networks and email providers and found at least one way of detecting the login status of a user. We found that apart from authenticated resources, authenticated URLs can also be used as a vector for login detection. Listing 2.4 shows the case where a script tag makes a cross origin HTTP request to a non-existing page on a target site, to detect the login status of a user on the target site.

```
<script src="http://example.com/user/nonExistingPage.php" onload=
   notLoggedIn() onerror=loggedIn()>
```

Listing 2.4 Login detection by fetching cross origin authenticated resources

Once the login status of a user is known, as explained by Bortz et al., spammers can perform invasive advertising and targeted phishing i.e., phishing a site which a user frequently uses, rather than phishing randomly.

Stealth mode Clickjacking Apart from these, we have identified an attack scenario that uses login detection, which we call *Stealth mode clickjacking*. Developers usually protect sensitive content using authentication. So in most cases, for a clickjacking attack to be successful, the victim should be logged in at the target site. Moreover, if the victim is not logged in and clicks on the framed target, authentication will be prompted, thereby raising suspicion. Using login detection techniques, an attacker can redesign the attack by ensuring that clickjacking code executes only if the victim is logged in at the target site, thereby removing any scope of suspicion. We observe that it is easy to compose such attacks with a comprehensive knowledge of the web.

Chapter 3

Related Work

This section explains about the related work done by researchers, along the web security aspects covered in this thesis. Section 3.1 contains the work done on Mashup Security, and Section 3.2 contains the work done on each of the web infiltration attacks.

3.1 Security of Web Mashups

Mashup security is an active area of research with contributions such as secure JavaScript subsets [23], Cross-domain interaction channels [33], Information flow tracking [45], Access control policies for JavaScript [49] etc. A survey of techniques proposed to solve the mashup security problem can be found at [16]. While some of them approaches require modifications in browser architectures, some make use of libraries or work arounds which can be used by developers in the existing browsers. In this section, we explain some of the approaches of the latter kind, which achieve effective separation of privileges between third party web content, while attaining interaction.

3.1.1 Fragment Identifier Messaging

A fragment is a part of a URL after the # symbol, which is used to navigate within a page via a hyperlink e.g., `http://example.com/article#section1`. If a web page is reloaded in a browser by changing only the fragment part of the URL, browsers do not send an HTTP request to the server. Therefore, a frame can send messages to a target frame by navigating the target frame with different fragment identifiers. The target frame can observe the value of the fragment by polling for *window.location.hash*. This feature has been used by developers as a workaround to communicate between cross origin iframes. Though this approache provides privilege separation along with interaction, the communication channel lacks authentication [11]. Also, since a URL can hold only a limited set of characters, this mechanism is not ideal to exchange long messages. The HTML5 *postMessage* API was built addressing these concerns of fragment identifier messaging and is the recommended approach to follow.

3.1.2 Subspace

Subspace [33] is a mechanism that allows cross-origin frame communication without sacrificing security. It is available as a JavaScript library which can be used by developers, without the need for any browser modification. Subspace relies on passing JavaScript objects across frames by manipulating the *document.domain* property of the frames. If two origins that want to communicate share a common suffix (e.g., `www.example.com` and `chat.example.com`), they can set their *document.domain* property to their hostname (in this case `example.com`). This technique is called domain relaxation and it allows both the origins to exchange JavaScript code and data with each other. By using nested iframes and domain relaxation techinique, subspace facilitates sharing JavaScript objects across origins, thereby achieving interaction. Though subspace guarantees confidentiality and integrity, managing subdomains turns out to be an expensive task for developers and hence it is not as widely adopted as other techniques.

3.1.3 Safe JavaScript Subsets

One of the popular techniques to embed third party JavaScript in a web page is to rewrite and restrict the code to a strict subset of JavaScript, which has desirable containment properties. By restricting the third party code to a subset of the language, an integrator can ensure that the code can interact only with object references explicitly provided by it. E.g., If an object in the language has no reference to *XMLHTTPRequest* object, it will not be able to trigger AJAX calls. By permitting the object a reference to *XMLHTTPRequest* object, capability to trigger AJAX calls is given. This is the core idea behind safe JavaScript subsets and is known as object-capability security model. Yahoo's ADSafe, Google's Caja, Facebook's FBJS follow this model. They allow restricted interactions via normal JavaScript objects and eliminating the usage of iframes. Several popular applications like Facebook, Orkut, iGoogle use JavaScript subsets while allowing untrusted third party code. The downside of this approach is the additional learning curve involved in using and maintaining the code.

3.1.4 HTML5 Enabled Privilege Separation

Akhawe et al. proposed a design [6] for achieving effective privilege separation in web applications using already available HTML5 security features. In their design, a HTML5 application has one privileged parent and any number of unprivileged children. The parent has three components-Bootstrap code, parent shim and policy code. The bootstrap code is the entry point of the applicaiton. It spawns the unprivileged children and controls their lifetime. The parent shim manages requests from children to make privileged calls. The policy code decides whether to allow or disallow calls initiated by the children. The children have two components- Child shim (which sends request to the parent shim for making privileged calls) and application specific code. The authors retrofit two popular Chrome extensions and a popular database management system (SQLBuddy) to use their design. They showed that the amount of trusted code running with full privileges reduced by a factor of 6 to 10000. Developers hosting applications of varying privileges under a single origin (e.g., `http://bank` and

`http://bank/sqlBuddy`) are suggested to use this architecture to ensure that a flaw in one application does not abuse the privileges of the other.

3.2 Mitigating Web Infiltration Attacks

In this section, we briefly describe existing defenses against each of web infiltration attacks - CSRF, clickjacking and cross-site timing attacks.

3.2.1 Approaches to Mitigate CSRF

In the case of CSRF, there are several server side (Secret tokens, NoForge, Origin header etc.) and client side defences (RequestRode, BEAP, CsFire etc.) to prevent the attack.

Secret Tokens: This is one of the most popular approaches used by developers. In this, a protected server generates a unique random secret token and embeds it into web pages in every HTTP response. When the server receives HTTP requests, it expects each request to contain the token it generated in the earlier response. It checks if the received token is the same as the one it generated earlier, and accepts the request only if the check succeeds. Since the token is not available to an attacker, request forgery cannot happen. CSRF Guard [55] and CSRFx [28] are a few server side frameworks which implement this technique. Though this technique is robust, most websites, including high profile ones, often miss them. Also, using XSS attacks and social engineering techniques, the secret tokens can be stolen thereby re-enabling request forgery.

NoForge: NoForge [39] is a server side proxy which inspects and modifies the HTTP responses sent to a browser. It modifies the responses such that future requests originating from the web page will contain a valid secret token. It takes countermeasures against requests that do not contain a valid token. The downside of this approach is, since it is a server side proxy, it will not be able to add tokens to dynamic content generated by JavaScript in the browser.

SOMA: Same Origin Mutual Approval (SOMA) [53] enforces constraints on HTTP traffic by mandating mutual approval from both the sites participating in an interaction. Websites send manifest files that inform a browser about the list of domains the site can communicate with. The domains whitelisted in the manifest expose a service which replies with a "yes" or "no" when queried for a domain name. When both the sites agree for the communication (via the manifest and the service), a cross origin request is allowed. Though SOMA enforces strict restrictions on cross origin interactions, it involves an additional network call to verify the permissions on a request. Moreover, it does not provide fine-grained control such as restricting only a subset of cross origin requests for a domain.

Origin Header: Barth [10] et al., proposed adding an *Origin* header to HTTP request headers, which indicates the origin from which each HTTP request initiates. It was an improvement over its predecessor - the Referer header, which includes the complete path of the page from where a request is originating. Due to privacy constraints, the Referer header is stripped by filtering proxies [4]. Since the Origin header

sends only the *Origin* in the request, it improves over Referer in terms of privacy. Majority of modern browsers already implemented this header. Using the origin information, the server can decide whether it should allow a particular cross origin request or not. However, origin header is not sent (set to *null*) if the request is initiated by hyperlinks, images, stylesheets and window navigation (e.g., *window.location*) since they are not meant to be used for state changing operations. Developers are forced to use *Form GET* if they want to check the origin of a GET request on the server. Such changes in application code require longer time for adoption by developer community.

Request Rodeo: Request Rodeo [38] is a client side proxy which sits in between web browser and the server. It intercepts HTTP responses and adds a secret random value to all URLs in the web page before it reaches the browser. It also strips authentication information from cross origin HTTP requests which do not have the correct random value, generated in the previous response. The downside of this is, it does not differentiate between genuine and malicious cross origin requests. Also, it fails to handle cases where HTML is generated dynamically by JavaScript, since this dynamic content has come after passing through the proxy.

BEAP: Browser Enforced Authenticity Protection [46] is a browser based solution which attempts to infer the intent of a user. It considers attack scenarios where a page has hidden iframes (clickjacking scenarios), on which users may click unintentionally. It strips authorization information from all cross origin requests by checking referer header on the client side. However, it also strips several genuine cross origin interactions, which are common on the web.

CsFire: CsFire [17, 18] builds on Maes et al. [44] and relies on stripping authentication information from HTTP requests. A client side enforcement policy is constructed which can autonomously mitigate CSRF attacks. The core idea behind this approach is - Client-side state is stripped from all cross-origin requests, except for expected requests. A cross-origin request from origin A to B is expected if B previously delegated to A by either issuing a POST request to A, or if B redirected to A using a URI that contains parameters. To remove false positives, the client policy is supplemented with server side policies or user supplied whitelist. The downside of this approach is that without the server-supplied or user-supplied whitelist, CsFire will not be able to handle complex, genuine cross origin scenarios and the whitelists need to be updated frequently.

ARLs: Allowed Referrer Lists (ARLs) [15] is a recent browser security policy proposed to mitigate CSRF. ARLs restrict a browser's ability to send ambient authority credentials with HTTP requests. The policy requires developers to identify and decouple credentials they use for authentication and authorization. Also, a whitelist of allowed referrer URLs has to be specified, to which browsers are allowed to attach authorization state. The policy is light weight, backward compatible and aims to eradicate CSRF, provided websites meet the policy's requirement. However, expecting all legacy, large websites to identify and decouple their authentication/authorization credentials may be unrealistic, since it could result in broken applications and also requires extensive regression testing. Our proposal, CORP, which uses whitelists like CSP and ARLs, does not require complex/breaking changes on the server. Details of the approach are explained in Section 5.2.1.

3.2.2 Approaches to Mitigate Clickjacking

There are several proposals to detect [8, 47] and prevent [57, 32] Clickjacking. At the same time, there are a few intelligent tricks [31, 43] that can bypass some of these proposals. Browser vendors and W3C have incorporated ideas from these efforts and are working towards a robust defense against clickjacking. Below are two important contributions in this direction:

X-Frame-Options (XFO) Header: The X-Frame-Options HTTP response header [50], was introduced by Microsoft in Internet Explorer 8, specifically to combat clickjacking. The value of the header takes two tokens-DENY, which does not allow an iframe to load any content, and SAMEORIGIN, which allows a frame to load only if its origin matches with the origin of the top frame. XFO was the first browser based solution for clickjacking.

CSP User Interface Security Directives: Content Security Policy (CSP) added a set of new directives- *User Interface Security Directives for Content Security Policy* [48] specifically to focus on User Interface Security. It supersedes XFO and encompasses the directives in it, along with providing a mechanism to enable heuristic input protections.

Both XFO and CSP, though promise to prevent clickjacking, leave CSRF wide open. Also, these solutions get invoked just before the frame is rendered, which is too late in the request/response lifecycle. Due to this, several bypasses such as *Double Clickjacking* [31], *Nested Clickjacking* [43] and *Login detection using XFO* [36] arise.

3.2.3 Approaches to Mitigate Cross-Site Timing Attacks

Bortz et al. [13] proposed that by ensuring that a web server takes a constant time to process a request, cross-site timing attacks can be mitigated. However, it is unlikely to get wider acceptance in web community as it involves complex server side changes. A popular recommendation by security researchers is to disable *onload/onerror* event handlers for cross origin requests, but this affects genuine cases. As of date, cross-site timing attacks are still unresolved.

Chapter 4

Building Secure Web Mashups

The advent of open data and APIs (Application Programming Interfaces) have led to the growth of insightful web applications, which cut across domains such as health care, retail, finance etc. As soon as organizations open up their data and provide APIs, developers compete with each other to build rich web applications, referred to as *Mashups*, by composing APIs from various service providers. While this newer development paradigm has made the web a collaborative platform, web developers have found it hard to understand and leverage browser security mechanisms like Same Origin, Content Security etc. As a result, the applications they develop often violate the principle of least privilege [72]. For example, a travel agency could develop a useful web-based visualization by composing a user's past travel itineraries (fetched using its own API), finance data (fetched using a bank's API), health records (fetched using a hospital's API). However, if the web application was not developed using the state-of-the-art security mechanisms, an advertisement on the application's web page could have complete access to the user's sensitive data and export it to a spammer. Therefore, with the growing adoption of open data by organizations, the need for developers to build secure applications by utilizing the security capabilities provided by the web platform has become even more important.

4.1 Evolution of Mashups

For close to a decade after the World Wide Web was made available to the public in 1993 [69], data belonging to websites existed in isolation. With the advent of technologies driving web services (SOAP [68], REST [22] etc.) in early 2000, websites started exposing their data via APIs, which made the web a collaborative platform. During the same period, the evolution of AJAX [24] (Asynchronous JavaScript and XML), JSON [14] (a lightweight data interchange format) and thereafter the rise of JavaScript libraries such as jQuery, Dojo, YUI etc. enabled web developers to build richer web applications. Leveraging these technologies, JavaScript in a web page could communicate asynchronously to the server from which the page was loaded and could dynamically update the page with fresh content. Utilizing the data exposed via APIs and JavaScript's capabilties, a new breed of web applications called Mashups evolved, which integrated content from various online resources and provided enriched

results. One of the popular examples of mashups is `HousingMaps.com`, which utilizes real estate information from *CraigsList* and overlays it on *Google maps*, providing a richer user experience. Web applications which come integrated with social widgets (e.g., Facebook's *Like*, Twitter's *Tweet*), comments systems (e.g., Disqus) etc., can also be called as mashups, since they aggregate content from more than one server.

Mashups can be broadly classified into two categories based on their mode of development: server side mashups and client side mashups. In server side mashups, the server hosting the mashup application plays the role of a proxy and relays data between a client and third party services. It aggregates data from various services and provides a unified user interface to the client. Several enterprise mashups take this approach since there is more control to deploy additional layers of defense over third party code. The limitations of this approach are the costs involved in maintaining the server, additional bandwidth and latency costs incurred due to routing via an intermediary server, instead of directly accessing the third party services from the client. In client side mashups, there are no intermediary proxies involved and communication takes place between the mashup application in the browser and the third party services. This approach is widely used by developers since it overcomes the limitations of server side mashups, apart from providing better interactivity and responsiveness. However, it has several security concerns whose roots lie in the architecture of web browsers. We refer to client side mashups as *Web mashups*, the content loaded from a third party website as a *widget* and the parent page integrating and unifying the widgets as the *integrator*.

In this work, we address the problem of enabling developers to build secure web mashups. We explain the need for newer security APIs in HTML5 and security concerns which arise due to their insecure usage. We present a security abstraction layer (as an API library) called "SafeMash", which handles low-level security checks and assists developers in creating secure and interactive mashups. We empirically demonstrate the usage of SafeMash by first building an interactive mashup using open APIs from *ProgrammableWeb* (without using state-of-the-art security mechanisms) and then rebuilding it using SafeMash without losing functionality.

The rest of this section is organized as follows: Section 4.1.1 explains the security concerns in mashups. Section 4.2 explains the newer, HTML5 enriched security model introduced in modern browsers. Section 4.3 shows the attacks that are possible due to insecure usage of HTML5 APIs. Section 4.4 explains how Safemash eases developer's tasks in creating safer mashups, leveraging the newer security model.

4.1.1 Security concerns in mashups

Web mashups are developed using two approaches: loading third party JavaScript files (e.g., Google maps) or embedding remote resources via iframes (e.g., Facebook widgets). Both the approaches have their own advantages and limitations. In the case of the script approach, the loaded JavaScript assumes the origin of the integrator, and not the origin of the server from which it has been served. Therefore, the script has complete access to the DOM (Document Object Model) of the integrator. Due to this, better

interactivity can be achieved, but at the cost of giving the script complete privileges to the integrator. Figure 4.1 depicts this behavior, where scripts loaded from various origins have equal access to the parent page's DOM.

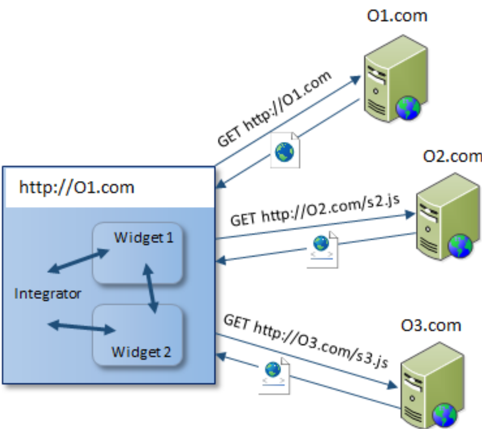

Figure 4.1 *A typical web mashup built by embedding third party JavaScript. The bidirectional arrows in the webpage indicate lack of privilege separation between content in the widgets and their parent.*

To overcome the security problems of using scripts and ensure separation of privileges, cautious developers use iframes to embed third party content in a web page. Iframes embedded in a webpage can load new documents. Elements inside an iframe inherit the origin of the document loaded in the iframe and do not not inherit the origin of the iframe's parent document. If the origin of the document inside the iframe does not match with the origin of its parent document, the restrictions of Same Origin Policy (Table 2.2) will be enforced by browsers. This means, a script in the parent page will not have any privileges on content inside the iframe and vice-versa, thereby isolating third party content from accessing the state of the parent. While the security concern is resolved, this draconion isolation defeats several benefits of web mashups.

4.1.2 Security versus Interactivity

Let us understand with an example the trade-off between security and interactivity in mashups. Consider a mashup having a widget displaying a map (using Google Maps API), a widget displaying weather information (using Yahoo weather API) and other useful widgets. A possible business requirement could involve clicking on any geographical location in the map widget, to display the corresponding weather information in the weather widget. If the mashup was designed by directly embedding third party scripts in the parent page, the requirement can be easily achieved, but every script will have access to every widget. So a script belonging to a malicious widget can tamper map data, weather information or steal sensitive user-data. On the other hand, if the mashup was designed using the iframes approach, scripts belonging to each widget will not be able to tamper content belonging to other widget due to SOP restric-

tions. However, the core functionality of communicating information between map and weather widgets cannot be achieved. Though developers have come up with fragment identifiers (see Section 3.1) as a work-around for communication between iframes, the approach does not provide authentication. Moreover, in spite of SOP, a script inside an iframe can open popups, redirect parent page and submit forms. These limitations demanded the need for secure mechanisms for building mashups.

4.2 Newer Browser Security Model

HTML5 introduces several security mechanisms such as PostMessage API [66], Sandboxed iframes [67], Cross Origin Resource Sharing [65] to overcome the limitations of existing browser security model. Apart from these, a new declarative policy called Content Security Policy (CSP) implemented by modern browsers helps developers in enforcing restrictions on the HTTP transactions originating from a webpage. These new specifications extend the traditional browser security model and encourage safer cross origin collaboration and communication. The rest of the section introduces the capabilities of the new APIs.

4.2.1 PostMessage API

The *postMessage* API of HTML5 enables documents belonging to different origins to authenticate and communicate with each other. This solves one of the main limitations in mashups which are built using iframes. Let *iframe1* and *iframe2* be the ids of two iframes loading their documents from `http://A.com` and `http://B.com` respectively.

```
var iframe2= queryElement('iframe2');
var targetOrigin = "http://B.com";
iframe2.postMessage("text", targetOrigin);
```

Listing 4.1 JavaScript code in *iframe1* sending messages to *iframe2* using postMessage API

```
window.addEventListener("message", receiveMessage, false);
function receiveMessage(event){
  if (event.origin !== "http://A.com")
    return;
  // ... (Authentication successful)
}
```

Listing 4.2 JavaScript code in *iframe2* receiving messages from *iframe1* using postMessage API

Listing 4.1 shows how *iframe1* can send a message to *iframe2* using *postMessage*. To receive a message, *iframe2* must add a “message” event listener and process the received data via a callback (see Listing 4.2). *PostMessage* improves over existing communication mechanisms (e.g., fragment identifiers) by providing confidentiality (senders can specify the intended recepient’s origin) as well as authentication (recepients can identify the sender via the *event.origin* property).

4.2.2 Iframe Sandbox

The *sandbox* attribute of iframes, introduced in HTML5, enables developers to assign fine-grained privileges on third-party content. By merely adding the *sandbox* attribute without any values (flags), an iframe will be assigned a unique and temporary origin. This means that even if the origin of the framed content is same as that of its parent, it is treated as cross-origin content and is denied privileges like script execution, form submission, opening popups etc.

4.2.2.1 Relaxing Sandbox Restrictions

Developers can relax sandbox’s restrictions by assigning a whitelist of flags such as allow-scripts, allow-forms etc., as values to the *sandbox* attribute, as shown in Listing 4.3. Some of the important flags of sandbox are described below:

- allow-forms: Allows form submission within the iframe
- allow-scripts: Allows script execution within the iframe
- allow-same-origin: Re-enables same-origin treatment to the sandboxed content.
- allow-top-navigation: Allows framed content to navigate and replace the top-level window.
- allow-popups: Allows framed content to open popups.

```
<iframe sandbox="allow-forms allow-scripts">
    ... Third party content...
</iframe>
```

Listing 4.3 Whitelisting privileges using sandbox

More details about a few experimental flags and nested browsing contexts can be found in the W3C recommendation [67].

4.2.3 Content Security Policy

Allowing HTTP traffic without any restrictions (e.g., via content inclusion and form submission etc.) is considered as one of the limitations of SOP and the cause of web attacks like Cross Site Scripting

(XSS) and data exfiltration. To fix this limitation, Mozilla pioneered the development of Content Security Policy (CSP) [58], which enables web administrators to declare a set of content restrictions for a web resource. Developers can declare CSP via HTTP response headers or meta tags of a page and inform the browser to which whitelisted origins HTTP requests will be allowed from the page. Apart from imposing restrictions on HTTP traffic, CSP automatically disables inline scripts (e.g., event attributes like *onclick*) and evaluation of strings (e.g., *eval(str)*, *setTimeout* etc.). These features are known to be responsible for XSS attacks and hence CSP disables them by default. Developers can re-enable these features using the *options* directive, but should carefully evaluate the possible threats.

```
Content-Security-Policy: script-src 'self' https://apis.google.com
```

Listing 4.4 Whitelisting HTTP traffic via CSP

The code in Listing 4.4 shows a sample CSP rule which restricts loading of scripts only from a document's own origin (self) and from `https://apis.google.com` and rejects any other loading attempts. Some of the important directives of CSP and their restrictions are listed below. A complete list of directives can be found at W3C's CSP specification [70].

- connect-src: restricts network activity (e.g., XHR, Web sockets etc.) only to whitelisted origins.
- default-src: sets source list for unspecified directives
- form-action: restricts form submissions
- frame-ancestors: restricts embedding content via iframes
- script-src: allows JavaScripts files to load only from whitelisted origins

Not specifying CSP directives (say, *img-src*) is equivalent to specifying *img-src:* '*'. This way CSP ensures that it does not break existing web applications by not being draconion. Figure 4.2 shows a mashup which utilizes these new security paradigms to achieve privilege separation and adding additional layers of defense.

Though the newer browser security model and the APIs greatly enhance the security of web applications, certain insecure usage patterns of these APIs introduce newer vulnerabilities into web applications. These are discussed next.

4.3 Insecure Usage of HTML5 APIs

HTML5 APIs were designed giving highest priority to security. They provide options of varying degree for web developers to tighten or relax the security of web applications. However, certain insecure usages of these APIs open doors to newer vulnerabilities. Though the HTML5 specification warns

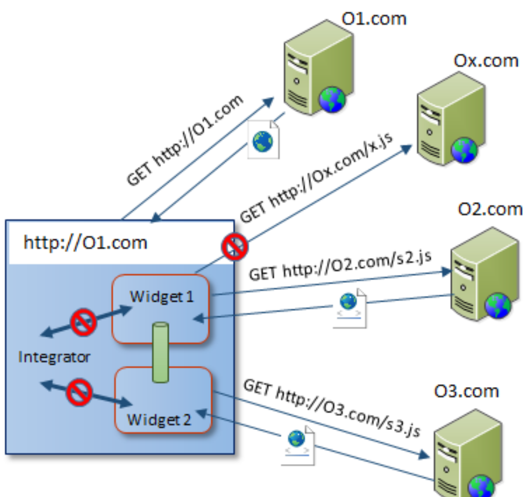

Figure 4.2 An interactive web mashup with privilege separation. Sandboxed iframes restrict JavaScript activity across widgets. PostMessage channel (depicted by a pipe) enables inter-widget communication. CSP enforces restrictions on HTTP traffic (e.g., blocking request to Ox.com in the figure).

against the insecure usage, even reputed development teams fail to follow the guidelines while developing web applications [26]. In this section, we explain how the insecure usage of *postMessage* and *sandbox* APIs can result in several vulnerabilities in web applications.

4.3.1 Security Considerations in Sandbox

The HTML5 *sandbox* API is an important DOM security feature which helps in completely switching off JavaScript's activity to completely switching it on, providing various options in between. Both the extremes can be used by attackers to cause undesirable consequences.

4.3.1.1 Sandbox Flags and Privilege Escalation

One of the security warnings in the HTML5 sandbox specification [67] highlights that if the framed page and its parent page belong to the same origin, then the flags "allow-same-origin" and "allow-scripts" should *not* be used together. This is because the "allow-same-origin" flag relaxes the unique origin treatment of sandboxed iframe while the "allow-scripts" flag enables script execution within the iframe. Due to this, the JavaScript code in the iframe can access the parent page, mutate its DOM and remove the sandbox restriction on the iframe altogether. As a result JavaScript code in the iframe achieves complete privileges on the mashup, which defeats the purpose of having privilege separation primitives.

4.3.1.2 Disabling Frame Busting Defense

Clickjacking or UI-Redressing is an attack technique discovered only in 2008 by Jeremiah Grossman and Robert Snake [27]. In this, an attacker loads a genuine webpage (say `http://G.com/index.php`) in an iframe and sets the CSS properties of the iframe such that it is invisible and positioned on top of a fake, attractive button (e.g., “Claim this lottery”). If a user who is already authenticated to the genuine site visits the attacker’s page and unwittingly clicks on the fake button, the user’s click is hijacked (i.e., intercepted by the hidden iframe), thereby leading to dangerous consequences. There are two popular approaches to defend against Clickjacking attacks. One approach is to disable the presentation of the page using JavaScript and enable it only if it opens in the topmost window (proposed by Rydstedt et al. [57]). This technique is called frame busting and can be achieved using the code snippet in Listing 4.5.

```
<style> body{display:none;} </style>
<script>
   if(self == top) {
       document.getElementsByTagName("body")[0].style.display='block';
   } else { top.location = self.location; }
</script>
```

Listing 4.5 Defeating Clickjacking using JavaScript

Frame busting is only a work around and is not considered as a standard defense against Clickjacking. The other approach is to add an HTTP response header called *X-Frame-Options* [50] on webpages. When browsers load any page having this header, they prevent rendering of the page in an iframe. Though most browsers support the enforcement of this header, very few developers use this and rely on frame busting as Clickjacking defense. We have seen in Section 4.2.2 that *sandbox* prevents script execution inside an iframe. Leveraging this, an attacker can create a fake page having sandboxed iframe and attempt a Clickjacking attack on a genuine page. If the genuine page relies only on frame busting, the defense will be broken since *sandbox* disables execution of the frame busting code. Since web standards recommend the usage of sandboxed iframes to build safer mashups, pages which are loaded as widgets will be vulnerable to clickjacking if frame busting is the only defense used. Therefore, it is very important that developers use *X-Frame-Options* header with *allow-from* directive to configure a whitelist of origins that are allowed to frame the page. Unfortunately, in some cases it is not possible to provide a whiltelist of allowed origins (e.g., social plugins such as Facebook *Like* which are used by millions of websites) and they continue to remain vulnerable to Clickjacking.

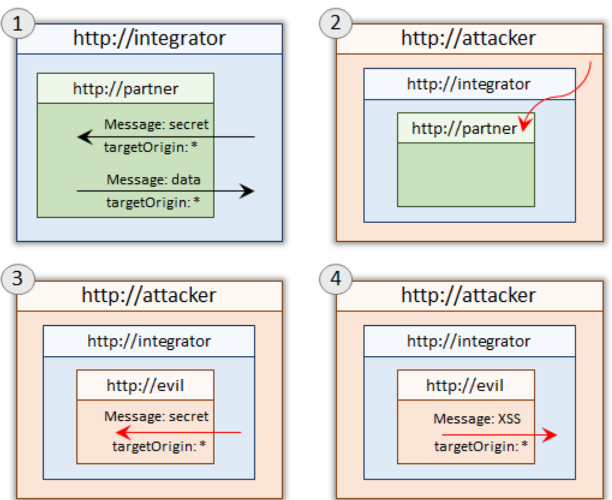

Figure 4.3 Attacks on postMessage API. (1) *postMessage* communication with targetOrigin set to '*'. (2) Attacker redirecting a frame via descendant policy. (3)Attack on confidentiality. (4) Attack on integrity.

4.3.2 Security Considerations in PostMessage

With respect to *postMessage* API, the HTML5 specification warns developers against the usage of '*' in the *targetOrigin* property and also suggests to verify authenticity of messages before replying. Unfortunately, developers ignore these guidelines, which are also not mandated by design.
In 2009, Barth et al. [11] showed that browsers implement a policy different from Same Origin Policy to determine whether a frame is allowed to reset the location of another frame. This is called *frame navigation*. Historically, browsers implemented frame navigation policies such as *Permissive policy* and *Window policy* which were too liberal and hence vulnerable to dangerous attacks. Modern browsers follow a stricter policy called *Descendant policy*, which says that a frame can navigate only its descendants. It is important to note that the policy is not dependent on origins i.e., a frame can navigate any of its descendant frames even though belong to different origins. This loophole is used by attackers to exploit applications that use the HTML5 *postMessage* API insecurely. The following subsections explain the attacks that can be launched due to insecure usage of postMessage API.

4.3.2.1 Attack on Confidentiality

Confidentiality refers to preventing disclosure of information to unauthorized entities. Using *postMessage* API, a frame can send a message that can be read only by an authorized recepient (i.e., a frame belonging to the origin specified in the *targetOrigin* property). This ensures that confidentialty is maintained. However, developers often set the *targetOrigin* property to '*', which allows any unauthorized recepient to read the message. Combining this insecure configuration with the loophole in descendant policy, attackers can hijack and steal the information being exchanged between frames. Figure 4.3 ex-

plains the sequence of steps involved in the attack. Step 1 shows the case where a mashup integrator uses *postMessage* insecurely to communicate with its widget. To hijack the communication in the mashup, an attacker loads the integrator in an iframe and navigate its child widget to an attacker-controlled evil widget (Step 2 in the figure). If the integrator sends any message to its widget (which was redirected to an evil widget in Step 2), the attacker will be able to receive the message (Step 3 in the figure). This way, an attacker can compromise the confidentiality of the communication with the insecure usage of *postMessage* API.

4.3.2.2 Attack on Integrity

Integrity refers to ensuring that data is not tampered by unauthorized entities in transit. *postMessage* API provides a way of verifying the origin of the received messages (i.e., verifying authenticity), thereby ensuring that integrity of messages is not lost. However, developers often miss verifying the origin of the received messages, thereby opening a vulnerability. To make use of this vulnerability, an attacker frames the mashup integrator and redirects its widget to an evil widget (Steps 1 and 2 as in the previous attack). Apart from stealing the integrator's message (Step 3), the attacker replies with a malicious piece of JavaScript code (e.g., a Cross Site Scripting (XSS) attack vector). Since the integrator does not verify the sender of the message and executes the incoming data, it will be prone to dangerous consequences.

Though a majority of modern browsers already support the new security model and HTML5 security APIs, their adoption by web developers has not picked up the expected pace. Hanna et al. [26] showed that Facebook Connect and Google Friend Connect, two new client-side protocols that are built using *postMessage*, were vulnerable to confidentiality and integrity attacks. To use these APIs effectively, a web developer is expected to understand the problems in traditional browser security architecture and the need for newer APIs (Section 4.1.1). A developer may not know how these APIs work in tandem with the newer browser model and how to compose them to build safer mashups. As we have explained, each of these APIs come with certain security precautions, failing to adhere to which results in a weak security configuration. Therefore, there is a need to simplify the task of developers and encourage them to embrace the new web standards while building complex web applications like mashups, in the era of open APIs.

4.4 Implementation of SafeMash

To assist web developers in utilizing the newer security model implemented by most browsers and creating safer mashups, we have developed SafeMash, a JavaScript API library. SafeMash exposes an API for creation of isolated widgets (using HTML5 Iframe sandbox), exchanging information between them (via HTML5 postMessage API) and impose content restrictions on the widgets (via CSP). It incorporates security checks which developers often miss while using direct HTML5 APIs. Also, it

educates developers by throwing useful warning messages as exceptions, in case certain configurations are missed.

To use SafeMash, developers need to embed the library "SafeMash.js" in the mashup page as well as widget pages and invoke the library's API methods in their JavaScript code. The library was designed following the design pattern of jQuery, a popular JavaScript library. It exposes a single global variable "$m" and accepts CSS selectors for querying DOM elements. However, it has no dependency on any JavaScript library.

4.4.1 SafeMash API

To embed third party content as widgets, developers should use the *createWidget* method (see Listing 4.6). It accepts a mandatory *loadPage* property which expects a widget's URL as its value and an optional *sandboxFlags* property, which expects a space-separated whitelist of HTML5 sandbox's flags (refer Section 4.2.2).

```
<script src="safemash.js"></script>
<script>
 $m('widgetContainer').createWidget({
  loadPage: 'http://domainName/gmaps',
  sandboxFlags: 'allow-scripts allow-forms' });
</script>
```

Listing 4.6 Widget creation using safeMash

To send a message to a widget (i.e., iframe/window), developers should use the *send* method by passing a CSS selector of the target widget (see Listing 4.7). The *send* method accepts a mandatory *message* property, which expects a message string as its value and an optional *targetOrigin* property, which expects the origin of the receiver as its value.

```
<script>
  $m(targetWidget).send({message: 'Hello'});
  $m.receive({from: 'http://domainName',
    callback: receiveMessage});

  function receiveMessage(event){
    console.log('Data received: ', event.data);
  }
</script>
```

Listing 4.7 Frame communication using safeMash

To receive a message from a widget, the *receive* method should be used. It accepts a mandatory, developer-supplied callback function which will be triggered once a message is received. It also accepts an optional property *from*, which expects the sender's origin as its value. A mashup's parent page and its widgets can leverage SafeMash and configure Content Security Policy (CSP) rules using the *applyCSP* method (see Listing 4.8). Internally, this method adds a HTML *Meta* tag to the corresponding page and applies the CSP rule as its value. Applying CSP via meta tags becomes handy when developers do not have permissions to set response headers.

```
<script>
$m.applyCSP('img-src http://*.flickr.com');
</script>
```

Listing 4.8 Applying CSP using SafeMash

4.4.2 Security Checks Built into SafeMash

Apart from simplifying the usage of modern security APIs, SafeMash comes with certain built-in security checks. As we have seen in Section 4.3, the HTML5 specification issues security warnings to developers against the usage of certain configurations. However, browsers do not give any hints to developers if such configurations exist in code. SafeMash checks for insecure configurations and warns developers by throwing exceptions.

4.4.2.1 Sandbox Related Checks

The *createWidget* method of SafeMash creates a sandboxed iframe with the flag "allow-scripts" by default, which allows JavaScript execution within the iframe. To constrain or relax the privileges of content in the iframe, developers can set the property "sandboxFlags" and pass the desired flags, which override the default flag. We have seen in Section 4.3 that the flags "allow-scripts" and "allow-same-origin" should not be used together since they neutralize the effect of the sandbox attribute by removing it completely. SafeMash issues warnings if developers configure these two flags together in their code, thereby reducing the scope of privilege escalation attack on sandboxed iframes.

4.4.2.2 PostMessage Related Checks

The postMessage API's specification [66] too comes with certain security warnings. While sending a message (line 1 in Listing 4.7), specifying the "targetOrigin" (which is optional) ensures that the message is sent only to a specific origin, which otherwise defaults to '*' (any origin). Browsers do not complain if the "targetOrigin" is not configured, which could result in the leakage of sensitive information. If SafeMash's "send" method is used, it warns the developer if the target origin is missing.

Also, before responding to a request using postMessage API, it is important to verify the origin of the sender. If this authentication is not performed, any random frame can send a message and invoke undesirable consequences. If SafeMash's "receive" method is used, it warns the developer if the sender's origin i.e., the "from" property is missing. If it is configured, SafeMash internally checks if sender's origin (retrieved from the browser's message event) matches with the configured origin and rejects the message if there is a mismatch. This is an important authentication check which developers often miss while using the HTML5 *postMessage* API and hence it is included in the library.

4.4.3 Evaluation

To evaluate SafeMash, we have built a widget-style toy mashup inspired by DropThings [73], a personalizable Web2.0 AJAX start page. With the growth of open APIs, this mashup architecture is still relevant and has manifestated into social widgets (e.g., *Tweet*, *Like* buttons), discussion systems (e.g., Disqus comments), dashboards etc. Our toy mashup consists of three widgets: a map widget, a gallery widget and a custom search widget. On submitting a search query for a location (e.g, "Hyderabad") in the search widget, the map widget uses Google Map's API to update the map and the gallery widget uses Flickr's API to fetch photos corresponding to the search query. Though the functionality is intact, there is no privilege separation in the mashup (as explained in Figure 4.1). To achieve privilege separation using state-of-the-art browser security mechanisms, we have rebuilt this mashup using SafeMash without losing any functionality. One of the key observations from this exercise is that web developers tend to focus primarily on achieving the desired functionality, with little or no thought about the *Principle of least privilege* [72]. Live demonstration of both the versions of the toy mashup, the source code of SafeMash and attacks on the insecure usage of *postMessage* API are available on GitHub and can be accessed at [62].

Chapter 5

Enhancing Browser Security Policies

In Chapter 2, the current security model of web browsers has been explained in detail. As mentioned in Section 2.2, the Same Origin Policy is insufficient to meet the security requirements of the modern web. In summary, the following are the limitations of the current browser security model:

1. Cross Origin Content Inclusion
2. Cross-Site Scripting (XSS)
3. Data-Exfiltration
4. Cross-Site Request Forgery (CSRF)
5. Clickjacking
6. Cross-Site Timing Attacks

Content Security Policy (CSP), introduced in 2010 [58] improves on SOP in mitigating the first three limitations in the above list, by restricting the sources of external content and disabling inline scripts. However, the problem of cross-origin requests from a malicious website to a genuine website i.e., the last three limitations in the above list, was left unanswered by SOP and CSP.

5.1 Web Infiltration attacks

Our work on enhancing browser security policies begins by seeking a common thread between CSRF, clickjacking and cross-site timing attacks with the goal of understanding the limitations of CSP in addressing these attacks. We label these attacks as *Web Infiltration attacks*. The root of web infiltration is a request initiated from an evil page to a genuine but unsuspecting server (Figure 5.1). In web infiltration attacks, a victim who is already logged in to a genuine site, *G.com*, unwittingly visits an attacker's site, *A.com* in a separate browser instance (or tab). The web page obtained from *A.com* triggers state-changing requests to *G.com* either through an automatic form submission initiated by a script or via an

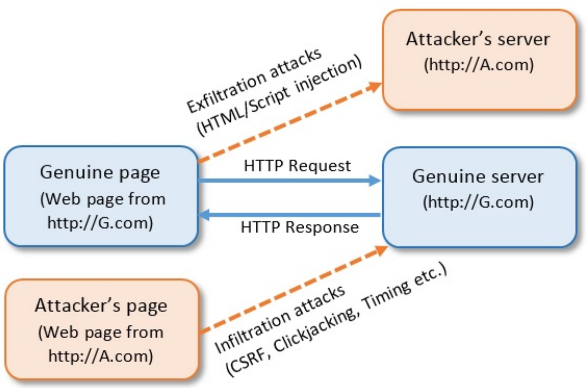

Figure 5.1 Exfiltration vs. Infiltration attacks

<img> tag, or through other similar vectors. The request to *G.com* goes from the victim's browser and uses the victim's credentials. *G.com* is unable to discriminate between genuine and forged requests. Web infiltration is complementary to exfiltration. Exfiltration is caused by XSS and can be controlled by CSP. Infiltration, on the other hand, cannot be controlled by CSP.

5.1.1 Observations and Inferences

We propose a novel approach to prevent web infiltration, based on the following observations:

- **Observation 1:** Irrespective of how a network event (HTTP request) is initiated, a web server responds with a resource. Therefore, any network event, e.g., loading an image can infiltrate and potentially change the server's state e.g., delete a resource.

- **Observation 2:** The prevention and detection techniques for web infiltration attacks that we have investigated are triggered too late. They apply either after an HTTP request leaves the browser [55, 13] or after the browser has already received the response [50, 48].

- **Observation 3:** Client side state information (cookies) of a website is shared across all tabs of a browser or multiple instances of the same browser, even though its access by other websites is restricted by Same Origin Policy.

- **Observation 4:** Website developers or administrators segregate the paths of various resources on the server, as a good engineering practice.

From *Observation 1*, we infer that a policy which monitors the initiator of web interactions is required. From *Observation 2*, we infer that every request must be subjected to the policy before it leaves the browser. From *Observation 3*, we infer that the policy should be available to and enforced by all tabs of the browser. From *Observation 4*, we infer that segregation of resource paths can be used as an important factor in the design of the policy.

Based on the above inferences, we propose a simple security policy, *Cross-Origin Request Policy (CORP)*, to prevent web infiltration attacks. The policy is a 3-way relation defined over the sets browser *event types*, *origins*, and the set of *resource paths* derived from the server's origin. CORP may therefore be seen as a policy that controls *who*, i.e., which site or origin, can access *what*, i.e., which resource on a cross-origin server, and *how*, i.e., through which browser event. CORP is declarative; it can be added as an HTTP response header to the landing page of a website. To implement the policy, web administrators need to segregate resources on the server based on the intended semantic effect of the resource. For example, all public resources could be in the path */public*, while all state changing resources could be sequestered in a different path. Thus the semantics of resources is mapped to paths. Fortunately, as discussed in Section 5.4, most website administrators already segregate resources along the lines proposed by the policy.

A web browser enforcing CORP would receive the policy and store it in memory accessible to all tabs or browser instances similar to the cookie storage mechanism. Assume that a tab t_A contains a page p_A from a server s_A. Along with the page p_A, the browser also receives a CORP policy $c(s_A)$ from s_A. Assume that the browser now opens a page p_B received from s_B in tab t_B and p_B attempts to make a cascading cross-origin request to s_A. The cross-origin request from p_B to s_A will be intercepted and allowed only if it complies with the permissions $c(s_A)$.

Threat Model: Throughout the work, we take into consideration only the threats that come under the capabilities of a web attacker. The threat model under consideration has been explained in Section 1.2. A web attacker has root access on at least one web server and can generate HTTP requests against any web server. However, the attacker has no special network privileges, which means threats like man-in-the-middle cannot be realized and HTTP headers generated by the browser or server cannot be tampered.

Contributions: Our contributions in this work are as follows: (1) We have identified a class of web infiltration attacks that include CSRF, clickjacking and cross-site timing attacks and designed a uniform browser policy to mitigate all of them. Recently, we have discovered that CORP can also be used to defend against application-level DDoS attacks. (2) We have formalized our proposal in Alloy [34], a finite state model checker, and verified that it is sound. (3) We have built two websites - one playing the role of a genuine website and the other a malicious website (a test suite) triggering malicious calls to the first. We have collected a large number of attack vectors from literature and incorporated them into the test suite. (4) We have implemented our proposal as an extension for Google Chrome web browser. We have evaluated the extension by configuring CORP on the genuine site and verified that infiltration attacks by the malicious site are blocked by the extension. (5) We have configured CORP on three popular open source web applications in our test environment to verify the effectiveness and ease of deployment on real world websites. (6) We have analyzed the home page traffic of over 15,000 popular websites and confirmed that the burden on web administrators to deploy CORP will be minimum.

We observe that CSRF, clickjacking and cross-site timing attacks have a common root, which is a cross origin HTTP request triggered by a malicious client to a genuine server without any restrictions.

We attempt to mitigate these attacks by devising a uniform browser security policy explained in detail in Section 5.2.

Organization of the chapter: The rest of the chapter is organized as follows: Section 5.2 explains the design of CORP and the security guarantees provided by it. Section 5.3 gives a brief introduction to Alloy and validates the soundness of CORP. Section 5.4 describes the implementation of CORP as a Chrome extension and the experimental methodology to evaluate its effectiveness.

5.2 Cross Origin Request Policy

In this section, we first explain the core idea behind Cross Origin Request Policy (CORP) and its importance in mitigating web infiltration attacks. Next, we explain the model of a browser which receives CORP and enforces it. Finally, we explain the directives which make the policy, with examples.

5.2.1 Core Idea Behind CORP

Based on our clear understanding of various types of web infiltration attacks (Section 5.1), we realize the need for a mechanism that enables a server to control cross origin interactions initiated by a browser. Precisely, a server should have fine-grained control on *Who* can access *What* resource on the server and *How*. By specifying these rules via a policy on the server and sending them to the browser, requests can be filtered/routed by the browser such that infiltrations attacks will be mitigated. This is the core idea behind CORP. Formally speaking, *Who* refers to the set of origins that can request a resource belonging to a server, *What* refers to the set of paths that map to resources on the server, *How* refers to the set of event-types that initiate network events (HTTP requests) to the server. We identify HTML tags such as <img>, <script>, <iframe> etc., and window events such as redirection, opening popups etc., as event-types (explained in Section 5.2.3). Therefore, CORP is a 3-way relation defined over the sets *Who*, *What* and *How*, as shown in Equation (1).

$$CORP \subseteq Origin \times ResourcePath \times EventType \tag{5.1}$$

Equation (2) shows an example of a policy which is a subset of the 3-way relation.

$$\begin{gathered} Origin = \{O_1, O_2, O_3\} \\ ResourcePath = \{P_1, P_2, P_3\} \\ EventType = \{Img, Script, Form\} \\ CORP, C_p = \{(O_1, P_1, Img), (O_2, P_2, Form), (O_2, P_3, Script)\} \end{gathered} \tag{5.2}$$

Let us say a website belonging to the origin O_0 sets this policy and a CORP-enabled browser receives it. Then, only the cross origin requests that satisfy the tuples in the policy will be allowed by the browser

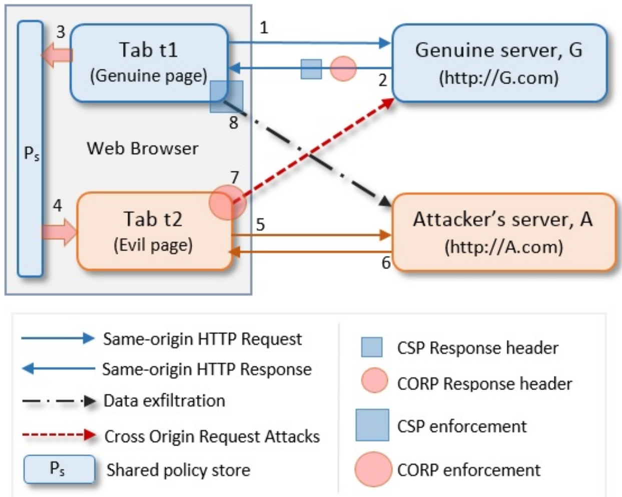

Figure 5.2 *Browser model showing exfiltration & infiltration and how they are mitigated by CSP & CORP*

and rest will be blocked. E.g., A webpage belonging to the origin O_1 will be allowed to request for images only under the path P_1, from a server belonging to the origin O_0 (refer to the first tuple in Equation (2)). Similarly, a webpage belonging to the origin O_1 will not be allowed to submit a form to the server belonging to O_0, since it is not defined in the policy.

5.2.2 Browser Model with CORP

Figure 5.2 shows the model of a browser which supports CORP. It shows the difference between exfiltration and infiltration attacks, thereby explaining how CORP differs from CSP. The figure shows a genuine server *G*, with origin `http://G.com`, an attacker's server *A*, with origin `http://A.com` and a browser with two tabs - *t1* and *t2*. A general browsing scenario, which is also the sufficient condition for a cross origin attack, where a user logs in at *G.com* in *t1* and (unwittingly) opens *A.com* in *t2* is depicted in the model.

5.2.2.1 Setting the Policy

Once a user requests the genuine site *G.com* by typing its URL in the address bar of *t1*, an HTTP request is sent from *t1* to *G*. In response, along with content, CORP is sent via HTTP response headers by G (shown by arrows 1 and 2 in the figure). The tab *t1* receives the policy and sends it to a shared policy store P_s where P_s ensures that CORP is available to every tab or instance (arrows 3 and 4 in the figure) of the browser. Now, when a user unwittingly visits a malicious page from *A* in *t2* (arrows 5 and 6 in the figure), every HTTP request initiated by the page in *t2* to *G* will be scrutinized and restrictions in *CORP* will be enforced (location 7 in the figure). Requests from *t2* to *G* will be allowed only if they comply with the configuration in the policy. As per the guidelines in Section 5.2.3, web administrators

will be able to configure rules in a way that web infiltration attacks will be prevented. It is sufficient to configure CORP on the login page/home page of a website. It is not a per-page policy like CSP and adding CORP on every page only overrides the policy.

5.2.2.2 Deleting the Policy

As users visit multiple websites, their browsers keep accumulating CORP policies and therefore, a mechanism to delete the policies is required. In CSP and HTML5 CORS, policies will be stored in the browser only till the participating websites remain open in browsers. The same mechanism cannot be used in CORP, because if a CORP-enabled website is closed accidentally by a user while being logged in and the policy is destroyed, malicious websites will be able to trigger infiltration attacks. To prevent this, it is important for the policy to be persistent in the browser. At the same time, its life-time in the browser should be under the control of the server. To meet both these objectives we follow the expiry mechanism of HTTP Strict Transport Security (HSTS) policy [29] and mandate the server to send a *max-age* attribute along with CORP directives. This attribute sets the amount of time (in seconds) for which CORP should be active in the browser. For example, a *max-age* value of 2592000 seconds ensures that the policy is active for 30 days, while a max-age of 0 deletes the policy immediately. If a user visits the website before the expiration time, the timer will be reset to the new time configured in *max-age*.

It is important to note that policy's set, get and delete operations are subjected to same origin checks on the browser, to prevent websites overwriting each other's policies. Also, since CORP aims to filter cross origin interactions, adding it to a website does not break the site's existing same origin HTTP transactions.

5.2.2.3 CORP and CSP - How They Differ

CORP and *CSP* together complement *SOP* and help in fixing *exfiltration* and *Infiltration*. *CSP* was designed to enforce restrictions on HTTP traffic leaving a genuine webpage, as shown by location 8 in Figure 5.2. *CORP* was designed to enforce restrictions on HTTP traffic sent by a malicious web page to a genuine server (location 7 in the figure). Also, *CSP* expects origins as directive values as they are sufficient to control exfiltration. *CORP* specifies a 3-way relation defined over the sets event-types, paths and origins. In a nutshell, *CORP* configured on a website *A.com* defines who (i.e., which origins) can probe what (i.e., which resource) on *A.com* and how (i.e., through which event).

5.2.3 Abstract Syntax of CORP

Listing 5.1 shows the abstract syntax of *CORP*.

```
policy ::= rule *...
rule ::= pattern permission
pattern ::=  origin-list eventType-list path-list
```

```
permission ::= ALLOW | DENY
origin-list ::= origin +... | ANY
eventType-list ::= eventType +... | ANY
path-list ::= path +... | ANY
origin ::= RFC 6454
eventType ::= img | media | style
                        | font | script | iframe
                        | form-action | xhr | hyperlink
                        | window | object
path ::= RFC 2396
```

Listing 5.1 Abstract syntax of CORP

For path, an additional pattern "resourcePath/*" is allowed to simplify the configuration of CORP. The wild card '*' in the pattern provides a way to refer to any resource under a specific resource path. E.g., Access to all paths under "admin" directory can be controlled using the pattern "/admin/*".

5.2.3.1 Order of Precedence for CORP rules

CORP rules are processed from top to bottom, till the default rule is reached. When a cross origin request is made by a website against a CORP-enabled site, the request is scrutinized by the first rule in the policy. If a match is found, the first rule is executed and rest of the rules are not evaluated. Else, the request is scrutinized by the next rule and the process continues till the last rule.

The last (default) rule is set to "* * * Allow", which means "Allow everything". If a server sends an empty policy, it is the same as not configuring CORP at all. In such cases, the default rule is evaluated and all cross origin requests are allowed. This approach ensures that CORP does not break existing cross origin interactions on a website. Also, it enables web administrators to incrementally build stricter rules and tighten the security of their servers. We demonstrate a few example policies in the following discussion.

5.2.3.2 Example Policies

- **Deny all:** A banking site may want to completely block all cross origin requests to its site. It may achieve this by setting the simple policy shown in Listing 5.2.

```
    *    *    *    DENY
```

Listing 5.2 Block all cross origin requests

- **Selective content:** A photo sharing site may want to respond only to authenticated cross origin requests involving scripts, images (from any site) and block any other authenticated cross origin request. It may set the policy shown in Listing 5.3.

```
*    img      /img          ALLOW
*    script   /scripts      ALLOW
*    *        *             DENY
```

Listing 5.3 Allow access to selective content

- **Partners only:** An e-commerce website might expose state-changing web services and expects only its partner sites, say *P1.com, P2.com*, to do a form submission to its services. It can set the policy shown in Listing 5.4.

```
{P1.com,  P2.com}   form   {/update, /delete}   ALLOW
*                   *      *                    DENY
```

Listing 5.4 Allow selective access to selective origins

5.2.4 Security Guarantees Provided by CORP

CORP helps website administrators use browser's capabilities in adding additional security to their sites. The following are the security guarantees provided by CORP:

5.2.4.1 Fine Grained Access Control

Through CORP, websites can decide *who* (i.e., which set of origins) can trigger cross origin requests to *what* resources on their sites and more importantly *how* (i.e., through which mechanism). Having such a fine grained access control helps web administrators selectively allow/deny cross origin requests, thereby enhancing the security of their site.

5.2.4.2 Combating CSRF

By binding various event types e.g., <img> to paths serving their corresponding resources e.g., `http://A.com/images/` via CORP, the semantics of request initiators is maintained. The implication of this binding is that active HTML elements can no longer be used as vectors for cross origin attacks. Also, by whitelisting sensitive paths and defining which origins can request them, automated requests triggered by scripts through various techniques can be blocked. If CORP is properly configured, CSRF attacks can be eliminated completely.

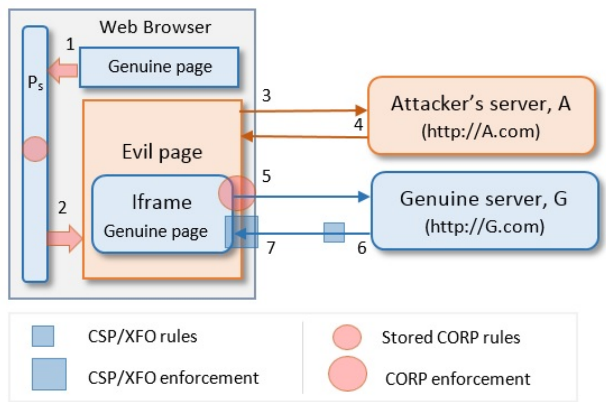

Figure 5.3 *Browser model showing the enforcement of Clickjacking defense in CSP/XFO and CORP*

5.2.4.3 Early Enforcement of Clickjacking Defense

As discussed in Section 3.2.2, XFO and CSP-UI-Security directives are two important proposals to mitigate clickjacking. Figure 5.3 explains how enforcement of clickjacking defense takes place in XFO/CSP and CORP. The workflow in the figure is similar to the workflow depicted Figure 5.2. As explained in Section 5.2.2, consider the normal browsing scenario where a user (victim) opens a genuine site *G.com* in tab *t1* and unwittingly opens an attacker's site *A.com* in tab *t2*. In this case, the evil page (belonging to *A.com*) embeds an iframe, which loads a page belonging to *G.com*, with an intention to hijack the victim's click. The iframe makes an HTTP request to the genuine server (*G*) and gets the HTML response along with HTTP headers. If the page is configured with either X-Frame-Options header or CSP clickjacking directive, browsers enforce XFO/CSP and do not render the HTML response (location 7 in the figure), thereby preventing clickjacking. However, since the request triggered by the iframe has already reached the server *G*, CSRF attack has already taken place. Also, due to this delayed enforcement, Clickjacking bypasses such as *Double Clickjacking* [31], *Nested Clickjacking* [43] and *Login detection using XFO* [36] arise. CORP mitigates these problems by ensuring that clickjacking enforcement take place even before a cross origin request is triggered. If the genuine site *G.com* in *t1* is configured with CORP, the policy will be stored in a shared policy store P_s, which is accessible to all instances of the browser. As soon as the iframe in the evil page (loaded in *t2*) triggers an HTTP request to *G.com*, CORP's enforcement triggers (location 5 in the figure), thereby blocking the request altogether. Since the request is blocked at the browser itself, CSRF is mitigated. The same logic applies to other bypasses for clickjacking. Hence, CORP is the right way to eliminate clickjacking completely. Listing 5.5 shows CORP configuration to mitigate clickjacking.

```
*        iframe        *        DENY
```

Listing 5.5 Defeating clickjacking with CORP

5.2.4.4 Controlling Social Engineering Attacks

Attackers attempt several social engineering tricks on end users by leveraging popups [60], iframes [51, 42] and hyperlinks. Spam emails having hyperlinks that point to sensitive web pages (e.g., delete.php) continue to be a common menace. Today, there are no standard defenses against these attacks as there is no mechanism for a server to instruct *how* a cross origin request should originate to itself. By configuring CORP, website administrators can block requests initiated by frames, popup windows, hyperlinks for all or specific paths. This ensures that end users do not succumb to most of the common social engineering tricks.

```
*       href                          /non-sensitive      ALLOW
*       {href, window, iframe}        *                   DENY
```

Listing 5.6 Controlling social engineering attacks

Listing 5.6 shows a sample CORP configuration, which blocks some of the vectors used in social engineering attacks. The configuration allows hyperlinks to navigate only to non-sensitive pages, denies requests which open popups or navigate to any location via *window* object and denies framing.

5.2.4.5 Defeating Cross-Site Timing Attacks

The vectors for cross-site timing attacks are same as that of CSRF, as discussed in Section 2.2.6. They use the *onload* and *onerror* event handlers of HTML elements for measuring the time taken for a resource to load under various conditions, thereby leaking sensitive information such as login status. One of the suggested defenses is to disable these event handlers for cross origin requests. This not only stops the attack but also breaks genuine scenarios. Website administrators who are cautious about cross-site timing attacks can configure CORP such that cross origin requests are allowed only to public resources i.e., resources which do not need authentication. CORP blocks cross origin requests to authenticated resources such as private pictures and URLs before they leave the browser, thereby defeating cross-site timing attacks. Listing 5.7 shows a sample CORP configuration for the same.

```
*       img     /public/images/*        ALLOW
*       *       *                       DENY
```

Listing 5.7 Defeating cross-site timing with CORP

5.2.4.6 Mitigating Application-level DDoS Attacks

On March 27, 2015, Github has witnessed a massive DDoS (distributed denial of service) attack, the largest in Github's history till date [2, 1]. As per the analysis of security researchers [3], the attack is an application-level DDoS attack caused by continuous cross origin JavaScript calls to Github from

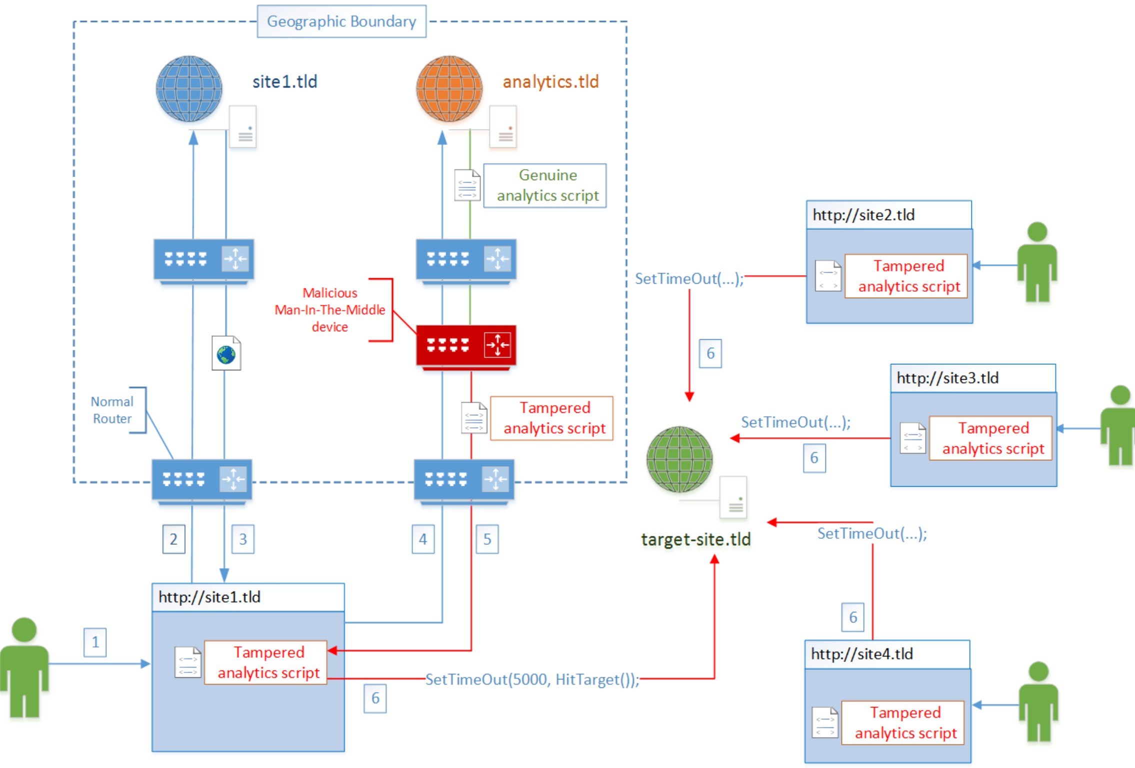

Figure 5.4 *Understanding an application-level DDoS attack*

several thousands of users. This variant of cross origin attacks were neither reported nor analyzed in any of the earlier studies (see Section 3). Also, we did not consider this variant when we defined the scope of Web infiltration attacks (see Section 5.1). Intuitively, it appeared that CORP can be enhanced to mitigate such attacks and we were interested in exploring this possibility further. In this attempt, we have generalized the attack on Github and came up with the flow diagram shown in Figure 5.4. The steps involved in the attack, which correspond to the numbers in the figure, are as follows:

1 A genuine user browses a website `http://site1.tld` in a browser (*tld* stands for Top Level Domain e.g., .com, .org).

2 An HTTP request is made to the server *site1.tld*

3 The server *site1.tld* responds with a web page, which is loaded in the user's browser.

4 The loaded web page references a third party analytics JavaScript file, which is hosted under `http://analytics.tld`.

5 While the server *analytics.tld* responds with a genuine analytics file, a malicious man-in-the-middle device intercepts the response and tampers the script with malicious content. The tampered script gets embedded in the web page in the user's browser. As explained in Section 2.2.1, the script inherits the origin *http://site1.tld*.

6 The tampered script continuously triggers cross origin HTTP requests to the server *target-site.tld* for every few seconds using JavaScript's *SetTimeOut* function. Therefore, the site `http://site1.tld` launches a Denial of Service (DoS) attack on the server *target-site.tld*. If the analytics JavaScript file is embeded by multiple sites (e.g., `http://site2.tld`, `http://site3.tld` etc.), then the users browsing those sites will unwittingly launch the DoS attack on *target-site.tld*. This makes the attack an application-level Distributed DoS (DDoS) attack.

Irrespective of the motive behind the attack on Github, we were interested in finding if CORP can help a server in mitigating against such application-level DDoS attacks. To explore this, we have simulated the DDoS attack in a lab environment. The setup has an analytics site which serves JavaScript content, a vector site which references the analytics JavaScript and a victim site which receives DDoS requests. For the sake of simplicity, instead of tampering the analytics script, we have injected the malicious script in the analytics script itself. We have enabled the victim site to monitor the incoming requests and maintain a threshold (say 10 requests/second). Once this threshold is reached, which is an indicative of a DDoS attack, the victim site responds with CORP that denies all cross origin requests to the site. Thus, by setting CORP dynamically, a server can protect itself from application-level DDoS attacks. Note that DDoS attacks can be launched even from outside a browser and CORP will not be able to prevent them. CORP only addresses the application-level DDoS attacks arising due to cross origin web interactions.

5.3 Validating the Soundness of CORP

Analyzing the security of the web platform is a daunting task, since it based on several complicated web specifications which are often written and implemented manually. Over the years, researchers have used formal verification to analyze the security of network protocols. Akhawe et al. [5] built a formal model of web security based on an abstraction of the web platform. They used Alloy [34], a finite state model finder, to build their formal model and showed that their model is useful in identifying well-known as well as new vulnerabilities in web specifications. Following this work, several researchers used Alloy to formalize security aspects of the web. They used it to verify the soundness of both existing security aspects as well as newer proposals. Inspired by the wide-spread application of Alloy in verifying web specifications and architectures, we have used it to formalize and verify the soundness of CORP.

5.3.1 A Brief Introduction to Alloy

Alloy is a model specification language based on first-order logic, used for creating relational models. Using Alloy, developers can write and test the specifications of their software designs by creating formal models. Alloy Analyzer is a tool used for analyzing and exploring the models created using the Alloy language. It takes the constraints written in Alloy language and reduces them to satisfiability (SAT) problem. It then uses built-in SAT solvers to find models satisfying or violating the constraints. To ensure that the model-finding problem is decidable, the Alloy Analyzer performs model-finding over a

restricted *scope*. The Alloy Analyzer can be used for *simulation* as well as *counterexample generation*. In *simulation*, Alloy tries to find instances of the model that satisfies all the constraints specified in the Alloy logic. In *counterexample generation*, Alloy tries to find instances of the model that violates an assertion. Together, Alloy's simulation and counterexample generation tests assist in verifying the soundness of a model.

5.3.1.1 Alloy specifications

All structures in Alloy models are built from atoms and relations. An *atom* is a primitive entity that is indivisible, immutable and uninterpreted. A *signature* (type) describes an entitiy that is reasoned about. It introduces a set of atoms. A signature can be created as an extension of another signature, thereby forming subsignatures (subtypes or subsets). A signature that does not extend another signature is called a *top-level* signature. For example, the declaration *sig Member* in Listing 5.8 introduces a new set named *Member*. The signatures *Faculty* and *Student* extend the signature *Member*. They are the subsets of the set *Member* and are mutually disjoint. When a signature is declared as *abstract*, it will have no atoms except those belonging to its extensions.

```
1 abstract sig Member{}
2 sig Faculty extends Member{}
3 sig Student extends Member{}
```

Listing 5.8 Alloy Signatures

Relations are declared as fields of Alloy signatures, and multiplicities help in constraining the size of signatures. A multiplicity keyword prefixing a signature constraints the number of elements in the signature's set. There are four multiplicities in Alloy. The multiplicity *set* says that the signature can contain any number of elements; *some* says that the set contains at least one element; *lone* says that the set contains at most one element; and *one* says that the set contains exactly one element. In an Alloy declaration, the default multiplicity constraint is *one*.

```
1 sig Class {
2     taughtBy: Faculty,
3     attendedBy: some Student
4 }
```

Listing 5.9 Alloy Relations and Multiplicities

The declaration in Listing 5.9 introduces a relation *taughtBy* whose domain is *Class* and whose image is *Faculty*. The relation says that each class is taught by exactly one faculty. The *attendedBy* relation says that each class is attended by at least one student.

Alloy has various forms of quantified constraints as shown in Listing 5.10. In the listing, *F* is a formula, a constraint, which contains the variable *x*, *e* is an expression bounding *x*, and the keywords preceding *x* are quantifiers.

```
1  all x:e | F      // F holds for every x in e.
2  some x:e | F     // F holds for at least one x in e.
3  no x:e | F       // F holds for no x in e.
4  lone x:e | F     // F holds for at most one x in e.
5  one x:e | F      // F holds for exactly one x in e.
```

Listing 5.10 Alloy Quantification

Alloy has several *set*, *logical* and *relational* operators. The *union* (+), *intersection* (&), *difference* (-), *subset* (in) and *equality* (=) operators are the standard set operators. The *not* (negation), *and* (conjunction), *or* (disjunction), *implies* (implication), *iff* (bi-implication) are the standard logical operators. They can also be written in shorter form (!, &&, ||, =>, <=>). While there are several relational operators, the *dot join* (.) operator is used more frequently. It is used to compose relations. E.g., if *p* and *q* are two relations, the join *p.q* is the relation obtained by taking every combination of a tuple from *p* and a tuple from *q* and adding their join if it exists. Listing 5.11 shows an example of composing relations using the *join* (.) operator.

```
1  p = {(a,b)}, q = {(b,c)}, r = {(a,c)}
2  p.q = {(a,c)} // The matching element (b) from p and q is omitted
3  p.r = {} // The last element of p and first element of r do not match
4  ~p.r = {(b,a)}.{(a,c)} = {(b.c)} // ~p is transpose of p
```

Listing 5.11 Composing relations using Alloy's dot join operator

A *fact* in Alloy is a set of constraints that are assumed always to hold. A *predicate* is a named constraint, with zero or more declaration parameters. It evaluates to true if the inputs satisfy all of the constraints in its body, and evalutes to false otherwise. An *assertion* is a constraint that is intended to be followed from the facts of the model. If an assertion does not follow from the facts, the Alloy analyzer produces a *counterexample*, thereby proving that the design of the model has a flaw. To analyze a model, the *run* command is used. It tells the Alloy analyzer to search for an instance of a predicate. The *check* command tells the analyzer to search for a counterexample of an assertion. The *scope* command bounds the size of instances or counterexamples that will be considered to make model finding feasible. If scope is omitted, the analyzer will use the default scope in which each top-level signature is limited to three elements.

5.3.1.2 Sample model

Appendix A explains in detail a sample Alloy model for a basic academic time table.

5.3.2 Design considerations of CORP Alloy model

This section explains in detail the formal models we created to validate the soundness of CORP. After a thorough analysis of the Alloy browser security model developed by Akhawe et al. [5], we have built a simpler model, which captures only the details required for web infiltration attacks. Below are the design considerations of our formal model:

5.3.2.1 Simpler Abstraction

Akhawe's Alloy model captures an abstraction of the web platform and could have been used as a baseline for our model. However, we have observed that it gets complicated when it is extended with DOM (Document Object Model) elements and the HTTP transactions initiated by them. Since the problem we are solving is related to cross origin intereactions, instead of extending Akhawe's model, we have borrowed the basic signatures and captured only the relevant details, thereby building a simpler abstraction of the web platform.

5.3.2.2 Non-empty browser context

In a general browsing scenario, when a user opens a new browser window, there is no initial context (we refer to this as "Empty context"). In such a context, the user initiates the first HTTP request by typing a URL in the address bar. Once the request gets a successful HTTP response, a document is constructed, which is the state of the browser. Subsequent HTTP requests occur in the context of this document, which we refer to as "Non-empty context". Our model assumes that a non-empty context of a victim website is available and a user has logged into the site. It does not model the HTTP transactions that built this context. Similarly, it assumes that a non-empty context of an evil website is available. In a typical web infiltration attack, the following steps take place:

Step 1 A user logs into a genuine website in one tab of a browser

Step 2 The user (unintentionally) opens an evil website in another tab

Step 3 The evil site generates malicious cross origin HTTP requests to the genuine site's server

By making the "Non-empty context" assumptions, the steps 1 and 2 above are eliminated from the model. These assumptions will not impact the depiction of cross origin interactions. Moreover, they will make the model simpler to analyze. The focus of our Alloy models will be in depicting the malicious cross origin requests in Step 3 initially, and later mitigating them with CORP.

5.3.2.3 Single browser instance

Since we are assuming non-empty contexts, we have restricted our model to contain only a single instance of a browser. As explained in the previous section, the intent of the model is to depict a

web infiltration attack, a cross origin HTTP request from an evil site to a genuine site's server, and to mitigate it. So it is sufficient to have a single browser window/tab (synonymous to the signature *Browser* in our model), which loads a document from an evil server. The contents of the document (HTML elements/JavaScript) initiate cascading HTTP requests to a genuine server.

For easier analysis and understanding, we have created two Alloy models: Pre-CORP and Post-CORP. The Pre-CORP model captures the current state of the web platform, where unrestricted cross origin requests are possible. The Post-CORP model is an extension of the Pre-CORP model wherein we add additional signatures, facts and predicates to describe CORP and the constraints it enforces on cross origin requests.

5.3.3 Modelling cross-origin requests in the web platform (Pre-CORP.als)

Figure 5.5 shows the meta model of Pre-CORP model. The core components of this model are: *HTTPTransaction*, *Origin* and *HTTPEventInitiator*. The relations and constraints around these components are explained in this section.

5.3.3.1 HTTP Transactions

The code in listing 5.12 shows our abstraction of an HTTP transaction. A *HTTPTransaction* is a type which consists of exactly one *HTTPRequest* and exactly one *HTTPResponse*. An HTTP request is initiated from a *Browser* and sent to a *Server*, while an HTTP response is initiated from a server and sent to a browser.

```
1  abstract sig HTTPTransaction{
2      req:  HTTPRequest,
3      resp: HTTPResponse
4  }
5  sig HTTPRequest {
6      from:  Browser,
7      to:  Server,
8      host: Origin
9  }
10 sig HTTPResponse{
11     from:  Server,
12     to:  Browser,
13     host: Origin
14 }
```

Listing 5.12 Basic HTTP Transactions in the Pre-CORP model

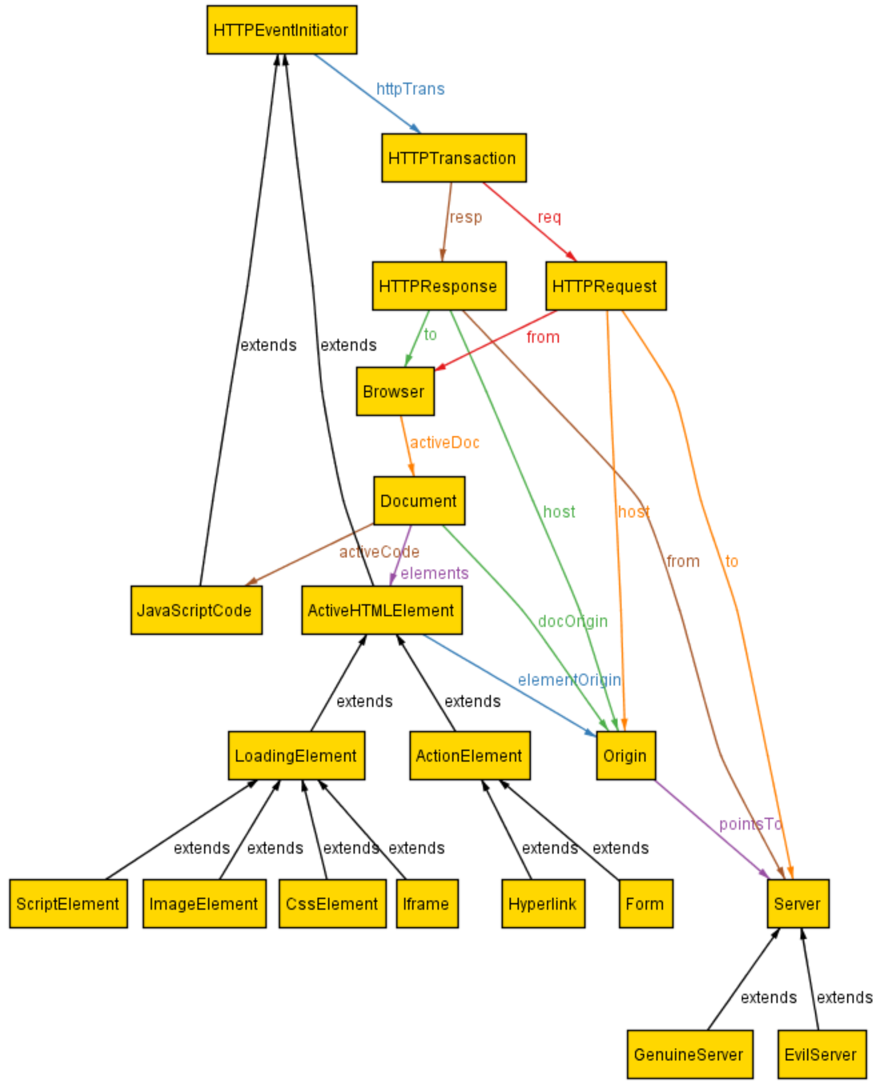

Figure 5.5 Meta model of the Pre-CORP Alloy model

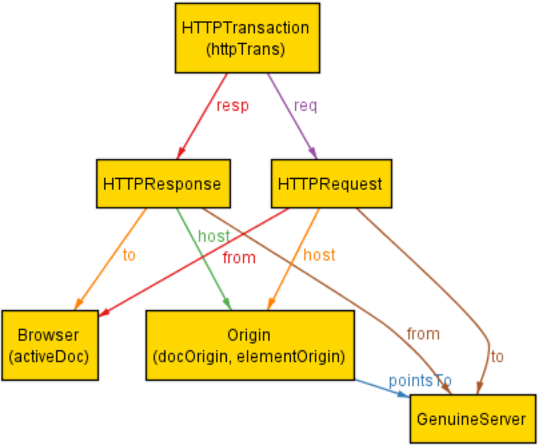

Figure 5.6 An instance of an HTTP transaction in the Pre-CORP model

Requests and responses are tied to their respective end points (i.e., browsers and servers) via the *Origin* signature. Figure 5.6 shows Alloy's instance of an HTTP transaction, projected over multiple signatures for simplicity.

5.3.3.2 Origin

As explained in Section 2.1.1, *Origin* is represented by the 3-tuple *(scheme, host, port)*. The code in listing 5.13 shows the basic web model, which depicts how an origin is related to a browser and a server. A browser consists of exactly one active document i.e., the document with which a user interacts at any point of time. Each document has exactly one origin, whose *host* resolves to the server from which the document has loaded. Figure 5.7 depicts this relation. It is produced by Alloy when the predicate *showBasicModel* shown in Listing 5.14 is run. The predicate asks Alloy to generate an instance of the model without any HTTP transactions.

```
1  sig Origin{
2      pointsTo:  Server
3  }
4  one sig Browser{
5      activeDoc: Document
6  }
7  abstract sig Server {}
8  sig Server1, Server2 extends Server{}
9  sig Document{
10     docOrigin: Origin,
```

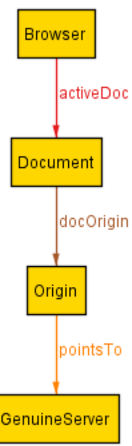

Figure 5.7 An instance of the predicate *showBasicModel*

```
11        elements: set ActiveHTMLElement,
12        activeCode: set JavaScriptCode
13    }
```

Listing 5.13 Relation between browser, origin and server

```
1 pred showBasicModel{
2     no HTTPTransaction
3 }
4 run showBasicModel
```

Listing 5.14 Predicate: showBasicModel

5.3.3.3 HTTP Event Initiators

The code in listing 5.15 shows our classification of components of a web page (document). A *HTTPEventInitiator* is any component that can trigger an HTTP transaction, and is associated with an origin. *JavaScriptCode* and *ActiveHTMLElement* are modelled as HTTP event initiators. DOM elements in a document are classified based on their capability to trigger HTTP requests. Those that cannot trigger HTTP calls are called as passive HTML elements (e.g., Div, Span, Textbox etc.), while those that can trigger HTTP calls are called as active HTML elements. For keeping the model concise, only active HTML elements have been included in our specification. *ActiveHTMLElement*s are further classified as *LoadingElement*s - elements that automatically trigger HTTP requests as soon as they get added to the DOM tree (e.g., img, script, iframe etc.) and *ActionElement*s - elements that require an action by humans to trigger HTTP requests (e.g., hyperlinks, forms).

```
1 abstract sig HTTPEventInitiator{
2     httpTrans: lone HTTPTransaction,
3     initiatorOrigin: Origin
4 }
5 sig JavaScriptCode extends HTTPEventInitiator{}
6 abstract sig ActiveHTMLElement extends HTTPEventInitiator {}
7 abstract sig LoadingElement, ActionElement extends ActiveHTMLElement {}
8 sig ScriptElement, ImageElement, CssElement, Iframe extends
     LoadingElement {}
9 sig Hyperlink, Form extends ActionElement{}
```

Listing 5.15 HTTP Event Initiators

The next sections show facts that are built into the Pre-CORP model. These are the constraints that are assumed to be true, to ensure that the Pre-CORP model conforms to the fundamentals guidelines of the W3C specifications [71].

5.3.3.4 Fact: EventInitiatorsInheritParentOrigin

The fact *EventInitiatorsInheritParentOrigin* in Listing 5.16, explains how *Origin* is inherited by *HTTPEventInitiator*.

```
1 fact EventInitiatorsInheritParentOrigin{
2     all elem: ActiveHTMLElement | elem.initiatorOrigin = elem.~elements.
     docOrigin
3     all js: JavaScriptCode | js.initiatorOrigin = js.~activeCode.
     docOrigin
4 }
```

Listing 5.16 Fact: OriginInheritence

It expresses the following constraints:

Line 2: The origin (*elem.initiatorOrigin*) of every active HTML element must be the same as the origin (*elem. elements.docOrigin*) of the document that contains the element.

Line 3: The origin (*js.initiatorOrigin*) of every JavaScript code in a document must be the same as the origin (*elem. elements.docOrigin*) of the document that contains the code.

As shown in Listing 5.15, *ActiveHTMLElement* and *JavaScriptCode* form the set *HTTPEventInitiator* i.e., they are capable of triggering HTTP transactions. Since these HTTP event initiators inherit their

parent document's origin, any HTTP transaction initiated by them will be tagged with their parent document's origin. An understanding of this constraint is crucial in understanding XSS attacks outlined in Section 2.2.1 and mashup security problem outlined in Section 4.1.1.

5.3.3.5 Fact: TransactionRules

The fact *TransactionRules*, shown in Listing 5.17, lists the set of constraints that ensure the sanity of HTTP transactions in the model.

```
1  fact TransactionRules{
2      all t:HTTPTransaction, b:Browser, s:Server| {
3          t.req.host = t.resp.host
4          t.req.host = t.req.to.~pointsTo
5          s=t.req.to => s = t.resp.from
6          b = t.req.from => b = t.resp.to
7          t in HTTPEventInitiator.httpTrans
8      }
9      all disj t1,t2: HTTPTransaction | {
10         no (t1.req & t2.req)
11         no (t1.resp & t2.resp)
12     }
13 }
```

Listing 5.17 Fact: TransactionRules

The fact says that for all instances of *HTTPTransaction*, *Browser* and *Server*, the following rules should hold:

Line 3: The request and the response of a transaction must belong to the same origin

Line 4: An HTTP request's origin must be the same as the origin whose *host* resolves to the request's destination (i.e., a server).

Line 5: If a request is sent to a server, then its corresponding response should be received from the same server

Line 6: If a request is sent from a browser, then its corresponding response should be received by the same browser

Line 7: Every HTTP transaction must be triggered by an HTTP event initiator (i.e., either an active HTML element or JavaScript code).

Line 9-12 : Requests and responses belonging to any two transactions must be disjoint.

5.3.3.6 Fact: Disjointness

The fact *Disjointness*, shown in Listing 5.18, lists the constraints which ensure that multiple instances of each signature do not interfere with each other.

```
1  fact Disjointness {
2      all disj b1,b2: Browser | {
3          no (b1.activeDoc & b2.activeDoc)
4          no (b1.activeDoc.docOrigin & b2.activeDoc.docOrigin)
5      }
6      all disj o1,o2:Origin | no (o1.pointsTo & o2.pointsTo)
7      all disj e1, e2: ActiveHTMLElement | no (e1.httpTrans & e2.httpTrans
       )
8      no GenuineServer.resourcePath &  EvilServer.resourcePath
9  }
```

Listing 5.18 Alloy Fact - Disjointness

5.3.3.7 Pred: SameOriginTransaction

The predicate *sameOriginTransaction* shown in Listing 5.19 asks Alloy to produce an instance of the model where there is only one server and at least one HTTP transaction. Figure 5.8 shows an instance of the model generated by Alloy, when this predicate is run. The figure shows a browser having a document, whose origin points to *Server2*. The document has a *ScriptElement* that initiates an HTTP transaction. An HTTP request is made from the browser to the server, and an HTTP response is returned from the server to the browser. Since the origin of the HTTP event initiator (*ScriptElement*) and the origin of the request's destination (*GenuineServer)* is the same, the transaction is said to be a same origin transaction.

```
1  pred sameOriginTransaction{
2      some HTTPTransaction
3      one Server
4  }
5  run sameOriginTransaction
```

Listing 5.19 Pred: sameOriginTransaction

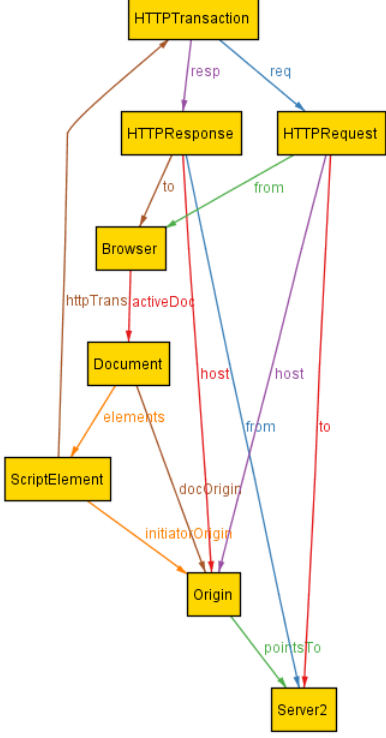

Figure 5.8 An instance of the predicate *sameOriginTransaction*

5.3.3.8 Pred: CrossOriginTransaction

The predicate *crossOriginTransaction* shown in Listing 5.20 asks Alloy to produce an instance of the model where the following constraints hold good for at least one HTTP transaction:

Line 3: The request must originate from Server1

Line 4: The request must be made to Server2

```
1 pred crossOriginTransaction{
2     some t:HTTPTransaction|{
3         t.req.from.activeDoc.docOrigin.pointsTo=Server1
4         t.req.to=Server2
5     }
6 }
7 run crossOriginTransaction
```

Listing 5.20 Pred: crossOriginTransaction

Figure 5.9 shows an instance of the model generated by Alloy, when this predicate is run. The figure shows a browser having a document, whose origin *Origin1* points to *Server1*. The document has a

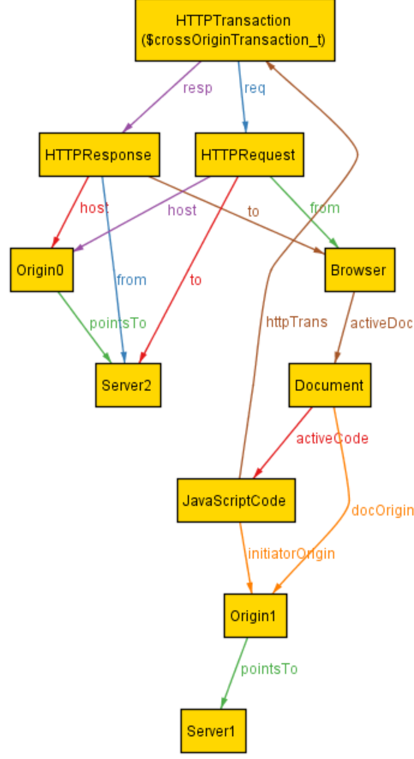

Figure 5.9 An instance of the predicate *crossOriginTransaction*

JavaScriptCode that initiates an HTTP transaction. An HTTP request is made from the browser to *Server2*, whose origin is *Origin0*. Since the origin of the HTTP event initiator (*JavaScriptCode*) and the origin of the request's destination (*Server2)* are different, the transaction is said to be a cross origin transaction.

5.3.4 Modelling restrictions introduced in CORP (Post-CORP.als)

We have extended our Pre-CORP model, which depicts unrestricted cross origin HTTP requests, with new signatures and constraints. The resultant model shows how a server can use CORP in imposing restrictions on cross origin transactions made to it, thereby mitigating web infiltration attacks.

Figure 5.10 shows the meta model of Post-CORP model.

5.3.4.1 Key idea of CORP

As explained in Section 5.2.1, a server should have fine-grained control on *Who* can access *What* resource on the server and *How*. *Who* refers to the set of origins that can request a resource belonging to a server, *What* refers to the set of paths that map to resources on the server, *How* refers to the set of event-types that initiate network events (HTTP requests) to the server. To capture these constraints, we have extended the Pre-CORP model with a few signatures and facts, as outlined in the next sections.

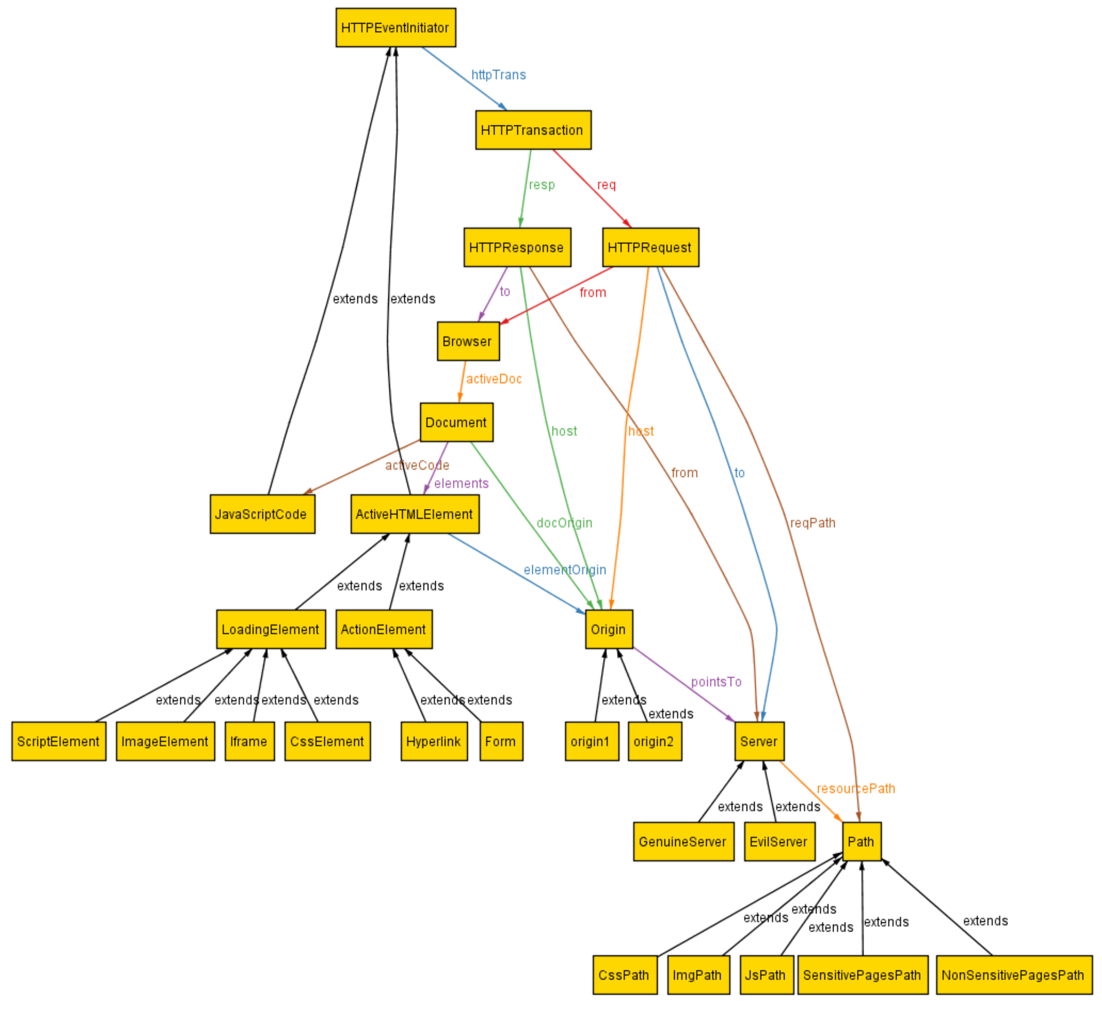

Figure 5.10 Meta model of the Post-CORP Alloy model

5.3.4.2 Resource Paths

Every resource on the web is identified by a unique path. As a good engineering practice, web administrators often organize different types of resources (e.g., images, scripts etc) under different directories (e.g., `http://A.com/images`, `http://A.com/js` etc.) on the server hosting the resources. Based on this observation, we have defined the signature *Path* in the Post-CORP model. It has various subtypes as shown in listing 5.21. The subtype *NonSensitivePagesPath* refers to pages that do not contain any sensitive content, while the subtype *SensitivePagesPath* refers to pages that contain sensitive content and need utmost protection.

```
1  abstract sig Path{}
2  one sig ImgPath, JsPath, CssPath, NonSensitivePagesPath,
       SensitivePagesPath extends Path{}
3  abstract sig Server {
4      resourcePath: set Path
5  }
```

Listing 5.21 Resource Paths

5.3.4.3 Pred: maliciousXOriginTransaction

The predicate *maliciousXOriginTransaction* is an enhancement to the predicate *crossOriginTransaction* defined in Section 5.3.3.8. The type *Server* has been extended with the subtypes *EvilServer* and *GenuineServer*. Listing 5.22 shows the contraints for this predicate.

```
1  sig EvilServer, GenuineServer extends Server{}
2  pred maliciousXOriginTransaction{
3      some t:HTTPTransaction|{
4          t.req.from.activeDoc.docOrigin.pointsTo=EvilServer
5          t.req.to=GenuineServer
6          t.req.reqPath = SensitivePagesPath
7      }
8  }
9  run maliciousXOriginTransaction  for 3 but exactly 1 HTTPTransaction
       expect 1
```

Listing 5.22 Predicate: maliciousXOriginTransaction

Figure 5.11 shows an instance of the model generated by Alloy when this predicate is run. The figure shows a browser having a document, whose origin *Origin2* points to *EvilServer*. The document has

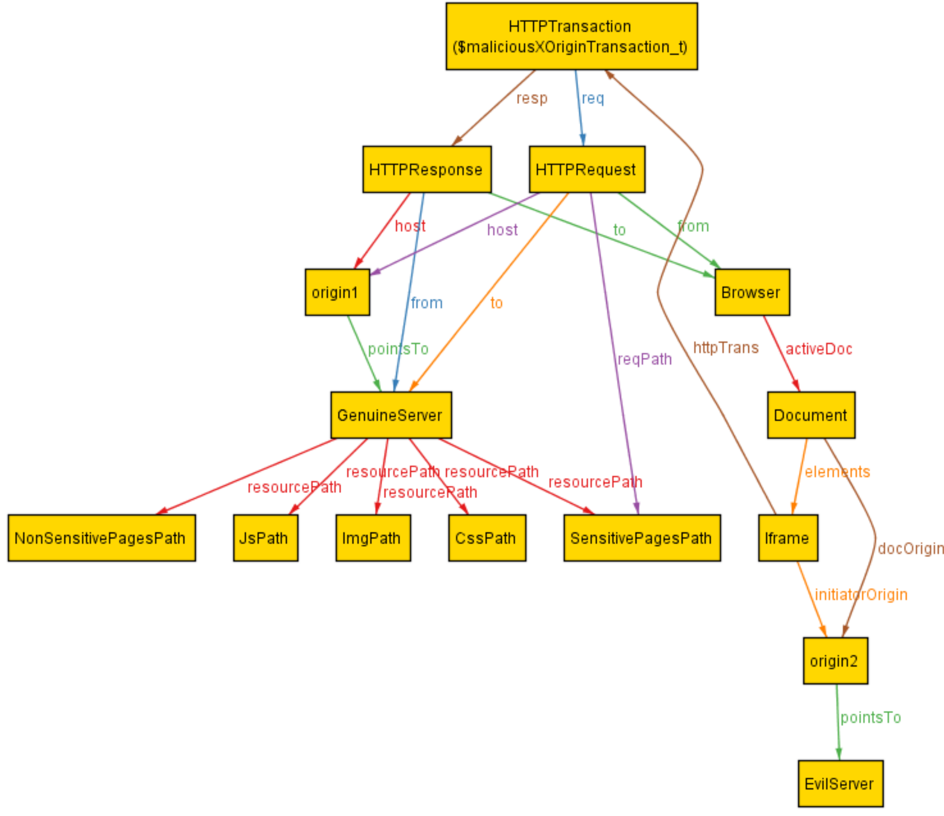

Figure 5.11 An instance of the predicate *maliciousXOriginTransaction*

an *Iframe* that initiates an HTTP transaction. An HTTP request is made from the browser to *GenuineServer*, whose origin is *Origin1*. Since the origin of the HTTP event initiator (*Iframe*) and the origin of the request's destination (*GenuineServer)* are different, the transaction is said to be a cross origin transaction. Since the request is initiated by an evil server and is sent to a sensitive page on a genuine server, the transaction is said to be a malicious cross origin transaction.

5.3.4.4 Pred: corpCompliantTransaction

The predicate *corpCompliantTransaction* shown in Listing 5.23 takes three arguments *o, ev, pt* which are of the type *Origin*, *HTTPEventInitiator* and *Path* respectively. It asks Alloy to produce an instance of the model where the following constraints hold good for at least one HTTP transaction:

Line 3: The request must originate from EvilServer

Line 4: The request must be made to GenuineServer

Line 5: The origin of the HTTP event initiator must be the same as the predicate's argument *o*.

Line 6: The HTTP event initiator that triggers an HTTP transaction must be the same as the predicate's argument *ev*

Line 7: The path to which the HTTP request is made must be the same as the predicate's argument *pt*

```
1  pred corpCompliantTransaction[o:Origin, ev: HTTPEventInitiator, pt: Path
      ] {
2      some t:HTTPTransaction {
3          t.req.from.activeDoc.docOrigin.pointsTo=EvilServer
4          t.req.to=GenuineServer
5          t.req.from.activeDoc.docOrigin = o
6          httpTrans.t= ev
7          t.req.reqPath=  pt
8      }
9  }
10 pred restrictImagesWithCorp{
11     corpCompliantTransaction[origin2, ImageElement, ImgPath]
12 }
13 pred restrictScriptsWithCorp{
14     corpCompliantTransaction[origin2, ScriptElement, JsPath]
15 }
16 pred restrictJsCodeWithCorp{
17     corpCompliantTransaction[origin2, JavaScriptCode,
      NonSensitivePagesPath]
18 }
19 run restrictImagesWithCorp  for 3 but exactly 1 HTTPTransaction expect 1
20 run restrictScriptsWithCorp for 3 but exactly 1 HTTPTransaction expect 1
21 run restrictJsCodeWithCorp for 3 but exactly 1 HTTPTransaction expect 1
```

Listing 5.23 Predicate: corpCompliantTransaction

The listing 5.23 also shows three more predicates-*restrictImagesWithCorp*, *restrictScriptsWithCorp* and *restrictJsCodeWithCorp*. Each of them in-turn invoke the predicate *corpCompliantTransaction* with various arguments. For example, the predicate *restrictJsCodeWithCorp* asks Alloy to produce an instance where an HTTP request is made to a *GenuineServer* by a *JavaScriptCode* from a document having an origin *origin2*. It also mandates the request to be made only to the path *NonSensitivePagesPath* on the *GenuineServer*. Figure 5.12 shows an instance of the model when this predicate is run.

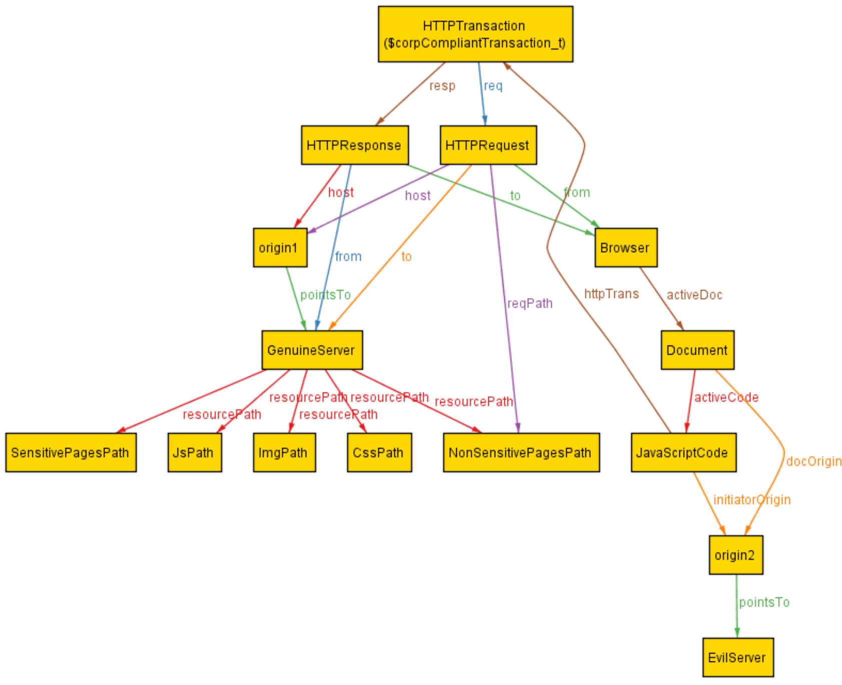

Figure 5.12 An instance of the predicate *restrictJsCodeWithCorp*

5.3.4.5 Assert: showMaliciousTransactionWithJsCode

The predicate *restrictJsCodeWithCorp* shows an instance of the model where a cross origin HTTP transaction initiated by a JavaScript code can be restricted through CORP to non-sensitive pages only. While the instances produced by Alloy show that the predicate is consistent, it is equally important to assert the negation. i.e., Can there exist a cross origin HTTP transaction triggered by JavaScript to the genuine server, where the predicate *corpCompliantTransaction* is violated?

```
1  assert showMaliciousTransactionWithJsCode {
2      no t:HTTPTransaction |{
3          corpCompliantTransaction[origin2, JavaScriptCode,
       NonSensitivePagesPath]
4          t.~httpTrans.initiatorOrigin = origin2
5          t.~httpTrans=JavaScriptCode
6          t.req.reqPath! = NonSensitivePagesPath
7      }
8  }
9  check showMaliciousTransactionWithJsCode for 20
```

Listing 5.24 Assert: showMaliciousTransactionWithJsCode

```
Executing "Check showMaliciousTransactionWithJsCode for 20"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   128967 vars. 5378 primary vars. 366729 clauses. 226ms.
   No counterexample found. Assertion may be valid. 183ms.
```

Figure 5.13 Checking the post-CORP assertion *showMaliciousTransactionWithJsCode*

The assertion *showMaliciousTransactionWithJsCode* shown in Listing 5.24 verifies if such a possibility exists. As shown in Figure 5.13, Alloy fails to produce a counterexample when the assertion *showMaliciousTransactionWithJsCode* is checked.

Thus, with the results shown by the predicate *restrictJsCodeWithCorp* and the assertion *showMaliciousTransactionWithJsCode*, it can be said that the post-CORP model is sound. Appendix B shows the complete code for Pre-CORP and Post-CORP models.

5.4 Experimentation and Analysis

In this section, we explain about the implementation of CORP as a Chrome extension, its evaluation and the results of our analysis.

5.4.1 Implementation

We have developed an extension for Google Chrome web browser to implement a prototype of CORP. When a user installs the extension and loads a CORP-enabled website, the extension receives the CORP header, parses it and stores it in the browser's memory using *HTML5 localstorage* API. The storage is accessible across all the tabs of the browser and the policies set by multiple websites are stored and retrieved using the *origin* of the site as the key.

Let us consider a genuine CORP-enabled website, `http://G.com` and an attacker's website, `http://A.com`. Assume that they are opened in two tabs of a browser that has the CORP extension enabled. They trigger HTTP transactions to the servers *G* and *A* respectively. When the `http://A.com` attempts to trigger a cross origin request to *G*, the extension intercepts every outgoing request from the web page at `http://A.com`. It checks if the request is made to the origin of *G* i.e., `http://G.com` and fetches the policy associated with `http://G.com`. Only if the request complies with the policy set by *G*, the extension will allow the request, else it will block it. The *chrome.webRequest.onHeadersReceived* event of Chrome extension API helps in receiving HTTP response headers. The *chrome.webRequest.onBeforeRequest* [25] event helps in the interception process. This event is fired before any TCP connection is made and can be used to cancel requests.

5.4.2 Experiments

Apart from validating the soundess of CORP using Alloy (see Section 5.3), we have conducted several experiments to evaluate the effectiveness and ease of deployment of CORP.

5.4.2.1 Evaluating CORP Against a Corpus of Attacks

We have built two websites to evaluate CORP against a corpus of attacks. One of them is a victim site that is vulnerable to web infiltration attacks, and the other is a malicious site that can launch attacks on the victim site. We have referred to the test suite created by De Ryck et al. [18] and added their CSRF attack vectors to the malicious website. We have also added vectors for clickjacking and timing to the malicious site. If a genuine user logs in at the victim site in one tab and opens the malicious site in another tab, malicious requests (GET and POST) will be triggered against the victim site's server. On configuring CORP headers on the victim website and enabling the CORP extension, all malicious cross origin calls to the victim website will be blocked.

The chrome extension, the vulnerable and malicious websites can be accessed online and the attacks discussed in the paper can be replayed before and after installing the extension. The source code for these is available on Github [61].

5.4.2.2 Configuring CORP on Open Source Web Applications

To understand how CORP performs on real world websites, we have deployed three popular open source web applications (Table 5.1) and CORP-enabled them. Instead of deploying vulnerable versions

Table 5.1 Summary of open source web applications we experimented with

Application	Type	Version	# of source files	Lines of code	# of CORP rules
Wordpress	Blog/CMS	3.9.1	2288	23.9K	14
Moodle	LMS	2.5.6	11950	92.9K	84
Mediawiki	Wiki software	1.15.5-7	1338	99K	11

of these applications and fixing them with CORP, we chose to deploy their respective latest versions. Our idea is to verify that CORP is at least as good as the previous defenses and additionally conforms to the security guarantees promised in Section 5.2.4. We first confirmed that these applications implement at least one of the popular defenses against each of the web infiltration attacks (Section 3.2). As we have seen that these defenses insufficiently deal with infiltration attacks, we started afresh by completely disabling them. Then we started enabling CORP on each of these applications and verified that they are resilient to infiltration attacks. Our analysis shows that the effort required to CORP-enable large applications greatly depends on how resources are organized on the server e.g., all images placed under a single"/images" directory as against being scattered along multiple directories. Table 5.1 shows the number of rules needed to enable CORP on each of the applications, without reorganizing resources on

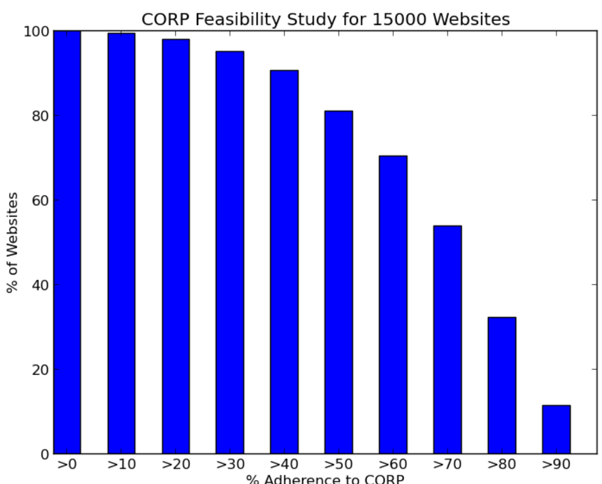

Figure 5.14 Bar chart showing adherence of Alexa Top 15,000 websites to CORP.

the server. With proper segregation of resources, the number of rules can be brought down to less than 10 per application.

5.4.2.3 Analyzing Adherence of Top Websites to CORP

We have analyzed the home page traffic of Alexa [7] Top 15,000 websites, to find if they adhere to CORP by segregating their content based on their type. The following content types were considered for analysis - images, css, scripts, html and flash. Figure 5.14 shows the results of the analysis. We find that more than 70% of sites already have an adherence greater than 60%. This is a positive indicator for the deployment of CORP, showing that website administrators can immediately use CORP on their existing sites and control their susceptibility to infiltration attacks.

Chapter 6

Conclusions and Future Work

The World Wide Web is undergoing changes faster than ever, with the demand for rich Internet applications that mimic desktop applications. Along with the evolution of richer capabilities (e.g., web mashups), the web has also seen the evolution of newer web-borne security threats (see 5.2.4.6). As explained in Section 2.2, the root cause of majority of these web-borne threats lie in the weak design of the core browser security model. Also, in a race to meet the growing demand, developers build several complex web applications without a proper understanding of the changing browser security model. The outcome of this are web applications that are inherently insecure by design (e.g., vulnerable to web infiltration attacks), and need multiple mitigations (e.g., CSRF tokens, frame busting etc.). Even if the mitigations are well known, developers often miss them (see 3.2.1) or use them insecurely (see 4.3).

6.1 Research Contributions

In this thesis, we have attempted to gain a deeper understanding of the security model of web browsers. We have studied that the existing model insufficiently protects web users against several web-borne threats (see 2.2), and observed certain gaps in the model (see 5.1.1). We have attempted to fix the gaps at a fundamental level, by enhancing the security model in a way that does not break the existing interactions on the web. In this process, we also came up with a developer-friendly library, which encapsulates certain security related intricacies from developers and makes the implementation of the modern web APIs less error-prone. The following are the key contributions of this thesis:

- To clearly understand the security model of web browsers and its limitations (see 2.2), we have created a corpus of web attacks (see 5.4). The source code for these attacks is available on Github [61]. Demonstrating these attacks practically was the starting point of our research. We believe this would be of immense help to researchers who wish to understand the security aspects of the web.
- We have identified a class of web attacks (CSRF, Clickjacking and Cross-Site Timing attacks) that are left unanswered in spite of the recent enhancements to the security model (via HTML5

APIs, CSP). We named them as *Web Infiltration Attacks* and found that they have a common root - a cross origin HTTP request triggered by a malicious client to a genuine server, without any restrictions. We attempted to mitigate these attacks by devising a uniform browser security policy called CORP (Cross Origin Request Policy). We have enforced CORP through an extension for Chrome browser, and evaluated it against the aforementioned corpus of web attacks.

- We have built a light weight formal model of a web browser in Alloy (Pre-CORP.als) to demonstrate our understanding of cross origin web interactions. We have enhanced this model to validate that our proposal, CORP, is sound (Post-CORP.als). Similar to the corpus of attacks, these formal models will serve as a starting point for researchers who are interested in formalizing the security aspects of the modern web.

- In an attempt to encapsulate browser security features from developers, we have built a developer-friendly library called "SafeMash". SafeMash enables developers to utilize modern browser security APIs (HTML5) and reduces the learning curve required to understand the intricacies of browser security. It ensures the development of secure web mashups. The source code of SafeMash and its accompanied demonstrations are available on GitHub and can be accessed at [62].

We believe that CSP and CORP together solve a large majority of exfiltration and infiltration attacks. The truth of this conjecture will, however, depend on the acceptance of CORP by browser vendors and its widespread adherence by web administrators.

6.2 Future Work

We have captured some of the interesting ideas that came up during our research group meetings and presentations. They are some of the directions in which the work presented in this thesis can be extended. They are as follows:

1 HTTP works at a level of abstraction that cannot anticipate the semantics of the transaction or of the resource sought by a client. CORP attempts to fill this semantic gap by conveying to the browser *who* (origins) can access *what* (resources) and *how* (events) as a result of a transaction. As new web standards emerge, declarative policies like CSP and CORP will need to carry richer semantic intent. Such information could, for example, be used to control other types of browser events like user interactions e.g., "no copy-paste" while visiting `https://bank.com` or force the browser to a canonical configuration e.g., disable browser extensions while visiting `https://bank.com`. It would be interesting to explore and expand the class of browser event types specifiable by declarative policies and study their impact on usability, security, and browsers for other form factors like mobiles and tablets.

2 To enforce CORP in a browser, we have created an extension for Chrome browser, as a proof of concept. The APIs exposed by Chrome extensions were sufficient to receive CORP via response

headers and enforce it on some of the requests leaving the browser. However, these APIs do not provide fine grained details such as the *origin* of all available types of HTTP event initiators. E.g., It is not possible to differentiate between the following two events via the APIs exposed by a browser extension: (1) A web page opened by a user by clicking on a hyperlink in an email client, (2) A web page opened by a user by typing its URL in the browser. This segregation is important to differentiate between cross origin HTTP requests triggered due to a social engineering attack, versus those triggered by a user's conscious action. To achieve such a fine grained control, access to low level browser APIs is required. Instead of enforcing CORP via a browser extension, modifying the source code of Chrome browser and incorporating CORP into it will provide finer control in restricting various types of HTTP events. This will also provide deeper insights that could potentially help in exploring newer possibilities while designing browser security policies.

3 The CORP Alloy models that we built capture only the details that are required for simulating/mitigating web infiltration attacks (see 5.3.2). They can be extended to capture complex cross origin web transactions e.g., federated authentication, OAuth handshake etc. Building a formal browser model that captures such interactions, along with CSP, CORP and the security policies in HTML5, will be a good direction of work that can validate the soundess of the modern web specifications.

4 To restrict the access privileges of JavaScript to a document's DOM, several solutions were proposed (see 3.1). Of all, Content Security Policy (CSP), which enforces restrictions on HTTP traffic leaving a genuine webpage, has gained wider acceptance. Though CSP mitigates XSS and exfiltration to a great extent, it does not help if the whitelisted script is compromised. E.g., In the case of the application-level DDoS attack explained in Section 5.2.4.6, the tampered analytics script would still have complete access to the news site's DOM, even if the site whitelisted the analytics script through CSP. This shows the need for a fine-grained access control model, where in a server can control the capabilities of third party scripts that are embedded in a document. Solving this problem, keeping in mind the complex interactions made by modern day web mashups, will be an interesting line of research.

5 JavaScript, in spite of its shortcomings as a language (e.g., single global namespace, dynamic typing etc.), has become ubiquitous on the modern web. While it is difficult for another programming language to replace JavaScript, it would be interesting to design a light weight browser that interprets a statically typed language instead. Various resources in a document such as DOM elements containing sensitive data (e.g., passwords, credit card information), cookies, non-sensitive data etc. may be defined as types, and type-safety rules may be defined such that third party code cannot access sensitive information.

While we have spent significant amount of time in analyzing various attacks and understanding the browser model, we would like to acknowledge that these contributions are merely the tip of the iceberg. Web security is a vast as well as a rapidly evolving area, with several novel attacks and bypasses to existing mitigations evolving on a regular basis. It is important for researchers in web security to track the

work of contemporary researchers in both academic as well as industry oriented security conferences. Also, it is important for the researchers to take feedback from developer communities as well. This is to ensure that the solutions they design stay relevant even to the most complicated scenarios on the web, while respecting backward compatibility.

Conference Publications

- Telikicherla, Krishna Chaitanya, and Venkatesh Choppella. “Enabling the development of safer mashups for open data." Proceedings of the 1st International Workshop on Inclusive Web Programming-Programming on the Web with Open Data for Societal Applications. ACM, 2014.
- Telikicherla, Krishna Chaitanya, Venkatesh Choppella, and Bruhadeshwar Bezawada. “CORP: A Browser Policy to Mitigate Web Infiltration Attacks." Information Systems Security. Springer International Publishing, 2014. 277-297.

Conference Submissions

- Telikicherla, Krishna Chaitanya, Venkatesh Choppella, and Bruhadeshwar Bezawada. “CORP: A Browser Policy to Mitigate Web Infiltration Attacks." WWW 2014.

Technical Reports

- Krishna Chaitanya Telikicherla and Venkatesh Choppella. “Alloy model for Cross Origin Request Policy (CORP)". Technical Report IIIT/TR/2013/31, IIIT-Hyderabad, August 2013.

Appendix A

Sample Alloy Model

A.1 A basic academic time table Alloy Model

The specification in Listing A.1 shows how Alloy can be used to model a basic academic time table. The model has the signatures *Faculty* and *Student*, who together form *Member*s. Each *Class* is *taughtBy* exactly one faculty and is *attendedBy* at least one student. A *Slot* is *associatedWith* at least one class. If there is more than one class in a slot, then those classes are said to be in parallel. Two constraints must always hold in this model - A faculty can teach only one class in a slot; A student can attend only one class in a slot. Accordingly, these two constraints are modelled as the Alloy facts- *facultyCannotTeachParallelClasses* and *studentCannotAttendParallelClasses* respectively.

```
1  abstract sig Member{}
2  sig Faculty extends Member{}
3  sig Student extends Member{}
4  sig Class {
5      taughtBy: one Faculty,
6      attendedBy: some Student
7  }
8  sig Slot{
9      allocatedWith: some Class
10 }
11 /* pred parallelClassesInASlot [c1: Class, c2: Class] -> Bool */
12 /*  If the inputs satisfy all of the constraints listed in the body,
       then the predicate evaluates to true. Else it evaluates to false. */
13 pred parallelClassesInASlot [c1: Class, c2: Class] {
14     /* Classes are parallel if they are allocated to the same slot. */
15     allocatedWith.c1 = allocatedWith.c2
```

```
16 }
17 /* If two classes c1 and c2 are in the same slot, i.e., parallel classes
     , then they should not be taught by the same faculty */
18 fact facultyCannotTeachParallelClasses{
19     no disj c1, c2: Class | {
20         parallelClassesInASlot[c1, c2] => some c1.taughtBy & c2.taughtBy
21     }
22 }
23 /* If two classes c1 and c2 are in the same slot, i.e., parallel classes
     , then they should not be attended by the same student */
24 fact studentCannotAttendParallelClasses{
25     no disj c1, c2: Class | {
26         parallelClassesInASlot[c1, c2] => some c1.attendedBy & c2.
     attendedBy
27     }
28 }
29 /* There must not be classes left out without being allocated to a slot.
      */
30 fact allClassMustBeAllocatedToASlot{
31     no Class - Slot.allocatedWith
32 }
33 /* There must not be students who do not attend any class */
34 fact allStudentsMustAttendClasses{
35     no Student - Class.attendedBy
36 }
37 /* There must not be faculty who do not teach any class */
38 fact allFacultyMustTakeClasses{
39     no Faculty-Class.taughtBy
40 }
41 /* A class can be taught in two different time slots. A student can
     prefer to attend the same class in two different slots. */
42 pred studentCanAttendSameClassInDifferentSlots {
43     some disj s1, s2: Slot, s: Student| {
44         /* class attended by student s is allocated to slot s1 */
45         s.~attendedBy = s1.allocatedWith
```

```
46          /* class attended by student s is allocated to slot s2  */
47          s.~attendedBy = s2.allocatedWith
48      }
49  }
50  assert studentCannotAttendDifferentClassesInSameSlot {
51      no disj c1, c2: Class | {
52          /* Disjoint classes c1, c2 are in same slot  */
53           c1.~allocatedWith = c2.~allocatedWith
54          /* There is some student who attends both the classes c1 and c2
        */
55          some c1.attendedBy & c2.attendedBy
56      }
57  }
58  pred showTimeTable{}
59  run showTimeTable for 3 but exactly 1 Slot expect 1
60  run showTimeTable for 5 but exactly 1 Slot expect 1
61  run studentCanAttendSameClassInDifferentSlots for 3 but exactly 1
       Student expect 1
62  check studentCannotAttendDifferentClassesInSameSlot for 20 expect 0
```

Listing A.1 Modelling academic time table using Alloy

Listing A.1 has three tests (two predicates and one assertion) to validate the soundness of the time table model. The predicate *showTimeTable* asks the Alloy analyzer to show an instance of the model with no additional constraints apart from the listed facts, for scopes 3 and 5. When this predicate is run, the Alloy analyzer generates the model instances shown in Figure A.1. The nodes in the directed graph represent the atoms that belong to a signature, while the edges represent the relation between the nodes that they connect. Each of the instances show a slot which is associated with at least one class. Each class is taught by exactly one faculty and is attended by at least one student. These are valid instances and are as per the envisioned design.

The predicate *studentCanAttendSameClassInDifferentSlots* asks the Alloy analyzer to show an instance of the model where a student can attend the same class in different slots. When this predicate is run, the Alloy analyzer generates the model instances shown in Figure A.2. The instance shows two slots associated with a single class. The class is taught by one faculty and attended by one student. Since the faculty and the student are involved only in a single class in a given slot, this is a valid instance.

The assertion *studentCannotAttendDifferentClassesInSameSlot* verifies the assumption that a student cannot attend parallel classes in a slot. When this assertion is checked, the Alloy analyzer fails to

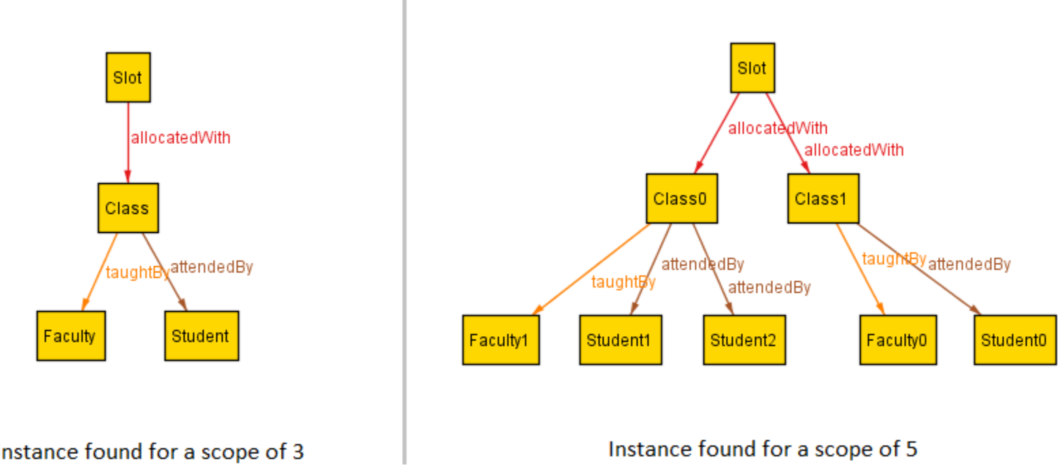

Figure A.1 Model instances generated by Alloy analyzer for the predicate *showTimeTable*

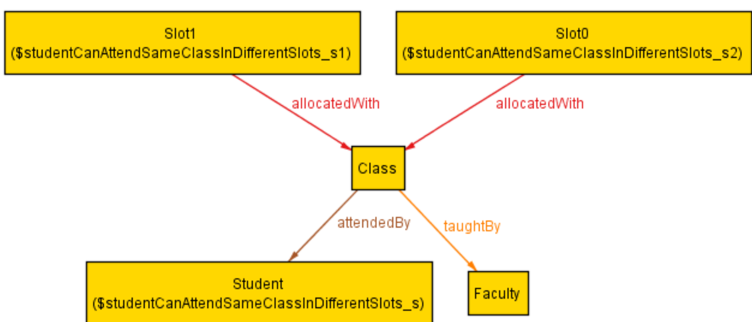

Figure A.2 Model instance generated by Alloy analyzer for the predicate *studentCanAttendSame-ClassInDifferentSlots*

```
Executing "Check studentCanAttendDifferentClassesInSameSlot for 20"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   28903 vars. 1320 primary vars. 54819 clauses. 275ms.
   No counterexample found. Assertion may be valid. 63ms.
```

Figure A.3 No counterexample is found for the assertion *studentCannotAttendDifferentClassesIn-SameSlot*

produce a counterexample even for a scope as large as 20, as shown in Figure A.3. This shows that the assertion is valid.

These tests establish the soundness of the sample academic time table model.

Appendix B

CORP Alloy Models

B.1 Pre-CORP Alloy Model

The specification in Listing B.1 shows the complete Pre-CORP Alloy model, which was explained in Section 5.3.3

```
1  abstract sig HTTPTransaction{
2      req:  HTTPRequest,
3      resp: HTTPResponse
4  }
5  sig HTTPRequest {
6      from:  Browser,
7      to:  Server,
8      host: Origin
9  }
10 sig HTTPResponse{
11     from:  Server,
12     to:  Browser,
13     host: Origin
14 }
15 /****************************
16                 END POINTS
17 ****************************/
18 sig Origin{
19     pointsTo:  Server
20 }
21 one sig Browser{
```

```
22      activeDoc: Document
23  }
24  abstract sig Server {}
25  sig Server1, Server2 extends Server{}
26  sig Document{
27      docOrigin: Origin,
28      elements: set ActiveHTMLElement,
29      activeCode: set JavaScriptCode
30  }
31  abstract sig HTTPEventInitiator{
32      httpTrans: lone HTTPTransaction,
33      initiatorOrigin: Origin
34  }
35  sig JavaScriptCode extends HTTPEventInitiator{}
36  /***************************
37                      ELEMENTS
38  ***************************/
39  abstract sig ActiveHTMLElement extends HTTPEventInitiator {}
40  abstract sig LoadingElement, ActionElement extends ActiveHTMLElement {}
41  sig ScriptElement, ImageElement, CssElement, Iframe extends
        LoadingElement {}
42  sig Hyperlink, Form extends ActionElement{}
43  /***************************
44                              FACTS
45  ***************************/
46  //Fact:: EventInitiators inherit parent document's origin
47  // elements = (Document, HTMLElement)
48  // ~elements = (HTMLElement, Document)
49  fact EventInitiatorsInheritParentOrigin{
50      all elem: ActiveHTMLElement | elem.initiatorOrigin = elem.~elements.
        docOrigin
51      all js: JavaScriptCode | js.initiatorOrigin = js.~activeCode.
        docOrigin
52  }
53  fact TransactionRules{
```

```
54      all t:HTTPTransaction, b:Browser, s:Server| {
55          //Request and response must belong to the same origin
56          t.req.host = t.resp.host
57          //A Http request's hostname and its destination server's origin
    must be the same
58          t.req.host = t.req.to.~pointsTo
59          //If a request is sent to a server, response should be received
    from the same server
60          s=t.req.to => s = t.resp.from
61          //If a request is sent from a browser, response should be
    received by the same browser
62          b = t.req.from => b = t.resp.to
63          //All HTTPTransactions should be due to HTTP event initiators (i
    .e., active HTML elements or JS code)
64          t in HTTPEventInitiator.httpTrans
65      }
66      // Two transactions should not interfere with each other's request/
    response
67      all disj t1,t2: HTTPTransaction | {
68          no (t1.req & t2.req)
69          no (t1.resp & t2.resp)
70      }
71  }
72  fact noOrphanElements{
73      // Ensures no unrelated/hanging elements exist
74      no (ActiveHTMLElement - Browser.activeDoc.elements)
75      no (Document - Browser.activeDoc)
76      no (Server - Origin.pointsTo)
77      //No orphan Request/Responses.
78      (no HTTPRequest-HTTPTransaction.req) and (no HTTPResponse-
    HTTPTransaction.resp)
79      no Origin - (HTTPRequest.host + HTTPResponse.host+Document.docOrigin
    )
80      no JavaScriptCode - Document.activeCode
81      no ActiveHTMLElement - httpTrans.HTTPTransaction
```

```
82       no JavaScriptCode - httpTrans.HTTPTransaction
83   }
84   fact Disjointness {
85       all disj b1,b2: Browser | {
86           no (b1.activeDoc & b2.activeDoc)
87           no (b1.activeDoc.docOrigin & b2.activeDoc.docOrigin)
88       }
89       //Two distinct origins do not point to the same server
90       all disj o1,o2:Origin | no (o1.pointsTo & o2.pointsTo)
91       all disj e1, e2: ActiveHTMLElement | no (e1.httpTrans & e2.httpTrans
        )
92   }
93   /****************************
94                       PREDICATES
95   ****************************/
96   pred showBasicModel{
97       no HTTPTransaction
98   }
99   pred sameOriginTransaction{
100      some HTTPTransaction
101      one Server
102  }
103  pred crossOriginTransaction{
104      some t:HTTPTransaction|{
105          t.req.from.activeDoc.docOrigin.pointsTo=Server1
106          t.req.to=Server2
107      }
108  }
109  run showBasicModel
110  run sameOriginTransaction //for 10
111  run crossOriginTransaction for 3 but exactly 1 HTTPEventInitiator
```

Listing B.1 Pre-CORP Alloy Model

B.2 Post-CORP Alloy Model

The specification in Listing B.2 shows the complete Post-CORP Alloy model, which was explained in Section 5.3.4.

```
1  abstract sig HTTPTransaction{
2      req:  HTTPRequest,
3      resp: HTTPResponse
4  }
5  sig HTTPRequest {
6      from:  Browser,
7      to:  Server,
8      host: Origin,
9      reqPath: one Path
10 }
11 sig HTTPResponse{
12     from:  Server,
13     to:  Browser,
14     host: Origin
15 }
16 /****************************
17                 END POINTS
18 ****************************/
19 sig Origin{
20     pointsTo:  Server
21 }
22 one sig Browser{
23     activeDoc: Document
24 }
25 abstract sig Server {
26     resourcePath: set Path
27 }
28 sig EvilServer, GenuineServer extends Server{}
29 sig Document{
30     docOrigin: Origin,
31     elements: set ActiveHTMLElement,
```

```
32     activeCode: set JavaScriptCode
33 }
34 abstract sig HTTPEventInitiator{
35     httpTrans: lone HTTPTransaction,
36     initiatorOrigin: Origin
37 }
38 sig JavaScriptCode extends HTTPEventInitiator{}
39 /****************************
40                     ELEMENTS
41 ****************************/
42 abstract sig ActiveHTMLElement extends HTTPEventInitiator {}
43 abstract sig LoadingElement, ActionElement extends ActiveHTMLElement {}
44 sig ScriptElement, ImageElement, CssElement, Iframe extends
      LoadingElement {}
45 sig Hyperlink, Form extends ActionElement{}
46 abstract sig Path{}
47 one sig ImgPath, JsPath, CssPath, NonSensitivePagesPath,
      SensitivePagesPath extends Path{}
48 one sig origin1, origin2 extends Origin{}
49 /****************************
50                             FACTS
51 ****************************/
52 //Fact:: EventInitiators inherit parent document's origin
53 // ~elements = (HTMLElement, Document)
54 fact EventInitiatorsInheritParentOrigin{
55     all elem: ActiveHTMLElement | elem.initiatorOrigin = elem.~elements.
      docOrigin
56     all js: JavaScriptCode | js.initiatorOrigin = js.~activeCode.
      docOrigin
57 }
58 fact TransactionRules{
59     all t:HTTPTransaction, b:Browser, s:Server| {
60         //Request and response must belong to the same origin
61         t.req.host = t.resp.host
```

```
62        //A Http request's hostname and its destination server's origin
    must be the same
63            /*pointsTo=(Origin, Server).
64                t.req.to.~pointsTo= (t.req.to).(Server, Origin) -> (
    Server. (Server, Origin)) -> Origin */
65        t.req.host = t.req.to.~pointsTo
66        //If a request is sent to a server, response should be received
    from the same server
67        s=t.req.to => s = t.resp.from
68        //If a request is sent from a browser, response should be
    received by the same browser
69        b = t.req.from => b = t.resp.to
70        //All HTTPTransactions should be due to HTTP event initiators (i
    .e., active HTML elements or JS code)
71        t in HTTPEventInitiator.httpTrans
72        //Request path and server's resource path must be the same
73        some (t.req.to.resourcePath & t.req.reqPath )
74    }
75    // Two transactions should not interfere with each other's request/
    response
76    all disj t1,t2: HTTPTransaction | {
77        no (t1.req & t2.req)
78        no (t1.resp & t2.resp)
79    }
80 }
81 fact noOrphanElements{
82    // Ensures no unrelated/hanging elements exist
83    no (ActiveHTMLElement - Browser.activeDoc.elements)
84    no (Document - Browser.activeDoc)
85    no (Server - Origin.pointsTo)
86    no (Path - (Server.resourcePath + HTTPRequest.reqPath))
87    //No orphan Request/Responses.
88    (no HTTPRequest-HTTPTransaction.req) and (no HTTPResponse-
    HTTPTransaction.resp)
```

```
89     no Origin - (HTTPRequest.host + HTTPResponse.host+Document.docOrigin
      )
90     no JavaScriptCode - Document.activeCode
91     no ActiveHTMLElement - httpTrans.HTTPTransaction
92     no JavaScriptCode - httpTrans.HTTPTransaction
93 }
94 fact Disjointness {
95     all disj b1,b2: Browser | {
96         no (b1.activeDoc & b2.activeDoc)
97         no (b1.activeDoc.docOrigin & b2.activeDoc.docOrigin)
98     }
99     //Two distinct origins do not point to the same server
100    all disj o1,o2:Origin | no (o1.pointsTo & o2.pointsTo)
101
102    all disj e1, e2: ActiveHTMLElement | no (e1.httpTrans & e2.httpTrans
      )
103
104    no GenuineServer.resourcePath &  EvilServer.resourcePath
105 }
106 /***************************
107                     PREDICATES
108 ***************************/
109 pred maliciousXOriginTransaction{
110     some t:HTTPTransaction|{
111         t.req.from.activeDoc.docOrigin.pointsTo=EvilServer
112         t.req.to=GenuineServer
113         t.req.reqPath = SensitivePagesPath
114     }
115 }
116 run maliciousXOriginTransaction  for 3 but exactly 1 HTTPTransaction
      expect 1
117 run maliciousXOriginTransaction  for 5
118
119 // httpTrans = (HTTPEventInitiator, HTTPTransaction)
```

```
120 pred corpCompliantTransaction[o:Origin, ev: HTTPEventInitiator, pt: Path
    ] {
121     some t:HTTPTransaction {
122         t.req.from.activeDoc.docOrigin.pointsTo=EvilServer
123         t.req.to=GenuineServer
124         t.req.from.activeDoc.docOrigin = o and
125         httpTrans.t= ev and
126         t.req.reqPath=  pt
127     }
128 }
129 pred restrictImagesWithCorp{
130     corpCompliantTransaction[origin2, ImageElement, ImgPath]
131 }
132 pred restrictScriptsWithCorp{
133     corpCompliantTransaction[origin2, ScriptElement, JsPath]
134 }
135 pred restrictJsCodeWithCorp{
136     corpCompliantTransaction[origin2, JavaScriptCode,
    NonSensitivePagesPath]
137 }
138 assert showMaliciousTransactionWithImage {
139     no t:HTTPTransaction |{
140         //restrictImagesWithCorp
141         corpCompliantTransaction[origin2, ImageElement, ImgPath]
142         t.~httpTrans.initiatorOrigin = origin2
143         t.~httpTrans=ImageElement
144         t.req.reqPath! = ImgPath
145     }
146 }
147 assert showMaliciousTransactionWithJsCode {
148     no t:HTTPTransaction |{
149         corpCompliantTransaction[origin2, JavaScriptCode,
    NonSensitivePagesPath]
150         t.~httpTrans.initiatorOrigin = origin2
151         t.~httpTrans=JavaScriptCode
```

```
152         t.req.reqPath! = NonSensitivePagesPath
153     }
154 }
155 run restrictImagesWithCorp  for 3 but exactly 1 HTTPTransaction expect 1
156 run restrictScriptsWithCorp for 3 but exactly 1 HTTPTransaction expect 1
157 run restrictJsCodeWithCorp for 3 but exactly 1 HTTPTransaction expect 1
158 check showMaliciousTransactionWithImage  for 20 // but exactly 1
        HTTPTransaction expect 0
159 check showMaliciousTransactionWithJsCode  for 20 // but exactly 1
        HTTPTransaction expect 0
```

Listing B.2 Post-CORP Alloy Model

Bibliography

[1] GitHub hit by DDoS attack-Hacker News, Mar 2015. https://news.ycombinator.com/item?id=9275041.

[2] Large Scale DDoS Attack on github.com, Mar 2015. https://github.com/blog/1981-large-scale-ddos-attack-on-github-com.

[3] Pin-pointing China's attack against GitHub, Mar 2015. http://blog.erratasec.com/2015/04/pin-pointing-chinas-attack-against.html#.ViObwfkrKhc.

[4] AdBlockPlus. HTTP Referer, 2008. http://adblockplus.org/blog/http-referer-header-wont-help-you-with-csrf.

[5] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 290–304. IEEE, 2010.

[6] D. Akhawe, P. Saxena, and D. Song. Privilege separation in HTML5 applications. In *Proceedings of the USENIX Security Symposium*, 2012.

[7] Alexa. Alexa top sites, Oct 2013. http://www.alexa.com/topsites.

[8] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A solution for the automated detection of clickjacking attacks. ASIACCS '10, pages 135–144, New York, NY, USA, 2010. ACM.

[9] A. Barth. RFC 6454 - The Web Origin Concept. Technical report, 2011. tools.ietf.org/search/rfc6454.

[10] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88. ACM, 2008.

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

[12] T. Berners-Lee and D. Connolly. Hypertext Markup Language – 2.0. Technical Report RFC1866, W3C, 1995. http://tools.ietf.org/html/rfc1866.

[13] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web*, pages 621–628. ACM, 2007.

[14] D. Crockford. Rfc4627: The application/json media type for javascript object notation (json), July 2006. http://tools.ietf.org/html/rfc4627.

[15] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang. Lightweight server support for browser-based CSRF protection. In *Proceedings of the 22nd international conference on World Wide Web*, pages 273–284, 2013.

[16] P. De Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey. In *Information Security Technology for Applications*, pages 223–238. Springer, 2012.

[17] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *Engineering Secure Software and Systems*, pages 18–34. Springer, 2010.

[18] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against csrf attacks. In *Computer Security–ESORICS 2011*, pages 100–116. Springer, 2011.

[19] F. et al. Rfc 2616, content negotation in http/1.1, June 1999. `http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html`.

[20] T. B.-L. et al. Uniform Resource Locators (URL). Technical Report RFC1738, IETF, 1994. `https://www.ietf.org/rfc/rfc1738.txt`.

[21] Facebook. Facebook, Washington State AG target clickjackers. Blog, Jan 2012. `https://www.facebook.com/notes/facebook-security/facebook-washington-state-ag-target-clickjackers/10150494427000766`.

[22] R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[23] M. Finifter, J. Weinberger, and A. Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *NDSS*, 2010.

[24] J. J. Garrett. AJAX: A New Approach to Web Applications, Feb 2005. `https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php`.

[25] Google. Life cycle of requests in Chrome.webRequest API, 2013. `http://developer.chrome.com/extensions/webRequest.html`.

[26] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song. The Emperor's New APIs: On the (In)Secure Usage of New Client-side Primitives. In *Proceedings of the Web*, volume 2, 2010.

[27] R. Hansen and J. Grossman. Clickjacking. Blog, Dec 2008. `http://www.sectheory.com/clickjacking.htm`.

[28] M. Heiderich. CSRFx, 2007. `https://code.google.com/p/csrfx/`.

[29] Hodges. RFC 6797, HTTP Strict Transport Security (HSTS), November 2012. `http://tools.ietf.org/html/rfc6797`.

[30] J. Hodges, A. Steingruebl, et al. The need for a coherent web security policy framework. Web.

[31] L. Huang and C. Jackson. Clickjacking attacks unresolved. *White paper, CyLab*, 2011. `http://mayscript.com/blog/david/clickjacking-attacks-unresolved`.

[32] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and Defenses. In *USENIX Security Symposium*, 2012.

[33] C. Jackson and H. J. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *Proceedings of the 16th international conference on World Wide Web*, pages 611–620. ACM, 2007.

[34] D. Jackson. Software Abstractions: Logic. *Language, and Analysis, The MIT Press*, 2006.

[35] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference*, pages 231–240. IEEE, 2010.

[36] G. Jeremiah. Introducing the 'I Know ...' series. Blog, October 2012. `https://blog.whitehatsec.com/introducing-the-i-know-series/`.

[37] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 601–610. ACM, 2007.

[38] M. Johns and J. Winter. RequestRodeo: Client side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference*, 2006.

[39] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Securecomm and Workshops, 2006*, pages 1–10. IEEE, 2006.

[40] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337. ACM, 2006.

[41] R. Kotcher, Y. Pei, and P. Jumde. Stealing cross-origin pixels: Timing attacks on css filters and shaders. 2013. `www.robertkotcher.com/pdf/TimingAttacks.pdf`.

[42] K. Kotowicz. Cross domain content extraction with fake captcha. `http://blog.kotowicz.net/2011/07/cross-domain-content-extraction-with.html`.

[43] S. Lekies, M. Heiderich, D. Appelt, T. Holz, and M. Johns. On the fragility and limitations of current browser-provided clickjacking protection schemes. In *Woot 2012, USENIX Security Symposium*. USENIX, 2012.

[44] W. Maes, T. Heyman, L. Desmet, and W. Joosen. Browser protection against cross-site request forgery. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 3–10. ACM, 2009.

[45] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the 5th ACM symposium on information, computer and communications security*, pages 15–23. ACM, 2010.

[46] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Cryptography and Data Security*, pages 238–255. Springer, 2009.

[47] G. Maone. Hello ClearClick, goodbye clickjacking! Blog, October 2008. `http://hackademix.net/2008/10/08/hello-clearclick-goodbye-clickjacking/`.

[48] G. Maone, D. L.-S. Huang, T. Gondrom, and B. Hill. User Interface Security Directives for Content Security Policy, September 2013. https://dvcs.w3.org/hg/user-interface-safety/raw-file/tip/user-interface-safety.html.

[49] L. A. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 481–496. IEEE, 2010.

[50] Microsoft. Combating ClickJacking With X-Frame-Options. Blog, March 2010. http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx.

[51] A. Nafeez. Stealing Facebook Graph API Access Token : Yet Another UI Redressing Vector, September 2011. http://blog.skepticfx.com/2011/09/facebook-graph-api-access-token.html.

[52] T. Oda and A. Somayaji. Enhancing web page security with security style sheets, 2011.

[53] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. SOMA: Mutual approval for included content in web pages. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 89–98. ACM, 2008.

[54] OWASP. OWASP Top Ten Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[55] OWASP. CSRF Guard, 2007. https://www.owasp.org/index.php/CSRF_Guard.

[56] M. Pilgrim. Dive into HTML5. Technical report. http://diveintohtml5.info/past.html#history-of-the-img-element.

[57] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[58] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, pages 921–930. ACM, 2010.

[59] P. Stone. Pixel perfect timing attacks with html5. 2013. http://contextis.com/files/Browser_Timing_Attacks.pdf.

[60] K. C. Telikicherla. Analyzing the new social engineering spam on facebook - lady with an axe. Blog post, June 2013. http://bit.ly/FBSpamAxe.

[61] K. C. Telikicherla. CORP repository. http://iiithyd-websec.github.io/corp/, Oct 2013.

[62] K. C. Telikicherla and V. Choppella. Source code and live demonstration of SafeMash, Feb 2014. https://github.com/iiithyd-websec/safemash.

[63] K. C. Telikicherla and V. Choppella. Source code and live demonstration of web security concepts, Feb 2014. http://iiithyd-websec.github.io/.

[64] M. Ter Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 331–346. IEEE, 2009.

[65] W3C. HTML5 CORS-W3C Candidate Recommendation 16 Jan 2014. Technical report. http://www.w3.org/TR/cors/.

[66] W3C. HTML5 Web Messaging-W3C Candidate Recommendation 1 May 2012. Technical report. http://www.w3.org/TR/webmessaging/.

[67] W3C. Iframe sandbox-W3C Candidate Recommendation 6 August 2013. Technical report. http://www.w3.org/TR/html5/embedded-content-0.html#the-iframe-element.

[68] W3C. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/soap/.

[69] W3C. History of the World Wide Web. Technical report, 1989. http://www.w3.org/Consortium/facts#history.

[70] W3C. Content Security Policy 1.1-W3C Working Draft 11 February 2014. Technical report, 2014. http://www.w3.org/TR/CSP11/#directives.

[71] W3C. HTML5-W3C Candidate Recommendation 6 August 2013. Technical report, 2014. http://www.w3.org/TR/html5/.

[72] Wikipedia. Principle of least privilege. http://en.wikipedia.org/wiki/Principle_of_least_privilege.

[73] O. A. Zabir. Dropthings, 2014. https://code.google.com/p/dropthings/.