CORP: A Browser Policy to Mitigate Web Infiltration Attacks

Krishna Chaitanya Telikicherla, Venkatesh Choppella, and Bruhadeshwar Bezawada

Software Engineering Research Center,
International Institute of Information Technology (IIIT),
Hyderabad - 500032, India
KrishnaChaitanya.T@research.iiit.ac.in, Venkatesh.Choppella@iiit.ac.in, Bezawada@mail.iiit.ac.in

Abstract. Cross origin interactions constitute the core of today's collaborative Word Wide Web. They are, however, also the cause of malicious behaviour like Cross-Site Request Forgery (CSRF), clickjacking, and cross-site timing attacks, which we collectively refer as *Web Infiltration attacks*. These attacks are a rampant source of information stealth and privacy intrusion on the web. Existing browser security policies like Same Origin Policy, either ignore this class of attacks or, like Content Security Policy, insufficiently deal with them.

In this paper, we propose a new declarative browser security policy — "Cross Origin Request Policy" (CORP) — to mitigate such attacks. 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. Based on lessons drawn from examining various types of cross origin attacks, we formulate CORP and demonstrate its effectiveness and ease of deployment. We formally verify the design of CORP by modelling it in the Alloy model checker. We also implement CORP as a browser extension for the Chrome web browser and evaluate it against real-world cross origin attacks on open source web applications. Our initial investigation reveals that most of the popular websites already segregate their resources in a way which makes deployment of CORP easier.

Keywords: Web Browser, Security, World Wide Web, Cross-site request forgery, Access control policy.

1 Introduction

When the World Wide Web was invented in 1989 [1], it only had a set of static pages interconnected via hyperlinks. With the addition of images in 1993[2], 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

A. Prakash and R. Shyamasundar (Eds.): ICISS 2014, LNCS 8880, pp. 277–297, 2014.
© Springer International Publishing Switzerland 2014

in 1995[3], cross-site interactions became a real security threat. For example, as shown Figure 1, a genuine website, say *G.com*, could now be compromised by an attacker who injects malicious content like an image tag pointing to attacker's site, say *A.com*. This is an example of a cross-site scripting (XSS) attack. A victim requesting the infected page could end up unwittingly participating in exfiltration, i.e., the leakage of private data to *A.com*.

Despite several proposals like whitelisting[4], input sanitization[5], static analysis[6], browser sandboxing[7], XSS vulnerabilities continue to be pervasive on the web. Browsers as early as 1995[8] introduced the Same Origin Policy (SOP)[9], which was designed to prevent scripts from accessing DOM, network and storage data belonging to other web origins. The earlier problem of cross-origin requests through automatic form submissions or content inclusion was, however, left unanswered by SOP. Content Security Policy (CSP), introduced in 2010[10] improves on SOP in mitigating the exfiltration problem by disabling inline scripts, restricting the sources of external scripts.

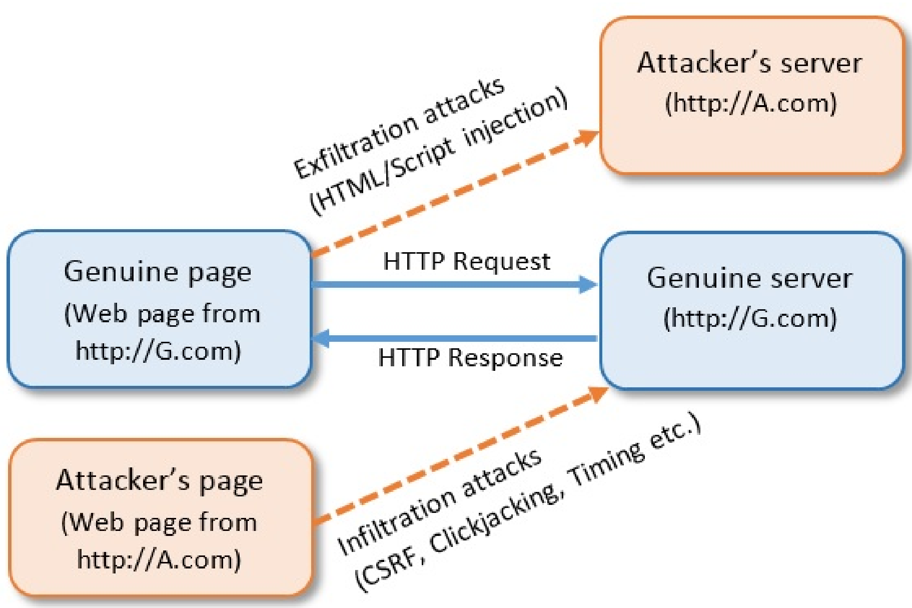


Fig. 1. Exfiltration vs. Infiltration attacks

1.1 Proposed Approach

Our work 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 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 <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, can not be controlled by CSP.

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 [11,12] or after the browser has already received the response [13,14].
– **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 of 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, 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 tab t_A contains a page p_A from a server s_A. Along with the 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: We follow the threat model classifications proposed by Akhawe et al. [15], which defines the capabilities of web, network and gadget attackers. Throughout the paper, we take into consideration only the threats that come under the capabilities of a web attacker. 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 paper 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. (2) We have formalized our proposal in Alloy [16], 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 also analyzed home page traffic of over 15,000 popular websites and confirmed that the burden on web administrators to deploy CORP will be minimum.

Organization of the Paper: The rest of the paper is organized as follows: Section 2 gives an overview of web infiltration attacks. Section 3 gives an overview of related work done in preventing these attacks. Section 4 explains the design of CORP. Section 5 describes the implementation of CORP as a Chrome extension and the experimental methodology to evaluate its effectiveness and Section 6 concludes with a discussion of future work.

2 Web Infiltration Attacks

In this section, we examine three common attacks: CSRF, clickjacking and cross-site timing. Each of these is an instance of a web infiltration attack.

2.1 Understanding CSRF

CSRF is a highly exploited web based vulnerability and is consistently listed in the OWASP Top Ten [17]. In a typical CSRF attack, a malicious site instructs a victim's browser to send an HTTP request to an honest site. This malicious request is sent on behalf of the victim, thereby disrupting the integrity of the victim's session.

In the example below, it is assumed that a user is logged in at a genuine site *G.com* and then opens an attacker's site *A.com* in a new browser tab. The page from the attacker's site contains the HTML shown in Listing 1.1.

```
<img src="http://G.com/user/delete.php">
```

Listing 1.1. Basic CSRF attack via image tag

As soon as the attacker's page is loaded, the image tag triggers a cascading HTTP request to *G.com*, which deletes the user's profile on the site. Though servers do not generally accept state changing requests using HTTP GET, generating HTTP POST requests using HTML forms is trivial. 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[1]. Also, in most cases servers do not have information about the origin which triggered the request (see Section 3.1 for details).

2.2 Understanding Clickjacking

Clickjacking was first reported in web browsers in 2008 [18]. 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 (if present) are submitted[2]. This makes clickjacking more dangerous than CSRF.

There are many online scams/spams, especially on social networks, which use clickjacking and make money. Facebook recently sued an ad network that used clickjacking and stole personal information of users, thereby making up to $1.2 million a month [19].

---

[1] This is an instance of the "Confused Deputy Problem", where the browser is the confused deputy.

[2] This is an instance of the "Confused Deputy Problem", where the user is the confused deputy.

2.3 Understanding Cross-Site Timing Attacks

Bortz et al. [12] 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 [20] and CSS shaders [21] 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 [22], which do not have CSRF tokens e.g., private profile pictures, script files etc. This means, 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, even authenticated URLs can also be used as a vector for login detection. Listing 1.2 shows the case where the script tag makes a cross origin HTTP request to a non-existing page on a target site to detect login status of the user.

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

Listing 1.2. 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.

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.

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 4.

3 Related Work

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

3.1 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, the server generates a unique random secret and embeds it into web pages in every HTTP response. The server checks if the secret received from the browser is the same as the one it generated earlier and accepts the request if the check succeeds. Since the token is not available to the attacker, request forgery cannot happen. CSRF Guard [11] and CSRFx [23] 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 social engineering techniques tokens can be stolen thereby re-enabling request forgery.

NoForge: NoForge [24] is a server side proxy which inspects and modifies client requests. It modifies 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) [25] enforcing constraints on HTTP traffic by mandating mutual approval from both the sites participating in an interaction. Websites send manifest files that inform a browser which 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 of 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 [26] 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 path or query strings that contain sensitive information. Due to privacy constraints, the Referer header is stripped by filtering proxies [27]. 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 [28] 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 [29] is a browser based solution which attempts to infer the intent of the 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 [30,31] builds on Maes et al. [32] 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 issues 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) [33] 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 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 4.1.

3.2 Approaches to Mitigate Clickjacking

There are several proposals to detect [34,35], prevent [36,37] Clickjacking and intelligent tricks [38,39] which bypass some of them. Browser vendors and W3C have incorporated ideas from these proposals and are working towards robust defense for clickjacking. Below are two important contributions in this direction:

X-Frame-Options (XFO) Header: The X-Frame-Options HTTP response header [13], 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 content of the frame to render, and SAMEORIGIN, which allows content of the frame to render 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* [14] 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 life-cycle. Due to this, several bypasses such as *Double Clickjacking* [38], *Nested Clickjacking* [39] and *Login detection using XFO* [22] arise.

3.3 Approaches to Mitigate Cross-Site Timing Attacks

Bortz et al. [12] proposed that by ensuring a web server takes constant time to process a request might help in mitigating cross-site timing attacks. 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.

4 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.

4.1 Core Idea Behind CORP

Based on our clear understanding of various types of web infiltration attacks (Section 2), we realize the need for a mechanism which 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 4.3). Therefore, CORP is a 3-way relation defined over the sets *Who*, *What* and *How*, as shown in Equation (1).

$$CORP \subseteq Origin \times ResourcePath \times EventType \tag{1}$$

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

$$\begin{aligned} Origin &= \{O_1, O_2, O_3\} \\ ResourcePath &= \{P_1, P_2, P_3\} \\ EventType &= \{Img, Script, Form\} \\ CORP, C_p &= \{(O_1, P_1, Img), (O_2, P_2, Form), (O_2, P_3, Script)\} \end{aligned} \tag{2}$$

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 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.

4.2 Browser Model with CORP

Figure 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.

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

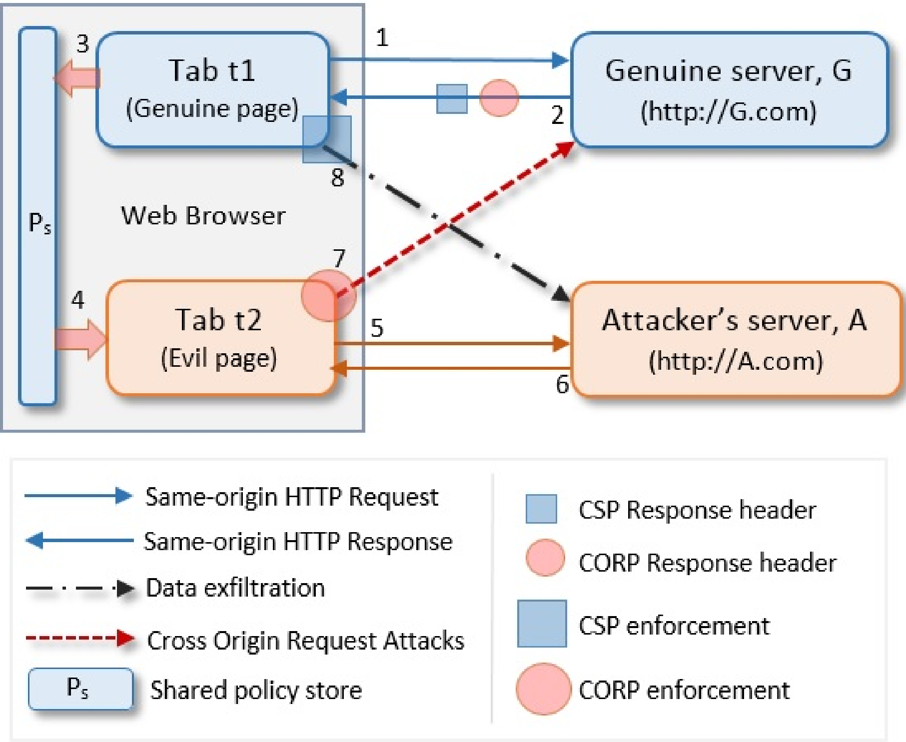


Fig. 2. Browser model showing exfiltration & infiltration and how they are mitigated by CSP & CORP

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 4.3, 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.

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 impor-

tant 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 [40] 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.

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 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., which event).

4.3 Abstract Syntax of CORP

Listing 1.3 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 1.3. 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/*".

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.

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 1.4.

```
*    *    *    DENY
```

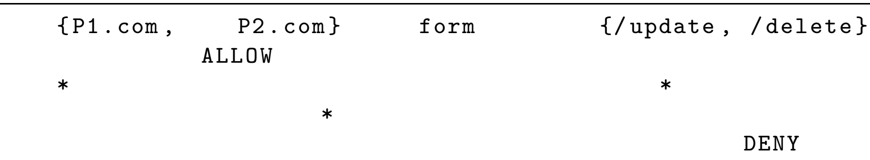
Listing 1.4. Block all cross origin requests

– **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 1.5.

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

Listing 1.5. 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 1.6.



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

Listing 1.6. Allow selective access to selective origins

4.4 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:

Fine Grained Access Control. Through CORP, websites can decide *who* (i.e., which set of origins) can trigger cross origin requests to 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.

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.

Early Enforcement of Clickjacking Defense. As discussed in Section 3.2, XFO and CSP-UI-Security directives are two important proposals to mitigate clickjacking. Figure 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 2. As explained in Section 4.2, 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 and points its *src* to 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* [38], *Nested Clickjacking* [39] and *Login detection using XFO* [22] 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 1.7 shows CORP configuration to mitigate clickjacking.

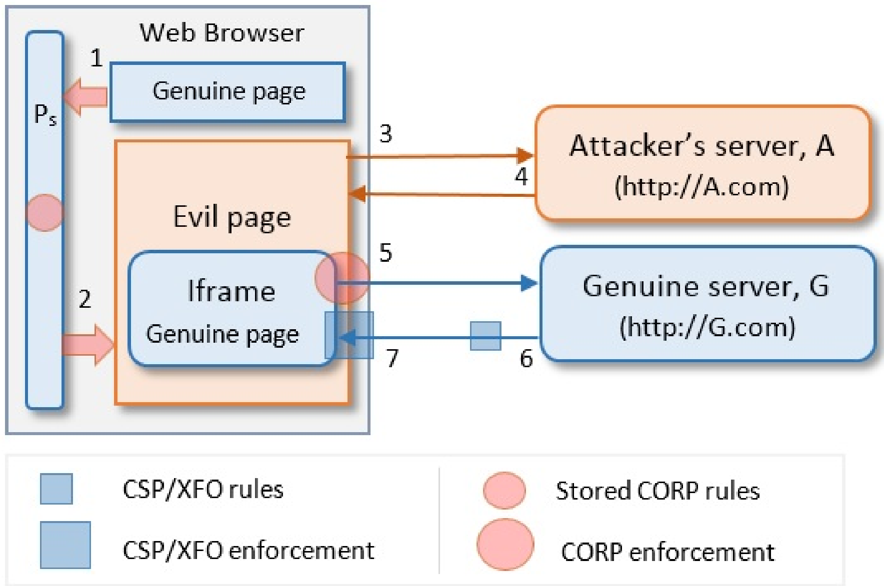


Fig. 3. Browser model showing the enforcement of Clickjacking defense in CSP/XFO and CORP

```
*        iframe       *        DENY
```

Listing 1.7. Defeating clickjacking with CORP

Controlling Social Engineering Attacks. Attackers attempt several social engineering tricks on end users by leveraging popups [41], iframes [42,43] 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 1.8. Controlling social engineering attacks

Listing 1.8 shows a sample CORP configuration, which blocks vectors for social engineering. 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.

Defeating Cross-Site Timing Attacks. The vectors for cross-site timing attacks are same as that of CSRF, as discussed in Section 2.3. 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 requests to authenticated resources such as private pictures and URLs before they leave the browser, thereby defeating cross-site timing attacks. Listing 1.9 shows a sample CORP configuration for the same.

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

Listing 1.9. Defeating cross-site timing with CORP

5 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.

5.1 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 browser's memory using *HTML5 localstorage* API. The storage is accessible across tabs of the browser and policies set by multiple websites are stored and retrieved using the *origin* of the site as the key. When a genuine, CORP-enabled site (G) is opened in one tab and an attacker's site (A) makes a cross origin request to G, the extension intercepts every outgoing request from A if it is made to the origin of G and checks the policy associated with it. 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* [44] event helps in the interception process. It is fired before any TCP connection is made and can be used to cancel requests.

5.2 Experiments

We have conducted several experiments to evaluate the soundness of CORP, its ease of deployment and effectiveness.

Validating the Soundness of CORP: We have used Alloy [16], a finite state model finder, to formalize and verify the soundness of our proposal, CORP. We have modelled cross origin web interactions and came up with predicates which show instances of web infiltration attacks. We verified that on configuring CORP, Alloy fails to produce attack instances. Details about the formal model of CORP shall be provided at a different venue.

Evaluating CORP against a Corpus of Attacks: We have built a web application which is vulnerable to web infiltration attacks and a malicious web application which can launch attacks on the vulnerable application. We have referred to the test suite created by De Ryck et al. [31] and added their CSRF attack vectors to the malicious web application. We have also added vectors for clickjacking and timing to the application. As in the general browsing scenario, if a genuine user logs in at the vulnerable application in one tab, opens the malicious application in another tab and interacts with it, malicious requests (GET and POST) will be triggered which affect the vulnerable application adversely. On configuring CORP headers on the vulnerable web application and enabling the extension, all malicious cross origin calls will be blocked.

The chrome extension, vulnerable and malicious web applications can be accessed online and the attacks discussed in the paper can be replayed before and after installing the extension. Source code is available on Github [45].

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 1) and CORP-enabled them. Instead of deploying vulnerable versions of these applications and fixing them with CORP, we chose to deploy 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 4.4. We first confirmed that these applications implement at least one of the popular defenses against each of the web infiltration attacks (Section 3). 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 1 shows the number of rules needed to enable CORP on each of the applications, without reorganizing resources on the server. With proper segregation of resources, the number of rules can be brought down to less than 10 per application.

Table 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

Analyzing Adherence of Top Websites to CORP: We have analyzed the home page traffic of Alexa [46] Top 15,000 websites, to find if they adhere to CORP by segregating their content based on types. The following content types were considered for analysis - images, css, scripts, html and flash. Figure 4 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.

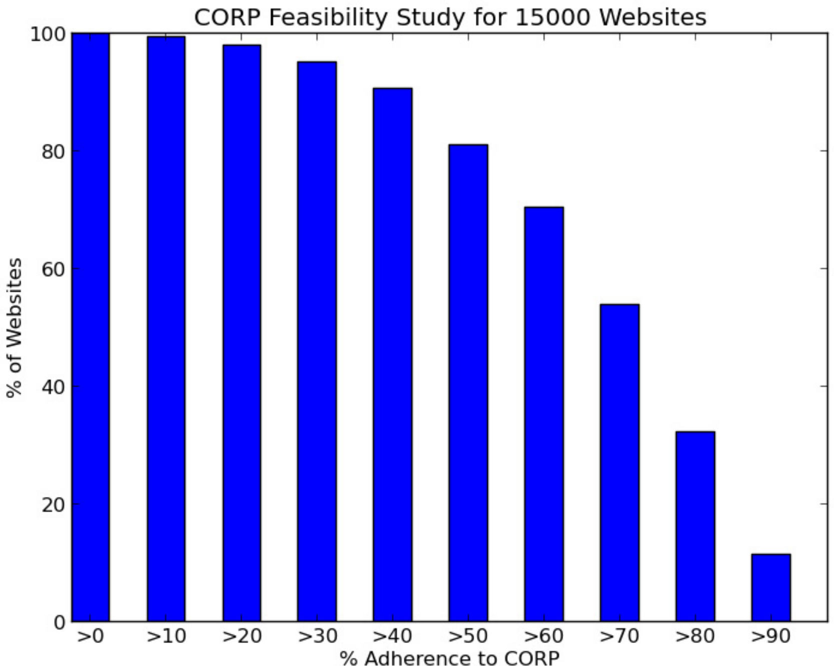


Fig. 4. Bar chart showing adherence of Alexa Top 15,000 websites to CORP

6 Conclusion and Future Work

HTTP works at a level of abstraction that cannot anticipate the semantics of the transaction or of the resource sought by a client. Declarative policies like CSP and CORP fill this semantic gap by conveying to the browser *who* (origins) can access *what* (resources) and *how* (events) as a result of a transaction. 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.

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 `Bank.com` or force the browser to a canonical configuration e.g., disable browser extensions while visiting `Bank.com`. As future work, we plan to explore and expand the class of browser event types specifiable by declarative policies and study their impact on usability and security. Browsers for other form factors like mobiles and tablets present other challenges. We plan to experiment the implementation of declarative policies on these platforms.

Acknowledgements. We thank Kaushik Srinivasan and Akshat Khandelwal for their assistance in analyzing the traffic of 15,000 websites; Amulya Sri for her assistance in implementing CORP on open source web applications.

References

1. W3C: History of the World Wide Web. Technical report (1989), http://www.w3.org/Consortium/facts#history
2. Pilgrim, M.: Dive into HTML5. Technical report, http://diveintohtml5.info/past.html#history-of-the-img-element
3. Berners-Lee, T., Connolly, D.: Hypertext Markup Language – 2.0. Technical Report RFC1866, W3C (1995), http://tools.ietf.org/html/rfc1866
4. Jim, T., Swamy, N., Hicks, M.: Defeating script injection attacks with browser-enforced embedded policies. In: Proceedings of the 16th International Conference on World Wide Web, pp. 601–610. ACM (2007)
5. OWASP: XSS Prevention Cheat Sheet, https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
6. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In: NDSS (2007)
7. Jayaraman, K., Du, W., Rajagopalan, B., Chapin, S.J.: Escudo: A fine-grained protection model for web browsers. In: 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS), pp. 231–240. IEEE (2010)
8. Wikipedia: Netscape navigator 2 (1995), http://en.wikipedia.org/wiki/Netscape_Navigator_2
9. Zalewski, M.: Browser Security Handbook. Technical report (2011), https://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy
10. Stamm, S., Sterne, B., Markham, G.: Reining in the web with content security policy. In: Proceedings of the 19th International Conference on World Wide Web, pp. 921–930. ACM (2010)
11. OWASP: CSRF Guard (2007), https://www.owasp.org/index.php/CSRF_Guard
12. Bortz, A., Boneh, D.: Exposing private information by timing web applications. In: Proceedings of the 16th International Conference on World Wide Web, pp. 621–628. ACM (2007)
13. 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

14. Maone, G., Huang, D.L.S., Gondrom, T., Hill, B.: 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
15. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: 2010 23rd IEEE Computer Security Foundations Symposium (CSF), pp. 290–304. IEEE (2010)
16. Jackson, D.: Software Abstractions: Logic. Language, and Analysis. The MIT Press (2006)
17. OWASP: OWASP Top Ten Project, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
18. Hansen, R., Grossman, J.: Clickjacking. Blog (December 2008), http://www.sectheory.com/clickjacking.htm
19. Facebook: Facebook, Washington State AG target clickjackers. Blog (January 2012), https://www.facebook.com/notes/facebook-security/facebook-washington-state-ag-target-clickjackers/10150494427000766
20. Stone, P.: Pixel perfect timing attacks with html5 (2013), http://contextis.com/files/Browser_Timing_Attacks.pdf
21. Kotcher, R., Pei, Y., Jumde, P.: Stealing cross-origin pixels: Timing attacks on css filters and shaders (2013), http://www.robertkotcher.com/pdf/TimingAttacks.pdf
22. Jeremiah, G.: Introducing the 'I Know...' series. Blog (October 2012), https://blog.whitehatsec.com/introducing-the-i-know-series/
23. Heiderich, M.: CSRFx (2007), https://code.google.com/p/csrfx/
24. Jovanovic, N., Kirda, E., Kruegel, C.: Preventing cross site request forgery attacks. In: Securecomm and Workshops, pp. 1–10. IEEE (2006)
25. Oda, T., Wurster, G., van Oorschot, P., Somayaji, A.: SOMA: Mutual approval for included content in web pages. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 89–98. ACM (2008)
26. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 75–88. ACM (2008)
27. AdBlockPlus: HTTP Referer (2008), http://adblockplus.org/blog/http-referer-header-wont-help-you-with-csrf
28. Johns, M., Winter, J.: RequestRodeo: Client side protection against session riding. In: Proceedings of the OWASP Europe 2006 Conference (2006)
29. Mao, Z., Li, N., Molloy, I.: Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 238–255. Springer, Heidelberg (2009)
30. De Ryck, P., Desmet, L., Heyman, T., Piessens, F., Joosen, W.: CsFire: Transparent client-side mitigation of malicious cross-domain requests. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 18–34. Springer, Heidelberg (2010)
31. De Ryck, P., Desmet, L., Joosen, W., Piessens, F.: Automatic and precise client-side protection against CSRF attacks. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 100–116. Springer, Heidelberg (2011)
32. Maes, W., Heyman, T., Desmet, L., Joosen, W.: Browser protection against cross-site request forgery. In: Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code, pp. 3–10. ACM (2009)

33. Czeskis, A., Moshchuk, A., Kohno, T., Wang, H.J.: Lightweight server support for browser-based CSRF protection. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 273–284 (2013)
34. Balduzzi, M., Egele, M., Kirda, E., Balzarotti, D., Kruegel, C.: A solution for the automated detection of clickjacking attacks. In: ASIACCS 2010, pp. 135–144. ACM, New York (2010)
35. Maone, G.: Hello ClearClick, goodbye clickjacking! Blog (October 2008), http://hackademix.net/2008/10/08/hello-clearclick-goodbye-clickjacking/
36. Rydstedt, G., Bursztein, E., Boneh, D., Jackson, C.: Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In: IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010) (2010)
37. Huang, L.S., Moshchuk, A., Wang, H.J., Schechter, S., Jackson, C.: Clickjacking: Attacks and Defenses. In: USENIX Security Symposium (2012)
38. Huang, L., Jackson, C.: Clickjacking attacks unresolved. White paper, CyLab (2011), http://mayscript.com/blog/david/clickjacking-attacks-unresolved
39. Lekies, S., Heiderich, M., Appelt, D., Holz, T., Johns, M.: On the fragility and limitations of current browser-provided clickjacking protection schemes. In: Woot 2012, USENIX Security Symposium. USENIX (2012)
40. Hodges: RFC 6797, HTTP Strict Transport Security (HSTS) (November 2012), http://tools.ietf.org/html/rfc6797
41. Telikicherla, K.C.: Analyzing the new social engineering spam on facebook - lady with an axe. Blog post (June 2013), http://bit.ly/FBSpamAxe
42. Nafeez, A.: 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
43. Kotowicz, K.: Cross domain content extraction with fake captcha, http://blog.kotowicz.net/2011/07/cross-domain-content-extraction-with.html
44. Google: Life cycle of requests in Chrome.webRequest API (2013), http://developer.chrome.com/extensions/webRequest.html
45. Telikicherla, K.C.: CORP repository (October 2013), http://iiithyd-websec.github.io/corp/
46. Alexa: Alexa top sites (October 2013), http://www.alexa.com/topsites