# A Server- and Browser-Transparent CSRF Defense for Web 2.0 Applications*

Riccardo Pelizzi and R. Sekar
Stony Brook University

## ABSTRACT

Cross-Site Request Forgery (CSRF) vulnerabilities constitute one of the most serious web application vulnerabilities, ranking fourth in the CWE/SANS Top 25 Most Dangerous Software Errors. By exploiting this vulnerability, an attacker can submit requests to a web application using a victim user's credentials. A successful attack can lead to compromised accounts, stolen bank funds or information leaks. This paper presents a new server-side defense against CSRF attacks. Our solution, called jCSRF, operates as a server-side proxy, and does not require any server or browser modifications. Thus, it can be deployed by a site administrator without requiring access to web application source code, or the need to understand it. Moreover, protection is achieved without requiring web-site users to make use of a specific browser or a browser plug-in. Unlike previous server-side solutions, jCSRF addresses two key aspects of Web 2.0: extensive use of client-side scripts that can create requests to URLs that do not appear in the HTML page returned to the client; and services provided by two or more collaborating web sites that need to make cross-domain requests.

## 1. INTRODUCTION

The stateless nature of HTTP necessitates mechanisms for maintaining authentication credentials across multiple HTTP requests. Most web applications rely on cookies for this purpose: on a successful login, a web application sets a cookie that serves as the authentication credential for future requests from the user's browser. As long as this login session is active, the user is no longer required to authenticate herself; instead, the user's browser automatically sends this cookie (and all other cookies set by the same server) with every request to the same web server.

The same origin policy (SOP), enforced by browsers, ensures confidentiality of cookies: in particular, it prevents one web site (say, `evil.com`) from reading or writing cookies for another site (say, `bank.com`). However, browsers enforce no restrictions on outgoing requests: if a user visits an `evil.com` web page, possibly because of an enticing email (see Figure 1), a script on this page can send a request to `bank.com`. Moreover, the user's browser will automatically include `bank.com`'s cookies with this request, thus enabling Cross-site Request Forgery (CSRF). A CSRF attack thus enables one site to "forge" a user's request to another site. Using this attack, `evil.com` may be able to transfer money from the user's account to the attacker's account [26]. Alternatively, `evil.com` may be able to reconfigure a firewall protecting a home or local area network, allowing it to connect to vulnerable services on this network [2, 3, 6].

Since CSRF attacks involve cross-domain requests, a web application can thwart them by ensuring that every sensitive request originates from its own pages. One easy way to do this is to rely on the `Referrer` header of an incoming HTTP request. This information is supplied by a browser and cannot be changed by scripts, and can thus provide a basis for verifying the domain of the page that originated a request. Unfortunately, referrer header is often suppressed by browsers, client-side proxies or network equipment due to privacy concerns [1]. An alternative to the `Referrer` header, called `Origin` header [1], has been proposed to mitigate these privacy concerns, but this header is not supported by most browsers. As a result, it becomes the responsibility of a web application developer to implement mechanisms to verify the originating web page of a request.

A common technique for identifying a same-origin request is to associate a nonce with each web page, and ensuring that all requests from this page will supply this nonce as one of the parameters. Since the SOP prevents attackers from reading the content of pages from other domains, they are unable to obtain the nonce value and include it in a request, thus providing a way to filter them out. Many web application frameworks further simplify the incorporation of this technique [25, 9, 14]. Nevertheless, it is ultimately the responsibility of a web application developer to incorporate these mechanisms. Unfortunately, some web application developers are not aware of CSRF threats and may not use these CSRF prevention techniques. Even when the developer is aware of CSRF, such a manual process is prone to programmer errors — a programmer may forget to include the checks for one of the pages, or may omit it because of a mistaken belief that a particular request is not vulnerable. As a result, CSRF vulnerabilities are one of the most commonly reported web application vulnerabilities, and is listed as the fourth most important software vulnerability in the

*This work was supported in part by ONR grant N000140710928, NSF grant CNS-0831298, and AFOSR grant FA9550-09-1-0539.

*ACSAC* '11 Dec. 5-9, 2011, Orlando, Florida USA
Copyright 2011 ACM 978-1-4503-0672-0/11/12 ...$10.00.

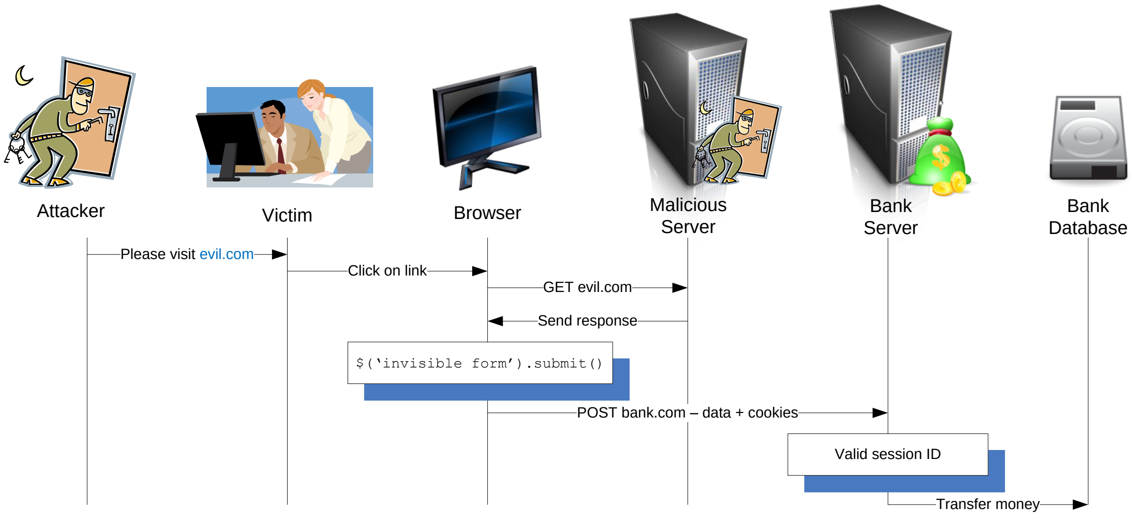

**Figure 1: Illustration of a CSRF attack**

CWE Top 25 list [7].

Prompted by the prevalence of CSRF vulnerabilities and their potential impact, researchers have developed techniques to retrofit CSRF protection into existing applications. NoForge [17] implements CSRF protection using the same basic nonce-based approach outlined earlier. On the server-side, it intercepts every page sent to a client, and rewrites URLs found on the page (including hyperlinks and form destinations) so that they supply the nonce when requested. RequestRodeo [16] is conceptually similar but is deployed on the client-side rather than the server side. Unfortunately, since these techniques rely on static rewriting of link names, they don't work well with Web 2.0 applications that construct web pages dynamically on the browser. More generally, existing CSRF defenses suffer from one or more of the following drawbacks:

1. *Need for programmer effort and/or server-side modifications.* Many existing defenses are designed to be used by programmers during the software development phase. In addition to requiring programmer effort, they are often specific to a development language or server environment. More importantly, they cannot be deployed by a site administrator or operator that doesn't have access to application source code, or the resources to undertake code modifications.

2. *Incompatibility with existing browsers.* Some techniques require browser modifications to provide additional information (e.g., the referrer or origin [1] header), while others rely on browsers enforcement of policies on cross-origin requests (e.g., NoScript [19], CsFire [8], SOMA [20], RequestRodeo [16]). These approaches thus leave server administrators at the mercy of browser vendors and users, who may or may not be willing to adopt these browser modifications.

3. *Inability to protect dynamically generated requests.* Existing server-side defenses, including NoForge [17], CSRFMagic [25], and CSRFGuard [23], do not work with requests that are dynamically created as a result of JavaScript execution on a browser.

4. *Lack of support for legitimate cross-origin requests.* Previous server-side token-based schemes similar to NoForge are aimed at identifying same-origin requests. However, there are many instances where one domain may trust another, and want to permit cross-origin requests from that domain. Such cross-origin requests are not supported by existing server-side solutions, and there does not seem to be any natural way to extend them to achieve this.

We therefore present a new approach for CSRF defense that does not suffer from any of the above drawbacks. Our solution, called jCSRF, is implemented in the form of a server-side proxy. Note that on web servers such as Apache that support a plug-in architecture, jCSRF can be implemented as a web server module, thus avoiding the drawbacks associated with proxies such as additional performance overheads and HTTPS compatibility.

jCSRF operates by interposing transparently on the communication between clients and servers, modifying them as needed to protect against CSRF attacks. As a server-side proxy, it avoids any need for server-side changes. jCSRF also avoids client-side changes by implementing client-side processing using a script that it injects into outgoing pages. It can protect requests for resources that are already present in the web page served to a client, as well as requests that are dynamically constructed subsequently within the browser by scripts. Finally, it incorporates a new protocol that enables support of legitimate cross-domain requests.

jCSRF protects all POST requests automatically, without any programmer effort, but as we describe later, it is difficult (for our technique and those of others) to protect against GET-based CSRF without some programmer effort. Moreover, GET-requests are supposed to be free of side-effects as per RFC2616 [11], in which case they won't be vulnerable to CSRF. For these reasons, jCSRF currently does not protect against GET-based CSRF.

## 2. APPROACH OVERVIEW

As described before, the essence of CSRF is a request to a web server that originates from an unauthorized page. We use the terms *target server,* and *protected server* to refer to such a server that is targeted for a CSRF attack. An authorized page is one that is from the same web server ("same-origin request"), or from a second server that is deemed acceptable by this server ("authorized cross-origin request"). In the former case, no special configuration of jCSRF is needed, but in the latter case, we envision the use of a configurable whitelist of authorized sources for a cross-origin request.

We have implemented jCSRF as a server-side proxy, but it can also be implemented as a server-side module for webservers that support modules, such as Apache. This proxy is transparent to web applications as well as clients (webbrowsers), and implements a server- and browser-independent method to check if the origin of a request is authorized. Conceptually, this authorization check involves three steps:

- In the first step, an authentication token is issued to pages served by the protected server.
- In the second step, a request is submitted to jCSRF, together with the authentication token.
- In the third step, jCSRF uses the authentication token to verify that the page from which the request originated is an authorized page. If so, the request is forwarded to the web server. Otherwise, the request is forwarded to the server *after stripping all the cookies.*

Note that stripping off all cookies will cause an authentication failure within the web application, except for requests requiring no authentication, e.g., access to the login page of the web application, or another informational page that contains no user-specific information. Thus, jCSRF is secure by design and will prevent CSRF attacks. Specifically, its security relies only on three factors: unforgeability of authentication tokens, secure binding between the token and the original page, and the correctness of the authorization policy used in the third step. Other design or implementation errors may lead to false positives (i.e., legitimate requests being denied) but not false negatives.

Note that conceptually, the first two steps are similar to those used in previous defenses such as NoForge [17]. Thus, the key novelty in our approach is the design of protocols and mechanisms that ensure that CSRF protection can be achieved for:

- *dynamically created requests:* requests that are constructed as a result of script execution on the client (web-browser). Such requests are common in Web 2.0 applications using AJAX.
- *cross-origin requests:* requests from a web page served by one web site $A$ to another website $B$, provided $B$ trusts $A$ for this purpose.

When requests are dynamically created, the strategy used by NoForge of statically rewriting the links (to include an authentication token) is not applicable. We have therefore developed a new approach that uses injected JavaScript to carry out this function. In particular, when a page is served by a web application, jCSRF injects some JavaScript code, called jCSRF-script, into this page. On the browser, jCSRF-script is responsible for obtaining the authentication token, and supplying it together with every request originating from this page. By comparing the domain of the current page and the domain of a request, this script can distinguish between same-origin and cross-origin requests, and use different means to obtain the authentication tokens in each case.

We also point out that a static rewriting strategy does not provide a way to validate cross-origin requests. In particular, if a server $A$ embeds a cross-origin request for server $B$ in its page, then the client would need a token for accessing $B$, but the server $A$ has no easy way to obtain such a token. Note that it cannot directly request such a token from $B$ since the token would have to be bound in some way to the *user's cookies* for $B$, and $A$ has no access to these cookies. In contrast, we develop a protocol that can support cross-origin requests naturally.

From a conceptual point of view, jCSRF approach can be applied to both GET and POST requests. However, in practice, the "authorized origin" constraint, which forms the basis of all CSRF defense mechanisms, should not be imposed on many GET-requests. Examples include (a) login pages and other pages that contain no security-sensitive data, and (b) book-marked pages, which may or may not contain sensitive data. Application-specific configuration would be required to list such *landing pages* (case (a)) for each application, and except them from authorization checks. Handling case (b) would require some level of browser cooperation, something we do not assume in our work. Moreover, since it is recommended practice to avoid side-effects in GET-request (as per RFC2616 [11]), they are less likely to be vulnerable as compared to POST-requests. Finally, certain HTML elements such as `img` and `frame` cause the browser to issue a GET request before jCSRF-script has a chance to add the authentication token, requiring jCSRF-script to resubmit the requests for these elements and complicate its logic. For these reasons, in our current implementation of jCSRF, GET requests are not subjected to the "authorized origin" constraint.

Below, we provide more details on the key steps in jCSRF.

## 2.1 Injecting jCSRF-script into web pages

When a page is served by a protected server, jCSRF-proxy automatically injects jCSRF-script into the page. This can be done without having to perform the complex task of parsing full HTML. Instead, the new script is added by inserting a line of the form

```
<script type="text/javascript" src=...  ></script>
```

immediately after the `<head>` tag. Also, jCSRF-proxy includes a new cookie in the HTTP response (unless one exists already) that can be used by jCSRF-script to authenticate same-origin requests. The rest of the page is neither examined nor modified by jCSRF-proxy. As a result, the proxy does not know whether the page contains any cross-origin (or same-origin) requests. It is left to the jCSRF-script to determine on the client-side whether a request being submitted is a same-origin or cross-origin request.

If jCSRF-proxy is implemented as a stand-alone proxy, then it may not be easy to handle HTTPS requests as the proxy will now intercept encrypted content. Although this can potentially be rectified by terminating the SSL sessions at the proxy, a simpler and more preferable alternative is to implement the proxy's logic as a module within the web server.

## 2.2 Protocol for Validating Requests

Although there is just a single protocol that uses different mechanisms to validate cross-origin and same-origin requests, it is easier to describe them separately. We first

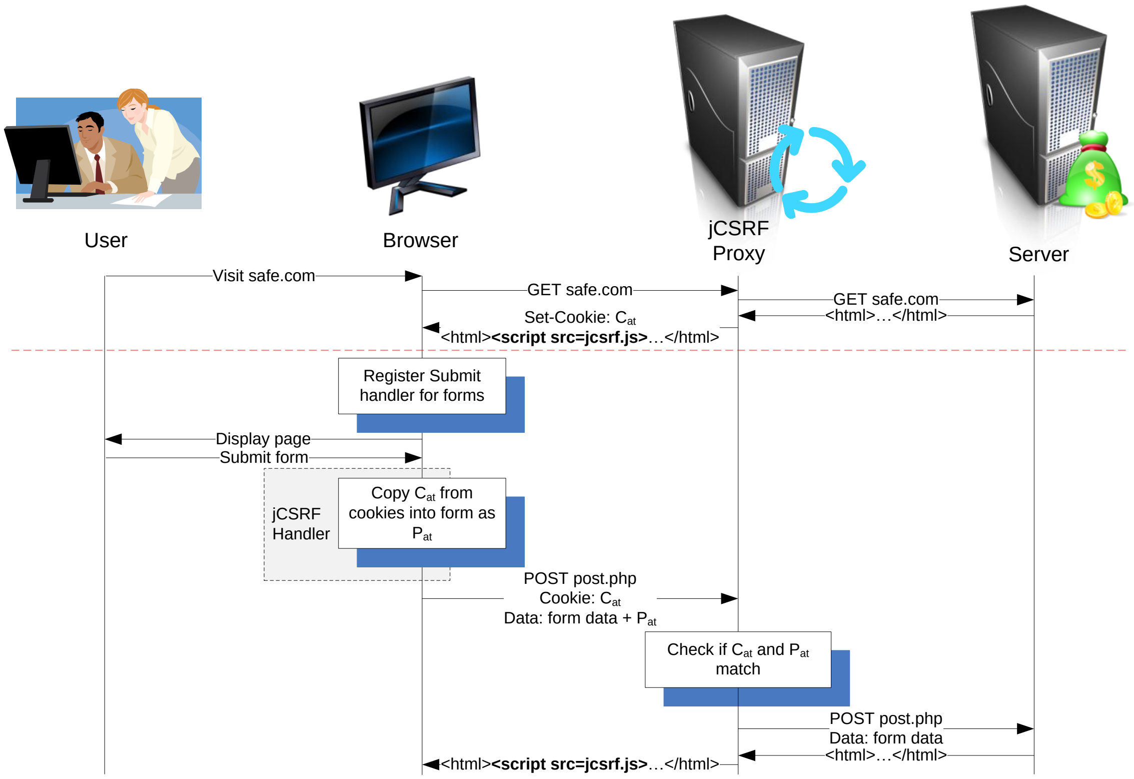
User
Browser
jCSRF Proxy
Server
Visit safe.com
GET safe.com
GET safe.com
<html>…</html>
Set-Cookie: $C_{at}$
<html><script src=jcsrf.js>…</html>
Register Submit handler for forms
Display page
Submit form
jCSRF Handler
Copy $C_{at}$ from cookies into form as $P_{at}$
POST post.php
Cookie: $C_{at}$
Data: form data + $P_{at}$
Check if $C_{at}$ and $P_{at}$ match
POST post.php
Data: form data
<html>…</html>
<html><script src=jcsrf.js>…</html>

**Figure 2: Same-Origin Protocol Workflow**

describe the same-origin validation since it is easier to understand, and then proceed to describe the cross-origin case.

### 2.2.1 *Same-Origin Protocol.*

The same-origin protocol, illustrated in Figure 2, is a simple stateless protocol which authenticates same-origin requests. Red dotted lines in the figure demarcate request-response cycles.

Initially, an authentication token needs to be issued to authorized pages. Since jCSRF permits POST requests only from authorized pages, the very first request from a user has to be a GET request. Such a request is characterized by the fact that a cookie $C_{at}$ used by jCSRF is not set. The server's response to this request is modified by jCSRF-proxy to set this cookie to a cryptographically secure random value. In addition, jCSRF-proxy also injects jCSRF-script in the response as previously described. When this page is received by the browser, jCSRF-script executes, and will ensure (as described further in Section 2.3) that the value of $C_{at}$ is copied into a new parameter $P_{at}$ for all requests originating from this page.

Note that all pages returned by a protected server are modified as above, not just the initial GET request. As such, subsequent requests can provide $C_{at}$ as well as $P_{at}$. This information is then used in the second step of the protocol in Figure 2 to validate POST requests. In particular, jCSRF-proxy checks if $P_{at} = C_{at}$, and if so, the request is forwarded to the server after stripping out $P_{at}$. A missing $P_{at}$ or $C_{at}$, or if $P_{at} \neq C_{at}$, it is deemed an unauthorized request. In this case, jCSRF-proxy strips off all cookies before the request is forwarded to the server. Since web applications typically use cookies to store authentication data, this ensures that the request will be accepted only if it requires no authentication. Note that cross-origin GET requests can be limited in the same way as POST requests, but for reasons described before, the current implementation of jCSRF does not do so.

#### *Correctness.*

In order to protect against CSRF, this protocol needs to guarantee the following properties:

- scripts running on an attacker-controlled page visited by user's browser cannot obtain the authentication token for the protected domain.
- any token that may be obtained by the attacker, say, using his own browser, cannot be used to authenticate a request from user's browser to the protected domain
- the attacker should not be able to guess an authentication token that is valid for the protected domain

The first property is immediate from the SOP: since the authentication token is stored as a cookie, attacker's code running on the user's browser runs on a different domain and has no access to it.

The second property holds because the attacker, apart from being prevented by the SOP from reading the token, is also prevented from setting the token. Therefore, any token obtained by the attacker and embedded into forms sent by the user would not match the cookie that jCSRF-proxy previously set for the user.

The third property is ensured by the fact that the authentication token is randomly chosen from a reasonably large keyspace. Specifically, jCSRF-proxy for a server $S$ generates $C_{at}$ as follows. First, a 128-bit random value $IR$ is generated from a true random source, such as `/dev/random`. A pseudo-random number generator, seeded with $IR$, is then used to

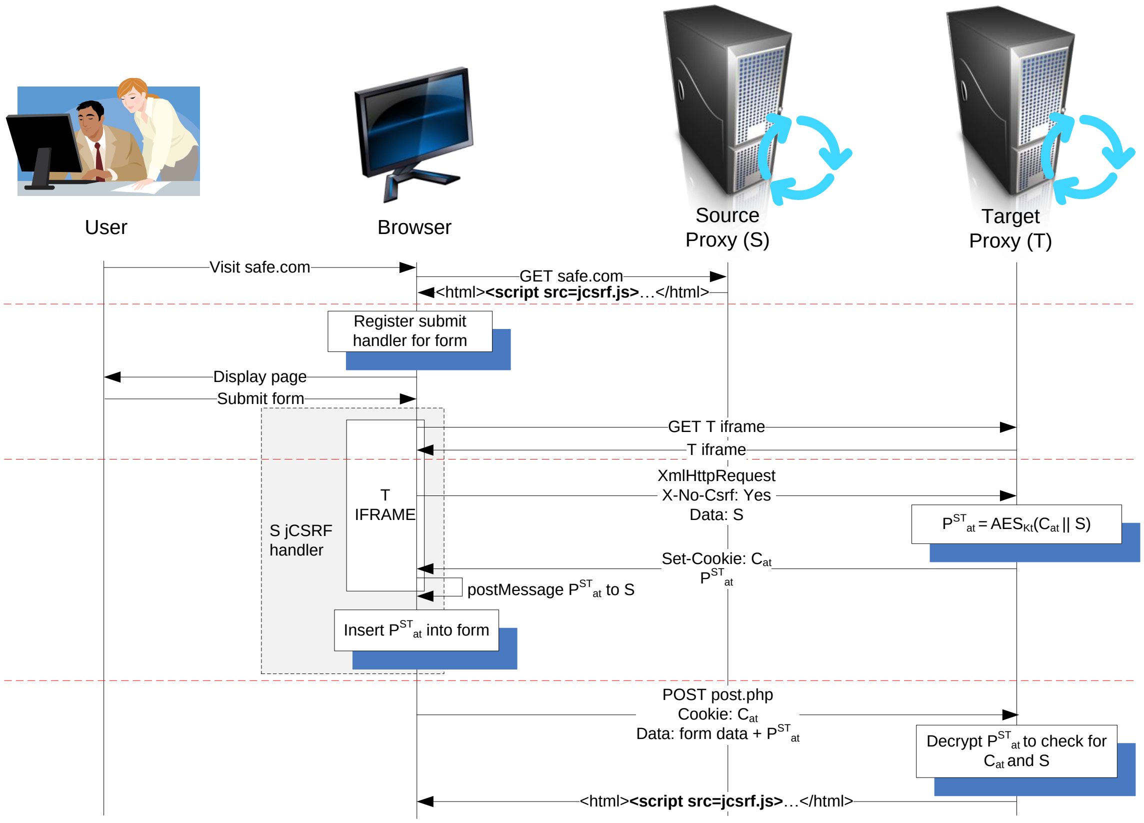
User
Browser
Source Proxy (S)
Target Proxy (T)
Visit safe.com
GET safe.com
<html><script src=jcsrf.js>…</html>
Register submit handler for form
Display page
Submit form
GET T iframe
T iframe
S jCSRF handler
T IFRAME
XmlHttpRequest
X-No-Csrf: Yes
Data: S
$P^{ST}_{at} = AES_{Kt}(C_{at} || S)$
Set-Cookie: $C_{at}$
$P^{ST}_{at}$
postMessage $P^{ST}_{at}$ to S
Insert $P^{ST}_{at}$ into form
POST post.php
Cookie: $C_{at}$
Data: form data + $P^{ST}_{at}$
Decrypt $P^{ST}_{at}$ to check for $C_{at}$ and S
<html><script src=jcsrf.js>…</html>

**Figure 3: Cross-Origin Protocol Workflow**

generate a sequence of pseudorandom numbers $R_1, R_2, \ldots$. From these, nonces $N_1, N_2, \ldots$ are generated using secret-key encryption (specifically, the AES algorithm) as follows:

$$K_s = IR,\ N_i = AES_{K_s}(R_i)$$

Whenever jCSRF-proxy receives a request with a missing (or invalid) $C_{at}$, it sets $C_{at}$ to $N_i$ and increments $i$.

Note that this protocol design does not require $N_i$ values to be stored persistently, since the validation check is stateless: jCSRF-proxy simply needs to compare $C_{at}$ and $P_{at}$ values in the submitted request. Hence, if jCSRF-proxy crashes, it can simply start all over, generating a new $IR$ and so on. Similarly, $K_s$ can be refreshed on a periodic basis by setting it to a new random value from `/dev/random`.

### 2.2.2 *Cross-Origin Protocol*

Figure 3 illustrates our protocol that enables pages from a (source) domain $S$ to submit requests to a (target) domain $T$. Note that servers have been omitted to reduce the number of actors involved in the picture. Before describing the specifics of the protocol, note that the mechanism used in the same-origin case cannot be used for cross-origin requests: jCSRF-script runs on the source domain and therefore has no access to the target domain's cookies, which should contain the authentication token for requests to that domain. An obvious approach for overcoming this problem is to have the source domain communicate directly with the target domain to obtain its authentication token, but this is not easy either. In particular, a correct protocol must bind the subset of user's cookies containing security credentials (for domain $T$) to jCSRF's authentication token (also for domain $T$). Unfortunately, jCSRF-proxy, being application-independent, is unaware of which cookies contain user credentials, and hence cannot achieve such a binding on its own. We therefore develop a protocol that exploits browser-side functionality to avoid the need for a new protocol between $S$ and $T$. In this protocol, javascript code executing on the user's browser communicates with $T$ to obtain an authentication token and communicates it to jCSRF-script. This enables jCSRF-script to include the right value of $P_{at}$ when it makes its cross-origin request to $T$.

Note that there may be many instances where the user loads a page from $S$ containing a form for $T$, but never actually submits it. To avoid the overhead of additional communication with $T$ in those instances, the steps for passing $T$'s authentication token to jCSRF-script are performed only when the user submits a cross-domain form. In addition to reducing the overhead, this approach has privacy benefits since $T$ does not get to know each time the user visits a page that allows submitting data to $T$.

The specifics of our cross-domain protocol are as follows. When a POST action is performed on a page from $S$, jCSRF-script checks if the target domain $T$ is different from $S$ and if $T$ accepts authenticated requests. This information can either be supplied by the web administrator of $S$ as a list of jCSRF-compatible origins or detected by attempting to load a special image `jCSRF-image.jpg` from $T$: the `error` and `load` events can be used to detect whether the resource was found. If the host does not support jCSRF, then jCSRF-script simply submits the post to $T$ without any authentication tokens. Otherwise, jCSRF-script injects an iframe into the page for the URL

`http://T/jCSRF-crossdomain.html?domain=S`

This page will contain javascript code that sets up the authentication token $P_{at}^{ST}$ that a page from $S$ can present to $T$. The steps involved in this process are as follows:

- First, the script within the iframe makes an `XmlHttpRequest` to the domain $T$, providing $S$ (given by the parameter `domain` in the above request) as an argument. `XmlHttpRequests` can only be issued to same-origin resources and, unlike ordinary requests, are allowed to include custom HTTP headers. Therefore, a request bearing the custom header `X-No-CSRF` proves to $T$ that the request came from a page served to the user's browser by $T$.

- This `XmlHttpRequest` is served by jCSRF-proxy. If the user's jCSRF cookie (i.e., the cookie $C_{at}$) is not set, it is set by jCSRF-proxy using a nonce value $N_i$ as described for the same origin case. In addition, jCSRF-proxy sends back the authentication token:

  $$P_{at}^{ST} = AES_{K_T}(C_{at}||S)$$

  Here, $K_T$ is a (random-valued) secret key generated for $T$ using the procedure described for the same-origin protocol.

- In the next step, $P_{at}^{ST}$ needs to be passed on jCSRF-script so that it can complete the request to server $T$. This is accomplished using the `postMessage` API, which provides a secure mechanism for the framed script from domain $T$ to communicate with a script from domain $S$. Note that a framing page from a malicious domain $A$ cannot trick the frame from $T$ into sending $P_{at}^{ST}$: `postMessage` can be instructed to deliver the message to a specific target origin which is chosen by $T$. Whenever $T$ is instructed to send $P_{at}^{ST}$, this will be sent to $S$ only, thus preventing $A$ from reading the message.

  Some of the older browsers do not support the `postMessage` API. In that case, a technique called *location hash polling*[1] can be used in its place.

- Once the framing page has received the token, the jCSRF-script from $S$ adds it to the form and submits the POST request to $T$.

- When jCSRF-proxy for domain $T$ receives a POST request, it decrypts it using $K_T$, and checks if the cookie $C_{at}$ included with the POST is a prefix of the decrypted data. If so, it checks if the domain $S$, which represents the remaining part of the decrypted data, is authorized to submit cross-domain POST requests. If so, the request is passed on the server. In all other cases, jCSRF-proxy treats the request as unauthorized, and strips all cookies before it is forwarded to $T$.

### *Correctness.*

Correctness of the cross-domain protocol relies on the same three properties as the same origin protocol:

- scripts running on an attacker-controlled page visited by user's browser cannot obtain the authentication token for the protected domain.

- any token that may be obtained by the attacker, say, using his own browser, cannot be used to authenticate a request from user's browser to the protected domain

- the attacker should not be able to guess an authentication token that is valid for the protected domain

For the first property, note that because of the semantics of the `postMessage` API, an attacker-controlled page can either receive an authentication token that encodes its true domain $A$, or it may lie about its origin and not receive the token at all. In the latter case, the first property obviously holds. In the former case, although there is an authentication token, it contains the true origin of the attacker. On receiving this token, jCSRF-proxy will deny the request, as the attacker domain $A$ is not authorized to make cross-origin posts.

The second property holds because the attacker is unable to set (or read) the value of user's cookie $C_{at}$ for the protected domain $T$. Thus, even if he obtains an authentication token $P$ by interacting with $T$ using his own browser, he cannot use it with user's cookie $C_{at}$ that will have a different value from the cookie value sent to the attacker by $T$. (Recall that $T$ uses cryptographically random nonces to initialize $C_{at}$.)

The third property is ensured by the fact that the authentication token is randomly chosen from a reasonably large keyspace.

## 2.3 Design and Operation of jCSRF-script

As noted before, jCSRF-script needs to intercept all POST-requests and add the authentication token as an additional parameter to these requests. There are two ways in which browsers may issue POST requests:

- *Submission of HTML forms,* represented by `form` tags. Note that it is not necessary for the page to contain a `form` tag, because the form can be constructed dynamically using Javascript. Also, it is not necessary for the user to submit the form explicitly, because the form can be submitted automatically using Javascript.

- *`XmlHttpRequest` submissions.* Unlike form submissions, where the response cannot be accessed by the submitting script, the response to `XmlHttpRequest` can be read by the script making the request.

Compatibility requires handling both types of primitives. We now describe how jCSRF-script achieves this.

### *2.3.1 HTML Form Submission*

Modern browsers allow Javascript code to register callbacks for specific events concerning the web page presented to the user. These functions are called *event handlers.* To ensure that every form is submitted to the web application with an authentication token, jCSRF-script registers a *submit* handler for each POST-based form. This handler then checks if the submission is to the same-origin or cross-origin.

[1] In location hash polling [12], a framing page sends its URL to a framed page as a parameter. The framed page can then append a token to the URL of the framed page using an anchor at the end of the URL. The basis for this technique is that this URL change does not cause the framing page to reload; instead the value appended to the URL is available for polling. If a malicious page from domain $A$ lies about its URL (pretending to be a page from $S$), then the update will cause the outer page to reload from domain $S$, thus defeating the attempt by $A$ to read data written by $T$.

| Application | Version | LOC | Type | Compatible |
|---|---|---|---|---|
| phpMyAdmin | 3.3.7 | 196K | MySQL Administration Tool | Yes |
| SquirrelMail | 1.4.21 | 35K | WebMail | Yes |
| punBB | 1.3 | 25K | Bulletin Board | Yes |
| WordPress | 3.0.1 | 87K | Content-Management System | Yes |
| Drupal | 6.18 | 20K | Content-Management System | Yes |
| MediaWiki | 1.15.5 | 548K | Content-Management System | Yes |
| phpBB | 3.0.7 | 150K | Bulletin Board | Yes |

**Figure 4: Compatibility Testing**

In the former case, jCSRF-script simply adds the authentication token as an additional parameter to the POST request. In the latter case, it uses the cross-domain protocol to first obtain a token for the target domain, and then adds this token as an additional parameter to the POST request.

Note that the web application might define its own event handlers for the submission event, mostly to validate the form contents. If the web application handler ran after jCSRF's handler, it would have access to the authentication token. In some rare cases, the presence of the token might confuse the web application handler which only expects a predefined set of fields. Therefore, jCSRF-script detects if the web application defines its own handlers and wraps them with a function which removes the token before calling the web application handler, reinserting the token afterward. There are two types of event handlers: DOM0 handlers (registered with the HTML attribute `onSubmit` or by assigning a function to the JavaScript property `form.submit`) and DOM2 handlers (registered with the `addEventListener` function). The former type of handler is detected by periodically checking all forms for new, unwrapped submit handlers, which can be done through the previously mentioned `submit` property of form elements, since handlers are JavaScript functions and functions are first-class objects in JavaScript. The latter type requires overriding the `addEventListener` method to directly wrap new handlers during their registration, since there is no way to query the set of listeners registered for a specific (event, object) pair.

### 2.3.2 *XmlHttpRequest*

For `XmlHttpRequests`, jCSRF-script modifies the `send` method of the class. For a browser supporting DOM prototypes [13], this can be done simply by substituting the `send` function, while on older browsers it is done by completely wrapping `XmlHttpRequest` functionality in a proxy object that hides the original class, and redirects all requests made by the web application to the proxy class. As explained in 2.2.2, adding a special header `X-No-CSRF` is enough to prove that the request is same-origin and therefore safe.

### 2.3.3 *Compatibility*

jCSRF-script uses jQuery's `live` method [24] to reliably interpose on submission of dynamically generated forms: instead of binding an event handler to a specific DOM element at call time, `live` registers a special handler for the root element which is then invoked once the event that fired on one of its descendants bubbles up the DOM tree. The purpose of the special handler is to find the element responsible for the event, check whether it matches the element type specified to `live` and apply the event handler supplied to `live` to it. jCSRF-script can thus bind its handler to the submit event for all future forms by calling

```
$('form').live('submit', handler)
```

Overriding `addEventListener` requires DOM prototypes support, which is not available on old browsers (IE7 and older). On these browsers, only DOM0 events can be wrapped.

Even though we did not encounter this scenario in any of the web application we tested, the techniques used to wrap DOM0 and DOM2 handlers may not work properly if the web application (or another software similar to jCSRF which transforms the HTML output) is using them as well. For example, if another piece of code other than jCSRF-script polls forms for the presence of submit handlers, a race condition can ensue: either the two wrapper functions are composed in a nondeterministic fashion, or one wrapper is overwritten by a subsequent attempt to wrap the function by the second piece of code.

Note that failure to wrap an event handler does not necessarily imply a compatibility issue with jCSRF: most web applications define their own handlers to predicate on specific form fields, enforcing constraints such as "the field `age` must be a number". Only handlers that predicate on classes of fields might be incompatible with jCSRF on older browsers. For example, the constraint "all fields must be shorter than 10 characters" could create a problem for the token field if the handler is not wrapped.

# 3. EVALUATION

## 3.1 Compatibility

To verify that jCSRF is compatible with existing applications, we deployed popular open-source Web applications and accessed them through the proxy, checking for false positives by manually testing their core functionality. We tested jCSRF with two browsers (Firefox and Google Chrome) and the following applications: phpMyAdmin, SquirrelMail, punBB, Wordpress, Drupal, Mediawiki, and phpBB. As Figure 4 shows, these are complex web applications consisting of thousands of lines of code that would require substantial developer effort to audit and fix CSRF vulnerabilities. jCSRF was able to protect all applications without breaking their functionality in any way.

Note that these web applications did not perform cross-origin requests, and therefore our evaluation did not cover the cross-origin protocol. Nevertheless, we believe that the primary source of incompatibility in the cross-origin protocol will remain the same as the same-origin protocol, namely, reliably interposing on submit events. As a result, we expect the compatibility results for the cross-origin protocol to be similar to those shown in Figure 4.

It is worth mentioning that jCSRF requires JavaScript enabled. If it is disabled, say, through the use of a browser extension such as NoScript [19], then requests would be sent out unauthenticated, resulting in false positives.

## 3.2 Protection

To test the protection offered by jCSRF, we selected 2 known CVE vulnerabilities and attempted to exploit them. The results are summarized in Figure 5.

First, we exploited the CVE-2009-4076 [4] vulnerability on the open source web mail application RoundCube [22]. Emails are sent using a POST request, but its origin is not authenticated. We built an attack page on an external website that fills out and submits an email message. jCSRF successfully blocked the attack, because the POST request was missing the authentication token. Second, we exploited the CVE-2009-4906 [5] vulnerability on the Acc PHP eMail web application. This vulnerability allows changing the admin password with a POST request from an external website. jCSRF was able to thwart this attack as well.

We limited our evaluation to two because the effectiveness of jCSRF does not need to be established purely through testing. Instead, we have provided systematic arguments as to why the design is secure against CSRF attacks. A secondary factor was that reproducing vulnerabilities is a very time-consuming task, and can be further complicated by difficulties in obtaining specific software versions that are vulnerable, and dependencies on particular configurations of applications, operating systems, etc.

Finally, two attacks are out of scope for a tool such as jCSRF, but should be mentioned for completeness: XSS attacks and same-domain CSRF attacks. XSS attacks can be used to break the assumption that same-origin scripts are under the control of the web developer, to issue token requests and leak results to the attacker, thus defeating the purpose of jCSRF. We point out that a successful XSS attack grants the attacker far more serious capabilities than the ability to craft requests on the victim's browser using his cookies. In fact, the attacker can simply steal the cookies directly and send authenticated requests as the victim from his own machine! To our knowledge, no other server-side CSRF defense can resist in case of an XSS attack. Same-origin CSRF attacks can be carried out by injecting a form in a server response and tricking the user into submitting it. jCSRF-script would add the correct authentication token, because it has no way to realize that the form present in the DOM tree was indeed supplied by the attacker [27].

## 3.3 Performance

In this section, we estimate the overhead imposed by jCSRF. A page embedding jCSRF-script issues three different type of requests to its target jCSRF-proxy:

1. GET requests. For these, the browser does not perform any special processing, and thus incurs no overhead. On the server-side, jCSRF-proxy only needs to generate a new token if the user does not have one already.

2. Same-Origin POST requests. Before the actual submission, jCSRF-script copies the authentication token $C_{at}$ from the cookies to the form as $P_{at}$. Therefore, no overhead is introduced on the client, and the proxy only needs to check that $C_{at} = P_{at}$, which is an inexpensive operation.

3. Cross-Origin POST requests. The cross-origin protocol requires three additional GET requests for authentication: one to detect whether the target web application is running jCSRF, one to fetch the iframe from it that requests the token and one for the actual `XmlHttpRequest` that fetches the token. Therefore, this additional network delay dominates any other delay introduced by token generation and verification by the proxy. Although this overhead is non-negligible, we point out that cross-origin POST requests make up only a small fraction of HTTP requests [18], and therefore the delay due to these roundtrips is not likely to affect the overall user browsing experience.

We built a simple web application, deployed it locally and compared the response time of unprotected vs. protected same-origin and cross-origin POST requests. jCSRF protection incurred at most 2ms overhead.

# 4. RELATED WORK

## 4.1 Server-side Defenses

NoForge [17] was the first approach that used tokens to ascertain same-origin requests without requiring modifications to the application's source code. Implemented as a server-side proxy, NoForge parses HTML pages served by a web server, and adds a token to every URL referring to this server. It associates this token with the cookie representing the session id for the application. When a subsequent GET- or POST-request is received, it checks if this request contains the token corresponding to the session id. jCSRF is clearly influenced by NoForge, but makes several significant improvements over it:

- NoForge requires developer help to specify the name of the cookie containing the session id. Not only is this effort unnecessary in our approach, but it is also the case that our technique is compatible with alternative schemes for authentication, such as those that store authentication credentials in multiple cookies, or schemes that support persistent logins that, at different times, may be associated with different session ids.

  Moreover, NoForge needs to maintain server-side state in the form of valid (session id, token) pairs. In contrast, jCSRF does not maintain state, and is less prone to DoS attacks.

- jCSRF supports web sites where URLs are dynamically created by client-side execution of scripts.

- jCSRF supports cross-origin requests whereas NoForge can only protect same-origin requests. NoForge's approach does not easily extend to cross-origin case since it relies on a mapping between cookies and tokens on the server side. In the cross-origin case, the cookies that are visible to the origin and target domains are different, and so it is unclear how the states maintained on the two domains can be correlated.

An important difference between NoForge and jCSRF is that the former protects GET-requests as well. However, as discussed before, there are a number of difficulties in CSRF protection for GET-requests: inability to bookmark pages, need for developer effort to identify "landing pages" that do not need CSRF protection (which are not supported by NoForge), and so on. In the interest of providing a simple, fully automated solution, jCSRF protects only POST-requests.

Bayawak [15] can be thought of as extending NoForge to enforce a stronger policy: URLs in the web application are

| Application | Version | LOC | Type | CVE | Stopped |
|---|---|---|---|---|---|
| RoundCube | 0.2.2 | 54K | Webmail | CVE-2009-4076 | Yes |
| Acc PHP eMail | 1.1 | 3K | Mailing List Manager | CVE-2009-4906 | Yes |

**Figure 5: Protection Evaluation**

augmented with a special token not only to ensure that the request is same-origin, but also to constrain the order in which web pages can be visited. As such, it also protects against *workflow attacks* that aim to disrupt the session integrity by sending out-of-sync requests. This increased protection is obtained at the cost of additional programmer effort needed to specify permissible workflows. Bayawak does not address cross-origin requests or URLs that are dynamically created on the client-side.

X-PROTECT [27] is a server side defense that employs white-box analysis and source code transformation to overcome the shortcomings of other black-box approaches, namely their inability to protect against same-origin CSRF and their need to specify landing pages manually.

### *4.1.1 Developer Tools*

Most web frameworks for rapid application development [9, 14, 21, 10, 26] include functionality to simplify CSRF protection, typically using a NoForge-like approach. For example, Django [9], a Python-based framework, provides CSRF protection for POST requests by requiring a specific template tag to be added to HTML forms, which is translated to a hidden form field containing a token that is also returned through cookies. To check whether the token in the cookies and the form match before executing the application logic, Django provides function wrappers to instrument *views* (python functions associated to URLs). CSRFMagic [25] provides a similar capability for PHP applications. Web developers need to include an import statement in their PHP files to activate this protection. The purpose of the included file is to register output and input filters. The output filter executes before the HTML page is sent to the client, and adds a token to POST forms. The input filter checks for the presence of this token.

CSRFGuard [23] is similar to CSRFMagic, but is designed for Java EE applications. It examines incoming GET and POST requests for the presence of a valid token. CSRFGuard introduced an option for client-side insertion of tokens using a script. Although this appears similar to our technique of injecting jCSRF-script, its operation is different. In particular, their client-side script adds the token to content available when the page fires the `load` event, and hence does not handle requests that may be dynamically constructed by various scripts associated with this page. Moreover, unlike NoForge, it allows web developers to configure a set of landing pages that do not require a valid token, thus mitigating the usability issues related to GET-request protection at the cost of additional developer effort.

CSRF tools for developers are an invaluable resource for rapid application development. Their benefit is that they provide a finer granularity of control for programmers, as compared to fully automated approaches such as jCSRF. The main drawback of developer tools is the need for programmer effort. Moreover, programmers may overlook to add checks in all places they are required, thus leaving vulnerabilities.

The basic idea of comparing a token and cookie value to verify same origin requests is similar between these approaches and jCSRF. However jCSRF goes beyond these tools by (a) providing support for cross-origin requests, and (b) supporting requests to URLs that are generated dynamically on the client-side.

## 4.2 Browser Defenses

Zeller and Felten [26] present a Firefox plugin which implements a simple policy to prevent POST-based CSRF: cross-site POST requests must be authorized by the user. The drawback of this simple approach is the fatigue stemming from repeated user prompts. NoScript [19] implements a more sophisticated policy that can avoid prompts. In particular, it restricts only those POST requests that go from an untrusted origin to a trusted origin. NoScript primarily targets sophisticated, security-conscious users who are willing to put in the effort needed to populate their list of trusted origins.

De Ryck et al [8] presents a CSRF protection plugin for Firefox, CsFire. It studies cross-domain interactions on the web, and uses the results to design a cross-domain policy that protects against CSRF attacks while optimizing compatibility with existing web applications. This policy relies on the concept of *relaxed SOP*, which allows communication between subdomains of the same registered domain (e.g. `mail.google.com` and `news.google.com`). Their policy also introduces the idea of *direct interaction*: since CsFire is a browser plugin, it has access to UI information such as whether a request was initiated by a user click. Cross-Origin GET requests initiated by user clicks are allowed, while cross-origin POST requests are not allowed in any case. Instead of blocking the request altogether, the plugins strips the cookies from the request, which are necessary to carry out a successful CSRF attack.

RequestRodeo [16] differs from the above techniques in that it is implemented outside of a browser as a client-side proxy. It relies on an approach similar to NoForge, but rather than blocking a request, it simply strips all cookies from such requests. Another difference is that it has no exceptions for landing pages. This can significantly affect usability.

A key advantage of browser-side defense is that it protects users even if web sites are not prompt in fixing their vulnerabilities. Moreover, they have accurate information about the origin of requests, whether they result from clicking on a bookmark, or a link on a web page trusted by a user. Their primary drawback is that the defense is applied to all web sites and pages, regardless of whether they have any significant security impact. Such indiscriminate application significantly increases the odds of false positives. Moreover, it is easier for a server-side solution residing on a target domain to determine whether it trusts the origin domain of a request. In contrast, browser-based defenses require the user to make this determination, and moreover, do it for all origins and target domains.

## 4.3 Hybrid Defenses

These defenses require both browser and server modifications. Referrer headers are the best known mechanism in this context. Using this HTTP header, a web browser can provide the crucial information that servers lack: namely, the origin domain of the current request. Given this information, a server can implement a simple CSRF protection mechanism that denies requests from domains that it does not trust. Unfortunately, due to privacy concerns, it is common to suppress referrer headers [1]. Barth et al [1] proposed a new header, called the *origin header*, to overcome these privacy concerns by suppressing some of the information provided by referrer headers, e.g., the query string. Currently, only Webkit-based browsers implement this header. Moreover, there may be lingering privacy concerns even with this origin header.

SOMA [20] is an alternate approach that aims to address a range of threats, including CSRF and XSS. With SOMA, a target domain is able to specify the set of allowable origin domains and vice-versa. Its implementation relies on a browser plug-in that retrieves these policies from the source and target domain and disallows any cross-origin requests that violate either policy.

Hybrid defenses often represent the best solutions that can be achieved by bringing together the information and mechanism that are available on browsers as well as webservers. Their key drawback is that both sides have to be modified simultaneously in order to achieve their benefits. For this reason, their adoption can take a long time. For instance, the origin header represents a relatively modest change, but even after about 3 years since that proposal was made, there is just a single major browser that supports it.

## 5. CONCLUSIONS

We introduced jCSRF, a tool to transparently protect web applications against CSRF attacks without requiring source code changes or configuration. Unlike similar solutions which modify all outgoing HTML responses to contain tokens, jCSRF uses JavaScript to augment requests dynamically. This allows jCSRF to also handle requests to resources that are not directly served by the protected web application, but rather generated dynamically on the browser. Moreover, jCSRF provides a protocol to authenticate cross-origin requests, which extends the applicability of the tool to complex, multi-domain deployments. Because it is implemented as a proxy, it should pose no compatibility problems with any web application, regardless of the language used or the web server it runs on. Our evaluation results show that jCSRF is a practical solution for automatically protecting web applications against CSRF attacks.

## 6. ACKNOWLEDGMENTS

We thank Siddhi Tadpatrikar for her work on implementing jCSRF-script for the same origin protocol.

## 7. REFERENCES

[1] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS*, 2008.

[2] CVE Editorial Board. CVE-2007-3574. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-3574, 2007.

[3] CVE Editorial Board. CVE-2009-2073. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2073, 2009.

[4] CVE Editorial Board. CVE-2009-4076. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-4076, 2009.

[5] CVE Editorial Board. CVE-2009-4906. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-4906, 2009.

[6] CVE Editorial Board. CVE-2010-1482. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1482, 2010.

[7] CWE and SANS Institute. 2010 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/, March 2011.

[8] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *ESSOS*, 2010.

[9] Django Software Foundation. Django. http://www.djangoproject.com, 2011.

[10] EllisLab Inc. Code Igniter. http://codeigniter.com/, 2002.

[11] R. Fielding et al. Hypertext Transfer Protocol – HTTP/1.1. http://www.ietf.org/rfc/rfc2616.txt, 1999.

[12] J. Fraser. Backwards compatible window.postMessage(). http://www.onlineaspect.com/2010/01/15/backwards-compatible-postmessage/, 2010.

[13] F. Guisset. JavaScript-DOM Prototypes in Mozilla. https://developer.mozilla.org/en/JavaScript-DOM_Prototypes_in_Mozilla, 2002.

[14] D. H. Hansson. Ruby on Rails. http://rubyonrails.org, 2011.

[15] K. Jayaraman, G. Lewandowski, P. Talaga, and S. Chapin. Enforcing Request Integrity in Web Applications. *Data and Applications Security and Privacy*, 2010.

[16] M. Johns and J. Winter. RequestRodeo : Client Side Protection against Session Riding. In *OWASP Europe*, 2006.

[17] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Securecomm*, 2007.

[18] W. Maes, T. Heyman, L. Desmet, and W. Joosen. Browser protection against cross-site request forgery. In *SecuCode*, 2009.

[19] G. Maone. NoScript. http://noscript.net/, 2011.

[20] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. SOMA: Mutual approval for included content in web pages. In *CCS*, 2008.

[21] Pylons. Pylons Project. http://pylonsproject.org/, 2011.

[22] RoundCube.net. RoundCube - Free Webmail for the Masses. http://roundcube.net/, 2010.

[23] E. Sheridan. OWASP: CSRFGuard Project. https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project, 2011.

[24] The jQuery Project. .live() - jQuery API. http://api.jquery.com/live/, 2011.

[25] E. Z. Yang. CSRFMagic. http://csrf.htmlpurifier.org/, 2008.

[26] W. Zeller and E. Felten. Cross-site request forgeries: Exploitation and prevention, 2008.

[27] M. Zhou, P. Bisht, and V. Venkatakrishnan. Strengthening XSRF Defenses for Legacy Web Applications Using Whitebox Analysis and Transformation. In *ICISS*, 2011.