Browser Protection against Cross-Site Request Forgery

Wim Maes, Thomas Heyman, Lieven Desmet, Wouter Joosen
IBBT-DistriNet
Katholieke Universiteit Leuven
3001 Leuven, Belgium
{Thomas.Heyman,Lieven.Desmet}@cs.kuleuven.be

ABSTRACT

As businesses are opening up to the web, securing their web applications becomes paramount. Nevertheless, the number of web application attacks is constantly increasing. Cross-Site Request Forgery (CSRF) is one of the more serious threats to web applications that gained a lot of attention lately. It allows an attacker to perform malicious authorized actions originating in the end-users browser, without his knowledge. This paper presents a client-side policy enforcement framework to transparently protect the end-user against CSRF. To do so, the framework monitors all outgoing web requests within the browser and enforces a configurable cross-domain policy. The default policy is carefully selected to transparently operate in a web 2.0 context. In addition, the paper also proposes an optional server-side policy to improve the accuracy of the client-side policy enforcement. A prototype is implemented as a Firefox extension, and is thoroughly evaluated in a web 2.0 context.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; D.4.6 [**Operating Systems**]: Security and Protection; H.4.3 [**Information Systems Applications**]: Communications Applications—*Information browsers*

General Terms

Security

Keywords

Cross-Site Request Forgery, Run-time policy enforcement, Web application security

1. INTRODUCTION

Cross-Site Request Forgery (CSRF) is a web application attack vector with which an attacker forces an unwitting user's browser to perform actions on a third party website, possibly reusing all cached authentication credentials of that user. In 2007, CSRF was listed as one of the most serious web application vulnerability in the OWASP Top Ten [18]. In 2008, Zeller and Felten documented a number of serious CSRF vulnerabilities in high-profile websites, among which a vulnerability on the home banking website of ING Direct, which allowed an attacker to transfer funds from any user account to an account chosen by the attacker [25].

One of the root causes of CSRF is the abuse of cached credentials in cross-domain requests. Web applications can easily trigger new requests to web applications in a different trust domain without any user intervention. This results in the browser sending out cross-domain requests, while implicitly using credentials cached in the browser (such as cookies, SSL certificates or login/password pairs). From a server point of view, these implicitly authenticated requests are legitimate and are requested on behalf of the user. The user, however, is not aware that he sent out those requests, nor that he approved them.

Currently, a whole range of techniques exist to mitigate CSRF, either protecting the server application or protecting the end-user (e.g. via a browser plugin or a client-side proxy). However, the server-side protection mechanisms are not yet widely adopted, and most of the client-side mitigation techniques only provide limited protection, or do not scale well to web 2.0 applications. As a result, even the most cautious web user is unable to appropriately protect himself against CSRF. Therefore, it is necessary to construct more robust client-side protection techniques against CSRF, capable of dealing with current and next-generation web applications.

This paper presents the following contributions. First, it describes a client-side mitigation technique against CSRF that intercepts outgoing requests within the browser and enforces a configurable cross-domain policy. This technique improves upon the state-of-the-art as it is able to use contextual information within the browser and selectively strip sensitive information from outgoing requests. Second, this paper proposes expressive, server-side cross-domain policies to improve the accuracy of client-side policy enforcement. Third, an proof-of-concept of the proposed technique is implemented as a Firefox extension, and is thoroughly evaluated in a web 2.0 context.

The remainder of this paper is structured as follows. Section 2 provides some background information on CSRF, explores the current state-of-the-art and defines the requirements of client-side mitigation solutions. Next, our solution

SecuCode'09, November 9, 2009, Chicago, Illinois, USA.

to transparently mitigate CSRF within the browser is presented in Section 3, as well as the underlying cross-domain policy. Section 4 describes our proof-of-concept implementation in Firefox, and discusses some of the implementation challenges. In addition, the presented approach and the proof-of-concept implementation are evaluated in Section 5. In Section 6, the presented work is related to alternative mitigation techniques and, finally, Section 7 summarizes the contributions of this paper.

2. BACKGROUND

This section provides some background information on CSRF and available countermeasures. Section 2.1 introduces CSRF and its enabling technologies in a nutshell. Next, Section 2.2 provides an overview of existing countermeasures, and identifies the gaps in the countermeasure landscape. Finally, Section 2.3 defines the requirements for client-side mitigation against CSRF.

2.1 Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) has been around for almost a decade, and is also known as XSRF, Confused Deputy, one-click-attack, Session Riding, ... CSRF is an attack specific to web applications, and allows an attacker to perform authorized actions on a third party website on behalf of an unwitting user. To do so, CSRF takes advantage of implicit authentication mechanisms built on top of HTTP and cached credentials in the browser.

HTTP is a stateless client-server protocol [10], which uses certain methods to transfer data between web servers and browsers. The two most frequently used HTTP methods are GET and POST. Conforming to the HTTP specification, GET methods are used to fetch data from the server, whereas POST methods are used to update server state. In practice however, GET and POST methods are used interchangeably, and both can trigger server-side state changes.

Because of the stateless nature of HTTP, session management is build on top of HTTP. This is typically done by means of cookies, which are small amounts of session-specific data, created by the server application and stored in the browser. Alternative approaches such as URL rewriting or hidden form parameters exist as well [20]. This session-specific data is sent back with each request to that particular web application, without any user intervention.

Similarly, HTTP basic authentication attaches encoded credentials to each individual request to enable server-side authentication and authorization. For user convenience, the credentials are typically cached within the browser, and are only requested once for the lifespan of the browser session. Other (albeit lesser used) authentication schemes include client side SSL and IP-based access control [13]. The latter is mainly used on intranets.

Cross-Site Request Forgery abuses the trust of a web application in the cached browser-provided user credentials. CSRF is an attack whereby "the attacker disrupts the integrity of the user's session with a website by injecting network requests in the browser" [7]. In a CSRF attack, the browser of the user is tricked into performing a request to a particular web application. If the user is authenticated to the target web application, this can result in malicious requests reusing the credentials of the user. Strictly speaking, CSRF can occur within the same domain, e.g. due to a cross-site scripting vulnerability or when multiple users can host a web site within the same domain (universities, ISP's, ...). This paper, however, focuses on CSRF between different domains.

To perform a successful CSRF attack, a number of conditions need to be met:

1. The target website must use implicit authentication, such as through cookies or HTTP authentication, as mentioned above.
2. The targeted user must already have been authenticated to the target web application, and the user's browser must have cached the authentication credentials.
3. An attacker forces the user's browser to make an HTTP request to the target web application.

When all three conditions are met, the browser will automatically add the implicit authentication, making the request appear as a legitimate one to the target server.

A multitude of vectors can be used to trigger HTTP requests including certain HTML tags, CSS attributes, HTTP headers and javascript. For example, an HTTP GET request can be launched by setting the `src` attribute of an `img` tag. Similarly, an HTTP POST request can be launched using a form that is automatically submitted by javascript. A more elaborate discussion of the different vectors can be found in Section 5.

2.2 Existing countermeasures

A number of techniques exist to protect against CSRF, by either protecting the server application or by protecting the end-user. Unfortunately, the server-side protection mechanisms are not yet widely adopted, and most of the client-side mitigation techniques only provide limited protection or do not scale well to web 2.0 applications.

Client-side countermeasures.

The most widespread countermeasure is the Same Origin Policy (SOP) [24], implemented in most browsers. This policy limits access to DOM properties and methods to scripts from the same 'origin', where origin is usually defined as the triple <domain name, protocol, tcp port>[1]. This prevents a malicious script on the website `evil.com` from reading out session identifiers stored in a cookie from `homebanking.com`, for example. Unfortunately, the protection offered by the SOP is insufficient. Although the SOP prevents the requesting script from accessing the cookies or DOM properties of a page from another origin, it does not prevent an attacker from making *requests* to other origins. The attacker can still trigger new requests and use cached credentials, even though the SOP prevents the attacker from processing responses sent back from the server.

On top of SOP, client-side countermeasures exist to monitor and filter cross-domain requests. They typically operate as a client-side proxy [13] or as an extension in the browser [21, 25]. These countermeasures monitor outgoing requests and incoming responses, and filter out implicit authentication or block cross-domain requests. Unfortunately, these client-side mitigation techniques typically scale not well to web 2.0 applications, suffer from degraded surfing

[1]This definition of 'origin' will be used throughout the remainder of this paper.

experience, or fail to protect against GET-based CSRF attacks. More details on the client-side mitigation techniques and their limitations can be found in Section 6.

Server-side countermeasures.

A number of server-side mitigation techniques exists as well. The most popular class of server-side CSRF protection is the use of secret tokens [14, 19, 9]. Each response sent by the server embeds a secret token into the web page (e.g. as a hidden parameter in an HTTP form). For each incoming request, the server verifies that the received token originated from the server, and is correctly bound to the user's session. Since the SOP prevents the attacker from reading responses from other origins, this is an effective server-side countermeasure.

Another class of server-side countermeasure relies on the HTTP `referer` header to determine the origin of the request. Unfortunately, quite often browsers or proxies block this header for privacy reasons [7]. Furthermore, an attacker can spoof HTTP headers, e.g. via Flash or request smuggling [15, 16].

More intrusively, Barth et al. propose an additional HTTP header to indicate the origin of the request [7]. This header should not be removed by the browser or proxies, and is less privacy-intrusive than the referer. In addition to such an `origin` header, W3C proposes to add additional headers to the responses as well, to give server-side cross-domain information to the browser [23].

2.3 Requirements for client-side protection

Current server-side mitigation techniques (as presented in the previous Section) can provide adequate protection against CSRF attacks. Unfortunately, they are not widely deployed yet. End-users can not rely on server-side protection mechanisms alone, as they are often missing, and have no choice but to fall back on client-side solutions.

As stated before, the quality and applicability of the client-side countermeasures is still inadequate. The client-side mechanisms are necessarily generic, as they have to work for every web application, in every application domain. This usually makes them too coarse grained, often resulting in too permissive or too restrictive enforcement. In addition, most countermeasures do not handle current technologies such as javascript and AJAX well.

Therefore, we propose the following requirements for a client-side CSRF solution in a contemporary web 2.0 context.

R1. The client-side protection should *not depend on user input*. Nowadays, a substantial fraction of web requests in an average browsing session is cross-domain (see Section 5). It is infeasible for the user to be called upon and validate requests. Furthermore, users can not be expected to know which third parties a web application needs to function correctly. Therefore, a transparent operation is essential.

R2. The protection mechanism should be *usable in a web 2.0 context*, without noticeable service degradation. The solution should support the dynamic interaction behavior of today's applications (i.e., 'mashups'), and embrace current and future web technologies. Ajax applications and mashup-based portals are becoming the norm instead of the exception.

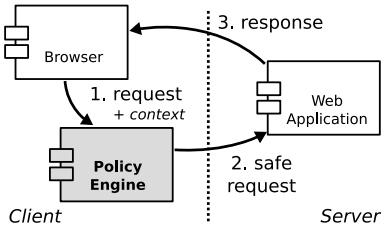

Figure 1: Request interception.

R3. *Secure by default.* The solution should not have false negatives using its default configuration.

The ideal CSRF protection combines the best of both worlds—offering a generic client-side solution, which can take extra application-specific, server-side context into account whenever this is available.

3. CLIENT-SIDE POLICY ENFORCEMENT

This Section describes our client side CSRF protection mechanism. It is implemented as a policy enforcement framework, described in Section 3.1, which intercepts cross-domain HTTP requests and allows, blocks or modifies them depending on the context and the current configuration. Section 3.2 illustrates the working of the enforcement engine and discusses the default cross-domain policy. Section 3.3 discusses and evaluates a more idealized setting, where the server hosting the web application cooperates with the client-side policy enforcement to explicitly define trusted cross-domain interactions.

3.1 The policy enforcement framework

The conceptual design of the approach is shown in Figure 1. As requests to access remote web application resources are made by the client, they are intercepted and evaluated by a policy engine. The engine uses a configurable policy in determining what requests comprise cross-domain communication and what actions should be taken on cross-domain requests. Its correct operation is based on two trust assumptions, 1) the execution environment of the enforcement framework (i.e., the user's browser) is trusted and 2) the actual policies are trusted, including the client-side policy (Section 3.2) and the server-side policy (Section 3.3).

The policy engine has a number of options to handle intercepted cross-domain requests, i.e.: *query the user*—ask the user for an appropriate action, by means of a dialog window; *allow the request*—do not intervene; *block the request*—stop the request from being executed; *modify the request*—remove data from the request, such as cookies, cached credentials, and extra headers. The first option, querying the user for every cross-domain request, conflicts with the proposed requirements as it severely hampers the usability of the enforcement solution, and is only intended for debugging purposes. Deployment policies should autonomously decide whether to allow cross-domain requests, or to intervene.

The policy engine offers an expressive interface to additional, contextual information, which enables the policy to autonomously decide on the appropriate action. This contextual information includes, among others, the origin of the document that spawned the request, the origin and the resource that is being accessed by the request, whether any

parameters are being passed along with the request, and if any cookies or credentials are attached to the request. Based on this information, the policy engine is able to take appropriate actions.

3.2 Default client-side policy

The default cross-domain policy is carefully selected to transparently operate in a web 2.0 context, and is as follows:

1. The origin policy is relaxed to top-level domains. This conforms to the domain relaxation in javascript, which is commonly used in contemporary web 2.0 applications.
2. All requests originating from the same origin are allowed.
3. Cross-domain POST requests are blocked, but all cross-domain GET requests (with and without parameters) are allowed. However, `cookie` and `authorization` headers are removed from these requests.

This choice prevents CSRF attacks from different origins, as authentication through implicit HTTP-authentication or cookies is defeated (i.e., no false negatives). On the other hand, the impact on the user's web browsing session is minimized (though still present), as the majority of cross-domain requests are not blocked, but modified. In a small number of cases (for details, see Section 5), this results in false positives and a degraded surfing experience. This conforms to the secure by default requirement, but leaves room for improvement in the web 2.0 usability department.

This configuration is encoded in the following policy:

```
domainRelaxation = true
getNoParameters = {allow=true, removeCookie=true,
       removeAuthorizationHeaders=true}
getWithParameters = {allow=true, removeCookie=true,
       removeAuthorizationHeaders=true}
post = {allow=false}
```

False positives, where the authentication credentials are stripped from non-malicious cross-domain requests, are possible, as the policy engine is unable to distinguish legitimate cross-domain requests, such as those often used in mashups, from abusive requests. As both malicious and legitimate cross-domain requests are identical on a technical level, the only way they can be distinguished is by gathering extra domain-specific knowledge per web application. How this extra contextual information can be encoded by the web application developer and used by the policy engine, is described in Section 3.3.

3.3 Cooperating server policy

This section illustrates how a cooperating server *can* provide additional contextual information by means of an additional server-side policy. This server-side input allows the client-side policy engine to distinguish malicious from legitimate cross-domain requests, thereby eliminating false positives all together. The server policy provides application-specific knowledge in the form of *intended cross-domain interactions*, and whether or not domain relaxation should be applied. This contextual information can significantly improve the quality of the client-side enforcement for particular web applications, such as mashup sites. However, the downside of such a server policy, is that the policy is application-specific, and therefore requires additional effort of the web application developer or deployer.

While the server hosting a web application is unable to know which user requests are actually initiated by the user, and which are malicious, it *is* able to enumerate the third parties it trusts to perform certain tasks. For instance, an online web banking application might trust requests issued by an online ticket vending website. If the client-side interceptor is aware of this information, it is able to leave the implicit authentication tokens intact on request to the web banking application from the ticket vending site, even if this request is cross-domain (and would normally be modified by the default client-side policy). As such, the main purpose of the cooperating server policy is to express which cross-domain interactions are allowed.

The cooperating server policy is built around the abstraction of intended cross-domain interaction expressions, which are defined between an array of *origins* (a tuple consisting of a host, port, protocol and path of the initiating web page) and *destinations* (identical to origins, except for the implied 'localhost' hostname). In essence, an intended cross-domain interaction links a group of origins with the samelevel of trust (i.e., subject to the same restrictions) to a number of destinations on that server. For every interaction that crosses the specified domains, the policy writer is able to express whether authentication credentials are allowed to be exchanged via HTTP authentication or stored in cookies, or stripped (the default behaviour). Furthermore, fine-grained control is facilitated by allowing the policy writer to select the type of request (GET with or without parameters, POST) and white list certain cookies (such as user preferences or other functionality-related data), even though authentication credentials are stripped. A JSON syntax for this policy expressed in the ABNF metasyntax language [8] can be found online[2].

The server policy, whenever present, modifies the behavior of the default client-side policy as follows. Cross-domain requests are not allowed (i.e., are modified or blocked), unless the request is matched by an intended cross-domain interaction definition. If multiple intended cross-domain interactions match, the first interaction that applies to an origin is taken into account. If it is, then the restrictions defined in the intended cross-domain interaction apply. The restrictions defined in the intended cross-domain interaction can relax those in the default client policy, by allowing certain named cookies or the authorization header, or by leaving the original request intact. The restrictions on the allowed cookies can be specified both in a white or black list fashion. Furthermore, separate restrictions can be applied on GET requests with parameters, GET requests without parameters, and POST requests. In addition, the server can also tighten the policy by requesting a strict enforcement of the Same Origin Policy, i.e. without the above described domain relaxation. As is the case with the default client-side policy, the overall policy enforcement does not depend on any interaction of the end user.

The server policy can either tighten or relax the behavior of the default client policy. While, in the best case, the server policy sufficiently relaxes the client policy to avoid false positives, it is possible to introduce false negatives (i.e., CSRF attacks) by making mistakes in the definition of the server policy. As such, the server policy places liability for protect-

[2] `http://www.cs.kuleuven.be/~lieven/research/SecuCode2009/serverpolicy-abnf.txt`

ing a web application in the hands of the web application developer.

Example.
An online bank is located on `https://online.bank.com` and cooperates with a ticket service located on `https://www.ticket.com`. Although every website on the Internet may link to the online bank, requests to the page `https://online.bank.com/confirm.php` should only originate from trusted partners, such as `https://www.ticket.com/request.php`. The corresponding cooperating server policy is:

```
{"strictDomainEnforcement": true,
 "intendedCrossDomainInteraction": [
   {"blockHttpAuth": false,
    "blockCookies": false,
    "methods": ["*"],
    "cookieExceptions": [],
    "origins": [{
       "host": "www.ticket.com",
       "port": 443,
       "protocol": "https",
       "path": "/request.php" }],
    "destinations"= [{
       "port": 443,
       "protocol": "https",
       "path": "/confirm.php" }]},
   {"blockHttpAuth": true,
    "blockCookies": true,
    "methods": ["getNoParam"],
    "cookieExceptions": ["language"],
    "origins": [{ "host": "*" }]
   }]}
```

This policy allows the cross-domain request from the ticket service without adaptations. Furthermore all websites are allowed to initiate a get request without parameters to the bank but the `authorization` header and all cookies except for the language `cookie` must be removed.

4. PROOF OF CONCEPT

To implement a client-side CSRF countermeasure, both a proxy or a browser extension can be used. The proxy has the advantage of being browser independent. The browser extension is necessarily browser-dependent, but has the advantage of easy access to crucial contextual information. As such, the proposed approach is implemented as an extension to the (slightly patched) Firefox browser, which is capable of intercepting cross-domain requests and provides a configurable policy to mitigate CSRF threats. The extension is written for Firefox 3 and works well. However, to obtain the full expressiveness of the approach, presented in Section 3, a small modification of the Firefox codebase is necessary, as will be discussed further.

Firefox, one of the more popular browsers, is extensible through an elaborate extension system. Firefox extensions are built around so called XPCOM components. XPCOM, which stands for cross platform component object model, has multiple language bindings (see, a.o., [3]), enabling components written in different languages to be transparently integrated as a Firefox extension. The component interfaces are defined in XPIDL [22], a language-neutral interface definition language. Next to a number of built-in core components, the Mozilla foundation itself provides a lot of XPCOM components.

To realize the implementation of the described technique, two things need to be available within Firefox. First, the necessary request information, such as origin of the requesting page, destination host and path on that host, cached credentials used, needs to be inspected. Second, the extension requires capabilities to block and/or modify the outgoing request. This includes access to the cookies attached to the request, and HTTP headers used.

Fortunately, the Firefox architecture (depicted in Figure 2) facilitates hooking into the request processing flow through two interfaces, i.e., `nsIContentPolicy` and `nsIObserver`. The former offers the `shouldLoad` method, which is called before a resource is loaded. The latter can be used to observe HTTP requests and responses while modifying them. The `nsIContentPolicy` gives access to contextual information, such as the origin URI, the location URI and the initiating DOM node. The `nsIObserver` allows, via the `nsIHTTPChannel` interface, to modify the HTTP headers or cancel the request.

Figure 2 shows the structure of the Mozilla platform, and the impact of the proposed extension (the shaded elements). This impact is localized to XPCOM and Necko. The design of the extension itself separates the interception and the policy enforcement to ensure that policies can easily be added or modified. The XPCOM component is responsible for the policy enforcement and delegates the decision making to a configurable policy. Each policy is implemented independently, and can store policy-specific state. The policies themselves can make specific modifications to a request and can be part of the policy enforcement point.

The server policy is expressed in JSON syntax, which poses a security challenge when evaluated with the javascript `eval` method. Indeed, when the policy file contains javascript code, this code is executed in the browser with the privileges of the extension. However, this can be mitigated by using the `nsIJson` interface for secure JSON processing.

The only limitation in implementing the approach, presented in Section 3, is that Firefox does not give access to the `Authorization` and `Proxy-Authorization` header via the `nsIHTTPChannel`. As access to these headers is necessary to strip `HTTP` authentication, a small modification of Firefox, allowing to strip authentication credentials from a request, was necessary to achieve the fine-grained policies (as used in a cooperating server setting). Alternatively, the privacy mode of Firefox can be used to to strip authentication credentials. The privacy mode is more coarse-grained, since it will strip both authentication credentials as well as cookies, but it is already available in the mainstream codebase.

5. EVALUATION

The evaluation consists of three parts. First, different small CSRF attack-scenarios are used to test the security added by the approach (Section 5.1). Second, real-world surf sessions are used to test if the policies cause functional degradation (Section 5.2). Finally, some metrics are applied to characterize the processed web traffic in Section 5.3, in order to rationalize some of the requirements made in Section 2.3.

5.1 Security evaluation with scenarios

In order to compose a test suite to evaluate the completeness of the implementation with respect to preventing CSRF, the specifications of the HTTP protocol, the HTML and CSS markup languages, and the javascript language were analyzed to identify all possible cross-domain interac-

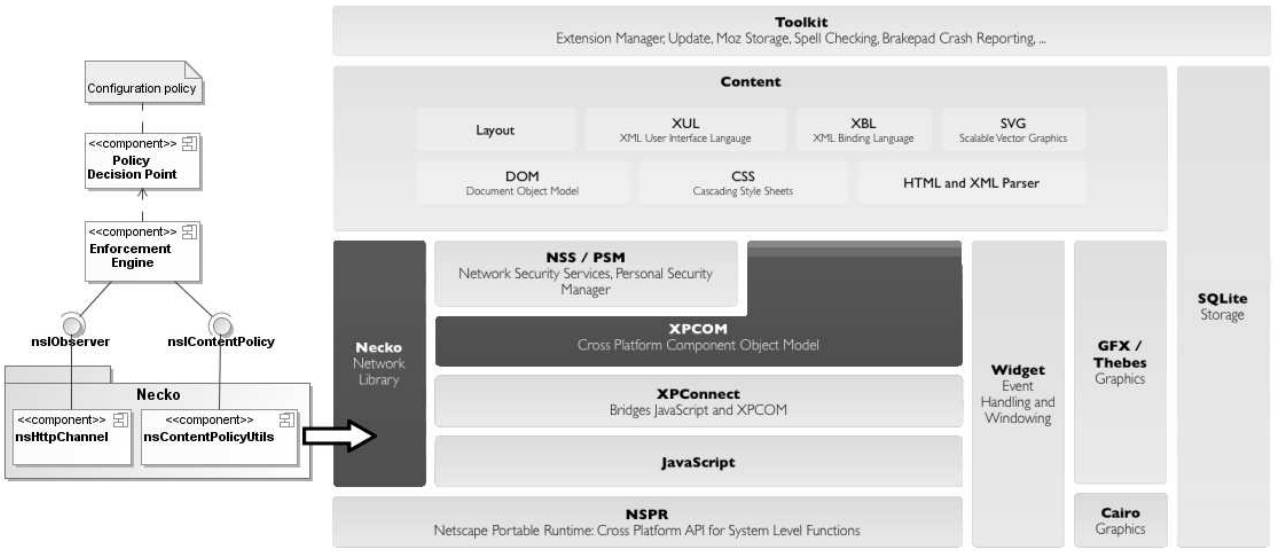

Figure 2: The components on the Mozilla platform (adapted from [11]).

tions. For each identified cross-domain interaction, a representable test was written as a simple web page (using PHP and javascript). This results in 59 different testing scenarios. The following paragraph highlights some of the identified interactions.

According to the HTTP specification [10], the `location` header can be used to stage a CSRF attack. Furthermore, the `refresh` header[3] can be used. For the scenarios related to HTML, the complete HTML 4.01 reference [5] was searched for tags with attributes that might initiate an HTTP request, e.g., the `src` attribute in the `img` tag and the `content` and `http-equiv` attributes in the `meta` tag. The CSS scenarios were listed by searching the W3C CSS reference [4] for all the properties that accept an `url` as their value, e.g. the `cursor` property. With javascript it is possible to change the DOM, thereby changing HTML and CSS content, which could be a possible way of initiating a CSRF attack. The first set of javascript scenarios covers these vectors. In addition, it is possible to change the document location using javascript. Finally, since javascript allows the use of timeouts, all scenarios are repeated with a timeout as well, for completeness.

The scenarios execute a CSRF attack against a mock website using HTTP authentication or a cookie (named *sessionid*, containing a session identifier) for authentication. The proof-of-concept Firefox extension is able to protect against all of the 59 scenarios, using the presented default policy. In addition, policy relaxation with the cooperating server policy was tested using the following server policy:

```
{"intendedCrossDomainInteraction":
  [{"blockHttpAuth": false,
    "blockCookies": false,
    "cookieExceptions": ["sessionid"]
    "origins": [{"host": "*"}]
   }]}
```

[3]The refresh header is not part of the HTTP specification [10], but is supported by Firefox.

The set of implemented test scenario's is solution-independent, so it can also be used to test other client-side or infrastructural CSRF protection mechanisms as well.

5.2 Evaluation in a web 2.0 context

This part of the evaluation uses the default policy to evaluate the functional degradation on a set of representative websites. A list of popular websites is composed, based on the rankings of Netcraft [2] and Alexa [1]. It contains the following sites, generating a substantial amount of cross-domain requests:

- iGoogle (`www.google.com/ig`)
- gmail (`mail.google.com`)
- Facebook (`www.facebook.com`)
- Google image search (`images.google.com`)
- Yahoo (`yahoo.com`)
- Live (`live.com`)
- Msn (`msn.com`)

The different sites were visited intensively, with the proof-of-concept Firefox extension enabled. All web sites except iGoogle show no signs of degradation. iGoogle shows some degradation for particular gadgets. When surfing to iGoogle via, for instance, `www.google.be/ig`, some of the gadgets (including Google Calendar) communicate with `google.com`, and require cookies for authentication. The only way the default policy can resolve this issue — without server cooperation — is by not removing cookies for all cross-domain requests. This is obviously not acceptable when it comes to security. This clearly shows the benefit of having server-side policies expressing intended cross-domain interactions to improve the accuracy of client-side enforcement techniques.

5.3 Analysis of real-world traffic

To obtain better insights in representative web traffic, we have closely monitored browser requests made by ten test persons during a two-week test period. This resulted in

about 1 million requests, of which all contextual information such as the origin, the requester (e.g. image, script, XmlHttpRequest, ...), the target, the HTTP method, whether or not cookies or authentication credentials were attached, whether or not parameters were supplied, the nesting of iframes, ..., were logged.

The high-level results of the real-world traffic analysis are presented in Table 1. The 991,529 requests represent 14,646 unique origin-destination pairs.

	GET	POST	Total
cross-domain requests (strict SOP)	460,899 (46.48%)	2,052 (0.21%)	462,951 (46.69%)
cross-domain requests (relaxed SOP)	291,552 (29.40%)	1,860 (0.19%)	293,412 (29.59%)
All requests	964,028 (97.23%)	27,501 (2.77%)	991,529 (100.00%)

Table 1: Cross domain communication metrics (total number of requests: 991,529

These numbers show that simply blocking cross-domain requests would block 46.69% of the requests, which is clearly unacceptable. Domain relaxation reduces this number to 29.59%. Furthermore, the amount of cross-domain requests is too high to rely on user interaction to facilitate the decision making process, favoring a fully automated decision making approach—such as that of the default client-side policy.

Blocking all cross-domain post requests, which is the case in the default policy, blocks only a small percentage of the requests (0.19%). Further analysis revealed that this percentage represent 27 unique origin-destination pairs, and consists of either illegitimate/unwanted requests (such as an unauthorized post request to paypal), requests between different top-level domains of the same company as identified in Section 5.2, and some gadgets included from `gmodules.com`.

6. RELATED WORK

In this section, some alternative client-side countermeasures against CSRF are discussed, and related to the proposed solution.

RequestRodeo [13] is a client-side proxy offering protection against CSRF. The approach is based on removing implicit authentication. Only requests classified as entitled may carry implicit authentication. A request is classified as entitled when it is initiated as a result of an interaction with the currently viewed web page (including javascript), and the request satisfies the SOP. RequestRodeo covers, next to cookie-based and HTTP authentication, also IP address-based authentication via an external proxy.

To classify requests, RequestRodeo processes incoming responses and augments them with tokens. Only requests carrying such a token are considered to be entitled. Because the classification depends on processing incoming responses, RequestRodeo does not scale well to web 2.0 applications, as it sometimes misses some crucial context information, and is not able to detect all dynamically created links in the responses. In contrast, stripping implicit authentication and using an external proxy as a means of protecting the user against CSRF is considered to be an important contribution of RequestRodeo, and was a direct inspiration of the approach presented in this paper.

A number of Firefox extensions exist to offer client-side CSRF protection within the browser. RequestPolicy [21] is an example of such an extension. The conceptual approach is similar to the client-side policy in this paper, but it has lower protection coverage. For instance, the evaluation of RequestPolicy with the attack scenarios in Section 5 revealed some important gaps, such as the protection against CSRF attacks using forms.

An interesting Firefox extension, proposed by Mao *et al.* is BEAP (formerly known as antiCSRF). This client-side enforcement records, within the browser, the user's intention for each requests (e.g. entering an address in the address bar, or implicitly fetching additional resources such as an image). For sensitive-critical operations (i.e. POST requests, or GET requests with session cookies over HTTPS), the authentication tokens are stripped for requests that do not explicitly reflect the user's intention. Although we agree with of the authors that requests that update server-state should not use the GET method and be send over HTTP, reality shows that web application developers do not adhere well to this practice. Well-known CSRF attacks, such as against `metafilter.com` and `youtube.com`, use GET over HTTP [25]. Therefore we argue the need to protect these requests as well.

CSRF Protector is another extension initially designed and implemented by Zeller and Felten in [25], and further developed by Mehmood. The initial implementation blocked POST requests that violate the SOP or the Adobe flash cross-domain policy (see below). The current implementation, adapted by Mehmood, removes the `cookies` instead of blocking the request. Despite a reasonably complete mitigation of POST CSRFs, the extension does not protect against real-world CSRF attacks via the GET method [25].

Adobe flash.

Adobe flash has a cross-domain policy [6] with server cooperation and is mostly related to the cooperating server solution proposed in this work. Our approach addresses CSRF specifically and has a more fine-grained expressiveness. Furthermore, it allows to remove implicit authentication, whereas Adobe's cross-domain policy focuses on allowed connectivity.

Another recent Firefox extension using server cooperation is SOMA [17], which protects against a large class of unrestricted information flow vulnerabilities, including CSRF. It does this by requiring mutual agreement between two different origins, before content of the first origin can be included in a page hosted on the second origin. Our approach focuses on protecting against CSRF only, making mutual agreement superfluous and policy writing easier—only sensitive cross-domain requests need to be considered. Moreover, our solution can easily be extended to take into account the parts of the SOMA policy relevant to CSRF.

7. DISCUSSION AND CONCLUSIONS

The default client-side policy does not require user intervention, but can cause functional degradation of certain 'mashup' based websites, as the behavior of these mashups is indistinguishable from CSRF attacks on a technical level. However, evaluation shows that our proposed configuration has a limited amount of these false positives. Furthermore, the cooperating server policy extension mitigates this problem. However, the protection offered by the server-side ex-

tension depends on the server policy writer—by relaxing the client-side policy, the server policy is able to reduce the amount of false positives, but it is also able to introduce false negatives if this relaxation is pushed too far.

The proposed implementation protects against CSRF vulnerabilities exploited in the browser, i.e., attacks that abuse CSS properties, HTML entities or HTTP headers to generate CSRF attack requests. Attacks that target lower layers, such as web applications using IP-based or client-side SSL authentication, are out of scope for this work. CSRF can still occur in this context, e.g., when the attacker combines CSRF with a DNS rebinding attack [12]. CSRF attacks within the same domain, which occur for instance when a website is vulnerable to cross-site scripting, are also not covered by this approach. In that case, the attacker can process the responses of the malicious requests (as they do not violate the same-origin policy), resulting in information leakage. Without cross-site scripting protection, CSRF is still a threat. And lastly, CSRF attacks in a browser caused by a request initiated by another application (e.g., a mail client) are also not covered.

To conclude, this paper introduces a novel client-side protection mechanism for CSRF attacks. It is able to operate stand-alone to provide default out-of-the-box protection, which covers (to our knowledge) all types of cross-site request attacks leveraging cached user credentials (both as cookies or implicit HTTP-authentication). Furthermore, a proposed server-side policy can improve the accuracy of the client-side enforcement by incorporating application-specific knowledge.

8. ACKNOWLEDGMENTS

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

9. REFERENCES

[1] Alexa top 500 global sites. `http://www.alexa.com/topsites/global`.

[2] Most visited web sites. `http://toolbar.netcraft.com/stats/topsites`.

[3] XPCOM - MDC. `https://developer.mozilla.org/en/XPCOM`, 2008.

[4] CSS reference. `http://www.w3schools.com/CSS/CSS_reference.asp`, 2009.

[5] HTML 4.01 / XHTML 1.0 Reference. `http://www.w3schools.com/tags/`, 2009.

[6] Adobe. Adobe Flash Player 9 security, July 2008.

[7] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 75–88, 2008.

[8] D. Crocker and P. Overell. Augmented BNF for syntax specifications: ABNF. `http://tools.ietf.org/html/rfc5234`, 2008.

[9] D. Esposito. Take advantage of ASP.NET built-in features to fend off web attacks. `http://msdn.microsoft.com/en-us/library/ms972969.aspx`, January 2005.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1 (rfc2616). `http://tools.ietf.org/html/rfc2616`, 1999.

[11] M. Finkle. Building on the Mozilla platform. Slides Toronto Developer Day, September 2008.

[12] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from DNS rebinding attacks. *ACM Trans. Web*, 3(1):1–26, 2009.

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

[14] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing Cross Site Request Forgery attacks. In *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, Baltimore, MD, USA, August 2006.

[15] A. Klein. Forging HTTP request headers with Flash. `http://www.securityfocus.com/archive/1/441014`, July 2006.

[16] C. Linhart, A. Klein, R. Heled, and S. Orrin. HTTP request smuggling. Technical report, Watchfire, 2005.

[17] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji. SOMA: mutual approval for included content in web pages. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[18] OWASP. The ten most critical web application security vulnerabilities.

[19] OWASP. CSRF Guard. `http://www.owasp.org/index.php/CSRF_Guard`, October 2008.

[20] V. Raghvendra. Session tracking on the web. *Internetworking*, 3(1), 2000.

[21] J. Samuel. Request Policy 0.5.3. `http://www.requestpolicy.com`.

[22] The Mozilla foundation. XPIDL - MDC. `https://developer.mozilla.org/en/XPIDL`, October 2007.

[23] A. van Kesteren. Cross-origin resource sharing. `http://www.w3.org/TR/2009/WD-cors-20090317/`, March 2009.

[24] M. Zalewski. *Browser Security Handbook*. 2008. `http://code.google.com/p/browsersec/wiki/Main`.

[25] W. Zeller and E. W. Felten. Cross-Site Request Forgeries: Exploitation and prevention. Technical report, October 2008. `http://www.freedom-to-tinker.com/sites/default/files/csrf.pdf`.