CsFire: Transparent client-side mitigation of malicious cross-domain requests

Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen

IBBT-DistriNet
Katholieke Universiteit Leuven
3001 Leuven, Belgium
{philippe.deryck,lieven.desmet}@cs.kuleuven.be

Abstract Protecting users in the ubiquitous online world is becoming more and more important, as shown by web application security – or the lack thereof – making the mainstream news. One of the more harmful attacks is cross-site request forgery (CSRF), which allows an attacker to make requests to certain web applications while impersonating the user without their awareness. Existing client-side protection mechanisms do not fully mitigate the problem or have a degrading effect on the browsing experience of the user, especially with web 2.0 techniques such as AJAX, mashups and single sign-on. To fill this gap, this paper makes three contributions: first, a thorough traffic analysis on real-world traffic quantifies the amount of cross-domain traffic and identifies its specific properties. Second, a client-side enforcement policy has been constructed and a Firefox extension, named CsFire (CeaseFire), has been implemented to autonomously mitigate CSRF attacks as precise as possible. Evaluation was done using specific CSRF scenarios, as well as in real-life by a group of test users. Third, the granularity of the client-side policy is improved even further by incorporating server-specific policy refinements about intended cross-domain traffic.

1 Introduction

Cross-Site Request Forgery (CSRF) is a web application attack vector that can be leveraged by an attacker to force 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 [15]. In 2008, Zeller and Felten documented a number of serious CSRF vulnerabilities in high-profile websites, among which was a vulnerability on the home banking website of ING Direct [23].

One of the root causes of CSRF is the abuse of cached credentials in cross-domain requests. A website 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 by protecting the server application or by protecting the end-user (e.g. via a browser extension or a client-side proxy). Unfortunately, the server-side protection mechanisms are not yet widely adopted. On the other side, most of the client-side solutions provide only limited protection or can not deal with complex web 2.0 applications, which use techniques such as AJAX, mashups or single sign-on (SSO). As a result, even the most cautious web user is unable to appropriately protect himself against CSRF, without compromising heavily on usability. 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 the results of an extensive, real-world traffic analysis. This analysis shows how many of the requests a browser makes are cross-domain and what their properties are. Second, we define CsFire, an autonomous client-side protection policy, which is independent of user-input or server-provided information. This policy will determine which cross-domain traffic is considered harmful and will propose an appropriate action, such as blocking the request or removing the implicit authentication credentials from the request. The paper also discusses how this policy can be enforced within the Firefox browser. Finally, this autonomous policy can be extended by server-specific refinements, to achieve a more flexible and fine-grained enforcement.

This research builds upon previous work [13]. The main achievements with respect to the previous preliminary results are (1) a much finer-grained policy allowing a secure-by-default solution without service degradation, and (2) a thorough evaluation of the proposed prototype.

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, the detailed traffic analysis results are presented, along with the autonomous client policy, in Section 3. Section 4 describes the client-side solution in great detail. The refinements with server-specific policies and its implementation are discussed in Section 5. Our solution is evaluated extensively by means of test scenarios, along with a group of test users. The evaluation results can be found in Section 6. In Section 7, the presented work is related to alternative mitigation techniques and, finally, Section 8 summarizes the contributions of this paper.

2 Background

This section provides some background information on CSRF and available countermeasures. We will also define the requirements for client-side mitigation against CSRF.

2.1 Cross-Site Request Forgery (CSRF)

HTTP is a stateless client-server protocol [5], 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 built 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 [17]. 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 [9], of which the latter is mainly used on intranets. This paper will focus on the first two authentication mechanisms.

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.

In general, CSRF attacks happen cross-domain, where an attacker tricks the user into connecting to another site. Strictly speaking, CSRF can also occur within the same domain, e.g. due to a script injection vulnerability or when multiple users can host a web site within the same domain (universities, ISP's, ...). The focus of this paper lies on the cross-domain CSRF attacks.

2.2 Existing countermeasures

A number of CSRF protection techniques exist, by either protecting the server application or by protecting the end-user. This section briefly discusses both.

Client-side countermeasures The most widespread countermeasure is the Same Origin Policy (SOP) [22], 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 [9] or as an extension in the browser [18,23]. These countermeasures monitor outgoing requests and incoming responses, and filter out implicit authentication or block cross-domain requests. Unfortunately, these client-side mitigation techniques suffer from various problems, as will be discussed in Section 7.

Server-side countermeasures A number of server-side mitigation techniques exists as well, of which the most popular class is the use of secret tokens [4,10,16]. Each response from the server embeds a secret token into the web page (e.g. a hidden parameter in an HTML 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 [2]. Furthermore, an attacker can spoof HTTP headers, e.g. via Flash or request smuggling [11,12].

More intrusively, Barth et al. propose an additional HTTP header to indicate the origin of the request [2]. 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 [19].

2.3 Requirements for client-side protection

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 well, such as JavaScript, AJAX and single sign-on (SSO).

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

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 6). It is infeasible for the user to 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.

R3. *Secure by default*. The solution should have minimal false negatives using its default configuration.

3 Secure cross-domain policy

The previous sections have made clear that even though there are effective countermeasures available, the user is left vulnerable due to the slow adoption of these countermeasures. In this section we will propose a policy defining exactly which cross-domain traffic is allowed and when protective measures need to be taken. This fine-grained policy, which is the main contribution of this paper, allows the users to protect themselves from malicious attackers and ill-secured websites.

To be able to determine an effective policy, information about cross-domain traffic is of crucial importance. By analyzing common cross-domain traffic, we can determine which request properties the policy enforcement mechanism can use to base its decisions on.

3.1 Properties of cross-domain traffic

We have collected real-life traffic from about 15 volunteers over a time period of 2 weeks, resulting in a total of 750334 requests. The analysis of this traffic has revealed a number of properties that can be used to determine a secure cross-domain policy. We will discuss the traffic by pointing out how the requests are distributed in the total data set. We will examine the data through the strict SOP, which uses the triple <full domain name, protocol, tcp port>, as well as the relaxed SOP, which only considers the actual domain name (e.g. `example.com`. These detailed results are consistent with one million earlier recorded requests, as reported in [13].

A first overview, presented in Table 1, shows the distribution between the different request methods (GET, POST and other). Striking is that for the strict SOP, almost 43% of the requests are cross-domain. For the relaxed SOP, this is nearly 33%. The number of cross-domain requests is dominated by GET requests, with the POST requests having a minimal share.

As far as the other methods are concerned, we have discovered a very small number of HEAD and OPTIONS requests. Due to their insignificant amount, we do not focus on this type of requests. We do acknowledge that they need to be investigated in future research.

	GET	POST	Other	Total
Cross-domain requests (strict SOP)	320529 (42.72%)	1793 (0.24%)	0 (0.00%)	322322 (42.96%)
Cross-domain requests (relaxed SOP)	242084 (32.26%)	1503 (0.20%)	0 (0.00%)	243587 (32.46%)
All requests	722298 (96.26%)	28025 (3.74%)	11 (0.00%)	750334 (100.00%)

Table 1. Traffic statistics: overview of all traffic (for each row, the percentages of the first 3 columns add up to the percentage of the last column)

Table 2 shows a detailed analysis of the GET requests. The columns show how much requests carry parameters (Params), how many requests are initiated by direct user interaction, such as clicking a link or submitting a form (User), how many requests contain a `cookie` header (Cookies) and how many requests carry HTTP authentication credentials (HTTP Auth). The results show that approximately 24% of the GET requests contain parameters. Furthermore, we can see that a very small amount of the GET-requests, especially of the cross-domain GET requests, originate from direct user interaction. The data also shows that cookies are a very popular authentication mechanism, whereas HTTP authentication is rarely used for cross-domain requests.

	Params	User	Cookies	HTTP Auth	Total
Cross-domain requests (strict SOP)	82587 (25.77%)	1734 (0.54%)	116632 (36.39%)	26 (0.08%)	320529 (42.72%)
Cross-domain requests (relaxed SOP)	58372 (24.11%)	1100 (0.45%)	59980 (24.78%)	1 (0.00%)	242084 (32.26%)
All GET requests	168509 (23.33%)	7132 (0.99%)	411056 (56.91%)	651 (0.89%)	722298 (96.26%)

Table 2. Traffic statistics: overview of GET requests (for each row, the percentages in the columns are independent of each other and calculated against the total in the last column)

The analysis of the POST requests is summarized in Table 3, categorized in the same way as the GET requests, except for the presence of parameters. This data shows the same patterns as for the GET requests, albeit on a much smaller scale. We do see a larger percentage of cross-domain requests being initiated by the user, instead of being conducted automatically. Again, the HTTP authentication mechanism suffers in popularity, but cookies are used quite often.

3.2 Defining a policy

A policy blocking all cross-domain traffic is undoubtedly the most secure policy. However, from the traffic analysis we can easily conclude that this would lead to a

	User	Cookies	HTTP Auth	Total
Cross-domain requests (strict SOP)	158 (8.81%)	1005 (56.05%)	0 (0.00%)	1793 (0.24%)
Cross-domain requests (relaxed SOP)	25 (1.66%)	753 (50.10%)	0 (0.00%)	1503 (0.20%)
All POST requests	930 (3.32%)	23056 (82.27%)	96 (1.99%)	28025 (3.74%)

Table 3. Traffic statistics: overview of POST requests (for each row, the percentages in the columns are independent of each other and calculated against the total in the last column)

severely degraded user experience, which conflicts with the second requirement of a good client-side solution. A policy that meets all three requirements will have to be more fine-grained and sophisticated. To achieve this goal, the policy can choose from three options for cross-domain requests: the two extremes are either allowing or blocking a cross-domain request. The road in the middle leads to stripping the request from authentication information, either in the form of cookies or HTTP authentication headers.

In order to preserve as much compatibility as possible, we have chosen to use the relaxed SOP to determine whether a request is cross-domain or not. This is comparable to JavaScript, where a relaxation of the origin is also allowed. We will now define the policy actions for each type of request and where possible, we will add further refinements. We will start by examining POST requests, followed by the GET requests. An overview of the policy is given in Table 4.

For a relaxed SOP, the traffic analysis shows that only 0.20 % of the cross-domain requests are POST requests, of which 1.66% is the result of direct user interaction. Therefore, we propose to strip all POST requests, even the manually submitted POST requests. This effectively protects the user from potentially dangerous UI redressing attacks [14], while having a minimal effect on the user experience.

Even though the HTTP protocol specification [5] states that GET requests should not have state-altering effects, we will ensure that CSRF attacks using GET requests are also prevented. This means that the policy for GET requests will have to be fine-grained. Since requests with parameters have a higher risk factor, we will define different rules for GET requests carrying parameters and GET requests without any parameters. The traffic analysis has shown that cross-domain GET requests with parameters are very common, which means that the attack vector is quite large too. Therefore, we propose to strip all GET requests with parameters from any authentication information. The GET requests without any parameters are less risky, since they are not likely to have a state-altering effect. Therefore, we have decided to allow GET requests with no parameters, if the request originates from user-interaction (e.g. clicking on a link). This helps preserving the unaltered user experience, because otherwise, when the user is logged in on a certain website, such as Facebook, and follows a link to `www.facebook.com` in for an example a search engine, the authentication information would be removed, which requires the user to re-authenticate.

GET requests without any parameters that are not the result of direct user interaction will be stripped to cover all bases. If such a request would depend on authentication information, a re-authentication will be necessary.

Properties			Decision
GET	Parameters		STRIP
	No parameters	User initiated	ACCEPT
		Not User initiated	STRIP
POST	User initiated		STRIP
	Not User initiated		STRIP

Table 4. The secure default policy for cross-domain traffic

4 Mitigating malicious cross-domain requests

In the previous section we have determined a policy to counter CSRF attacks. The policy will be enforced by a few specific components, each with their own responsibility. One of these components is the policy information point (PIP), where all the available information is collected. This information can be leveraged by the policy decision point (PDP) to make a decision about a certain request. This decision is used by the policy enforcement point (PEP), which will provide active protection for the user. We have implemented this policy as CsFire, an extension[2] for Mozilla Firefox that incorporates each of these components. The technical details of CsFire will now be discussed.

4.1 The Firefox architecture

Mozilla Firefox, the second most popular browser, comes with an open and extensible architecture. This architecture is fully aimed at accommodating possible browser extensions. Extension development for Firefox is fairly simple and is done using provided XPCOM components [21]. Our Firefox extension has been developed using JavaScript and XPCOM components provided by Firefox itself.

To facilitate extensions wishing to influence the browsing experience, Firefox provides several possibilities to examine or modify the traffic. For our extension, the following four capabilities are extremely important:

- Influencing the user interface using XUL overlays
- Intercepting content-influencing actions by means of the `content-policy` event
- Intercepting HTTP requests before they are sent by observing the `http-on-modify-request` event (This is the point where the policy needs to be enforced).
- Intercepting HTTP responses before they are processed by observing the `http-on-examine-response` event

[2] The extension can be downloaded from `https://distrinet.cs.kuleuven.be/software/CsFire/`.

4.2 Policy enforcement in Firefox

When a new HTTP request is received, the PEP needs to actively enforce the policy to prevent CSRF attacks. To determine how to handle the request, the PEP contacts the PDP which can either decide to *allow* the request, *block* it or *strip* it from any authentication information.

Enforcing an allow or block decision is straightforward: allowing a request requires no interaction, while blocking a request is simply done by signaling an error to Firefox. Upon receiving this error message, Firefox will abort the request. Stripping authentication information is less straightforward and consists of two parts: stripping cookies and stripping HTTP authentication credentials. How this can be done will be explained in the following paragraphs.

Firefox 3.5 has introduced a *private browsing mode*, which causes Firefox to switch to a mode where no cookies or HTTP authentication from the user's database are used. Private browsing mode stores no information about surfing sessions and uses a separate cookie and authentication store, which is deleted upon leaving private browsing mode. Unfortunately, we were not able to leverage this private browsing mode to strip authentication information from cross-domain requests, due to some difficulties. The major setback is the fact that Firefox makes an entire context switch when entering private browsing mode. This causes active tabs to be reloaded in this private mode, which essentially causes false origin information and influences all parallel surfing sessions.

Another approach is to manually remove the necessary HTTP headers when examining the HTTP request, before it is sent out. This technique is very effective on the `cookie` headers, but does not work for `authorization` headers. These headers are either added upon receiving a response code of 401 or 407 or appended automatically during an already authenticated session. In the former case, the headers are available upon examining the request and can be removed, but in the latter case, they are only added after the request has been examined. Obviously, this poses a problem, since the headers can not be easily removed.

Investigating this problem revealed that to implement the private browsing mode, the Firefox developers have added a new load flag[3], `LOAD_ANONYMOUS`, which prevents the addition of any `cookie` or `authorization` headers. If we set this flag when we are examining the HTTP request, we can prevent the addition of the `authorization` header. This is not the case for the `cookie` header, but as mentioned before, the `cookie` header, which at this point is already added to the request, can be easily removed.

4.3 Considerations of web 2.0

The difficulty of preventing CSRF can not necessarily be contributed to the nature of the attack, but more to the complex traffic patterns that are present in the modern web 2.0 context. Especially sites extensively using AJAX, single

[3] Load flags can be set from everywhere in the browser and are checked by the Firefox core during the construction of the request.

sign-on (SSO) mechanisms or mashup techniques, which combines content of multiple different websites, make it hard to distinguish intended user traffic from unintended or malicious traffic. Web 2.0 techniques such as AJAX and SSO can be dealt with appropriately, but mashups are extremely difficult to distinguish from CSRF attacks. Our solution has no degrading effect on websites using AJAX and SSO, but can be inadequate on mashup sites depending on implicit authentication to construct their content.

A SSO session typically uses multiple redirects to go from the site the user is visiting to an SSO service. During these redirects, authentication tokens are exchanged. When the original site receives a valid authentication token, the user is authenticated. Since all these redirects are usually cross-domain, no cookies or HTTP authentication headers can be used anyway, due to the SOP restrictions implemented in browsers. The authentication tokens are typically encoded in the redirection URLs. Our extension is able to deal with these multiple redirects and does not break SSO sessions.

5 Server contributions to a more fine-grained policy

The policy up until now was completely based on information available at the client-side. Unfortunately, such a policy fails to reflect intentions of web applications, where cross-domain traffic may be desired in certain occasions. To be able to obtain a more fine-grained policy, containing per site directives about cross-domain traffic, we have introduced an optional server policy. This server policy can tighten or weaken the default client policy. For instance, a server can prohibit any cross-domain traffic, even if authentication information is stripped, but can also allow intended cross-domain traffic from certain sites.

The technical implementation of server-side policies are fairly straightforward: the server defines a cross-domain policy in a pre-determined location, using a pre-determined syntax. The policy syntax, which is based on JSON, is expressed in the ABNF metasyntax language [3] and is available online[4]. The policy is retrieved and parsed by the browser extension at the client side. The policy is used by the PDP when decisions about cross-domain traffic need to be made.

The server policy has been made as expressive as possible, without requiring too many details to be filled out. The server policy can specify whether the strict SOP or the relaxed SOP needs to be used. Next to this, a list of intended cross-domain traffic can be specified. This intended traffic consists of a set of origins and a set of destinations, along with the policy actions to take. We have also provided the option to specify certain cookies that are allowed, instead of stripping all cookies. Finally, we also allow the use of the wild card character *, to specify rules for all hosts. An example policy can be found in Figure 1.

Technically, the server policy is enforced as follows: when the PDP has to decide about a certain request, the request is checked against the target server

[4] `http://www.cs.kuleuven.be/~lieven/research/ESSoS2010/serverpolicy-abnf.txt`

```
{"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": "*" }]
    }]}
```

Figure 1. An example server policy

policy. If a match is found, the decision specified by the policy will be found. If no match is found, the request is handled by the default client policy.

At the time, the composition of the server policy and the *secure by default* client policy to a unified policy is very rudimentary. This needs to be refined, such that the server can introduce policy refinements, without severely compromising the client-side policy. These refinements are left to future research.

6 Evaluation

CSRF attacks, as described earlier, are cross-domain requests abusing the cached authentication credentials of a user, to make state-altering changes to the target application. Knowing whether a request is state altering or not, is very application specific and very hard to detect at the client-side. The solution we proposed in the previous sections, examines all cross-domain traffic (intended and unintended) and limits the capabilities of such cross-domain requests, thus also prevents CSRF attacks. The extension is evaluated using a testbed of specially created test scenarios, to confirm that the capabilities of cross-domain requests are indeed limited as specified by the policy. A second part of the evaluation is done by a group of test users, that have used the extension during their normal, everyday surfing sessions. This part of the evaluation will confirm that even though the extension intercepts all cross-domain traffic – and not only the CSRF attacks –, the user experience is not affected by this fact. We conclude by presenting a few complex scenarios, that have been tested separately.

6.1 Extensive evaluation using the scenario testbed

To evaluate the effectiveness of CSRF prevention techniques, including our own solution, we have created a suite of test scenarios. These scenarios try to execute a CSRF attack in all different ways possible in the HTTP protocol, the HTML specification and the CSS markup language. The protocol and language specifications have been examined for cross-domain traffic possibilities. Each possible security risk was captured in a single scenario. Where applicable, a scenario has a version where the user initiates the request, as well as an automated version using JavaScript. For completeness, an additional JavaScript version using timeouts was created. In total, we have used a test suite of 59 scenarios. For requests originating from JavaScript, no scenarios are created, since these requests are typically blocked by the SOP.

Some highlights of these testing scenarios are redirects, either by the `Location` header, the `Refresh` header[5] or the `meta`-tag. For CSS, all attributes that use an URL as a value are possible CSRF attack vectors.

The extension has been evaluated against each of these scenarios. For every scenario, the CSRF attack was effectively prevented. Some scenarios conducted a hidden CSRF attack, in which case the user does not notice the attack being prevented. In case the attack is clearly visible, such as by opening a link in the current window, the user is presented with an authentication prompt for the targeted site. This is an altered user experience, but since an unwanted attack is prevented, this can not be considered a degradation of the surfing experience.

When discussing related work, these test scenarios will be used for the evaluation of other CSRF prevention solutions.

6.2 Real-life evaluation

A group of more than 50 test users, consisting of colleagues, students and members of the OWASP Chapter Belgium, has used CsFire with the policy as defined in this paper for over three months. The extension provides a feedback button, where users can easily enter a comment whenever they encounter unexpected effects. The results of a 1 month time slice are presented in Table 5. These numbers show that the extension has processed 1,561,389 requests, of which 27% was stripped of authentication credentials. The feedback logged by the users was limited to 3 messages, which was confirmed verbally after the testing period.

Apart from the transparent evaluation by a group of test users, certain specific scenarios have been tested as well. This includes the use of single sign-on services such as Shibboleth and OpenID. The use of typical web 2.0 sites such as Facebook or iGoogle was also included in the evaluation.

The only minor issue we detected, with the help of the feedback possibility of the extension, was with sites using multiple top-level domains. For instance, when Google performs authentication, a couple of redirects happen between several

[5] Even though the `Refresh` header is not part of the official HTTP specification, it is supported by browsers.

Number of processed requests	1,561,389
Number of ACCEPT decisions	1,141,807
Number of BLOCK decisions	0
Number of STRIP decisions	419,582
Number of feedback messages	3

Table 5. A 1 month time slice of evaluation data

`google.com` domains and the `google.be` domain. This causes certain important session cookies to be stripped, which invalidates the newly authenticated session. This problem occurs for example with the calendar gadget of iGoogle, as well as the login form for *code.google.com*. This issue has not been registered on other Google services, such as Gmail, or any other websites.

These issues show why it is very difficult for an autonomous client-side policy to determine legitimate cross-domain traffic from malicious cross-domain traffic. This problem shows the importance of a server-side policy, which could relax the client-side policy in such a way that requests between `google.be` and `google.com` would be allowed.

A side-effect from the way Firefox handles its tabs becomes visible when using multiple tabs to access the same website. If a user is authenticated in tab one, a session cookie has probably been established. If the user now accesses this site in another tab, using a cross-domain request, the cookies will be stripped. This will cause the sessions in both tabs to be invalidated, which is a minor degrading experience. This behavior is very application-specific, since it depends on the way the application handles authentication and session management. This behavior has been experienced on LinkedIn, but does not appear on Facebook or Wikipedia. This problem can be mitigated with the integration of tab-isolation techniques. Such techniques are not yet available for Mozilla Firefox, but are in place in Google Chrome [7] and Microsoft Gazelle [20].

7 Related work

In this section, we discuss CSRF protection mechanisms that where an inspiration to our solution: *RequestRodeo* and the *Adobe Flash cross-domain policy*. We also discuss two competing solutions: *BEAP* and *RequestPolicy*. Finally, we discuss *BEEP*, which proposes server-enforced policies, which can lead to future improvements of CSRF protection techniques.

RequestRodeo The work of Johns and Winter aptly describes the issues with CSRF and a way to resolve these issues [9]. They propose *RequestRodeo*, a client-side proxy which protects the user from CSRF attacks. The proxy processes incoming responses and augments each URL with a unique token. These tokens are stored, along with the URL where they originated. Whenever the proxy receives an outgoing request, the token is stripped off and the origin is retrieved. If the origin does not match the destination of the request, the request is considered suspicious. Suspicious requests will be stripped of authentication credentials in the form of cookies or HTTP `authorization` headers. RequestRodeo also protects against IP address based attacks, by using an external

proxy to check the global accessibility of web servers. If a server is not reachable from the outside world, it is considered to be an intranet server that requires additional protection. The user will have to explicitly confirm the validity of such internal cross-domain requests.

By stripping authentication credentials instead of blocking the request, RequestRodeo makes an important contribution, which lies at the basis of this work. Protecting against IP address based attacks is novel, and could also be added to our browser extension using the same approach. Johns and Winter do encounter some difficulties due to the use of a client-side proxy, which lacks context information. They also rely on a rewriting technique to include the unique token in each URL. These issues gain in importance in a web 2.0 world, where web pages are becoming more and more complex, which can be dealt with gracefully by means of a browser extension. Our solution is able to use full context information to determine which requests are authorized or initiated by a user.

Adobe Flash By default, the Adobe flash player does not allow flash objects to access content retrieved from other websites. By means of a server-provided configurable policy, Adobe provides the option to relax the *secure by default* policy [1]. The target server can specify trusted origins, which have access to its resources, even with cross-domain requests. This technique was a source of inspiration for our own server-provided policies.

One unfortunate side-effect with the Adobe cross-domain policy and the example above is that a lot of sites have implemented an *allow all* policy [23]. To obtain a secure unified policy, smart composition of client and server-provided policies is crucial.

BEAP (AntiCSRF) Mao, Li and Molloy present a technique called *Browser-Enforced Authenticity Protection* [14]. Their solution resembles our solution, in a sense that they also aim to remove authentication credentials and have implemented a Firefox extension. Their policy to determine suspicious requests is quite flexible and based on the fact that GET requests should not be used for sensitive operations. As this may hold in theory, practice tells us otherwise, especially in the modern web 2.0 world. GET requests not carrying an `authorization` header are not considered sensitive, which leaves certain windows of attack open. BEAP addresses one of these issues, namely UI redressing, by using a *source-set* instead of a single source, the origin of the page. All the origins in the set, which are the origins of all ancestor frames, need to match the destination of the new request before it is allowed.

We have tested the provided Firefox extension against various test scenarios. The extension only works effectively against cross-domain POST requests, which is an expected consequence of the protection policy they propose. Unfortunately, the provided extension does not remove the `authorization` header and only seems to remove the `cookie` header. Our solution proposes a more realistic and more secure policy, and contributes technically by providing a clean way to actually remove an `authorization` header in Firefox.

RequestPolicy Samuel has implemented a Firefox extension against CSRF attacks [18]. RequestPolicy is aimed at fully preventing CSRF attacks, which is realized by blocking cross-domain traffic, unless the sites are whitelisted. The criteria used to identify suspicious cross-domain traffic are user interaction and a relaxed SOP. Whenever a user is directly responsible for a cross-domain request, by clicking on a link or submitting a form, the request is allowed. Otherwise, traffic going to another domain is blocked. the extension allows a way to add whitelisted sites, such that traffic from `x.com` is allowed to retrieve content from `y.com`. By default, the extension proposes some whitelist entries, such as traffic from `facebook.com` to `fbcdn.com`.

When testing the RequestPolicy extension against our test scenarios, we found that almost all CSRF attacks are effectively blocked. Only the attacks which are caused by direct user interaction succeeded, which was expected. Unfortunately, when testing the extension by surfing on the internet, our experience was severely degraded. For starters, opening search results on Google stopped working. The cause for this issue is that clicking a search result actually retrieves a Google page[6], which uses JavaScript to redirect the user to the correct website. This JavaScript redirect is considered suspicious and therefore blocked. Apart from the Google issue, other sites suffered very noticeable effects. For instance, the popular site `slashdot.org` suffers major UI problems, since the stylesheet is loaded from a separate domain. These issues do not occur in our solution, since we only strip authentication information, instead of completely blocking cross-domain traffic.

BEEP Jim, Swamy and Hicks propose *Browser-enforced Embedded Policies*, which uses a server-provided policy to defeat malicious JavaScript. They argue that the browser is the ideal point to prevent the execution of malicious scripts, since the browser has the entire script at execution time, even if it is obfuscated or comes from multiple sources. Their solution is to inject a protection script at the server, which will be run first by the browser and validates other scripts.

Such a server-provided but client-enforced technique is very similar to our solution, which is able to use server-provided policies and enforces security at the client-side. The solution proposed in BEEP can be an inspiration to work towards unified client-server CSRF protection mechanisms.

8 Conclusion

We have shown the need for an autonomous client-side CSRF protection mechanism, especially since the already existing server-side protection mechanisms fail to get widely adopted and the available client-side solutions do not suffice. We have provided an answer to this requirement with our client-side browser extension which effectively protects the user from CSRF attacks. A predefined policy

[6] When investigating this issue, not much information about this issue was found. We have noticed that this effect does not happen consistently, but have not found any logic behind this Google behavior.

is used to determine which cross-domain requests need to be restricted, by either blocking the request or stripping the request from authentication credentials.

This work builds on preliminary results presented in an earlier paper, but presents much more detailed results in every section. We have conducted an extensive traffic analysis, which resulted in a number of request properties that can be used to determine the appropriate policy action for a cross-domain request. The predefined client-side policy uses these fine-grained criteria to achieve a policy that protects the user, without influencing the user experience in a negative way. In case cross-domain traffic is intended, which is not known by the client, servers can provide a cross-domain policy specifying which cross-domain requests are allowed. The browser extension merges both the client-side policy and the server-provided policy, to preserve the available protection mechanisms but also to offer as much flexibility as possible.

The policy and the enforcement mechanism have been thoroughly evaluated against CSRF attack scenarios, which cover all possibilities to mount a CSRF attack using HTTP, HTML or CSS properties. We have also collected results from a group of test users, which have actively used the extension during their normal surfing sessions. Finally, we have evaluated the extension against a few complex scenarios, such as sites using a single sign-on (SSO) mechanism or mashup techniques. Aside from one minor issue with sites spanning multiple top-level domains, no degrading effects where monitored, while all CSRF attack scenarios where successfully prevented. Even on mashup sites and sites using SSO mechanisms, no problems where detected.

The solution in this paper is not yet perfect and there is still room for improvement. Future research will focus on the refinement of the composition of a client-side policy and server-provided policies. The policies need to be extended to include other traffic besides GET and POST. Finally, the use of other authentication mechanisms, such as for instance SSL authentication, needs to be further investigated to prevent CSRF attacks abusing such credentials.

Acknowledgements

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

We would also like to thank our colleagues, students and members of the OWASP Chapter Belgium, who volunteered to collect traffic information and test CsFire.

References

1. Adobe. Adobe Flash Player 9 security, July 2008.
2. 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.
3. D. Crocker and P. Overell. Augmented BNF for syntax specifications: ABNF. `http://tools.ietf.org/html/rfc5234`, 2008.
4. 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.
5. 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.
6. E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns. Addison-Wesley, 1995.
7. Chromium Developer Documentation `http://dev.chromium.org/developers/design-documents/process-models`
8. T. Jim and N. Swamy and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, 2007.
9. M. Johns and J. Winter. RequestRodeo: Client side protection against session riding. In *In Proceedings of the OWASP Europe 2006 Conference*, 2006.
10. 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.
11. A. Klein. Forging HTTP request headers with Flash. `http://www.securityfocus.com/archive/1/441014`, July 2006.
12. C. Linhart, A. Klein, R. Heled, and S. Orrin. HTTP request smuggling. Technical report, Watchfire, 2005.
13. W. Maes and T. Heyman and L. Desmet and W. Joosen. Browser protection against Cross-Site Request Forgery In *Workshop on Secure Execution of Untrusted Code (SecuCode)*, Chicago, IL, USA, November 2009.
14. Z. Mao and N. Li and I. Molloy. Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection. In *LNCS*, 2009.
15. OWASP. The ten most critical web application security vulnerabilities.
16. OWASP. CSRF Guard. `http://www.owasp.org/index.php/CSRF_Guard`, October 2008.
17. V. Raghvendra. Session tracking on the web. *Internetworking*, 3(1), 2000.
18. J. Samuel. Request Policy 0.5.8. `http://www.requestpolicy.com`.
19. A. van Kesteren. Cross-origin resource sharing. `http://www.w3.org/TR/2009/WD-cors-20090317/`, March 2009.
20. H.J. Wang and C. Grier and A. Moshchuk and S.T. King and P. Choudhury and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. Microsoft Research Technical Report (MSR-TR-2009-16) 2009.
21. XPCOM - MDC. `https://developer.mozilla.org/en/XPCOM`, 2008.
22. M. Zalewski. *Browser Security Handbook*. 2008. `http://code.google.com/p/browsersec/wiki/Main`.
23. 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`.