# Automatic and Precise Client-Side Protection against CSRF Attacks

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

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

**Abstract.** A common client-side countermeasure against Cross Site Request Forgery (CSRF) is to strip session and authentication information from malicious requests. The difficulty however is in determining when a request is malicious. Existing client-side countermeasures are typically too strict, thus breaking many existing websites that rely on authenticated cross-origin requests, such as sites that use third-party payment or single sign-on solutions.

The contribution of this paper is the design, implementation and evaluation of a request filtering algorithm that automatically and precisely identifies *expected* cross-origin requests, based on whether they are preceded by certain indicators of collaboration between sites. We formally show through bounded-scope model checking that our algorithm protects against CSRF attacks under one specific assumption about the way in which good sites collaborate cross-origin. We provide experimental evidence that this assumption is realistic: in a data set of 4.7 million HTTP requests involving over 20.000 origins, we only found 10 origins that violate the assumption. Hence, the remaining attack surface for CSRF attacks is very small. In addition, we show that our filtering does not break typical non-malicious cross-origin collaboration scenarios such as payment and single sign-on.

**Keywords:** CSRF, web security, browser security.

## 1 Introduction

From a security perspective, web browsers are a key component of today's software infrastructure. A browser user might have a session with a trusted site A (e.g. a bank, or a webmail provider) open in one tab, and a session with a potentially dangerous site B (e.g. a site offering cracks for games) open in another tab. Hence, the browser enforces some form of isolation between these two origins A and B through a heterogeneous collection of security controls collectively known as the *same-origin-policy* [18]. An *origin* is a (protocol, domain name, port) triple, and restrictions are imposed on the way in which code and data from different origins can interact. This includes for instance restrictions that prevent scripts from origin B to access content from origin A.

V. Atluri and C. Diaz (Eds.): ESORICS 2011, LNCS 6879, pp. 100–116, 2011.

An important known vulnerability in this isolation is the fact that content from origin B can initiate requests to origin A, and that the browser will treat these requests as being part of the ongoing session with A. In particular, if the session with A was authenticated, the injected requests will appear to A as part of this authenticated session. This enables an attack known as *Cross Site Request Forgery (CSRF)*: B can initiate effectful requests to A (e.g. a bank transaction, or manipulations of the victim's mailbox or address book) without the user being involved.

CSRF has been recognized since several years as one of the most important web vulnerabilities [3], and many countermeasures have been proposed. Several authors have proposed server-side countermeasures [3,4,10]. However, an important disadvantage of server-side countermeasures is that they require modifications of server-side programs, have a direct operational impact (e.g. on performance or maintenance), and it will take many years before a substantial fraction of the web has been updated.

Alternatively, countermeasures can be applied on the client-side, as browser extensions. The basic idea is simple: the browser can strip session and authentication information from malicious requests, or it can block such requests. The difficulty however is in determining when a request is malicious. Existing client-side countermeasures [9,5,12,13,16,19] are typically too strict: they block or strip all cross-origin requests of a specific type (e.g. GET, POST, any). This effectively protects against CSRF attacks, but it unfortunately also breaks many existing websites that rely on authenticated cross-origin requests. Two important examples are sites that use third-party payment (such as PayPal) or single sign-on solutions (such as OpenID). Hence, these existing client-side countermeasures require extensive help from the user, for instance by asking the user to define white-lists of trusted sites or by popping up user confirmation dialogs. This is suboptimal, as it is well-known that the average web user can not be expected to make accurate security decisions.

This paper proposes a novel client-side CSRF countermeasure, that includes an automatic and precise filtering algorithm for cross-origin requests. It is *automatic* in the sense that no user interaction or configuration is required. It is *precise* in the sense that it distinguishes well between malicious and non-malicious requests. More specifically, through a systematic analysis of logs of web traffic, we identify a characteristic of non-malicious cross-origin requests that we call the *trusted-delegation assumption*: a request from B to A can be considered non-malicious if, earlier in the session, A explicitly delegated control to B in some specific ways. Our filtering algorithm relies on this assumption: it will strip session and authentication information from cross-origin requests, unless it can determine that such explicit delegation has happened.

We validate our proposed countermeasure in several ways. First, we formalize the algorithm and the trusted-delegation assumption in Alloy, building on the formal model of the web proposed by [1], and we show through bounded-scope model checking that our algorithm protects against CSRF attacks under this assumption. Next, we provide experimental evidence that this assumption is

realistic: through a detailed analysis of logs of web traffic, we quantify how often the trusted-delegation assumption holds, and show that the remaining attack surface for CSRF attacks is very small. Finally, we have implemented our filtering algorithm as an extension of an existing client-side CSRF protection mechanism, and we show that our filtering does not break typical non-malicious cross-origin collaboration scenarios such as payment and single sign-on.

In summary, the contributions of this paper are:

- The design of a novel client-side CSRF protection mechanism based on request filtering.
- A formalization of the algorithm, and formal evidence of the security of the algorithm under one specific assumption, the trusted-delegation assumption.
- An implementation of the countermeasure, and a validation of its compatibility with important web scenarios broken by other state-of-the-art countermeasures.
- An experimental evaluation of the validity of the trusted-delegation assumption.

The remainder of the paper is structured as follows. Section 2 explains the problem using both malicious and non-malicious scenarios. Section 3 discusses our request filtering mechanism. Section 4 introduces the formalization and results, followed by the implementation in Section 5. Section 6 experimentally evaluates the trusted-delegation assumption. Finally, Section 7 extensively discusses related work, followed by a brief conclusion (Section 8).

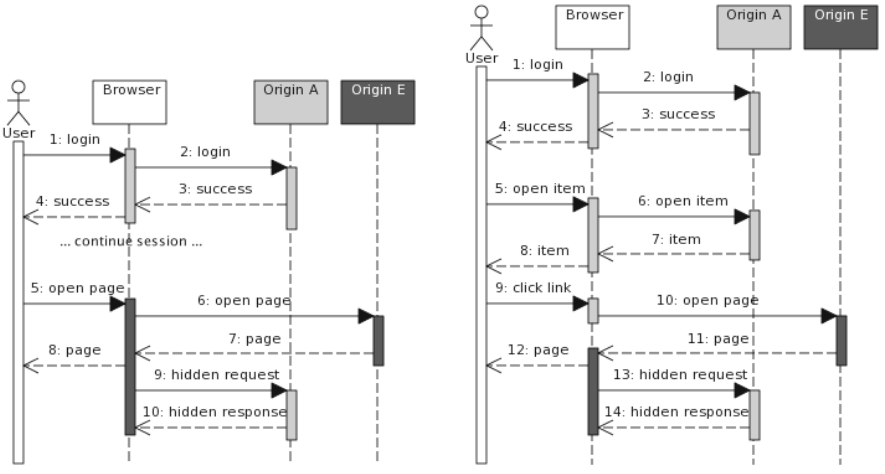
## 2 Cross-Origin HTTP Requests

The key challenge for a client-side CSRF prevention mechanism is to distinguish malicious from non-malicious cross-origin requests. This section illustrates the difficulty of this distinction by describing some attack scenarios and some important non-malicious scenarios that intrinsically rely on cross-origin requests.

### 2.1 Attack Scenarios

*A1. Classic CSRF* Figure 1(a) shows a classic CSRF attack. In steps 1–4, the user establishes an authenticated session with site A, and later (steps 5–8) the user opens the malicious site E in another tab of the browser. The malicious page from E triggers a request to A (step 9), the browser considers this request to be part of the ongoing session with A and automatically adds the necessary authentication and session information. The browser internally maintains different *browsing contexts* for each origin it is interacting with. The shade of the browser-lifeline in the figure indicates the origin associated with the browsing context from which the outgoing request originates (also known as *the referrer*). Since the attack request originates from an E browsing context and goes to origin A, it is *cross-origin*.

For the attack to be successful, an authenticated session with A must exist when the user surfs to the malicious site E. The likelihood of success can be




(a) Classic CSRF (b) Link injection

**Fig. 1.** CSRF attack scenarios

increased by making E content-related to A, for instance to attack a banking site, the attacker poses as a site offering financial advice.

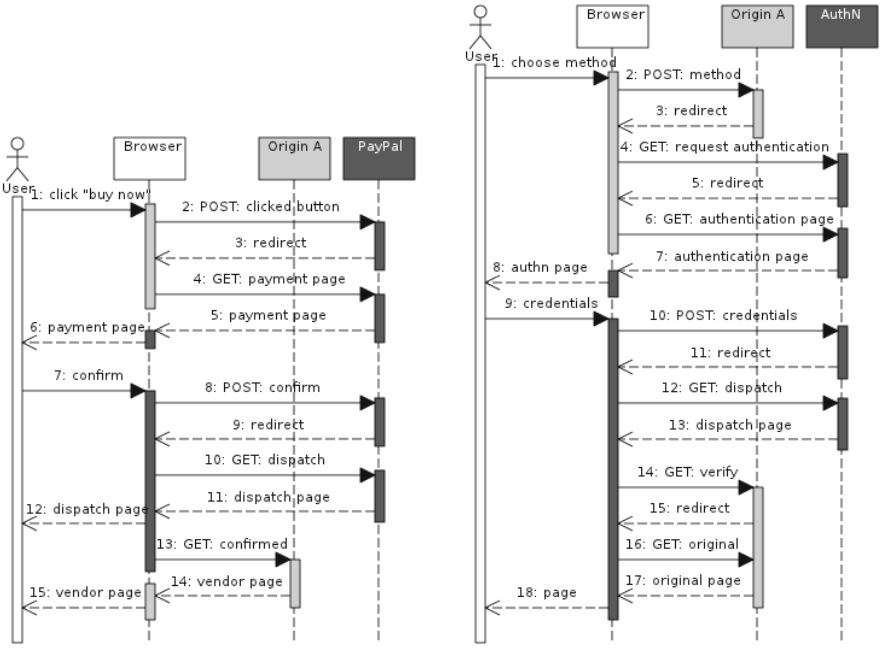
*A2. Link Injection.* To further increase the likelihood of success, the attacker can inject links to E into the site A. Many sites, for instance social networking sites, allow users to generate content which is displayed to other users. For such a site A, the attacker creates a content item which contains a link to E. Figure 1(b) shows the resulting CSRF scenario, where A is a social networking site and E is the malicious site. The user logs into A (steps 1–4), opens the attacker injected content (steps 5–8), and clicks on the link to E (step 9) which launches the CSRF attack (step 13). Again, the attack request is cross-origin.

### 2.2 Non-malicious Cross-Origin Scenarios

CSRF attack requests are cross-origin requests in an authenticated session. Hence, forbidding such requests is a secure countermeasure. Unfortunately, this also breaks many useful non-malicious scenarios. We illustrate two important ones.

*F1. Payment Provider.* Third-party payment providers such as PayPal or Visa 3D-secure offer payment services to a variety of sites on the web.

Figure 2(a) shows the scenario for PayPal's *Buy Now* button. When a user clicks on this button, the browser sends a request to PayPal (step 2), that redirects the user to the payment page (step 4). When the user accepts the payment (step 7), the processing page redirects the browser to the dispatch page (step 10), that loads the landing page of the site that requested the payment (step 13).




(a) Payment scenario (b) Central authentication scenario

**Fig. 2.** Non-malicious cross-origin scenarios

Note that step 13 is a cross-origin request from PayPal to A in an authenticated session, for instance a shopping session in web shop A.

*F2. Central Authentication.* The majority of interactive websites require some form of authentication. As an alternative to each site using its own authentication mechanism, a single sign-on service (such as OpenID or Windows Live ID) provides a central point of authentication.

An example scenario for OpenID authentication using MyOpenID is shown in Figure 2(b). The user chooses the authentication method (step 1), followed by a redirect from the site to the authentication provider (step 4). The authentication provider redirects the user to the login form (step 6). The user enters the required credentials, which are processed by the provider (step 10). After verification, the provider redirects the browser to the dispatching page (step 12), that redirects to the processing page on the original site (step 14). After processing the authentication result, a redirect loads the requested page on the original site (step 16).

Again, note that step 16 is a cross-origin request in an authenticated session.

These two scenarios illustrate that mitigating CSRF attacks by preventing cross-origin requests in authenticated sessions breaks important and useful web scenarios. Existing client-side countermeasures against CSRF attacks [5,13,16] either are incompatible with such scenarios or require user interaction for these cases.

## 3 Automatic and Precise Request Stripping

The core idea of our new countermeasure is the following: client-side state (i.e. session cookie headers and authentication headers) is stripped from all cross-origin requests, except for *expected* requests. A cross-origin request from origin A to B is *expected* if B previously (earlier in the browsing session) *delegated* to A. We say that B *delegates* to A if B either issues a POST request to A, or if B redirects to A using a URI that contains parameters.

The rationale behind this core idea is that (1) non-malicious collaboration scenarios follow this pattern, and (2) it is hard for an attacker to trick A into delegating to a site of the attacker: forcing A to do a POST or parametrized redirect to an evil site E requires the attacker to either identify a cross-site scripting (XSS) vulnerability in A, or to break into A's webserver. In both these cases, A has more serious problems than CSRF.

Obviously, a GET request from A to B is not considered a delegation, as it is very common for sites to issue GET requests to other sites, and as it is easy for an attacker to trick A into issuing such a GET request (see for instance attack scenario A2 in Section 2).

Unfortunately, the elaboration of this simple core idea is complicated somewhat by the existence of HTTP redirects. A web server can respond to a request with a *redirect* response, indicating to the browser that it should resend the request elsewhere, for instance because the requested resource was moved. The browser will follow the redirect automatically, without user intervention. Redirects are used widely and for a variety of purposes, so we cannot ignore them. For instance, both non-malicious scenarios in Section 2 heavily depend on the use of redirects. In addition, attacker-controlled websites can also use redirects in an attempt to bypass client-side CSRF protection. Akhawe et al. [1] discuss several examples of how attackers can use redirects to attack web applications, including an attack against a CSRF countermeasure. Hence, correctly dealing with redirects is a key requirement for security.

The flowgraph in Figure 3 summarizes our filtering algorithm. For a given request, it determines what session state (cookies and authentication headers) the browser should attach to the request. The algorithm differentiates between simple requests and requests that are the result of a redirect.

*Simple Requests.* Simple requests that are not cross-origin, as well as expected cross-origin requests are handled as unprotected browsers handle them today. The browser automatically attaches the last known client-side state associated with the destination origin (point 1). The browser does not attach any state to non-expected cross-origin requests (point 3).

*Redirect Requests.* If a request is the consequence of a redirect response, then the algorithm determines if the redirect points to the origin where the response came from. If this is the case, the client-side state for the new request is limited to the client-side state known to the previous request (i.e. the request that triggered this redirect) (point 2). If the redirect points to another origin, then, depending

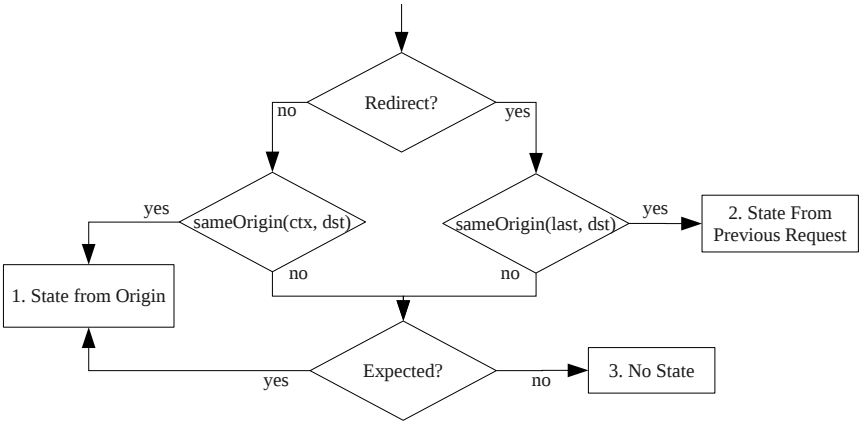



**Fig. 3.** The request filtering algorithm

on whether this cross-origin request is expected or not, it either gets session-state automatically attached (point 1) or not (point 3).

*When Is a Request Expected?.* A key element of the algorithm is determining whether a request is *expected* or not. As discussed above, the intuition is: a cross-origin request from B to A is expected if and only if A first delegated to B by issuing a POST request to B, or by a parametrized redirect to B. Our algorithm stores such trusted delegations, and an assumption that we rely on (and that we refer to as the *trusted-delegation assumption*) is that sites will only perform such delegations to sites that they trust. In other words, a site A remains vulnerable to CSRF attacks from origins to which it delegates. Section 6 provides experimental evidence for the validity of this assumption.

The algorithm to decide whether a request is expected goes as follows.

For a simple cross-origin request from site B to site A, a trusted delegation from site A to B needs to be present in the delegation store.

For a redirect request that redirects a request to origin Y (light gray) to another origin Z (dark gray) in a browsing context associated with some origin $\alpha$, the following rules apply.

1. First, if the destination (Z) equals the source (i.e. $\alpha = Z$) (Figure 4(a)), then the request is expected if there is a trusted delegation from Z to Y in the delegation store. Indeed, Y is effectively doing a cross-origin request to Z by redirecting to Z. Since the browsing context has the same origin as the destination, it can be expected not to manipulate redirect requests to misrepresent source origins of redirects (cfr next case).
2. Alternatively, if the destination (Z) is not equal to the source (i.e. $\alpha \neq Z$) (Figure 4(b)), then the request is expected if there is a trusted delegation from Z to Y in the delegation store, since Y is effectively doing a cross-origin request to Z. Now, the browsing context might misrepresent source

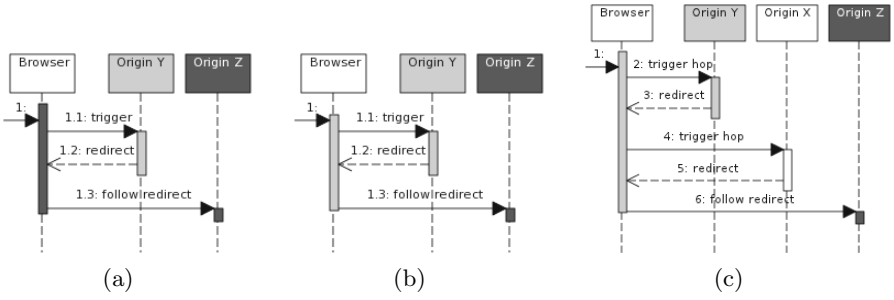



**Fig. 4.** Complex cross-origin redirect scenarios

origins of redirects by including additional redirect hops (origin X (white) in Figure 4(c)). Hence, our decision to classify the request does not involve X.

Finally, our algorithm imposes that expected cross-origin requests can only use the GET method and that only two origins can be involved in the request chain. These restrictions limit the potential power an attacker might have, even if the attacker successfully deceives the trusted-delegation mechanism.

*Mapping to Scenarios.* The reader can easily check that the algorithm blocks the attack scenarios from Section 2, and supports the non-malicious scenarios from that section. We discuss two of them in more detail.

In the PayPal scenario (Figure 2(a)), step 13 needs to re-use the state already established in step 2, which means that according to the algorithm, the request from PayPal to A should be expected. A trusted delegation happens in step 2, where a cross-origin POST is sent from origin A to PayPal. Hence the GET request in step 13 is considered expected and can use the state associated with origin A. Also note how the algorithm maintains the established session with PayPal throughout the scenario. The session is first established in step 3. Step 4 can use this session, because the redirect is an internal redirect on the PayPal origin. Step 8 can use the last known state for the PayPal origin and step 10 is yet another internal redirect.

In the link injection attack (Figure 1(b)), the attack happens in step 13 and is launched from origin E to site A. In this scenario, an explicit link between A and E exists because of the link injected by the attacker. This link is however not a POST or parametrized redirect, so it is not a trusted delegation. This means that the request in step 10 is not considered to be expected, so it can not access the previously established client-side state, and the attack is mitigated.

## 4 Formal Modeling and Checking

The design of web security mechanisms is complex: the behaviour of (same-origin and cross-origin) browser requests, server responses and redirects, cookie and session management, as well as the often implicit threat models of web

security can lead to subtle security bugs in new features or countermeasures. In order to evaluate proposals for new web mechanisms more rigorously, Akhawe et al. [1] have proposed a model of the Web infrastructure, formalized in Alloy.

The base model is some 2000 lines of Alloy source code, describing (1) the essential characteristics of browsers, web servers, cookie management and the HTTP protocol, and (2) a collection of relevant threat models for the web. The Alloy Analyzer – a bounded-scope model checker – can then produce counterexamples that violate intended security properties if they exist in a specified finite scope.

In this section, we briefly introduce Akhawe's model and present our extensions to the model. We also discuss how the model was used to verify the absence of attack scenarios and the presence of functional scenarios.

### 4.1 Modeling Our Countermeasure

The model of Akhawe et al. defines different principals, of which `GOOD` and `WEBATTACKER` are most relevant. `GOOD` represents an honest principal, who follows the rules imposed by the technical specifications. A `WEBATTACKER` is a malicious user who can control malicious web servers, but has no extended networking capabilities.

The concept of `Origin` is used to differentiate between origins, which correspond to domains in the real world. An origin is linked with a server on the web, that can be controlled by a principal. The browsing context, modeled as a `ScriptContext`, is also associated with an `origin`, that represents the origin of the currently loaded page, also known as the referrer.

A `ScriptContext` can be the source of a set of `HTTPTransaction` objects, which are a pair of an `HTTPRequest` and `HTTPResponse`. An HTTP request and response are also associated with their remote destination origin. Both an HTTP request and response can have headers, where respectively the `CookieHeader` and `SetCookieHeader` are the most relevant ones. An HTTP request also has a `method`, such as `GET` or `POST`, and a `queryString`, representing URI parameters. An HTTP response has a `statusCode`, such as `c200` for a content result or `c302` for a redirect. Finally, an HTTP transaction has a cause, which can be none, such as the user opening a new page, a `RequestAPI`, such as a scripting API, or another `HTTPTransaction`, in case of a redirect.

To model our approach, we need to extend the model of Akhawe et al. to include (a) the accessible client-side state at a certain point in time, (b) the trusted delegation assumption and (c) our filtering algorithm. We discuss (a) and (b) in detail, but due to space constraints we omit the code for the filtering algorithm (c), which is simply a literal implementation of the algorithm discussed in Section 3.

*Client-Side State.* We introduced a new signature `CSState` that represents a client-side state (Listing 1.1). Such a state is associated with an `Origin` and contains a set of `Cookie` objects. To associate a client-side state with a given request or response and a given point in time, we have opted to extend the `HTTPTransaction` from the original model into a `CSStateHTTPTransaction`. Such an extended transaction includes a `beforeState` and `afterState`, respectively

representing the accessible client-side state at the time of sending the request and the updated client-side state after having received the response. The `afterState` is equal to the `beforeState`, with the potential addition of new cookies, set in the response.

```
sig CSState {
  dst: Origin,
  cookies: set Cookie
}

sig CSStateHTTPTransaction extends HTTPTransaction {
  beforeState : CSState,
  afterState : CSState
} {
 //The after state of a transaction is equal to the before state + any additional cookies set in
      the response
 beforeState.dst = afterState.dst
 afterState.cookies = beforeState.cookies + (resp.headers & SetCookieHeader).thecookie

 // The destination origin of the state must correspond to the transaction destination origin
    beforeState.dst = req.host
}
```

**Listing 1.1.** Signatures representing our data in the model

*Trusted-delegation Assumption.* We model the trusted-delegation assumption as a fact, that honest servers do not send a POST or parametrized redirect to web attackers ((Listing 1.2).

```
fact TrustedDelegation {
  all r : HTTPRequest | {
   (r.method = POST || some (req.r).cause & CSStateHTTPTransaction)
     &&
  ((some (req.r).cause & CSStateHTTPTransaction && getPrincipalFromOrigin[(req.r).cause.req.
       host] in GOOD) || getPrincipalFromOrigin[transactions.(req.r).owner] in GOOD)
   implies
     getPrincipalFromOrigin[r.host] not in WEBATTACKER
  }
}
```

**Listing 1.2.** The fact modeling the trusted-delegation assumption

### 4.2 Using Model Checking for Security and Functionality

We formally define a CSRF attack as the possibility for a web attacker (defined in the base model) to inject a request with at least one existing cookie attached to it (this cookie models the session/authentication information attached to requests) in a session between a user and an honest server (Listing 1.3).

We provided the Alloy Analyzer with a universe of at most 9 HTTP events and where an attacker can control up to 3 origins and servers (a similar size as used in [1]). In such a universe, no examples of an attacker injecting a request through the user's browser were found. This gives strong assurance that the countermeasure does indeed protect against CSRF under the trusted delegation assumption.

```
pred CSRF[r : HTTPRequest] {
   //Ensure that the request goes to an honest server
   some getPrincipalFromOrigin[r.host]
   getPrincipalFromOrigin[r.host] in GOOD

    //Ensure that an attacker is involved in the request
    some (WEBATTACKER.servers & involvedServers[req.r]) || getPrincipalFromOrigin[(
         transactions.(req.r)).owner] in WEBATTACKER

    // Make sure that at least one cookie is present
    some c : (r.headers & CookieHeader).thecookie | {
       //Ensure that the cookie value is fresh (i.e. that it is not a renewed value in a redirect
            chain)
       not c in ((req.r).*cause.resp.headers & SetCookieHeader).thecookie
    }
}
```

**Listing 1.3.** The predicate modeling a CSRF attack

We also modeled the non-malicious scenarios from Section 2, and the Alloy Analyzer reports that these scenarios are indeed permitted. From this, we can also conclude that our extension of the base model is consistent.

Space limitations do not permit us to discuss the detailed scenarios present in our model, but the interested reader can find the complete model available for download at [6].

**Table 1.** CSRF benchmark

| | Test scenarios | Result |
|---|---|---|
| HTML | 29 cross-origin test scenarios | ✓ |
| CSS | 12 cross-origin test scenarios | ✓ |
| ECMAScript | 9 cross-origin test scenarios | ✓ |
| Redirects | 20 cross-origin redirect scenarios | ✓ |

## 5 Implementation

We have implemented our request filtering algorithm in a proof-of-concept add-on for the Firefox browser, and used this implementation to conduct an extensive practical evaluation. First, we have created simulations for both the common attack scenarios as well as the two functional scenarios discussed in the paper (third party payment and centralized authentication), and verified that they behaved as expected.

Second, in addition to these simulated scenarios, we have verified that the prototype supports actual instances of these scenarios, such as for example the use of MyOpenID authentication on sourceforge.net.

Third, we have constructed and performed a CSRF benchmark[1], consisting of 70 CSRF attack scenarios to evaluate the effectiveness of our CSRF prevention

[1] The benchmark can be applied to other client-side solutions as well, and is downloadable at [6].

technique (see Table 1). These scenarios are the result of a CSRF-specific study of the HTTP protocol, the HTML specification and the CSS markup language to examine their cross-origin traffic capabilities, and include complex redirect scenarios as well. Our implementation has been evaluated against each of these scenarios, and our prototype passed all tests successfully.

The prototype, the scenario simulations and the CSRF benchmark suite are available for download [6].

## 6 Evaluating the Trusted-Delegation Assumption

Our countermeasure drastically reduces the attack surface for CSRF attacks. Without CSRF countermeasures in place, an origin can be attacked by any other origin on the web. With our countermeasure, an origin can only be attacked by another origin to which it has delegated control explicitly by means of a cross-origin POST or redirect. We have already argued in Section 3 that it is difficult for an attacker to cause unintended delegations. In this section, we measure the remaining attack surface experimentally.

We conducted an extensive traffic analysis using a real-life data set of 4.729.217 HTTP requests, collected from 50 unique users over a period of 10 weeks. The analysis revealed that 1.17% of the 4.7 million requests are treated as delegations in our approach. We manually analyzed all these 55.300 requests, and classified them in the interaction categories summarized in Table 2.

For each of the categories, we discuss the resulting attack surface:

**Third Party Service Mashups.** This category consists of various third party services that can be integrated in other websites. Except for the single sign-on services, this is typically done by script inclusion, after which the included script can launch a sequence of cross-origin GET and/or POST requests towards offered AJAX APIs. In addition, the service providers themselves often use cross-origin redirects for further delegation towards content delivery networks.

As a consequence, the origin A including the third-party service S becomes vulnerable to CSRF attacks from S. This attack surface is unimportant, as in these scenarios, S can already attack A through script inclusion, a more powerful attack than CSRF.

In addition, advertisement service providers P that further redirect to content delivery services D are vulnerable to CSRF attacks from D whenever a user clicks an advertisement. Again, this attack surface is unimportant: the delegation from P to D correctly reflects a level of trust that P has in D, and P and D will typically have a legal contract or SLA in place.

**Multi-origin Websites.** Quite a number of larger companies and organizations have websites spanning multiple origins (such as *live.com* - *microsoft.com* and *google.be* - *google.com*). Cross-origin POST requests and redirects between these origins make it possible for such origins to attack each other. For instance, *google.be* could attack *google.com*. Again, this attack

**Table 2.** Analysis of the trusted-delegation assumption in a real-life data set of 4.729.217 HTTP requests

| | # requests | | POST | redir. |
|---|---|---|---|---|
| Third party service mashups | 29.282 | (52,95%) | 5.321 | 23.961 |
| *Advertisement services* | *22.343* | *(40,40%)* | *1.987* | *20.356* |
| *Gadget provider services (appspot, mochibot, gmodules, ...)* | *2.879* | *(5,21%)* | *2.757* | *122* |
| *Tracking services (metriweb, sitestat, uts.amazon, ...)* | *2.864* | *(5,18%)* | *411* | *2.453* |
| *Single Sign-On services (Shibboleth, Live ID, OpenId, ...)* | *1.156* | *(2,09%)* | *137* | *1.019* |
| *3rd party payment services (Paypal, Ogone)* | *27* | *(0,05%)* | *19* | *8* |
| *Content sharing services (addtoany, sharethis, ...)* | *13* | *(0,02%)* | *10* | *3* |
| Multi-origin websites | 13.973 | (25,27%) | 198 | 13.775 |
| Content aggregators | 8.276 | (14,97%) | 0 | 8.276 |
| *Feeds (RSS feeds, News aggregators, mozilla fxfeeds, ...)* | *4.857* | *(8,78%)* | *0* | *4.857* |
| *Redirecting search engines (Google, Comicranks, Ohnorobot)* | *3.344* | *(6,05%)* | *0* | *3.344* |
| *Document repositories (ACM digital library, dx.doi.org, ...)* | *75* | *(0,14%)* | *0* | *75* |
| False positives (wireless network access gateways) | 1.215 | (2,20%) | 12 | 1.203 |
| URL shorteners (gravatar, bit.ly, tinyurl, ...) | 759 | (1,37%) | 0 | 759 |
| Others (unclassified) | 1.795 | (3,24%) | 302 | 1.493 |
| **Total number of 3rd party delegation initiators** | **55.300** | **(100%)** | **5.833** | **49.467** |

surface is unimportant, as all origins of such a multi-origin website belong to a single organization.

**Content Aggregators.** Content aggregators collect searchable content and redirect end-users towards a specific content provider. For news feeds and document repositories (such as the ACM digital library), the set of content providers is typically stable and trusted by the content aggregator, and therefore again a negligible attack vector.

Redirecting search engines register the fact that a web user is following a link, before redirecting the web user to the landing page (e.g. as Google does for logged in users). Since the entries in the search repository come from all over the web, our CSRF countermeasure provides little protection for such search engines. Our analysis identified 4 such origins in the data set: *google.be*, *google.com*, *comicrank.com*, and *ohnorobot.com*.

**False Positives.** Some fraction of the cross-origin requests are caused by network access gateways (e.g. on public Wifi) that intercept and reroute requests towards a payment gateway. Since such devices have man-in-the-middle capabilities, and hence more attack power than CSRF attacks, the resulting attack surface is again negligible.

**URL Shorteners.** To ease URL sharing, URL shorteners transform a shortened URL into a preconfigured URL via a redirect. Since such URL shortening services are open, an attacker can easily control a new redirect target. The effect is similar to the redirecting search engines; URL shorteners are essentially left unprotected by our countermeasure. Our analysis identified 6 such services in the data set: *bit.ly*, *gravatar.com*, *post.ly*, *tiny.cc*, *tinyurl.com*, and *twitpic.com*.

**Others(unclassified).** For some of the requests in our data set, the origins involved in the request were no longer online, or the (partially anonymized) data did not contain sufficient information to reconstruct what was happening, and we were unable to classify or further investigate these requests.

In summary, our experimental analysis shows that the trusted delegation assumption is realistic. Only 10 out of 23.592 origins (i.e. 0.0042% of the examined origins) – the redirecting search engines and the URL shorteners – perform delegations to arbitrary other origins. They are left unprotected by our countermeasure. But the overwhelming majority of origins delegates (in our precise technical sense, i.e. using cross-origin POST or redirect) only to other origins with whom they have a trust relationship.

## 7 Related Work

The most straightforward protection technique against CSRF attacks is server-side mitigation via validation tokens [4,10]. In this approach, web forms are augmented with a server-generated, unique validation token (e.g. embedded as a hidden field in the form), and at form submission the server checks the validity of the token before executing the requested action. At the client-side, validation tokens are protected from cross-origin attackers by the same-origin-policy, distinguishing them from session cookies or authentication credentials that are automatically attached to any outgoing request. Such a token based approach can be offered as part of the web application framework [14,7], as a server-side library or filter [15], or as a server-side proxy [10].

Recently, the `Origin` header has been proposed as a new server-side countermeasure [3,2]. With the `Origin` header, clients unambiguously inform the server about the origin of the request (or the absence of it) in a more privacy-friendly way than the `Referer` header. Based on this origin information, the server can safely decide whether or not to accept the request. In follow-up work, the `Origin` header has been improved, after a formal evaluation revealed a previously unknown vulnerability [1]. The Alloy model used in this evaluation also formed the basis for the formal validation of our presented technique in Section 4.

Unfortunately, the adoption rate of these server-side protection mechanisms is slow, giving momentum to client-side mitigation techniques as important (but hopefully transitional) solutions. In the next paragraphs, we will discuss the client-side proxy RequestRodeo, as well as 4 mature and popular browser addons (CsFire, NoScript ABE, RequestPolicy, and CSD[2]). In addition, we will evaluate how well the browser addons enable the functional scenarios and protect against the attack scenarios discussed in this paper (see Table 3).

RequestRodeo [9] is a client-side proxy proposed by Johns and Winter. The proxy applies a client-side token-based approach to tie requests to the correct source origin. In case a valid token is lacking for an outgoing request, the request is considered suspicious and gets stripped of cookies and HTTP `authorization` headers. RequestRodeo lies at the basis of most of the client-side CSRF solutions [5,12,13,16,19], but because of the choice of a proxy, RequestRodeo often lacks context information, and the rewriting technique on raw responses does not scale well in a web 2.0 world.

[2] Since the client-side detection technique described in [17] is not available for download, the evaluation is done based on the description in the paper.

**Table 3.** Evaluation of browser-addons

| | Functional scenarios | | Attack scenarios | |
|---|---|---|---|---|
| | F1. Payment Provider | F2. Central Authentication | A1. Classic CSRF | A2. Link Injection |
| CsFire [5] | × | × | ✓ | ✓ |
| NoScript ABE [13] [a] | × | × | ✓ | ✓ |
| RequestPolicy [16] | ⋈ [b] | ⋈ [b] | ✓[c] | ✓[c] |
| Client-Side Detection [17] | × | × | ✓ | ✓ |
| **Our Approach** | ✓ | ✓ | ✓ | ✓ |

[a] ABE configured as described in [8]
[b] Requires interactive feedback from end-user to make the decision
[c] Requests are blocked instead of stripped, impacting the end-user experience

CsFire [5] extends the work of Maes *et al.* [11], and strips cookies and HTTP `authorization` headers from a cross-origin request. The advantage of stripping is that there are no side-effects for cross-origin requests that do not require credentials in the first place. CsFire operates autonomously by using a default client policy which is extended by centrally distributed policy rules. Additionally, CsFire supports users creating custom policy rules, which can be used to blacklist or whitelist certain traffic patterns. Without a central or user-supplied whitelist, CsFire does not support the payment and central authentication scenario.

To this extent, we plan to integrate the approach presented in this paper to the CsFire Mozilla Add-On distribution in the near future.

NoScript ABE [13], or Application Boundary Enforcer, restricts an application within its origin, which effectively strips credentials from cross-origin requests, unless specified otherwise. The default ABE policy only prevents CSRF attacks from the internet to an intranet page. The user can add specific policies, such as a CsFire-alike stripping policy [8], or a site-specific blacklist or whitelist. If configured with [8], ABE successfully blocks the three attack scenarios, but disables the payment and central authentication scenario.

RequestPolicy [16] protects against CSRF by blocking all cross-origin requests. In contrast to stripping credentials, blocking a request can have a very noticeable effect on the user experience. When detecting a cross-origin redirect, RequestPolicy injects an intermediate page where the user can explicitly allow the redirect. RequestPolicy also includes a predefined whitelist of hosts that are allowed to send cross-origin requests to each other. Users can add exceptions to the policy using a whitelist. RequestPolicy successfully blocks the three attack scenarios (by blocking instead of stripping all cross-origin requests) and requires interactive end-user feedback to enable the payment and central authentication scenario.

Finally, in contrast to the CSRF *prevention* techniques discussed in this paper, Shahriar and Zulkernine proposes a client-side *detection* technique for CSRF [17]. In their approach, malicious and benign cross-origin requests are distinguished from each other based on the existence and visibility of the submitted form or link in the originating page, as well as the visibility of the target. In addition, the expected content type of the response is taken into account to detect false negatives during execution. Although the visibility check closely approximates the end-user intent, their technique fails to support the script inclusions of

third party service mashups as discussed in Section 6. Moreover, without taking into account the delegation requests, expected redirect requests (as defined in Section 3) will be falsely detected as CSRF attacks, although these requests are crucial enablers for the payment and central authentication scenario.

## 8 Conclusion

We have proposed a novel technique for protecting at client-side against CSRF attacks. The main novelty with respect to existing client-side countermeasures is the good trade-off between security and compatibility: existing countermeasures break important web scenarios such as third-party payment and single-sign-on, whereas our countermeasure can permit them.

**Acknowledgements.** This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IBBT, IWT, the Research Fund K.U. Leuven and the EU-funded FP7-projects WebSand and NESSoS.

## References

1. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: IEEE Computer Security Foundations Symposium, pp. 290–304 (2010)
2. Barth, A., Jackson, C., Hickson, I.: The web origin concept (November 2010), `http://tools.ietf.org/html/draft-abarth-origin-09`
3. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: 15th ACM Conference on Computer and Communications Security, CCS 2008 (2008)
4. Burns, J.: Cross site reference forgery: An introduction to a common web application weakness. In: Security Partners, LLC (2005)
5. De Ryck, P., Desmet, L., Heyman, T., Piessens, F., Joosen, W.: CsFire: Transparent client-side mitigation of malicious cross-domain requests. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 18–34. Springer, Heidelberg (2010)
6. De Ryck, P., Desmet, L., Piessens, F., Joosen, W.: Automatic and precise client-side protection against csrf attacks - downloads (2011), `https://distrinet.cs.kuleuven.be/software/CsFire/esorics2011/`
7. Django. Cross site request forgery protection (2011), `http://docs.djangoproject.com/en/dev/ref/contrib/csrf/`
8. Informaction Forums. Which is the best way to configure ABE? (July 2010), `http://forums.informaction.com/viewtopic.php?f=23\&t=4752`
9. Johns, M., Winter, J.: RequestRodeo: client side protection against session riding. In: Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448, pp. 5–17 (2006)
10. Jovanovic, N., Kirda, E., Kruegel, C.: Preventing cross site request forgery attacks. In: IEEE International Conference on Security and Privacy in Communication Networks (SecureComm), pp. 1–10 (2006)

11. Maes, W., Heyman, T., Desmet, L., Joosen, W.: Browser protection against cross-site request forgery. In: Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code, pp. 3–10. ACM, New York (2009)
12. Mao, Z., Li, N., Molloy, I.: Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 238–255. Springer, Heidelberg (2009)
13. Giorgio Maone. Noscript 2.0.9.9 (2011), http://noscript.net/
14. Ruby on Rails. Actioncontroller::requestforgeryprotection (2011), http://api.rubyonrails.org/classes/ActionController/RequestForgeryProtection.html
15. Owasp. Csrf guard (October 2008), http://www.owasp.org/index.php/CSRF_Guard
16. Samuel, J.: Requestpolicy 0.5.20 (2011), http://www.requestpolicy.com
17. Shahriar, H., Zulkernine, M.: Client-side detection of cross-site request forgery attacks. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), pp. 358–367 (November 2010)
18. Zalewski, M.: Browser security handbook (2010), http://code.google.com/p/browsersec/wiki/Main
19. Zeller, W., Felten, E.W.: Cross-site request forgeries: Exploitation and prevention. Technical report, Princeton University (2008)