Lightweight Server Support for Browser-Based CSRF Protection

Alexei Czeskis
University of Washington
aczeskis@cs.uw.edu

Alexander Moshchuk
Microsoft Research
alexmos@microsoft.com

Tadayoshi Kohno
University of Washington
yoshi@cs.uw.edu

Helen J. Wang
Microsoft Research
helenw@microsoft.com

ABSTRACT

Cross-Site Request Forgery (CSRF) attacks are one of the top threats on the web today. These attacks exploit ambient authority in browsers (*e.g.,* cookies, HTTP authentication state), turning them into confused deputies and causing undesired side effects on vulnerable web sites. Existing defenses against CSRFs fall short in their coverage and/or ease of deployment. In this paper, we present a browser/server solution, *Allowed Referrer Lists (ARLs)*, that addresses the root cause of CSRFs and removes ambient authority for participating web sites that want to be resilient to CSRF attacks. Our solution is easy for web sites to adopt and does not affect any functionality on non-participating sites. We have implemented our design in Firefox and have evaluated it with real-world sites. We found that ARLs successfully block CSRF attacks, are simpler to implement than existing defenses, and do not significantly impact browser performance.

Categories and Subject Descriptors

D.m [**Software**]: Miscellaneous

General Terms

Security, Design

Keywords

Web Browser, CSRF, XSRF, Authentication, Authorization

1. INTRODUCTION

Web application developers have relied on web cookies to not only provide simple state management across HTTP requests, but also to be bearers of authentication and authorization state. This programming paradigm, combined with the fact that web browsers send cookies by default with every HTTP request, has led to the proliferation of ambient authority, whereby HTTP requests can be automatically authenticated and authorized with the transport of a cookie. Other sources of ambient authority include state in HTTP authentication headers, client-side TLS certificates, and even IP addresses (which are used for authorization in some intranets or home networks). Such ambient authority, in turn, has led to the proliferation of Cross-Site Request Forgery (CSRF) attacks.

CSRF attacks occur when malicious web sites cause a user's web browser to make unsolicited (or forged) requests to a legitimate site on the user's behalf. Browsers act as confused deputies and attach any existing cookies (or other ambient authority state) to the forged request to the victim site. If the web application looks at the cookie or other state attached to an HTTP request as an indication of authorization, the application may be tricked into performing an unwanted action. For example, when a user visits *bad.com*, the displayed page may force the browser to make requests to *bank.com/transfer-funds* (*e.g.,* by including an image). When making the request to *bank.com*, the user's browser will attach any cookies it has stored for *bank.com*. If *bank.com* verifies the request only via the attached cookies, it may erroneously execute the attacker's bidding.

CSRF attacks are a major concern for the web. In 2008, Zeller et al. [39] demonstrated how several prominent web sites were vulnerable to CSRF attacks that allowed an attacker to transfer money from bank accounts, harvest email addresses, violate user privacy, and compromise accounts. From January 1 to November 16 in 2012, 153 CSRF attacks have been reported, making 2012 one of the most CSRF-active years [26]. These vulnerabilities are not merely theoretical; they have a history of being actively exploited in the wild [3, 31].

CSRF defenses exist, and some may believe that defending against CSRF attacks is a solved problem. For example, tokenization is a well-known approach adopted in various forms in numerous web development frameworks. In a tokenization-based defense, a web server associates a secret token with HTTP requests that are allowed to cause side-effects on the backend. Assuming an attacker site cannot capture this token, the attacker cannot launch CSRF. However, our in-depth analysis (Section 3.1.1) reveals that tokenization has a number of practical drawbacks, such as lack of protection for GET requests, possible token extraction by adversaries, and challenges dealing with third-party web development components. We show that other defenses are also limited: either too rigid (thereby blocking legitimate content) or too yielding (thereby allowing certain attacks to occur).

By studying drawbacks in existing approaches, we set out to build a new CSRF defense that is (1) developer-friendly, (2) backward compatible (not blocking legitimate content), and (3) has complete coverage (defending against all CSRF attack vectors). We propose a new mechanism called *Allowed Referrer Lists (ARLs)* that allows browsers to withhold sending ambient authority credentials for web sites wishing to be resilient against CSRF attacks. We let participating sites specify their authorization structure through a whitelist of referrer URLs for which browsers are allowed to attach authorization credentials to HTTP requests.

This approach takes advantage of the fact that browsers know the browsing context, while web developers understand the application-specific authorization semantics. By letting browsers carry out the enforcement and having web developers only specify the policies, we ease the enforcement burden on developers.

WWW, 2013, Rio de Janeiro, Brazil
ACM 978-1-4503-2035-1/13/05.

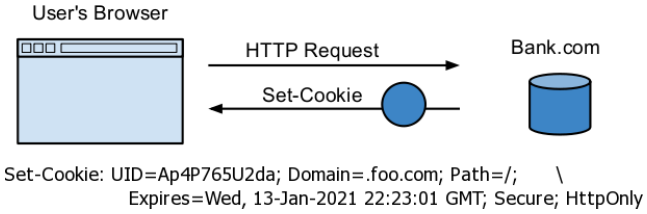

Figure 1: Example of a server *bank.com* setting a cookie. The cookie name is "UID" with a value of 11Ap4P765U2da. The cookie will be sent by the browser with every request to **.foo.com/ ("domain" and "path"), but only over HTTPS ("secure"). The cookie cannot be read or set from JavaScript ("HttpOnly"), and will expire on 13-Jan-2021.**

By only having participating sites specify the policies and receive CSRF protection from browsers, we leave other web sites' behavior unchanged, thus providing backward compatibility.

We have implemented ARLs in the Firefox browser. To evaluate ARLs, we studied four open-source web applications for which source code was available and for which the CSRF attacks were reported via the public vulnerability e-mail list "full-disclosure". We analyzed each application and reproduced the reported CSRF attacks. We then developed ARLs for each application and showed that the attacks were no longer possible. We also compared the amount of effort needed to implement CSRF protection using ARLs versus a traditional tokenization patch, finding ARLs to be the easier solution and one that provides better coverage.

We also studied how ARLs could be deployed on three large, real-world sites: Gmail, Facebook, and Woot. We found that most features offered by these sites could be supported with ARLs with very few modifications. We also considered ARL compatibility for complicated real-world web constructs such as as nested iframes, multiple redirections, and federation identity protocols.

We summarize our contributions of this paper as follows: (1) we give an in-depth analysis of existing CSRF defenses and analyze their limitations. (2) we propose, implement, and evaluate a new browser mechanism for not sending ambient authorization state for participating sites based on their policies.

In Section 2 we give a background of authentication on the web and how ambient authority leads to CSRF attacks. In Section 3, we conduct an in-depth analysis of existing CSRF defenses and their drawbacks. We present the ARL design in Section 4 and describe our browser implementation in Section 5. In Section 6, we evaluate ARLs against real CSRF attacks on open-source applications and discuss how ARLs would fare on real-world sites. We discuss privacy and limitations in Section 7, and we conclude in Section 8.

2. BACKGROUND

E-commerce web sites, webmail services, and many other web applications require the browser and server to maintain state about user sessions. Today, the *de facto* method of doing so is through HTTP Cookies, which are simply key/value pairs that a server can pass to and retrieve from the user's browser. A server "sets" a cookie by adding a `Set-Cookie` HTTP header to an HTTP response. By default, the browser stores the cookie for the current browsing session and uses the `Cookie` header to attach it to any subsequent HTTP requests it makes to the same web domain. The server may add attributes in the `Set-Cookie` header to change how the browser should handle the cookie. For example, the server can set a cookie's expiration date with the "Expires" attribute (making the cookie persistent), restrict the cookie to be sent only over HTTPS with the "Secure" attribute, and disallow JavaScript access to the cookie with the "HttpOnly" attribute. Additionally, the server may limit the cookie's scope to particular sub-domains and/or URL paths via the "Domain" and "Path" attributes. Figure 2 shows an example of how a server sets a cookie.

Figure 2: An example CSRF attack. When the user visits the adversary's page, the HTTP reply (step 2) includes code that causes the user's browser to make a request to *bank.com* (and attach *bank.com's* cookie). *Bank.com* erroneously treats this request as legitimate since it has the user's cookie.

Web servers use cookies to store a variety of client-side state. For example, to tie HTTP requests to users, many servers store the user ID or session ID in a cookie. Some web applications also reduce load on their backend servers by using cookies to store frequently queried values, such as language preferences or UI settings.

Cross-Site Request Forgery. As mentioned above, cookies are often used for authentication — as bearers of user identity. Many web applications, however, mistakenly use the same cookie not only for *authentication*, but also for *authorization*. Specifically, many sites assume that an HTTP request bearing the user's cookie must have been initiated by the user. Unfortunately, this is not necessarily true in today's web. In fact, if the user visits an attacker's web page, the attacker can cause the user's browser to make HTTP requests to *any* web origin. This attack is known as Cross-Site Request Forgery (CSRF). An example of this attack is shown in Figure 2.

2.1 Cause of CSRFs: Ambient Authority

The root cause of CSRFs is the prevalence of *ambient authority* on today's web. Ambient authority means that web sites rely on data *automatically* sent by browsers as an indication of authority and thereby legitimate user intent. While cookie misuse is the most widespread cause of ambient authority and CSRF attacks, there are a number of lesser-known means by which CSRF can happen:

- **HTTP Authentication**: Some web sites use HTTP Authentication [6] to authenticate users. Browsers prompt the user for a username/password and send the user-supplied credentials in an "Authorization" HTTP header. The browser then caches these credentials and resends them whenever the server requests them. Note that *authentication* data is being sent in an *authorization* header — both confusing and misleading. Attackers may create CSRFs by causing the user's browser to send requests to an origin with cached HTTP Authentication credentials. A separate "Proxy-Authorization" header similarly authenticates users to proxies, with similar implications for CSRF. More advanced techniques such as NTLM [24] exist for verifying authenticated clients, but they eventually cause similar tokens to be sent in an HTTP header.
- **Source IP Address**: A corporation may grant access to intranet sites based on a client's source IP. For example, employees may request vacation days, add dependents, or divert parts of their paycheck towards a charitable organization through the intranet. When visiting an attacker's site, a user's browser may be instructed to connect to an intranet IP address with potentially malicious consequences.
- **Client-Side TLS Certificates**: The TLS protocol has support for both TLS server and (less popular) TLS client certificates. Client certificates can encode the user identity and, just like cookies, be used to identify a user. Unlike cookies or HTTP

Authentication, TLS client certificates are not sent with every request. Instead, they are used to initially establish an authenticated TLS session. A web application will then consider any data sent through the authenticated TLS session as belonging to the respective user. However, if a site uses TLS client certificates for authorization (rather than purely authentication), the site may be vulnerable to CSRFs, since attackers can cause browsers to send requests over authenticated TLS sessions.

In each of the above scenarios, a web application relies on a single "token" (IP address, cookie, HTTP header, or client certificate) as an indication of authorization. We call these tokens *bearer tokens*. Because most of today's web sites implement authorization based on cookies, the majority of known CSRF attacks are cookie-based. Nevertheless, other types of attacks have been observed in the wild. For example, "router pharming" attacks [34] use JavaScript in the user's browser to change DNS settings in home routers, many of which use Basic Authentication or source IP for user authorization.

3. RELATED WORK

We now describe existing anti-CSRF defenses and explain how they fall short of a comprehensive solution. We dive deep into how current solutions are designed, deployed, and used across the web, looking at both currently deployed defenses and those that have been proposed but not yet adopted. CSRF defenses come in three flavors: server-side, client-side, and server/client hybrids; we will discuss each in turn. We view the synthesis and systematization below as a contribution of independent interest.

3.1 Server-side Approaches

Server-side solutions rely solely on server logic for CSRF protection. They are currently the most popular type of CSRF defense.

3.1.1 Tokenization

The current best practice for CSRF protection involves the use of a secret token. This approach works as follows:

1. When the user loads a page from the web application, the web server generates a secret token (a string) and includes it in the body of the web page.
2. As the user interacts with the web page and causes state-changing requests to be issued back to the web server, those requests include the secret token.
3. The web server then verifies the existence and correctness of the token received in the request before continuing execution.

Note that the secret token is not sent automatically by the browser (as is the case with cookies). Instead, the secret token is stored in the web page's DOM, and the page attaches it to requests programmatically via scripts or HTML forms. The security of this approach stems from the token being tied to the user's current session and being random enough to not be guessable by an attacker.

Implementing anti-CSRF tokenization involves three steps: (1) limit all "unsafe" operations to POST requests (as per RFC 2616 [13]), (2) include tokens in all HTML forms and AJAX requests that issue POSTs back to the server, and (3) verify the existence of the correct CSRF token when processing POST requests at the server.

Traditionally, developers implemented tokenization in a painstakingly manual manner. A developer would write code to generate and validate tokens and then find and secure each part of the application that generates or handles POST requests. To simplify this daunting process, several CSRF protection frameworks have been developed (*e.g.,* CSRF Guard [27, 28]). Most frameworks automate POST request tokenization by rewriting HTML forms and adding token information to AJAX queries. Although these frameworks exist, many web applications still implement CSRF protection manually; this appears to be especially true for applications written using older web platforms, such as PHP or Perl.

More recent web development platforms (*e.g.,* Ruby on Rails, ASP.NET, and Django) include token-based CSRF protection as part of the standard development platform package. In some cases, CSRF protection is enabled for all pages; in others, developers must mark specific classes, controllers, views, or other platform components as requiring CSRF protection. In these cases, the web platform issues CSRF tokens when creating HTML output and validates the tokens when processing POST data submissions.

While CSRF frameworks and integrated tokenization in web platforms have simplified tokenization's deployment, we argue that tokenization is an incomplete defense having many drawbacks:

Incompatible with GET requests. Tokens must not be sent over GET requests since GET requests may be logged by proxies or other services, or may be posted on the web by users, thus leaking the token. One may think that this problem does not arise in practice, as RFC 2616 specifically designates GET as a safe and idempotent method, which would make tokens unnecessary for GETs. However, real web applications don't follow this paradigm. For example, we investigated several popular web sites and found that (as of late 2012) Google, Amazon, live.com, and PayPal all use GET URLs to log users out. This is clearly not an idempotent action and, because none of the four web applications use CSRF protection for the logout action, an attacker can terminate a user's session with any of these applications without user consent. As another example, we found that Flickr.com (as of early 2012) uses GET requests for actions like changing the display language. Flickr does protect these requests with a CSRF token (sent as a URL parameter), but unfortunately uses the same token for POST requests as well. Because a Flickr user's token is the same from session to session, token leakage over a GET request could lead to more serious CSRF attacks that target POST APIs. Tokens may leak because URLs for GET requests may be stored in the browser history, server and proxy log files, bookmarks, etc. Attackers may then use techniques such as browser history sniffing to discover CSRF tokens [15].

Potentially extractable. In some situations, attackers may be able to extract CSRF tokens directly. For example, attackers could convince users to drag content that includes tokens from one web frame (the victim) to another (the attackers), or to copy-and-paste the token into the attacker's frame [18]. Attackers have used these techniques to trick Facebook users into exposing their CSRF tokens [40]. Researchers have also shown that many web sites are vulnerable to CSRF token extraction through a variety of HTML and script injection attacks [38, 9]. Recent work shows how CSRF tokens may be extracted using only cleverly formed CSS [14].

Error-prone manual implementation. Tokenization has many "moving parts", and custom implementations may thus be quite prone to errors. For example, a developer can easily overlook an important location where tokenization is needed and leave the application open to CSRFs. On the other hand, if the developer is overzealous with tokenization, tokens can leak; this is particularly bad if tokens are not ephemeral (like with Flickr) or made easily reversible to a session ID as suggested by some tutorials [12].

Frameworks confuse developers. On the other end of the spectrum, developers using a CSRF protection framework may misuse it, or they may falsely believe that it protects them from all types of CSRF attacks. For example, some frameworks do not tokenize AJAX requests [11, 7]. Other frameworks may only rewrite forms generated using web platform calls, leaving forms written using raw HTML unprotected. As another example, without under-

standing how CSRF frameworks work, developers can accidentally cause cross-domain POSTs to send a CSRF token to an untrusted third party. Finally, it's possible for developers to unwittingly accept GET requests while thinking the data came from POSTs — popular libraries such as Perl's CGI.PM module allow a developer to fetch a parameter without caring if it came in via a GET or POST request. Thus, POST requests could be converted to GET by the attacker, and the application action will still be performed [2].

Poor third-party subcomponent support. Many modern web development platforms (such as Drupal, Django, and Ruby on Rails) allow developers to use third-party components or plug-ins for added functionality. By integrating a poorly written component, developers might introduce a CSRF into their application. In such cases, it may be difficult to check whether a component correctly protects all actions, especially if it has a large code base [32].

Language dependence. For large, complicated web applications (such as large e-commerce sites) each part of the page may be generated using a different programming language. CSRF token generation and verification may need to be implemented separately by every one of those components.

3.1.2 Origin Checking

Besides tokenization, web application developers may use the proposed `Origin` HTTP header [4, 37], a privacy-preserving replacement for the `Referer` header. Like its predecessor, the Origin header is used by browsers to convey the originating domain of a request to the server. Web application developers can use that information to decide whether a request originated from a web origin the application trusts and hence is a legitimate request. For example, *bank.com* may trust *broker.com* and treat any request having an Origin header value of *broker.com* or *bank.com* as a valid state-modifying request, and treat all other requests as untrusted. As of late 2012, the Origin header is supported in Chrome and Safari.

In practice, the Origin header has restrictions that complicate and impede the ability of developers to use it as an anti-CSRF mechanism. Next, we discuss two such challenges.

No path information. To preserve privacy, the Origin header does not contain the path part of the request's origin. For example, suppose *bank.com* wants to assert that only requests from *broker.com/accounts/* can have side-effects on *bank.com*, but requests from any other location from *broker.com*, such as *broker.com/forum/*, may not. By design, the Origin header lacks the path information necessary for a web application to make this decision; making it impossible for *bank.com* to distinguish a request for the (legitimate) *accounts* page from a CSRF attack inside a malicious post on the *forum* page.

One workaround could be for *broker.com* to separate the */forum/* and */accounts/* parts of the web application into multiple subdomains (*e.g., accounts.broker.com* and *forum.broker.com*), but subdomain configuration may be problematic. Many open-source web applications (such as forums or blogs in a box) do not support subdomain creation via scripts, instead forcing the web developer to manually perform potentially confusing server configuration. Moreover, subdomain creation may be disallowed or may incur additional cost from hosting providers [33]. Finally, if using TLS, web developers would have to procure additional costly TLS certificates for subdomains or pay more for a wildcard certificate. Because of these complications, developers often separate web application modules by path rather than by subdomain.

Origin sent as null. If a request originated from an anchor tag or a window navigation command such as *window.open*, the Origin header is sent as null. The rationale is that "hyperlinks are common ways to jump from one site to another without trust. They should not be used to initiate state-changing procedures" and "many sites allow users to post links, so we don't want to send Origin with links" [25]. The suggested workaround is to convert all anchor tags and window navigation calls to a form GET. Such overhauls may be tough for maintainers of legacy sites, making it difficult for them to rely on the Origin header for CSRF protection.

Additionally, for business and security reasons, many sites (such as banking sites) do not allow users to post links. For these sites, using anchor tags as trusted sources of state-changing requests may be a valid decision. However, since the Origin header is *null* for all requests originating from anchor tags, these sites would be forced into using forms instead if they wish to leverage the Origin header.

3.2 Client-side Approaches

Some defenses check for CSRF on the client side (browser) rather than the server side. Client-side solutions first identify "suspicious" cross-origin requests and then either block the request outright [39] or strip the request of all cookies and HTTP authentication data [30, 10, 16, 22, 21]. The biggest advantage of client-only solutions is that they do not require any web site modifications, relying instead on either heuristics or central policy sources. Unfortunately, this makes them prone to false positives which break legitimate sites and/or false negatives which fail to detect CSRF.

CsFire. CsFire [30, 10] is a browser plug-in that strips cookies and authentication headers from outgoing HTTP requests that are deemed unsafe. By default, all cross-origin requests are considered unsafe, except for requests that pass a fairly strict set of rules to identify trust relationships between sites (*e.g., a.com* may make a request to *b.com* if *b.com* redirected to *a.com* earlier in the same browsing session). This policy breaks many legitimate sites, so CsFire maintains a whitelist of exceptions on a central server (maintained by the CsFire authors) and also allows users to add exceptions manually.

Unfortunately, in our experience, CsFire still results in false positives and breaks legitimate functionality during normal browsing, such as logging into Flickr or Yahoo via OpenID. This shows that such an architecture would need to rely on users and/or CsFire developers to constantly update the list of whitelisted sites that are allowed to send authentication data. Moreover, existing sites may change at any moment and break existing CsFire policies. We believe maintaining such an exception list for the whole web is close to impractical, as is relying on users to manually add exceptions.

In addition, CsFire's policies make a binary decision to either send or strip *all* cookies and HTTP authentication headers. This may cause overly restrictive policies, as cross-origin requests could harmlessly include cookies that are not used for authentication (such as user settings). Worse, this may also lead to insecure policies: if a site needs a non-sensitive cookie in order to function, a user may be tempted to add a CsFire exception for such a site, which would allow requests to the site to attach *all* cookies, including sensitive authentication cookies, which could lead to CSRF.

RequestRodeo. RequestRodeo [16] is a client-side proxy, positioned between the browser and web sites, that stops CSRFs by stripping authentication data from suspicious requests. Requests may contain authentication headers only if they are initiated "because of interaction with a web page (*i.e.,* clicking on a link, submitting a form or through JavaScript), and if the URLs of the originating page and the requested page satisfy the same-origin policy." To trace request sources, the proxy rewrites "HTML forms, links, and other means of initiating HTTP requests" with a random URL token. The token and response URL is then stored by the proxy for future reference. When the browser makes a request, the proxy looks up the URL token and compares the destination URL with the request's referring URL. If they do not match, the request is con-

sidered suspicious. The proxy also detects intranet cross-domain requests and validates them with explicit user confirmation.

This approach has several downsides. First, it mandates that all user traffic must go through a TLS man-in-the-middle proxy. Second, many applications use JavaScript and other active content to dynamically create forms and issue HTTP requests directly from the client, making rewriting of such requests impossible in a proxy. Third, rewriting all HTML content may have unpleasant latency implications. Finally, some cross-domain requests can be legitimate (*e.g.,* for federated login/single-sign-on or for showing users personalized content when they visit a link); RequestRodeo does not support these cases.

BEAP. Browser-Enforced Authenticity Protection (BEAP) [21] attempts to infer whether a user intended the browser to issue a particular request. User intent is determined via a browser extension that monitors user actions. For example, if the user enters a URL into the address bar, the request is considered intended. However, if the user clicks on a link in an e-mail message or from the browser history list, the request is considered unintended. The browser extension strips unintended requests of "sensitive" authentication tokens, which include (1) all *session* (non-persistent) cookies sent over POST requests and (2) HTTP Authorization headers. Persistent cookies are treated as non-sensitive, as are session cookies sent over GET requests. BEAP authors note that these rules were generated by analyzing real-world applications. However, this analysis does not hold today: for some web sites we analyzed, such as Woot or Google, some sensitive cookies were persistent.

Cross-site Request Forgeries: Exploitation and Prevention. In their 2008 paper [39], Zeller and Felten identified a variety of high-profile CSRF vulnerabilities and gave general guidelines for how to prevent them, recommending tokenization and using POSTs for state modifying requests. The authors also provided two tools for CSRF prevention: a plug-in for developers to perform automatic tokenization, and a plug-in for browsers to block cross-domain POST requests (unless the site has an Adobe cross-domain policy that specifies exceptions).

We believe the auto-tokenization plug-in is useful, but suffers from the same drawbacks as other tokenization frameworks (Section 3.1.1). The client-side solution is both too coarse in how it handles POST requests and too permissive when it freely passes any GET requests through, including potentially dangerous GETs (see examples in Section 3.1.1).

Adobe Cross-Domain Policy. Adobe cross-domain files specify how Adobe clients (such as Flash Player and Reader) should make requests across domains. These policies are hosted on remote domains and grant "read access to data, permit a client to include custom headers in cross-domain requests, and are also used with sockets to grant permissions for socket-based connections." [1]

We believe these policies do not provide enough control over ambient authority to prevent CSRFs without restricting functionality. For example, developers cannot specify names of cookies to which the policies apply or control valid embedding.

3.3 Comparing to Our Approach

In summary, we found that server-side approaches required non-trivial development effort, failed to protect web applications against state-modifying GET requests, and were vulnerable to some types of attacks (*e.g.,* token extraction). Client-side approaches used heuristics to identify "suspicious" requests and "important" authentication tokens. However, because client-side solutions cannot know developer intentions, they were prone to either false positives which break sites or to false negatives which miss attacks due to being too lenient with allowed requests.

In contrast to these approaches, our solution takes a middle ground. We let web application developers write policies specifying how browsers should handle data and requests for their sites; this eliminates guesswork in approaches like CsFire, BEAP, or RequestRodeo. We also add browser support for handling policy enforcement, avoiding the pitfalls such as forgetting to check a CSRF token. This hybrid client/server design build on principles from two recent efforts aimed at securing sites, Content Security Policy (CSP) [35] and Cross-Origin Resource Sharing (CORS) [37]. Neither CSP nor CORS are designed for mitigating CSRF, instead aiming at preventing XSS and relaxing the same-origin policy, respectively. Our solution fills this gap.

In concurrent work, the popular Firefox extension NoScript [22] recently released a module called ABE (Application Boundaries Enforcer) [23], which provides generic firewall-like protection for web sites, including an ability to stop CSRF by stripping authentication headers from requests that match policy rules. Policies may be configured by users or stored on a site in a "rules.abe" file. Although this general approach is similar to our solution, there are important differences as well. ABE's design will cause the browser to make an extra request for every new domain, even if that site doesn't have "rules.abe". Furthermore, since NoScript allows "rules.abe" to only be served over HTTPS, web sites must have an HTTPS server to use ABE for CSRF protection. While ABE allows stripping authentication data from requests, it cannot set policies for individual cookies, running into the same limitations as CsFire. Our solution does not have these drawbacks, and unlike NoScript, it does not require any user involvement and is implemented directly in the browser. Finally, and most importantly, we not only design a mechanism, but also evaluate the feasibility of modifying real web sites to use it in Section 6.1; this evaluation shines light on practicality of not just our solution, but ABE as well.

4. DESIGN

Learning from our analysis of pitfalls in existing defenses, our anti-CSRF design should be (1) **easy for developers** (*e.g.,* unlike checking of anti-CSRF tokens), (2) **transparent to users** (*e.g.,* unlike CsFire or NoScript which ask users to define or approve policies), (3) **backwards compatible** to not break legitimate sites that do not opt in to our defense (*e.g.,* unlike CsFire breaking OpenID), and most importantly, we should (4) **address the root cause** of CSRFs, namely ambient authority, to provide more comprehensive coverage against CSRF than existing solutions.

Recall from Section 2 that fundamentally, CSRFs result when browsers exercise ambient authority: (1) browsers automatically attach credentials to HTTP requests and (2) web application developers treat the presence of those credentials in a request as implicit authorization for some action. Keeping these root causes in mind, we make the following key observations.

- While splitting authentication and authorization is a classical best practice in computer security [19, 36], it is underutilized on the web. In our experience, most websites use a single token (such as a session cookie or the HTTP basic authentication header) for both authentication and authorization. Decoupling this concept on the web could have significant security benefits.
- The developers of site X are in the best position to determine which other sites are authorized to cause the user's browser to issue HTTP requests that perform state-modifying actions on X. However, server-side code on site X cannot reliably tell which other site caused the user's browser to issue an HTTP request.
- On the other hand, browsers know the full context in which requests are issued. Specifically, browsers know the full DOM layout and can infer whether a request was triggered by a user action, a nested iframe, or a redirection.

In light of these observations, we introduce a new browser mechanism called *Allowed Referrer Lists (ARLs)*, which restricts a browser's ability to send ambient authority credentials with HTTP requests. Sites wishing to be resilient against CSRF must opt in to use ARLs. With ARLs, participating sites specify which state (*e.g.,* specific cookies) serves authorization purposes. Sites also specify a whitelist of allowed referrer URLs; browsers are allowed to attach authorization state to HTTP requests made from these URLs only. This policy is transmitted to the user's browser in an HTTP header upon first visit to a site, before any authorization state is stored.

This approach capitalizes on the fact that browsers know the browsing context, namely determining which web site principal issued an HTTP request, while web developers understand site-specific authorization semantics, namely whether a request-issuing web site should be authorized. By having developers only specify policies and letting browsers carry out enforcement, we ease the enforcement burden on developers. By having only participating sites specify policies to receive CSRF protection, we leave other sites' behavior unchanged, thus providing backward compatibility.

4.1 Identifying and Decoupling Authentication and Authorization

To use ARL, developers should identify and decouple the credentials they use for authenticating and authorizing users' requests.

First, developers must determine which credential is being used to identify users. Recall from Section 2.1 that developers use various methods to identify users: HTTP authentication, source IP, TLS client certificates, and cookies. In many cases we studied, applications use a single cookie to authenticate users.

Next, developers should create a credential for *authorizing* user requests. For example, developers may choose to set a new authorization cookie called *authz* on the user's browser. Any HTTP request not bearing the authorization credential must not be allowed to induce state-changing behavior (even if the request has the authentication credential). A request bearing *only* the authorization credential should be treated as an unauthenticated request.

Some sites can benefit from ARLs without needing to separate authentication and authorization credentials, whereas others will require this separation to properly work with ARLs. In Section 6, we will further discuss these two cases.

4.2 Defining ARL Policies

To define an ARL policy, developers list authorization credentials, which may include specific cookie name(s), HTTP Authentication, or even the whole request (for source IP or TLS client cert authentication). If a credential is mentioned in an ARL policy, we say that the credential has been *arled*. By default, *arling* a credential prevents that credential from ever being attached to any HTTP request. Developers specify additional rules to relax this restriction in a least-privilege way. ARLs have two core directives:

- **allow-referrers**: Developers provide a whitelist of referrers that can issue authorized requests. A referrer is a URL with optional wildcards. Referrers can be as generic as *https://*.bank.com/** or as specific as *https://bank.com/accounts/modify.php*. Wildcards and paths allow web sites to be flexible in expressing their trust relationships. For example *bank.com* may be owned and run by the same entity as *broker.com*, but *bank.com* may only want to receive authorized requests from *https://broker.com/transfer/**.
- **referrer-frame-options**: Malicious sites could cause a protected credential to be sent by embedding content (*e.g.,* in an iframe) from a referrer specified in the ARL policy. We therefore allow developers to restrict framing of referrers. Similarly to the HTTP Header Frame Options RFC [29], we allow three framing options: DENY, SAMEORIGIN, or ALLOW-FROM. DENY states that the referrer must not be framed when issuing an authorized request. SAMEORIGIN allows the referrer to make an authorized request while being framed by "itself" (*i.e.,* by a URL matching the ARL whitelist entry for that referrer) or by the target of the request. The ALLOW-FROM option takes additional values in the form of domain URLs such as *https://broker.com/*. We allow framing depth of at most one embedding to prevent attackers from mounting attacks on the embedder. If this directive is omitted, the default value is DENY.

```
arl {
  apply-to-cookie = authz,
  allow-referrers = https://*.bank.com/*
                    https://broker.com/finance/*,
  referrer-frame-options = SAMEORIGIN
}
```

Figure 3: ARL policy example. With this policy, bank.com forbids the browser from sending the `authz` cookie, except when the request was generated by *https://*.bank.com/ or *https://broker.com/finance/**, and only if the requesting page was framed by a page of the same origin.**

```
arl {
  apply-to-http-auth = true,
  allow-referrers = https://*.bank.com/*
                    https://broker.com/finance/*,
  referrer-frame-options = SAMEORIGIN
}
```

Figure 4: Restricting HTTP Authentication. With this policy, bank.com forbids the browser from sending the HTTP Authentication header, except when the request was generated by *https://*.bank.com/ or *https://broker.com/finance/**, and only if the requesting page was framed by a page of the same origin.**

Using these two directives, developers can generate simple yet powerful policies. Figure 3 shows a policy that may be used by *bank.com*. Here, the cookie `authz` is arled and *https://*.bank.com/** and *https://broker.com/finance/** are listed as the only referrers. This means that the browser will only attach the `authz` cookie to HTTP requests generated by an element from *https://*.bank.com/** or *https://broker.com/finance/**. Note that the *allow-referrers* directive specifies not only the domain, but also the scheme (HTTPS in this case) from which the requesting element must have been loaded. This differs from the Secure attribute of a cookie, which only specifies how to send the cookie itself. By including the HTTPS scheme, a web application developer specifies that protected credentials are never sent by an element not loaded over TLS. This control is not possible with any of the techniques we studied. This policy also states that the `authz` cookie may be attached only if the referrer was either not framed or framed only by a page from SAMEORIGIN (either target *bank.com* or referrer *broker.com/finance*). Note that unlike origin-based framing rules in the X-Frame-Options HTTP header, our rules will check for the full *broker.com/finance* path, *e.g.,* to avoid potentially malicious framing from *broker.com/forum* that *bank.com* did not intend.

As another example, Figure 4 shows how an ARL policy can be applied to HTTP basic authentication credentials instead of cookies (though we suggest that developers use cookies for authorization). Finally, Figure 5 shows how ARLs can be used to disallow any requests to a particular destination unless the request is being made by a particular referrer (specified via the **apply-to-requests-to** directive). This directive can protect sites which rely on source IP or TLS client certificates for authorization.

It is important to reiterate that ARLs should be applied to authorization, *not* authentication credentials. That is, from a security point of view, it is always acceptable for a web site to know from which user's browser an HTTP request came. However, it is not

```
arl {
  apply-to-requests-to = https://accounts.bank.com/modify,
  allow-referrers = https://accounts.bank.com/*,
  referrer-frame-options = DENY
}
```

Figure 5: Disallowing requests. Here, *bank.com* forbids *any* requests (with or without credentials) to *https://accounts.bank.com/modify*, except when the request was generated by *https://accounts.bank.com/, and only if the requesting page was not framed.**

always acceptable for sites to take actions based on those requests.

Application: Defeating Login and Logout CSRFs. Barth, Jackson, and Mitchell described a Login CSRF attack whereby an attacker "forges a login request to an honest site using the attacker's user name and password at that site" [5]. A successful attack causes the user to be logged into the site with the attacker's credentials, allowing the attacker to "snoop" on the user.

A Logout CSRF attack is similar in that it allows attackers to disrupt the user's session on legitimate sites. Many sites implement logout by having users visit a URL (*e.g., site.com/Logout*). For many sites, this URL is vulnerable to a CSRF attack: malicious sites can embed an iframe pointing to a site's logout URL and cause any visitor of the malicious site to be logged out of the legitimate site. Google, for example, is vulnerable to this attack.

ARLs can be used to protect web applications against both login and logout CSRF attacks. To protect against login CSRF, a web site may set an arled dummy authorization credential when displaying the login form. Then, the site should verify that the dummy authorization credential is returned along with the user's other login credentials before setting the real authorization and authentication credentials. Similarly, logout CSRF can be stopped by simply checking that the (real) arled authorization credential is present before signing the user out.

4.3 Enforcing ARL Policies

Once the browser obtains an ARL policy for a site, the browser must examine each outgoing HTTP request's context to check whether any of a site's current ARL policies apply, and if so, whether or not to attach arled credentials to the request.

Referrers. To enforce the "allow-referrers" directive, we leverage the fact that browsers already determine each request's referrer. Broadly speaking, a referrer is the URL of the item which led to the generation of a request. For example, if a user clicks on a link, the referrer of the generated request is the URL of the page on which the link was shown. The referrer of an image request is the URL of the page in which that image was embedded.

Browsers also handle complex cases of referrer determination. For the redirect chain $a.com \Rightarrow b.com \Rightarrow c.com$, modern browsers (IE, Chrome, Firefox, and Safari) will choose *a.com* as the referrer if the redirection from $b.com \Rightarrow c.com$ was made via the HTTP `Location` header and will choose *b.com* as the referrer if the redirection from $b.com \Rightarrow c.com$ was made by assigning *window.location* in JavaScript. This is not arbitrary. If the redirect is caused by the HTTP header, then none of the subsequent page content is interpreted by the browser. However, a JavaScript page redirect can occur at any point in time after an arbitrary amount of interaction with the user. Further discussion of this issue is beyond the scope of this paper, but this issue shows that browsers already take great care to identify the source of every request so that they can enforce the same-origin policy — the foundation of many web security properties. We therefore use the default definition of referrer. If a credential is arled, then the request's referrer must match one of the "allowed-referrers". Otherwise, the credential will not be sent with the request.

Frame-options. To enforce the "referrer-frame-options" directive, we are helped by the fact that browsers maintain the page embedding hierarchy (*e.g.,* to enforce the X-FRAME-OPTIONS header). Browsers record complicated cases with nested and dynamically created iframes. We simply consult this internal hierarchy when enforcing "referrer-frame-options".

If a credential (*e.g.,* cookie) is arled, then the embedding hierarchy of the referrer must match the "referrer-frame-options" policy as defined above. Our mechanism extends frame-checking to also consider popup windows to prevent attackers from causing CSRFs by opening victim pages in popups.

Mixed content sites. Some web sites mix HTTP and HTTPS content when presenting web pages to users. For example, a site may choose to serve images over HTTP to increase performance, while all JavaScript and HTML are served over HTTPS. This practice may introduce certain security vulnerabilities. For example, since network attackers can manipulate content sent over HTTP, they can modify "secure" cookies. That is, although cookies bearing the "secure" attribute will only ever be *sent* over HTTPS, they can be *set* or overwritten via a `Set-cookie` header over HTTP [8].

To avoid these vulnerabilities, we introduce two rules for browsers implementing ARLs:

- If an ARL policy is received over HTTP, it may only overwrite an old policy if the old policy was also received over HTTP.
- If an ARL policy is received over HTTPS, it may overwrite any old policy.

These rules prevent ARL hijacking by network attackers who are able to insert an ARL header into an HTTP response.

Requests with no referrer. In several situations, a request may completely lack a referrer. Examples of such events are when users type URLs into the browser location bar, click on a bookmark, or follow a link from an e-mail. In these cases, any arled credential should not be sent, while all the other state should be sent. Note that this means that web sites will not initiate any state changing behavior as a result of this request, but can still show personalized content to the user (since the authentication credentials were sent).

5. IMPLEMENTATION

The implementation and deployment of ARLs involves several steps. First, web applications need to be modified to implement ARL policies. This includes designing a policy, potentially separating authentication and authorization credentials, and then inserting the policy into all relevant HTTP responses. Second, the user's browser needs to be modified to understand, store, and enforce ARL policies for arled credentials.

Modifications to web applications are specific to each application's own logic and framework. We explore how this is done with real-world applications in Section 6.1. In this section, we focus on options for delivering policies and modifications that browsers need to support ARLs.

5.1 Policy Delivery

The simplest delivery option, and one used in our implementation, is to piggyback ARL policies onto existing cookie definitions. Today, the Set-Cookie header allows cookies to specify constraints through attributes such as HttpOnly, Secure, Domain, and Path. We added a new "arl" attribute (bearing an ARL policy) to cookies, allowing each cookie to specify its own ARL policy. Unfortunately, this delivery method makes it difficult to address HTTP authentication and IP-based ambient authority.

An alternate ARL policy delivery mechanism is to integrate ARLs as a new directive for Content Security Policy (CSP) [35], which is specified through its own HTTP header. CSP already

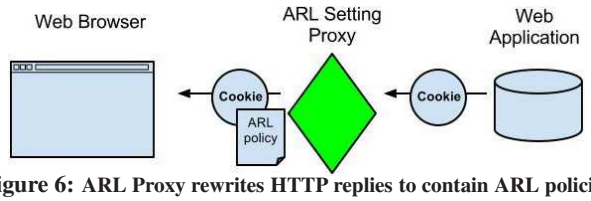

Figure 6: ARL Proxy rewrites HTTP replies to contain ARL policies.

specifies a number of rules regulating content interaction on web applications, primarily for mitigating XSS and data injection [35]. Although we did not yet implement this approach, we recommend it over using cookie attributes due to added flexibility, expressiveness, and usability. For example, if a web application used multiple authorization tokens (*e.g.,* combining HTTP Authentication headers with cookies or using multiple authorization cookies), a developer would have to specify multiple (potentially duplicate) policies with the first approach but not with a centralized, HTTP-header-based policy. As well, HTTP headers can transmit longer ARL policies than cookies, which are typically limited to 4 kilobytes.

5.2 Browser Modification

To validate the ARL design, we implemented it in the Mozilla Firefox 3.6 browser. Although the 3.6 version of Firefox is slightly dated, it was the most stable version when this project began in 2011. We believe that our modifications and implications of our implementation generalize to more recent versions of Firefox and other browsers.

Rather than create a browser extension or plug-in (as done by CsFire or NoScript ABE), we decided to directly modify Firefox. First, we wanted to have direct access to internal representations of cookies, referrer information and frame embedding hierarchy. Second, we wanted to validate the feasibility of adding native browser support for ARLs.

For our proof-of-concept, we used attribute-based policy delivery described above. We modified Firefox in three key aspects:

- **Parsing**. We modified the cookie parsing code for HTTP responses to accept a new cookie attribute "arl" and parse the contents into an ARL policy.
- **Storage.** To store and retrieve all of the parsed ARL policies, we modified Firefox's cookie manager, cookie database (sqlite), and all of the related structures and methods to allow setting and retrieving a new "ARLPolicy" data structure for each cookie.
- **Enforcement.** Finally, we enforce ARL policies by adding code that checks each cookie's policy before attaching it to an HTTP request. Firefox already considers many factors before attaching cookies to requests, such as verifying that requests adhere to the same-origin policy or that the request is going over TLS (if the cookie's *Secure* attribute is set). We appended our ARL enforcement logic (described in Section 4.3) to this code.

In total, our modifications consisted of approximately 700 lines of C++ code spread across 13 files.

6. EVALUATION

We conducted experiments to answer several questions about the effectiveness and feasibility of ARLs, including: (1) how well do ARLs guard against CSRF attacks, (2) how difficult is it to write ARL policies for real sites, (3) how much developer effort is required to deploy ARLs, (4) does deploying ARLs break existing web applications, and (5) do ARLs have a performance impact?

To answer these questions, we first studied four open-source web applications with known CSRF vulnerabilities, implemented ARLs in the applications' code bases, and analyzed how well ARLs protected them against CSRF attacks. Second, we used the Fiddler SDK [20] to develop an ARL Setting Proxy that allowed us to experiment with ARL policies on arbitrary web sites without modifying their code. This proxy, implemented in 120 lines of JavaScript and illustrated in Figure 6, allowed us to study how ARLs interact with large, complex, on-line web applications such as Google. We also evaluated ARL performance using browser benchmarks.

6.1 Studying Applications with Source

We monitored the public security mailing list "full disclosure" during the summer of 2011 for CSRF reports. Using those reports, we chose four projects for which source code was available and for which the CSRF attacks were reproducible. The web applications we studied are summarized in Table 1.

For each application, we first performed an in-depth analysis to understand the intended authorization structure and then developed ARL policies to enforce that structure. We confirmed that without modifications, the applications were indeed vulnerable to CSRF attacks. Next, we deployed the applications with ARL policies and tested whether the attacks were now thwarted. Finally, we tested each application to check that ARL deployment did not inhibit any functionality. We report on an in-depth case study of one of the applications below. We found that ARL polices for the other three applications had a similar level of complexity, and that ARLs were effective at protecting these applications against CSRF attacks.

6.1.1 Case Study: UseBB

UseBB is an open-source web application for lightweight forum management. The project's web site advertises that rather than providing as many features as possible, UseBB strives to provide an easy and usable forum with only the most necessary features. UseBB is written in PHP with a MySQL database backend.

Initial vulnerability report and analysis. In the summer of 2011, a submission to the security mailing list *full-disclosure* reported "Multiple CSRF (Cross-Site Request Forgery) in UseBB". This report was for version 1.011 of UseBB, which we downloaded and analyzed.

This version of UseBB consisted of 25k lines of code spread over 83 source files. After studying UseBB, we understood that the application's state management worked as follows: when a user logs into the application, UseBB sets a single cookie, called usebb_sid, to authenticate further HTTP requests as coming from that user.

Building a corpus of CSRF attacks. The bug report to *full-disclosure* mentioned multiple vulnerabilities, but only identified one page as having a vulnerability. After studying the code, we discovered that UseBB had no CSRF protection on any of its pages. This resulted in any request bearing the *usebb_sid* cookie to be authenticated and authorized to perform arbitrary actions. For example, by exploiting this vulnerability, attackers may have been able to perform a variety of attacks including changing user e-mail addresses, name, or adding illegitimate administrator accounts. We implemented these attacks and formed a corpus of CSRF attacks.

Developing an ARL Policy. Next, we developed an ARL policy and analyzed its effectiveness. First, so that we could study an ARL policy in action, we deployed an instance of UseBB on an internal network at the URL: `http://usebb.com`. Next, we verified that all CSRF attacks from our corpus worked as expected. Then, we developed an ARL policy and deployed it via the ARL Setting Proxy. The policy we developed is:

```
arl {
  apply-to-cookie = usebb_sid,
  allow-referrers = http://usebb.com/,
  referrer-frame-options = SAMEORIGIN
}
```

Next, we tested UseBB (now protected by ARLs) against our cor-

Application	Version	# of source files	Lines of Code	Type of Application
Selectapix	1.4.1	39	6k	Image gallery
UseBB	1.011	83	21k	Forum
PHPDug	2.0.0	133	25k	URL/link sharing app (similar to digg.com)
PoMMo	PR16.1	234	32k	Mailing list manager

Table 1: Summary of web applications we studied. All applications were written in PHP and used a MySQL backend database.

pus of CSRF attacks and found the attacks to be no longer functional. Our ARL policy fully protected this deployed application instance against any CSRF attacks that leverage the *usebb_id* cookie.

Deploying an ARL Policy in UseBB. The next step was to deploy the ARL policy within the UseBB application itself (rather than via the proxy). We explored several ways in which this may be done. For example, one approach we used to deploy ARLs in UseBB was to add one line to the Apache server configuration file:

```
Header add X-ARL-Policy "arl{ \
apply-to-cookie = usebb_sid, allow-referrers = self, \
referrer-frame-options = SAMEORIGIN }"
```

To use this method, the web application developer must have access to the global Apache configuration file, which may be unavailable on some shared hosting providers. In that case, the developer could also deploy ARLs by modifying the *.htaccess* file (*i.e.,* the local web server configuration file). We verified that this deployment strategy worked as well. The additions to *.htaccess* were:

```
<IfModule mod_headers.c>
 Header set X-ARL-POLICY "arl{ \
 apply-to-cookie = usebb_sid, allow-referrers = self, \
 referrer-frame-options = SAMEORIGIN }"
</IfModule>
```

A similar modification is possible for local configuration files for IIS web servers. We verified that these additions were possible, but did not deploy UseBB via IIS. There may, however, be cases where developers are not allowed to create or modify even local web server configuration files. In such cases, the developer would have to modify the application source directly to implement ARLs. Since UseBB is written in PHP, we accomplished this by adding the following line of code to files which produce HTML code:

```
header('X-ARL-Policy: arl{ \
 apply-to-cookie = usebb_sid, allow-referrers = self, \
 referrer-frame-options = SAMEORIGIN }');
```

We implemented all of the aforementioned approaches and verified that each of them secured UseBB against our corpus of CSRF attacks.

Comparing ARLs to Traditional Patch. The official repository of UseBB was later updated to fix the CSRF vulnerabilities in version 1.0.12 of the code. The developers protected UseBB against CSRFs by writing a custom CSRF token framework. By manually inspecting changes introduced in version 1.0.12, we counted approximately 190 line changes that were related to the new CSRF defense. We tested version 1.0.12 against our corpus of CSRF attacks and found that it did prevent them. We believe our solution to be better than the traditional patch in several ways. First, as we saw, implementing ARLs requires many fewer code modifications. Second, while ARLs would protect against any new CSRF attacks that leverage the *usebb_id* cookie, the patch would not — additional code would have to be written for any new pages added to UseBB.

Backwards Compatibility. Though ARLs clearly protected UseBB against CSRF attacks, we wanted to investigate whether the deployment of ARLs impeded any UseBB functionality. Since the UseBB source code did not include any unit (or other type of) tests, we performed all functionality testing manually. Even though UseBB did not have separate authentication and authorization credentials, we found that all existing legitimate functionality was maintained.

6.2 Studying Large Sites without Source

Next, we studied the feasibility of ARLs in real-world, proprietary web sites. Modern web applications, such as Google's Gmail, are incredibly complex. The server source code is proprietary, and the code that is shipped to the browser is often obfuscated or has been run through an optimizer that makes the code difficult to read. Furthermore, to optimize user experience, such sites set dozens of cookies on the client, some of which represent preferences, while others are responsible for authenticating the user's requests.

We chose to study three web applications which we believe to be quite complex and which, in our opinion, serve as good representatives of state-of-the-art web applications: Gmail, Facebook, and Woot (a flash deals e-commerce site owned by Amazon.com). For each application, we first identified the "important" application cookies. Next, we observed how those cookies were sent during normal operation. Finally, we created ARL policies for those cookies, deployed them using our ARL Setting Proxy, and examined whether normal functionality was maintained.

Selecting Important Cookies. Modern, large web applications use a large number of cookies. For example, Gmail and Facebook set around 40 cookies, while Woot sets about 20[1]. Many of these cookies have to do with preferences, screen resolution, locale, and other application state. Some of these cookies, however, deal with authentication and (potentially) authorization — these are the cookies that need to be protected by ARLs.

The majority of cookies have cryptic names, making it difficult to infer their intended function. We identified authentication cookies by experimentally removing cookies set in the browser until the web site refused to respond to requests or performed a signout. This cookie elimination process was done through manual analysis by individually removing each cookie and testing the result.

Using this strategy, we were able to narrow the set of all cookies down to just a few important cookies for each application. For Gmail, we found the important cookies to be `LSID`, `SID`, and `GX`; for Facebook the important cookies were `xs` and `c_user`; for Woot, the important cookies were `authentication` and `verification`.

Developing ARL Policies. Next, we needed to determine what the legitimate referrers were for sending these authentication cookies. We accomplished this by performing normal actions (such as sending and checking e-mail on Gmail, viewing photos and friends on Facebook, and browsing items on Woot) and observing the network traffic through the Fiddler web proxy.

Using these traces, we then developed an ARL policy for each site. The policies were less complex than we had assumed they would be. For example, the policy for Gmail was:

```
arl {
  apply-to-cookie = SID LSID GX,
  allow-referrers = https://accounts.google.com
                    https://mail.google.com,
  referrer-frame-options = DENY
}
```

[1]The number of cookies varied based on user actions.

Policies for Facebook and Woot were of similar complexity; that is, they mentioned the important cookies and only a handful of referrers. Furthermore, we found that these policies did not inhibit regular in-app functionality.

6.2.1 Splitting Authentication and Authorization

While the simple ARL policies above can support many interactions with Google, Facebook, and Woot, we discovered some desirable actions that these policies cannot support with the sites' existing cookie structure. For example, these policies are not compatible with the use of Google as a federated login provider or the use of Facebook's Like buttons on other sites[2].

Recall from Section 4.1 that web sites should clearly separate authentication credentials from authorization credentials. After doing so, it is only necessary to arl the authorization credential. The authentication credential can be sent about as before. We found that the limitations above are all due to the fact that none of our applications separate authentication and authorization in their cookies, and making this modification re-enables the unsupported functionality.

Embedded Widgets. Facebook's Like Button and Google's +1 Button are just two examples of a large class of "embedded widgets" that allow users to like, pin, tweet, and otherwise share a web page with their social graph. A web page that wants to enable a specific widget on its page includes HTML or JavaScript, usually provided by the social network, which renders an iframe sourcing the specific social network and displays the social button. The iframe's content is loaded from the social network; this has two key features. First, it prevents the host page from programmatically clicking on the widget. Second, it gives the widget access to the user's cookies from that social network, so that when the user clicks on the widget, the user's state in the social network can be properly affected.

When a user clicks on such a widget, the browser infers the referrer of the consequent request to be the social network and not the host page. This is because the content inside the iframe has already been loaded from the social network. Consequently, one would expect ARLs to work out of the box with embedded widgets. Indeed, this is the case with Google's +1 button. However, ARLs do not currently work with Facebook's Like button because when sending the HTTP request for the iframe's initial contents, the browser determines that the referrer is the host page, preventing any arled cookies from being sent. If Facebook used the authentication/authorization splitting paradigm as above, then only the authorization cookie would not be sent on the initial request, and the iframe could still be loaded with authenticated content (*e.g.,* the number of friends that have liked the host page), allowing the like button to work.

Exploring Federated Login. Federated identity management is the mechanism whereby web applications can rely on other web sites, such as Facebook Connect or OpenID, to authenticate users. One of our selected test apps, Woot, supports federated login from Google. That is, a user can click on a button on the Woot login page, which will redirect the user to Google, which will verify the user's authentication credentials and log them into Woot via another redirect. Unfortunately, since Woot did not separate authentication and authorization credentials, the only cookie we can arl is the single authentication cookie, which would then be stripped from redirection requests between the two parties. By splitting authentication and authorization credentials into two cookies (as above) and only arling the authorization credentials, federated login on Woot could be supported.

[2]This was an artifact of Facebook's implementation; ARLs supported Google's +1 button.

6.3 Browser Performance

To make sure ARLs are feasible for browsers to implement, we checked performance overhead of ARLs. More specifically, we set up a 4K benchmarking page that sets a cookie and, upon loading, sends the cookie back to the server using AJAX. We measured the latency of this HTTP response-request roundtrip with unmodified Firefox browser and with ARL-enabled Firefox and an ARL policy for the cookie. We evaluated two ARL policies: a 1-referrer policy, and a 30-referrer policy where only the last referrer was valid. We averaged our results over 100 runs. Our client ran on a Windows 7 laptop with 2.20GHz CPU and 2GB RAM, and our server ran on a Macbook Pro laptop with a 2.66GHz CPU with 8GB RAM, connected with a 100MBps network.

We found that the latency difference between unmodified and ARL-enabled browsers was negligible for requests with the small (single referrer) ARL policy. For the longer 30-referrer ARL policy, the median latency difference increased to 3ms (2%), which is still very acceptable. We expect most sites to have shorter ARL policies, and we note that our implementation of ARL parsing and matching was not optimized for performance.

7. DISCUSSION

Privacy. Adversaries may try to learn the referrer of a request by creating many cookies with different ARL policies and then seeing which ones are sent back. However, ARLs introduce no additional privacy leaks compared to the `Referer` header, which is already freely sent by browsers. Some organizations may need to conceal the referrer to prevent revealing secret internal URLs. To do this, they install proxies that remove the `Referer` header [17]. To prevent any additional referrer leaks from ARLs, these proxies (or ARL-enabled browsers) can strip all cookies and HTTP authentication headers from requests generated by following a link from an intranet site to an internet site. This should not negatively impact any functionality, since links from the intranet to outside sites should not be state-modifying.

Limitations. ARLs help protect against many types of CSRF attacks, but they are not a panacea. For example, ARLs are unable to protect against "same origin and path" attacks. If a page accepts legitimate requests from itself and somehow an attacker is able to forge attacks from that page, then ARLs may be ineffective.

Another limitation involves deployment. Until ARLs are supported in most browsers, web sites must use older CSRF defenses alongside ARLs. This is also true for any browser-based security feature. For example, many sites still have to use JavaScript-based framebusting because some older browsers still do not support the X-FRAME-OPTIONS HTTP header.

8. CONCLUSION

CSRF attacks are still a large threat on the web; 2012 was one of the most CSRF-active years on record [26]. In this paper, we give a background of CSRF attacks, highlighting their fundamental cause — ambient authority. Next, we study existing CSRF defenses and show how they stop short of a complete solution. We then present ARLs, a browser/server solution that removes ambient authority for participating web sites that want to be resilient to CSRF attacks. We implement our design in Firefox and evaluate it against a variety of real-world CSRF attacks on real web sites. We believe that ARLs are a robust, efficient, and fundamentally correct way to mitigate CSRF attacks.

9. ACKNOWLEDGEMENTS

We thank Collin Jackson for feedback about our approach.

10. REFERENCES

[1] Adobe. Cross-domain policy file specification, 2013. http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html.

[2] R. Auger. The Cross-Site Request Forgery (CSRF/XSRF) FAQ, 2010. http://www.cgisecurity.com/csrf-faq.html.

[3] M. Baldwin. OpenX CSRF Vulnerability Being Actively Exploited, 2012. http://www.infosecstuff.com/openx-csrf-vulnerability-being-actively-exploited/.

[4] A. Barth. The web origin concept, 2011. http://tools.ietf.org/html/draft-abarth-origin.

[5] 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.

[6] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen. Hypertext Transfer Protocol – HTTP/1.0, 1996. http://www.ietf.org/rfc/rfc1945.txt.

[7] blowdart. AntiCSRF, 2008. http://anticsrf.codeplex.com/.

[8] A. Bortz, A. Barth, and A. Czeskis. Origin Cookies: Session Integrity for Web Applications. In *Web 2.0 Security and Privacy (W2SP)*, 2011.

[9] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In *Web 2.0 Security & Privacy (W2SP)*, 2012.

[10] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against CSRF attacks. In *Lecture Notes in Computer Science*. Springer, Sept. 2011.

[11] Django Software Foundation. Cross Site Request Forgery protection, 2012. https://docs.djangoproject.com/en/dev/ref/contrib/csrf/.

[12] D. Esposito. Take advantage of asp.net built-in features to fend off web attacks. Microsoft MSDN, 2005. http://msdn.microsoft.com/en-us/library/ms972969.aspx.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999. http://www.ietf.org/rfc/rfc2616.txt.

[14] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless Attacks - Stealing the Pie Without Touching the Sill. In *CCS*, 2012.

[15] Inferno. Hacking CSRF Tokens using CSS History Hack, 2009. http://securethoughts.com/2009/07/hacking-csrf-tokens-using-css-history-hack/.

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

[17] A. Johnson. The referer header, intranets and privacy, 2007. http://cephas.net/blog/2007/02/06/the-referer-header-intranets-and-privacy/.

[18] K. Kotowicz. Cross domain content extraction with fake captcha, 2011. http://blog.kotowicz.net/2011/07/cross-domain-content-extraction-with.html.

[19] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, Nov. 1992.

[20] E. Lawrence. Fiddler Web Debugging Proxy, 2012. http://www.fiddler2.com/fiddler2/.

[21] Z. Mao, N. Li, and I. Molloy. *Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection*. Financial Cryptography and Data Security. Springer-Verlag, Berlin, Heidelberg, 2009.

[22] G. Maone. NoScript, 2012. http://noscript.net/.

[23] G. Maone. NoScript ABE - Application Boundaries Enforcer, 2012. http://noscript.net/abe/.

[24] Microsoft. Microsoft NTML, 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/aa378749(v=vs.85).aspx.

[25] Mozilla Wiki. Origin header proposal for csrf and clickjacking mitigation, 2011. https://wiki.mozilla.org/Security/Origin.

[26] National Institute of Standards and Technology (NIST). National vulnerability database, 2012. http://web.nvd.nist.gov/.

[27] OWASP: The Open Web Application Security Project. OWASP CSRFGuard Project, 2012. https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project.

[28] H. Purifier. CSRF Magic, 2012. http://csrf.htmlpurifier.org/.

[29] D. Ross and T. Gondrom. Http header frame options – draft-gondrom-frame-options-01, 2012. http://tools.ietf.org/html/draft-ietf-websec-frame-options-00.

[30] P. D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *Proceedings of the Second international conference on Engineering Secure Software and Systems (ESSoS)*, 2010.

[31] O. Shezaf. WHID 2008-05: Drive-by Pharming in the Wild, 2008. http://www.xiom.com/whid-2008-05.

[32] A. Sidashin. CSRF: Avoid security holes in your Drupal forms, 2011. http://pixeljets.com/blog/csrf-avoid-security-holes-your-drupal-forms.

[33] Softflare Limited. Hosting/e-mail account prices, 2011.

[34] S. Stamm, Z. Ramzan, and M. Jakobsson. Drive-by pharming, 2006. https://www.symantec.com/avcenter/reference/Driveby_Pharming.pdf.

[35] B. Sterne. Content Security Policy – unofficial draft 12, 2011. https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html.

[36] T. Y. Woo, T. Y. C, W. Simon, and S. S. Lam. Designing a distributed authorization service. In *INFOCOM*, 1998.

[37] World Wide Web Consortium. Cross-Origin Resource Sharing, 2012. http://www.w3.org/TR/cors/.

[38] M. Zalewski. Postcards from the post-XSS world, 2012. http://lcamtuf.coredump.cx/postxss/.

[39] W. Zeller and E. W. Felten. Cross-Site Request Forgeries: Exploitation and prevention, 2008. www.cs.utexas.edu/users/shmat/courses/library/zeller.pdf.

[40] Z. Zorz. Facebook spammers trick users into sharing anti-csrf tokens, 2011. http://www.net-security.org/secworld.php?id=11857.