Robust Defenses for Cross-Site Request Forgery

Adam Barth
Stanford University
abarth@cs.stanford.edu

Collin Jackson
Stanford University
collinj@cs.stanford.edu

John C. Mitchell
Stanford University
mitchell@cs.stanford.edu

ABSTRACT

Cross-Site Request Forgery (CSRF) is a widely exploited web site vulnerability. In this paper, we present a new variation on CSRF attacks, *login CSRF*, in which the attacker forges a cross-site request to the login form, logging the victim into the honest web site as the attacker. The severity of a login CSRF vulnerability varies by site, but it can be as severe as a cross-site scripting vulnerability. We detail three major CSRF defense techniques and find shortcomings with each technique. Although the HTTP `Referer` header could provide an effective defense, our experimental observation of 283,945 advertisement impressions indicates that the header is widely blocked at the network layer due to privacy concerns. Our observations do suggest, however, that the header can be used today as a reliable CSRF defense over HTTPS, making it particularly well-suited for defending against login CSRF. For the long term, we propose that browsers implement the `Origin` header, which provides the security benefits of the `Referer` header while responding to privacy concerns.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Design, Experimentation

Keywords

Cross-Site Request Forgery, Web Application Firewall, HTTP `Referer` Header, Same-Origin Policy

1. INTRODUCTION

Cross-Site Request Forgery (CSRF) is among the twenty most-exploited security vulnerabilities of 2007 [10], along with Cross-Site Scripting (XSS) and SQL Injection. In contrast to cross-site scripting, which has received a great deal of attention [14], and the effective mitigation of SQL injection through parameterized SQL queries [8], cross-site request forgery has received comparatively little attention. In a CSRF attack, a malicious site instructs a victim's browser to send a request to an honest site, as if the request were part of the victim's interaction with the honest site, leveraging the victim's network connectivity and the browser's state, such as cookies, to disrupt the integrity of the victim's session with the honest site.

For example, in late 2007 [42], Gmail had a CSRF vulnerability. When a Gmail user visited a malicious site, the malicious site could generate a request to Gmail that Gmail treated as part of its ongoing session with the victim. In November 2007, a web attacker exploited this CSRF vulnerability to inject an email filter into David Airey's Gmail account [1].[1] This filter forwarded all of David Airey's email to the attacker's email address, which allowed the attacker to assume control of `davidairey.com` because Airey's domain registrar used email authentication, leading to significant inconvenience and financial loss.

In this paper, we examine the scope and diversity of CSRF vulnerabilities, study existing defenses, and describe incremental and new defenses based on headers and web application firewall rules. We introduce *login cross-site request forgery attacks*, which are currently widely possible, damaging, and under-appreciated. In login CSRF, an attacker uses the victim's browser to forge a cross-site request to the honest site's login URL, supplying the *attacker's* user name and password. A vulnerable site will interpret this request and log the victim into the site as the attacker. Many web sites, including Yahoo, PayPal, and Google, are vulnerable to login CSRF. The impact of login CSRF attacks vary by site, ranging from allowing the attacker to mount XSS attacks on Google to allowing the attacker to obtain sensitive financial information from PayPal.

There are three widely used techniques for defending against CSRF attacks: validating a secret request token, validating the HTTP `Referer` header, and validating custom headers attached to XMLHttpRequests. None of these techniques are satisfactory, for a variety of reasons.

1. The most popular CSRF defense is to include a secret token with each request and to validate that the received token is correctly bound to the user's session, preventing CSRF by forcing the attacker to guess the session's token. There are a number of variations on this approach, each fraught with pitfalls, and even sites

[1]David Airey later repudiated this incident [2].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

Copyright 2008 ACM 978-1-59593-810-7/08/10 ...$5.00.

that implement the technique correctly often overlook their login requests because login request lack a session to which to bind the token.

2. The simplest CSRF defense is to validate the HTTP `Referer` header, preventing CSRF by accepting requests only from trusted sources. While effective in principle, this technique must deal with requests that lack a `Referer` header entirely. Sites can either process these requests or block them. If a site processes requests that lack a `Referer` header, the defense is ineffective because the `Referer` header can be suppressed by an attacker. If the site refuses to process these requests, our experimental measurements indicate that the site will exclude an appreciable fraction of users.

3. XMLHttpRequest's popularity has increased recently with more sites implementing AJAX interfaces. Sites can defend against CSRF by setting a custom header via XMLHttpRequest and validating that the header is present before processing state-modifying requests. Although effective, this defense requires sites to make all state-modifying requests via XMLHttpRequest, a requirement that prevents many natural site designs.

`Referer` validation is an appealing CSRF defense, but the technique is hampered by the widespread suppression of the `Referer` header. To evaluate this defense, we conducted an experiment to determine how frequently, and under what circumstances, `Referer` header is blocked. We placed advertisements on two different advertising networks that caused 283,945 browsers displaying the advertisements to issue network requests to servers in our laboratory. Our results show that although the `Referer` header is suppressed often over HTTP, the header is rarely suppressed over HTTPS, letting current sites prevent CSRF by using HTTPS and strict `Referer` validation.

To create a robust CSRF defense, we propose that browsers include an "`Origin`" header with POST requests. This header provides the security benefits of the `Referer` header while addressing the privacy concerns that have lead to the widespread suppression of the `Referer` header. The `Origin` header lets sites defend against CSRF by deploying a few simple web application firewall rules. This mechanism also protects login forms without requiring additional effort from the site's developers.

Although CSRF defenses are necessary to protect session integrity, other session integrity attacks are possible, even against sites without XSS or CSRF vulnerabilities. We describe other attacks on session initialization in which the user becomes authenticated to the honest site as the attacker. Although similar to login CSRF, these attacks do not require CSRF vulnerabilities. We describe session initialization vulnerabilities in OpenID [13], PHP cookieless session management [37], and HTTPS `Secure` cookies [40]. For each vulnerability, we propose an improved session management protocol to prevent attacks on session initialization.

Contributions. Our main contributions include:

1. An explanation of the CSRF threat model, including often-overlooked variations based on network connectivity and login CSRF. We demonstrate the severity of login CSRF vulnerabilities by describing the consequences of the vulnerability for a small sample of the many widely used web sites that are vulnerable.

2. A study of current browser behavior, including experimental measurement of `Referer` header suppression based on 283,945 advertising impression on two advertisement networks. Based on our experimental data, we propose a refinement to `Referer` validation: employ HTTPS and strict `Referer` validation. This technique is secure because browsers ensure the integrity of the `Referer` header and is compatible with 99.9% of the web users we observed in our experiment.

3. A proposal for an `Origin` header that contains only the the scheme, host, and port parts of the referring URL, addressing the privacy concerns of the `Referer` header while containing the information necessary for CSRF defense. For browsers, we have implemented this proposal as a 466-line extension to Firefox and as a eight-line patch to WebKit. For sites, we have implemented `Origin` validation in three lines of ModSecurity, a web application firewall for Apache.

4. A study of related session initialization vulnerabilities and defenses for OpenID, PHP cookieless sessions, and HTTPS cookies. We implement our cookie defense as a 202-line extension to Firefox.

Organization. The remainder of the paper is organized as follows. Section 2 reviews the threat model. Section 3 provides examples of login CSRF. Section 4 analyzes existing defenses using experimental data. Section 5 proposes the `Origin` header as a defense mechanism. Section 6 generalizes login CSRF to other session initialization vulnerabilities. Section 7 describes related work. Section 8 concludes.

2. CSRF DEFINED

In a *cross-site request forgery* (CSRF) attack, the attacker disrupts the integrity of the user's session with a web site by injecting network requests via the user's browser. The browser's security policy allows web sites to send HTTP requests to any network address. This policy allows an attacker that controls content rendered by the browser to use resources not otherwise under his or her control:

1. **Network Connectivity.** For example, if the user is behind a firewall, the attacker is able to leverage the user's browser to send network requests to other machines behind the firewall that might not be directly reachable from the attacker's machine. Even if the user is not behind a firewall, the requests carry the user's IP address and might confuse services relying on IP address authentication [36].

2. **Read Browser State.** Requests sent via the browser's network stack typically include browser state, such as cookies, client certificates, or basic authentication headers. Sites that rely on this authentication state might be confused by these requests.

3. **Write Browser State.** When the attacker causes the browser to issue a network request, the browser also parses and acts on the response. For example, if the response contains a `Set-Cookie` header, the browser will modify its cookie store. These modifications can lead to subtle attacks, which we describe in Section 3.

In-Scope Threats. We consider three different threat models, varying by attacker capability:

- **Forum Poster.** Many web sites, such as forums, let users to supply limited kinds of content. For example, sites often permit users to submit passive content, such as images or hyperlinks. If an attacker chooses the "image's" URL maliciously, the network request might lead to a CSRF attack. The *forum poster* can issue requests from the honest site's origin, but these requests cannot have custom HTTP headers and must use the HTTP "GET" method. Although the HTTP specification [6] requires GET requests to be free of side effects, some sites do not comply with this requirement.

- **Web Attacker.** A *web attacker* is a malicious principal who owns a domain name, e.g. `attacker.com`, has a valid HTTPS certificate for `attacker.com`, and operates a web server. These capabilities can all be obtained for $10. If the user visits `attacker.com`, the attacker can mount a CSRF attack by instructing the user's browser to issue cross-site requests using both the GET and POST methods.

- **Network Attacker.** A *active network attacker* is a malicious principal who controls the user's network connection. For example, an "evil twin" wireless router or a compromised DNS server can be exploited by an attacker to control the user's network connection. These attacks require more resources than web attacks, but we consider this threat in-scope for HTTPS sites because HTTPS is designed to protect against active network attacks.

Out-of-Scope Threats. There are a number of related threat models we do not consider in this paper. CSRF defenses are complementary to defenses against these threats.

- **Cross-site Scripting (XSS).** If the attacker is able to inject script into a site's security origin, the attacker can disrupt both the integrity and confidentiality of the user's session with the site. Some XSS attacks involve network requests, for example to transfer the user's bank balance to the attacker, but CSRF defenses do not attempt to guard against these attacks. To be secure, a site must implement XSS and CSRF defenses.

- **Malware.** If the attacker is able to run malicious software on the user's machine, the attacker can compromise the user's browser and inject script into the honest web site's security origin. Browser-based defenses are helpless against such an attacker because the malware attacker can replace the browser with a browser of malicious design.

- **DNS Rebinding.** Like CSRF, DNS rebinding [25] can be used to obtain network connectivity to a server of an attacker's choice using the browser's IP address. Web servers that are behind firewalls or that use the IP address of the browser to make policy decisions require DNS rebinding defenses. Although DNS rebinding attacks often have a similar purpose to CSRF attacks, they require different defenses. A simple DNS rebinding defense is to validate the `Host` header of HTTP requests to ensure that it contains an expected value. An alternative DNS rebinding defense is to filter DNS traffic, preventing external DNS names from resolving to private IP addresses.

- **Certificate Errors.** If the user is willing to click through HTTPS certificate errors, much of the protection afforded by HTTPS against network attackers evaporates. A number of researchers [38, 31, 24] have addressed this threat model, but, in this paper, we assume users do not click through certificate errors.

- **Phishing.** Phishing attacks [12] occur when an attacker's web site solicits authentication credentials from the user. Phishing attacks can be very effective because users find it difficult to distinguish the real site from a fake web site [11].

- **User Tracking.** Cross-site requests can be used by cooperating web sites to build a combined profile of a user's browsing activities. Most browsers include third-party cookie blocking features that are designed to discourage such tracking, but these features can be circumvented [26].

3. LOGIN CSRF

Most discussions of cross-site request forgery focus on requests that mutate server-side state, either by leveraging browser's network connectivity or by leveraging the browser's state. CSRF attacks that mutate browser state have received less attention even though these attacks also disrupt the integrity of the user's session with honest sites. In a *login CSRF* attack, the attacker forges a login request to an honest site using the attacker's user name and password at that site. If the forgery succeeds, the honest server responds with a `Set-Cookie` header that instructs the browser to mutate its state by storing a session cookie, logging the user into the honest site as the attacker. This session cookie is used to bind subsequent requests to the user's session and hence to the attacker's authentication credentials. Login CSRF attacks can have serious consequences, depending on other site behavior:

Search History. Many search engines, including Yahoo! and Google, allow their users to opt-in to saving their search history and provide an interface for a user to review his or her personal search history. Search queries contain sensitive details about the user's interests and activities [41, 4] and could be used by an attacker to embarrass the user, to steal the user's identity, or to spy on the user. An attacker can spy on a user's search history by logging the user into the search engine as the attacker; see Figure 1. The user's search queries are then stored in the attacker's search history, and the attacker can retrieve the queries by logging into his or her own account.

PayPal. PayPal lets its users transfer funds to each other. To fund a PayPal account, users enroll their credit card or their bank account. A web attacker can use login CSRF to mount the following attack:

1. The victim visits a malicious merchant's site and chooses to pay using PayPal.

2. As usual, victim is redirected to PayPal and logs into his or her account.

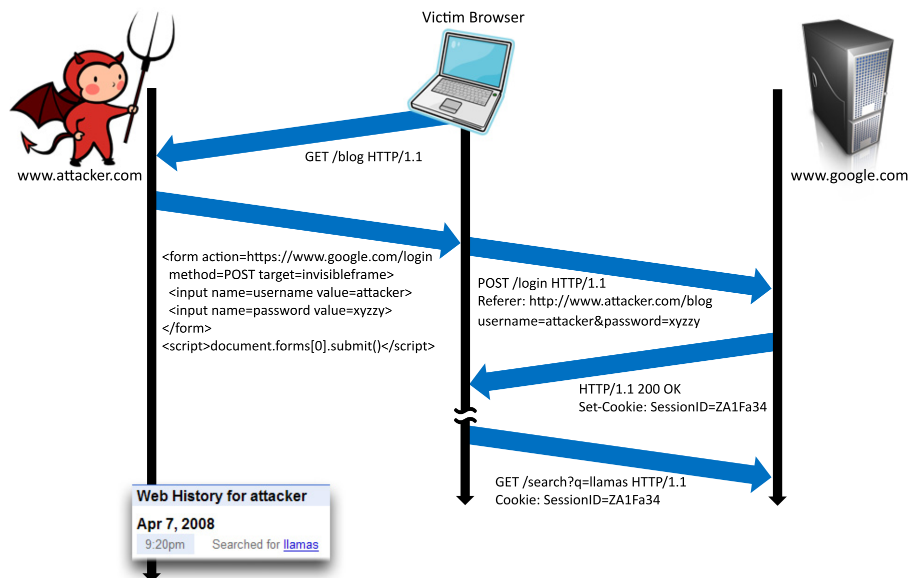

Figure 1: Event trace diagram for a login CSRF attack. The victim visits the attacker's site, and the attacker forges a cross-site request to Google's login form, causing the victim to be logged into Google as the attacker. Later, the victim makes a web search, which is logged in the attacker's search history.

3. The merchant silently logs the victim into his or her PayPal account.
4. To fund her purchase, the victim enrolls his or her credit card, but the credit card has actually been added to the merchant's PayPal account.

iGoogle. Using iGoogle, users can customize their Google homepage by including gadgets. For usability, some gadgets are "inline," meaning they run in the security context of iGoogle. Before adding such gadgets, users are asked to make a trust decision, but in a login CSRF attack, a web attacker makes the trust decision on behalf of the user:

1. Using his or her own browser, the attacker authors an inline iGoogle gadget (containing a malicious script) and adds it to his or her own personalized home page.
2. The attacker logs the victim into Google as the attacker and opens a frame to iGoogle.
3. Google believes the victim to be the attacker and serves the attacker's gadget to the victim, letting the attacker to run script in the `https://www.google.com` origin.
4. The attacker can now either (a) create a fake login page at the correct URL, (b) steal the user's autocompleted password, or (c) wait for the user to log in using another window and read `document.cookie`.

We disclosed this vulnerability to Google, and they have mitigated the vulnerability in two ways. First, they have deprecated the use of inline gadgets. Developers cannot create new inline gadgets, and only a few of the most popular inline gadgets are still allowed [22]. Second, they have deployed the secret token validation defense against login CSRF (discussed below), but the defense is deployed only in logging mode. We expect Google to begin denying login CSRF attempts once they have fully tested their defense.

4. EXISTING CSRF DEFENSES

There are three mechanisms a site can use to defend itself against cross-site request forgery attacks: validating a secret token, validating the HTTP `Referer` header, and including additional headers with XMLHttpRequest. All of these mechanisms are in use on the web today, but none of them are entirely satisfactory.

4.1 Secret Validation Token

One approach to defending against CSRF attacks is to send additional information in each HTTP request that can be used to determine whether the request came from an authorized source. This "validation token" should be hard to guess for attacker who does not already have access to the user's account. If a request is missing a validation token or the token does not match the expected value, the server should reject the request.

Secret validation tokens can defend against login CSRF, but developers often forget to implement the defense because, before login, there is no session to which to bind the CSRF token. To use secret validation tokens to protect against login CSRF, the site must first create a "pre-session," implement token-based CSRF protection, and then transition to a real session after successful authentication.

Token Designs. There are a number techniques for generating and validating tokens:

- **Session Identifier.** The browser's cookie store is designed to prevent unrelated domains from gaining access to each other's cookies. One common design is to use the user's session identifier as the secret validation token. On every request, the server validates that the token matches the user's session identifier. An attacker who can guess the validation token can already access the user's account. One disadvantage of this technique is that, occasionally, users reveal the contents of web

pages they view to third parties, for example via email or uploading the web page to a browser vendor's bug tracking database. If the page contains the user's session identifier in the form of a CSRF token, anyone who reads the contents of the page can impersonate the user to the web site until the session expires.

- **Session-Independent Nonce.** Instead of using the user's session identifier, the server can generate a random nonce and store it as a cookie when the user first visits the site. On every request, the server validates that the token matches the value stored in the cookie. For example, the widely used Trac issue tracking system [49] implements this technique. This approach fails to protect against active network attackers, even if the entire web application is hosted over HTTPS, because an active network attacker can overwrite the session-independent nonce (see Section 6.2) with his or her own CSRF token value and proceed to forge a cross-site request with a matching token.

- **Session-Dependent Nonce.** An refinement of the nonce technique is to store state on the server that binds the user's CSRF token value to the user's session identifier. On every request, the server validates that the supplied CSRF token is associated with the user's session identifier. This approach is used by CSRFx [19], CSRFGuard [48], and NoForge [30] but has the disadvantage that the site must maintain a large state table in order to validate the tokens.

- **HMAC of Session Identifier.** Instead of using server-side state to bind the CSRF token to the session identifier, the site can use cryptography to bind the two values. For example, the Ruby on Rails [46] web application framework implements this technique and uses the HMAC of the session identifier as a CSRF token. As long as all the site's servers share the HMAC key, each server can validate that the CSRF token is correctly bound to the session identifier. Properties of HMAC ensure that an attacker who learns a user's CSRF token cannot infer the user's session identifier.

Given sufficient engineering resources, a web site can use the HMAC technique to defend itself against CSRF attacks. However, many web sites and CSRF defense frameworks (such as NoForge [30], CSRFx [19] and CSRFGuard [48]), fail to implement the secret token defense correctly. One common mistake is to leak the CSRF token during cross-site requests. For example, if the honest site appends the CSRF token to hyperlinks another sites, that site gains the ability to forge cross-site requests against the honest site.

Case Study: NoForge. NoForge [30] implements CSRF defense using secret validation token bound to the session identifier using server-side state. Instead of modifying the web application to handle the CSRF token, NoForge parses the site's HTML as it is serialized onto the network and appends the CSRF token to all hyperlinks and form submissions. This technique is not robust for three reasons:

1. HTML dynamically created in the browser will not be re-written to include the CSRF token. Some sites create most of their HTML on the client. For example, Gmail, Flickr, and Digg all use JavaScript to create forms that require CSRF protection.
2. NoForge does not discriminate between hyperlinks back to the web application and hyperlinks to other web sites. If the web application links to another site, the remote site will receive a copy of the user's CSRF token. For example, if phpBB [44] adopted NoForge, forum posters would receive a copy of the user's CSRF token if the user clicked a link in their post. Even if NoForge discriminated between same-site and cross-site hyperlinks, the HTTP `Referer` header would leak the user's CSRF token.
3. NoForge does not defend against login CSRF because it only validates the CSRF token if the user already has a session identifier. Although this oversight is repairable, it demonstrates the complexity of implementing secret token validation correctly.

Although each is repairable, these vulnerabilities illustrate the complexity of implementing the secret validation technique correctly. CSRFx and CSRFGuard, as well as many web sites, contain similar issues.

4.2 The Referer Header

In many cases, when the browser issues an HTTP request, it includes a `Referer` header that indicates which URL initiated the request. The `Referer` header, if present, distinguishes a same-site request from a cross-site request because the header contains the URL of the site making the request. A site can defend itself against cross-site request forgery attacks by checking whether the request in question was issued by the site itself.

Unfortunately, the `Referer` contains sensitive information that impinges on the privacy of web users [18]. For example, the `Referer` header reveals the contents of the search query that lead the user to visit a particular site. Although this information is useful to web site owner, who can use the information to optimize their search engine rankings, this information disclosure leads some users to feel their privacy has been violated. Additionally, many organizations are concerned [28] that confidential information about their corporate intranets might leak to external web sites via the `Referer` header.

Bugs. Historically, browsers and have contained vulnerabilities that let malicious web sites spoof value of the `Referer` header, especially in conjunction with proxy servers. Discussions of `Referer` spoofing often cite [32] as evidence that browsers permit the `Referer` header to spoofed. Mozilla has patched the `Referer` spoofing vulnerabilities in Firefox 1.0.7 [15]. Internet Explorer currently contains known `Referer` spoofing vulnerabilities [47], but these vulnerabilities affect only XMLHttpRequest and can be used only to spoof `Referer`s directly back to the attacker's own site.

Strictness. If a site elects to use the `Referer` header to defend against CSRF attacks, the site's developers must decide whether implement lenient or strict `Referer` validation.

- In *lenient Referer validation*, the site blocks requests whose `Referer` header has an incorrect value. If a requests lacks the header, the site accepts the request. Although widely implemented, lenient `Referer` validation is easily circumvented because a web attacker can cause the browser to suppress the `Referer` header [27]. For example, requests issued from `ftp` and `data` URLs do not carry `Referer` headers.

- In *strict Referer validation*, the site also blocks requests that lack a `Referer` header. Blocking requests that lack a `Referer` header protects against malicious `Referer` suppression but incurs a compatibility penalty as some browsers and network configurations suppress the `Referer` header for legitimate requests. The magnitude of this compatibility penalty is an empirical question, which we investigate in Section 4.2.1.

Case Study: Facebook. Throughout the majority of its site, Facebook uses secret token validation to protect against CSRF. Facebook's login form, however, uses lenient `Referer` validation to defend against CSRF attacks. This approach to login CSRF protection is ineffective against web attackers. For example, a web attacker can redirect the user from `http://attacker.com/` to `ftp://attacker.com/index.html` and then issue a cross-site login request to Facebook. Because it originates from an `ftp` URL, none of the major browsers send a `Referer` header.

4.2.1 Experiment

To evaluate the compatibility of strict `Referer` validation, we conducted an experiment to measure how often, and under which circumstances, the `Referer` header is suppressed during legitimate requests.

Design. Advertisement networks provide a convenient platform for measuring browser and network characteristics [25]. To assess how often the `Referer` header is suppressed, we purchased 283,945 advertisement impressions from 163,767 unique IP addresses using two advertisement networks from 5 April 2008 to 8 April 2008. On Ad Network A, we purchased banner advertisements by bidding $0.50 per thousand impressions for the keywords "Firefox," "Game," "Internet Explorer," "Video," and "YouTube." On Ad Network B, we purchased interstitial advertisements by bidding $5 per thousand impressions for the keywords "Ballet," "Finance," "Flowers," "Food," and "Gardening." We spent $100 on each ad network, obtaining 241,483 impressions (146,310 unique IP addresses) on Ad Network A and 42,406 impressions (18,314 unique IP addresses) on Ad Network B.

The advertisement was served from two machines in our laboratory. The servers used two domain names purchased through separate registrars. When displayed, the advertisement generates a unique identifier that accompanies all subsequent requests generated by the impression and randomly chooses one of the two machines to be the primary server. The primary server sends the client HTML that issues a sequence of GET and POST requests to our servers, both over HTTP and HTTPS. The requests are generated programmatically by submitting forms, requesting images, and issuing XMLHttpRequests. The requests are generated in a random order and occur automatically without involving the user. When permitted by the browser security policy, the advertisement generates both same-domain requests to the primary server and cross-domain requests to the secondary server. Each server cost $400, each domain name cost $7, and each 90-day domain-validated HTTPS certificate was obtained for free from a legitimate certificate authority.

Upon receiving network requests, the servers logged a number of request parameters, including the `Referer` header, the `User-Agent` header, the date, the client's class C network, and the session identifier. Using JavaScript, the servers recorded the value of `document.referrer` DOM API as well. The servers did *not* log the client's IP address. To count unique IP addresses, the servers instead logged the HMAC of the client's IP address using a randomly generated key, which was discarded. None of the information recorded by the servers is sufficient to individually identify the viewer of the advertisement.

Ethics. The experimental design complied with the terms of service of both advertisement networks. The actions taken by the experiment are routine for web advertisements, which typically request additional resources from advertisers, including images, audio, and video. While the number of HTTP requests generated by our advertisement is likely greater than a typical advertisement, the bandwidth required to run our advertisement is significantly smaller than a typical video advertisement. The servers logged only information that is typically logged by advertisers when their advertisements are displayed. By not recording the client's IP address, our servers actually recorded significantly *less* information than is recorded by commercial advertisers.

Results. Our observations are summarized in Figure 2 and Figure 3. We observe the following results at the 95% confidence level:

- Over HTTP, the `Referer` header is suppressed more often for cross-domain requests than for same-domain requests, both for POST (chi-square = 2130, p-value < 0.001) and for GET (chi-square = 2175, p-value < 0.001) requests.

- The `Referer` header is suppressed more often for HTTP requests than HTTPS requests for cross-domain POST (chi-square = 6754, p-value < 0.001), for cross-domain GET (chi-square = 6940, p-value < 0.001), for same-domain POST (chi-square = 2286, p-value < 0.001), and for same-domain GET (chi-square = 2377, p-value < 0.001) requests.

- Over HTTP, the `Referer` header is suppressed more often than the `document.referrer` value for cross-domain POST (chi-square = 3096, p-value < 0.001), for cross-domain GET (chi-square = 3146, p-value < 0.001), for same-domain POST (chi-square = 786, p-value < 0.001), and for same-domain GET (chi-square = 754, p-value < 0.001) requests.

- The `Referer` header is suppressed more often on Ad Network B than on Ad Nework A for all types of request, including HTTP cross-domain POST (chi-square = 3060, p-value < 0.001), HTTP same-domain POST (chi-square = 6537, p-value < 0.001), HTTPS cross-domain POST (chi-square = 49.13, p-value < 0.001), and HTTPS same-domain POST (chi-square = 44.52, p-value < 0.001) requests.

- We also measured suppression of the custom headers `X-Requested-By` (see Section 4.3) and `Origin` (see Section 5). `X-Requested-By` was suppressed for 0.029–0.047% of HTTP POST requests, for 0.084–0.112% of HTTP GET requests, for 0.008–0.018% of HTTPS POST requests, and for 0.009–0.020% of HTTPS GET requests. `Origin` was suppressed for the same requests.

Discussion. There are two strong pieces of evidence that the `Referer` header is usually suppressed in the network and not in the browser.

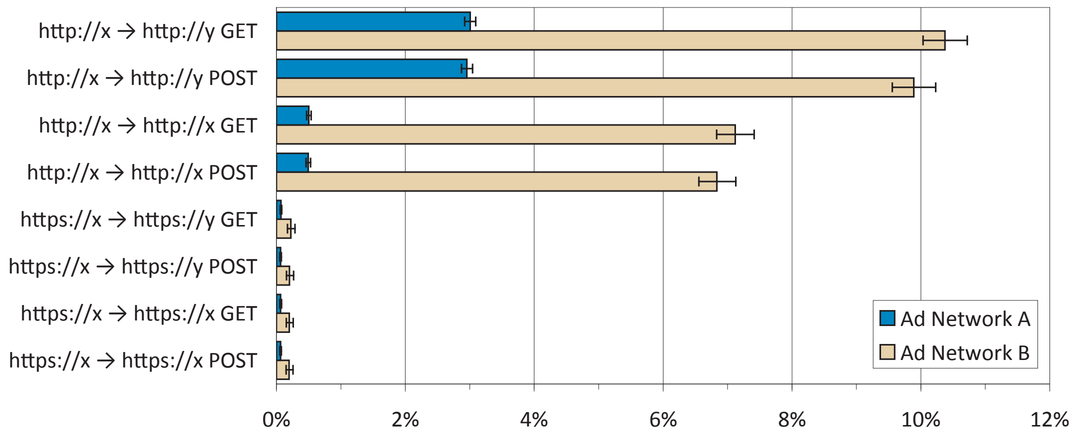

Figure 2: Requests with a Missing or Incorrect `Referer` Header (283,945 observations). The "x" and "y" represent the domain names of the primary and secondary web servers, respectively.

1. The `Referer` header is suppressed more often for HTTP requests than for HTTPS requests because network proxies are able to remove the header from HTTP traffic but are unable to tamper with HTTPS traffic. In some corporate networks, a network proxy serves as the HTTPS endpoint and can alter HTTPS requests, but this configuration is fairly rare.

2. Browsers that suppress the `Referer` header also suppress the `document.referrer` value, but when `Referer` is suppressed in the network, the `document.referrer` value is not suppressed. If the `Referer` header were suppressed in the browser, the browser would also suppress the value of `document.referrer`, but we observed that the `document.referrer` is suppressed less often than the `Referer` header.

In fact, most observations of the `document.referrer` value being suppressed are explainable by two facts about specific browsers: the PlayStation 3 browser does not support `document.referer` and Opera suppresses `document.referrer` (but not the `Referer` header) for cross-site HTTPS requests. The higher percentage of `Referer` suppression for XMLHttpRequest is due to a bug in Firefox 1.0 and 1.5. These observations indicates that extremely few browsers are configured to block referrers.

There is also evidence that the `Referer` header is suppressed due to privacy concerns. The user's privacy is degraded to a greater extent when the browser sends a `Referer` header from one site to another because the second site learns about the user's activities on the first site. By contrast, sending a `Referer` header back to the same site does not incur much privacy cost because the site can easily correlate multiple requests from the same user using cookies. We observed more `Referer` blocking for cross-site requests than for same-site requests, suggesting that the entity suppressing the header is cognizant of the differential privacy impact of these types of requests.

Conclusions. We draw two main conclusions:

1. **CSRF Defense over HTTPS.** The `Referer` header can be used as a CSRF defense for HTTPS requests. In order to use the `Referer` header as a CSRF defense, a site must reject requests that omit the header because an attacker can cause the browser to suppress the header. Over HTTP, sites cannot afford to block requests that lack a `Referer` header because they would cease to be compatible with the sizable percentage (roughly 3–11%) of users. Over HTTPS, however, strict `Referer` validation is feasible because only a tiny percentage (0.05–0.22%) of browsers suppress the header. In particular, strict `Referer` validation is well-suited for preventing login CSRF because login requests are typically issued over HTTPS.

2. **Privacy Matters.** Strict `Referer` validation is an appealing CSRF defense because the defense is simple to implement. Unfortunately, the poor privacy properties of the `Referer` header hamper attempts to use the header for security over HTTP. New browser security features, including new CSRF defense mechanisms, must address privacy concerns in order to be effective in large-scale deployments.

4.3 Custom HTTP Headers

Custom HTTP headers can be used to prevent CSRF because the browser prevents sites from sending custom HTTP headers to another site but allows sites to send custom HTTP headers to themselves using XMLHttpRequest. For example, the `prototype.js` JavaScript library [45] uses this approach and attaches the `X-Requested-By` header with the value `XMLHttpRequest`. Google Web Toolkit also recommends [16] that web developers defend against CSRF attacks by attaching a `X-XSRF-Cookie` header to XMLHttpRequets that contains a cookie value. The cookie value is not actually required to prevent CSRF attacks: the mere presence of the header is sufficient.

To use custom headers as a CSRF defense, a site must issue all state-modifying requests using XMLHttpRequest, attach the custom header (e.g., `X-Requested-By`), and reject all state-modifying requests that are not accompanied by the header. For example, to defend against login CSRF, the site must send the user's authentication credentials to

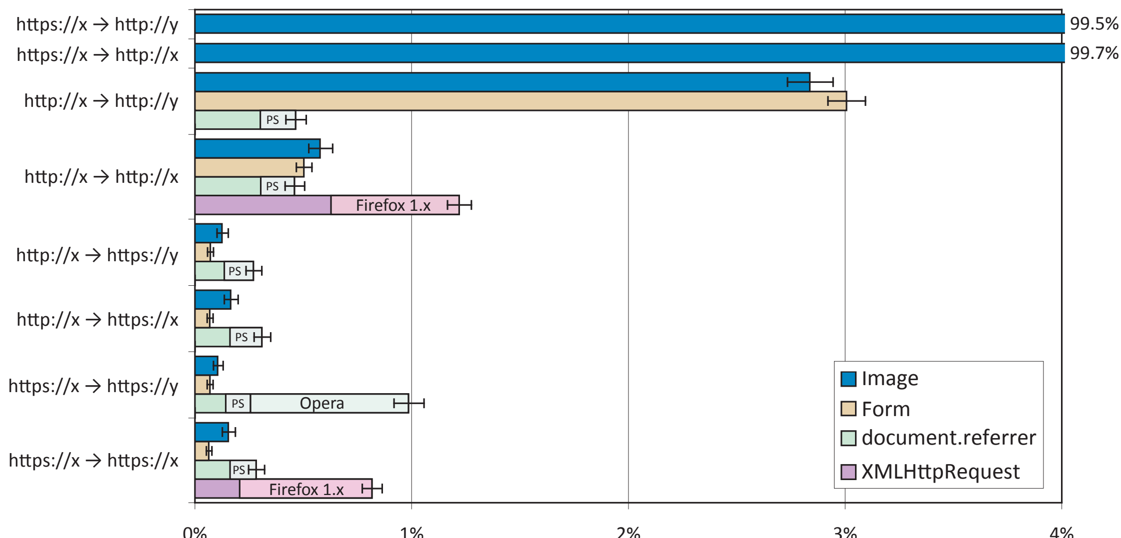

Figure 3: Requests with a Missing or Incorrect `Referer` Header on Ad Network A (241,483 observations). Opera blocks cross-site `document.referrer` for HTTPS. Firefox 1.0 and 1.5 do not send `Referer` for XMLHttpRequest due to a bug. The PlayStation 3 (denoted PS) does not support `document.referrer`.

the server via XMLHttpRequest. In our experiment, the `X-Requested-By` header is correctly delivered to servers approximately 99.90–99.99% of the time, suggesting that this technique works for a large percentage of users.

5. PROPOSAL: ORIGIN HEADER

To prevent CSRF attacks, we propose modifying browsers to send a `Origin` header with POST requests that identifies the origin that initiated the request. If the browser cannot determine the origin, the browser sends the value `null`.

Privacy. The `Origin` header improves on the `Referer` header by respecting the user's privacy:

1. The `Origin` header includes only the information required to identify the principal that initiated the request (typically the scheme, host, and port of the active document's URL). In particular, the `Origin` header does not contain the path or query portions of the URL included in the `Referer` header that invade privacy without providing additional security.

2. The `Origin` header is sent only for POST requests, whereas the `Referer` header is sent for all requests. Simply following a hyperlink (e.g., from a list of search results or from a corporate intranet) does not send the `Origin` header, preventing the majority of accidental leakage of sensitive information.

By responding to privacy concerns, the `Origin` header will likely not be widely suppressed.

Server Behavior. To use the `Origin` header as a CSRF defense, sites should behave as follows:

1. All state-modifying requests, including login requests, must be sent using the POST method [6]. In particular, state-modifying GET requests must be blocked in order to address the forum poster threat model.

2. If the `Origin` header is present, the server must reject any requests whose `Origin` header contains an undesired value (including `null`). For example, a site could reject all requests whose `Origin` indicated the request was initiated from another site.

Security Analysis. Although the `Origin` header has a simple design, the use of the header as a CSRF defense has a number of subtleties.

- **Rollback and Suppression.** Because a supporting browser will always include the `Origin` header when making POST requests, sites can detect that a request was initiated by a supporting browser by observing the presence of the header. This design prevents an attacker from making a supporting browser appear to be a non-supporting browser. Unlike the `Referer` header, which is absent when suppressed by the browser, the `Origin` header takes on the value `null` when suppressed by the browser.

- **DNS Rebinding.** In existing browsers, The `Origin` header can be spoofed for same-site XMLHttpRequests. Sites that rely only on network connectivity for authentication should use one of the DNS rebinding defenses in Section 2, such as validating the `Host` header. This requirement is complementary to CSRF protection and also applies to all the other existing CSRF defenses described in Section 4.

- **Plug-ins.** If a site opts into cross-site HTTP requests via `crossdomain.xml`, an attacker can use Flash Player to set the `Origin` header in cross-site requests. Opting into cross-site HTTP requests also defeats secret token validation CSRF defenses because the tokens leak during cross-site HTTP requests. To prevent these (and other) attacks, sites should not opt into cross-site HTTP requests from untrusted origins.

Adoption. The Origin header is similar to four other proposals that identify the initiator of a request. The Origin header improves and unifies these proposals and has been adopted by several working groups.

- **Cross-Site XMLHttpRequest.** The proposed standard for cross-site XMLHttpRequest [50] included a Access-Control-Origin header to identify the origin issuing the request. This header is sent for all HTTP methods, but it is sent only for XMLHttpRequests. Our specification for the Origin header is modeled off this header. The working group accepted our proposal to rename the header to Origin.

- **XDomainRequest.** The XDomainRequest API [39] in Internet Explorer 8 Beta 1 sends cross-site HTTP requests that omit the path and query from the Referer header. This truncated Referer header identifies the origin of the request. Our experimental results suggest that the Referer header is frequently blocked by the network, whereas the Origin header is rarely blocked. Microsoft has announced that it will adopt our suggestion and rename XDomainRequest's truncated Referer header to Origin.

- **JSONRequest.** The JSONRequest API for cross-site HTTP requests [7] included a Domain header that identifies the host name of the requester. The Origin improves on the Domain header by including the requester's scheme and port. The JSONRequest specification editor accepted our proposal to replace the Domain header with the Origin header in order to defend against active network attackers.

- **Cross-Document Messaging.** The HTML 5 specification proposes a new browser API for authenticated client-side communication between HTML documents [20]. Each message is accompanied by an origin property that cannot be overwritten. The process for validating this property is the same as the process for validating the Origin header, except that the validation occurs on the client rather than on the server.

Implementation. We implemented both the browser and server components of the Origin header CSRF defense. On the browser side, we implemented the Origin header in a eight-line patch to WebKit, the open source component of Safari, and in a 466 line extension to Firefox. On the server side, we used the Origin header to implement a web application firewall for CSRF in three lines of ModSecurity, a web application firewall language for Apache; see Figure 4. These rules validate that, for POST requests, the Host header and the Origin header contain an acceptable values. These rules implement CSRF protection without modification to the site itself, provided GET requests are free of side effects (and that browsers implement the Origin header).

6. SESSION INITIALIZATION

Login CSRF is one example of a more general class of vulnerabilities in session initialization. After initializing a session, the web server typically associates a user identity with some form a session identifier. There are two types of session initialization vulnerabilities, one in which the server associates the honest user's identity with the newly initialized session and another in which the server associates the attacker's identity with the session.

- **Authenticated as User.** In some cases, the attacker can force the site to use a predictable session identifier for a new session. These vulnerabilities are often referred to as *session fixation* vulnerabilities (see, for example, [52]). After the user supplies their authentication credentials to the honest site, the site associates the user's authorization with the predictable session identifier. The attacker can then access the honest site direct using the session identifier and can act as the user.

- **Authenticated as Attacker.** Alternately, the attacker cause the honest site to begin a new session with the user's browser but force the session to be associated with the attacker's authorization. (Section 3 contains examples of how this vulnerability can be exploited.) The simplest form of this type of session initialization vulnerability is login CSRF, but there are other ways to force the user's browser to participate in a session associated with the attacker.

There are two common approaches to mounting an attack on session initialization: HTTP requests and cookie overwriting. In the HTTP requests approach, a web attacker causes the user's browser to issue HTTP requests to the honest site and confuse the site into incorrectly initializing a session. In the cookie overwriting approach, a network attacker uses a design flaw in Secure cookies to overwrite HTTPS cookies from an unauthenticated HTTP connection.

6.1 HTTP Requests

OpenID. The OpenID protocol [13], used by many web sites including LiveJournal, Movable Type, and Wordpress, recommends that sites include a self-signed nonce to protect against reply attacks, but does not suggest (nor do sites implement) a mechanism to bind the OpenID session to the user's browser, letting a web attacker force the user's browser to initialize a session authenticated as the attacker:

1. Using his or her own machine, the web attacker visits the Relying Party (such as Blogger) and begins the authentication process with the Identity Provider (such as Yahoo!).

2. In the final step of the OpenID protocol, the Identity Provider redirects the attacker's browser to the "return_to" URL of the Relying Party.

3. Instead of following the redirect, the attacker directs the user's browser to the return_to URL.

4. The Relying Party completes the OpenID protocol and stores a session cookie in the user's browser.

5. The user is now logged in as the attacker.

The specification states "the return_to URL MAY be used as a mechanism for the Relying Party to attach context about the authentication request to the authentication response," but this behavior is neither required nor implemented by LiveJournal, Movable Type, or Wordpress.

```
SecRule REQUEST_HEADERS:Host !^www\.example\.com(:\d+)?$ deny,status:403
SecRule REQUEST_METHOD ^POST$ chain,deny,status:403
SecRule REQUEST_HEADERS:Origin !^(https?://www\.example\.com(:\d+)?)?$
```

Figure 4: ModSecurity rules needed to implement CSRF protection using the `Origin` header.

To defend against these attacks, the Relying Party should generate a fresh nonce at the start of the protocol, store the nonce in the browser's cookie store and include the nonce in the `return_to` parameter of the OpenID protocol. Upon receiving a positive identity assertion from the user's Identity Provider, the Replying Party should validate that the nonce included in the `return_to` URL matches the nonce stored the cookie store. This defense is similar to the secret token validation technique and ensures that the OpenID protocol session completes on the same browser as it began.

PHP Cookieless Authentication. PHP cookieless authentication [37] is used by sites like Hushmail to avoid leaving cookies on the user's machine. Cookieless authentication stores the user's session identifier in a query parameter instead. This technique fails to bind the session to the user's browser, letting a web attacker force the user's browser to initialize a session authenticated as the attacker:

1. Using his or her own machine, the web attacker logs into the honest web site.
2. The web attacker redirects the user's browser to the URL currently displayed in the attacker's location bar. (Recall that the web attacker can navigate any top-level frame in the user's browser [5].)
3. Because this URL contains the attacker's session identifier, the user is now logged in as the attacker.

To prevent this session initialization attack without cookies, a site must use some other mechanism to bind to the session identifier to the user's browsers. For example, the site could maintain a long-lived frame that contains the session identifier token. This frame binds the session to the user's browser by storing the session identifier in memory.

Sites that use PHP cookieless authentication often contain a session initialization vulnerability that lets a web attacker impersonate an honest user:

1. Using his or her own machine, the web attacker visits the honest web site's login page.
2. The web attacker redirects the user's browser to the URL currently displayed in the attacker's location bar. (Recall that the web attacker can navigate any top-level frame in the user's browser [5].)
3. The user read the location bar, accurately determines that displayed URL corresponds to the honest site, and logs into the site.
4. Because the URL supplied by the attacker contains the attacker's session identifier, the attacker's session is now authenticated as the user.

This session fixation vulnerability has a number of standard defenses [9]. For example, the site can regenerate the session identifier after the user logs in.

6.2 Cookie Overwriting

Vulnerability. A server can include the `Secure` flag in the `Set-Cookie` header to instruct the browser that the cookie should be sent only over HTTPS connections. All modern browsers respect this attribute, and it is commonly used to protect sessions at high-security sites. However, the `Secure` flag does not offer any *integrity* protection [40, 35, 34] in the cross-scheme threat model. An active network attacker can supply a `Set-Cookie` header over a HTTP connection to the same host name as the site and install either a `Secure` or a non-`Secure` cookie of the same name. When the browser sends the cookie back to the site over HTTPS, the site has no mechanism for determining whether the cookie has been overwritten by the attacker. If the `Secure` cookie contains the user's session identifier, the attacker can mount an attack on session initialization simply by overwriting the user's session identifier with his or her own session identifier.

Most often, this attack can be used to force the user's browser to initialize a session authenticated as the attacker. There is little sites can do to protect themselves from this attack because they require the browser to provide client-side storage with integrity against network attackers. However, some proposed browser features, such as `localStorage` [21], provide the needed integrity to work around the deficiencies of the `Cookie` header. Alternately, if a site maintains its application-layer authentication session independently of its cookie-based HTTP-layer session, a network attacker can overwrite the user's session cookie prior to authentication and act as the user after the use authenticates to the site.

Security professionals have known for a number of years that an active network attacker can overwrite `Secure` cookies [29], but the browser vendors have been unable to find a deployable defense. The vendors have considered preventing HTTP requests from overwriting `Secure` cookies, but this defense cannot be deployed "without breaking standards and existing web apps" [29]. Worse, this defense does not actually provide cookie integrity because the `Cookie` header provides no way to distinguish a `Secure` cookie from a non-`Secure` cookie under either the *de facto* or the proposed cookie standards [40, 35, 34].

Defense. To provide integrity without modifying the `Cookie` header (and thereby maintain backwards compatibility), we propose browsers report the integrity state of cookies using a `Cookie-Integrity` header in HTTPS requests:

```
Cookie: SID=DQAAAHQA...; pref=ac81a9...; TM=1203...
Cookie-Integrity: 0, 2
```

The header identifies the index of the cookies in the request's `Cookie` header that were set using HTTPS. If none of the cookies in the request were set over HTTPS, the `Cookie-Integrity` contains the value `none`. This header's integrity protection is complementary to the confidentiality provided by `Set-Cookie`'s `Secure` flag and is backwards-compatible because servers ignore unrecognized headers. Below are several design decisions:

- **Bandwidth.** Adding bytes to every HTTP request increases the latency of all web traffic. To save bandwidth, we include only the index of the cookies as they appear in the `Cookie` header. Another proposal for changing the behavior of cookies [43] includes a redundant copy of the `Cookie` header named `Cookie2`.

- **Multiplicity.** If the current host sets a cookie with the same name as a domain cookie, the `Cookie` header can contain two cookies with the same name. Were the `Cookie` integrity header to designate cookies by name, this case could cause confusion. Designating cookies by index avoids this difficulty.

- **Rollback.** Always including the `Cookie-Integrity` header for HTTPS requests prevents a rollback attack. If the header were absent when none of the cookies had integrity, the server would be unable to distinguish between a request in which none of the cookies had integrity from a request issued by a down-level client that did not support the `Cookie-Integrity` header.

- **Sibling Domains.** Consider a deployment in which a registry-controlled domain, such as `example.com`, contains a trusted and untrusted subdomain, `www.example.com` and `users.example.com`, respectively. By setting a domain cookie for `.example.com`, the untrusted domain can inject cookies into the trusted domain's `Cookie` header. The `Cookie-Integrity` header does not defend against this attack, but an extension of the header could by including the origin of each cookie (at the cost of bandwidth and complexity).

We implemented the `Cookie-Integrity` header as a Firefox extension with 202 lines of JavaScript. The extension augments the cookie store to include an `Integrity` flag that records which cookies were set using HTTPS.

7. RELATED WORK

Our analysis of the main existing CSRF defenses is provided in Section 4. In this section, we describe a few other CSRF mitigations.

RequestRodeo. RequestRodeo [27] is a client-side CSRF mitigation that strips implicit authorization information, such as the `Cookie` header, from outgoing cross-site HTTP requests. It aims to prevent CSRF by preventing the site from associating cross-site requests with existing user sessions. RequestRodeo is unable to prevent login cross-site request forgery because the forged login request does not require implicit authorization information to be used in an attack. The authors of RequestRodeo conceptualize CSRF as "Session Riding" and missed login CSRF because there is no "session to ride" when forging a login request across sites. Another limitation of RequestRodeo is that it breaks existing web site functionality because it cannot automatically distinguish legitimate cross-site requests from attacks.

CAPTCHAs. Another proposal [3] for mitigating CSRF is to require the user to solve a CAPTCHA [51] before allowing an important request to proceed. Although CAPTCHAs have many other applications, they offer few advantages over secret validation tokens as a CSRF defense. If it is known to the attacker which CAPTCHA is displayed, then the attacker can manually solve CAPTCHAs and attack one user per CAPTCHA solved, which is expensive but probably still cost-effective. If the decision of which CAPTCHA to display is a session-dependent secret, then this information could be used as a session-dependent secret validation token without burdening the user with the task of solving a CAPTCHA.

8. CONCLUSIONS AND ADVICE

Cross-site request forgery is a widely exploited vulnerability in web sites. Many web sites that have repaired their CSRF vulnerabilities contain login CSRF vulnerabilities that let an attacker force a user to authenticate as the attacker. Based on our analysis and experiments, we recommend different CSRF defenses for different use cases.

- **Login CSRF.** We recommend strict `Referer` validation to protect against login CSRF because login forms typically submit over HTTPS, where the `Referer` header is reliably present for legitimate requests. If a login request lacks a `Referer` header, the site should reject the request to defend against malicious suppression.

- **HTTPS.** For sites exclusively served over HTTPS, such as banking sites, we recommend strict `Referer` validation to protect against CSRF. Sites should whitelist specific "landing" pages, such as the home page, that accept cross-site requests.

- **Third-party Content.** Sites that incorporate third-party content, such as images and hyperlinks, should use a framework, such as Ruby-on-Rails, that implements secret token validation correctly. If such a framework is unavailable, sites should spend the engineering effort to implement secret token validation and use HMAC to bind the token to the user's session.

In the long term, our proposed `Origin` header improves on `Referer` header by eliminating the privacy concerns that lead to `Referer` blocking, and eliminates the need for secret token defenses, allowing sites to protect both HTTPS and non-HTTPS requests without having to worry about keeping secret tokens from leaking.

Future Work. To use the `Origin` header as a CSRF defense, sites must take care not to perform side-effecting operations in response to GET requests. Although required by the HTTP specification, many sites do not adhere to this discipline. Techniques for enforcing this discipline are an important area of future work.

A variant on CSRF involves a web attacker embedding a frame to an honest site and tricking the user into clicking a button inside the frame [17]. Although this attack is not technically a CSRF attack by our definition, the attack is similar to CSRF in that an attacker causes the user's browser to issue a network request to an honest web site. The traditional defense for this attack is frame busting [33], but this defense is problematic because it relies on JavaScript, which might be disabled by the user or suppressed by the attacker [23]. Another approach to preventing this attack is to extend the `Origin` header to report the active frame's ancestors in the frame hierarchy, allowing the honest site to reject requests that originate from within frames controlled by the attacker.

9. REFERENCES

[1] David Airey. Google's Gmail security failure leaves my business sabotaged, December 2007. http://www.davidairey.co.uk/google-gmail-security-hijack/.

[2] David Airey. An informal chat with Google, March 2008. http://www.davidairey.com/google-site-links-gmail-hack-search-penalty/.

[3] Robert Auger. The cross-site request forgery (CSRF/XSRF) FAQ, 2007. http://www.cgisecurity.com/articles/csrf-faq.shtml.

[4] Michael Barbaro and Tom Zeller Jr. A face is exposed for AOL searcher no. 4417749. *The New York Times*, August 2006. http://www.nytimes.com/2006/08/09/technology/09aol.htm.

[5] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *In Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*, July 2008.

[6] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext Transfer Protocol—HTTP/1.0. RFC 1945, May 1996.

[7] Douglas Crockford. JSONRequest, 2006. http://json.org/JSONRequest.html.

[8] Neil Daswani, Christoph Kern, and Anita Kesavan. *Foundations of Security: What Every Programmer Needs to Know*. Apress, 2007.

[9] Rogan Dawes. Session Fixation, 2008. http://www.owasp.org/index.php/Session_Fixation_Protection.

[10] Rohit Dhamankar et al. Sans top-20 security risks, 2007. http://www.sans.org/top20/2007/.

[11] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, 2006.

[12] E. W. Felten, D. Balfanz, D. Dean, and D. S. Wallach. Web Spoofing: An Internet Con Game. In *20th National Information Systems Security Conference*, October 1997.

[13] Brad Fitzpatrick, David Recordon, Dick Hardt, Johnny Bufu, Josh Hoyt, et al. OpenID authentication 2.0, December 2007. http://openid.net/specs/openid-authentication-2_0.html.

[14] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2007.

[15] Mozilla Foundation. Security advisory 2005-58, September 2005. http://www.mozilla.org/security/announce/2005/mfsa2005-58.html.

[16] Google. Security for GWT Applications. http://groups.google.com/group/Google-Web-Toolkit/web/security-for-gwt-applications.

[17] Robert Hansen and Tom Stracener. Xploiting Google gadgets: Gmalware and beyond, August 2008. Black Hat briefing.

[18] Elliotte Rusty Harold. Privacy tip #3: Block Referer headers in Firefox, October 2006. http://cafe.elharo.com/privacy/privacy-tip-3-block-referer-headers-in-firefox/.

[19] Mario Heiderich. CSRFx, 2007. http://php-ids.org/category/csrfx/.

[20] Ian Hickson et al. Cross-document messaging. http://www.w3.org/html/wg/html5/#crossDocumentMessages.

[21] Ian Hickson et al. HTML 5 Working Draft. http://www.whatwg.org/specs/web-apps/current-work/.

[22] Dan Holevoet. Changes to inline gadgets, August 2008. http://igoogledeveloper.blogspot.com/2008/08/changes-to-inlined-gadgets.html.

[23] Collin Jackson. Defeating frame busting techniques, 2005. http://crypto.stanford.edu/framebust/.

[24] Collin Jackson and Adam Barth. ForceHTTPS: Protecting high-security web sites from network attacks. In *Proceedings of the 17th International World Wide Web Conference (WWW)*, April 2008.

[25] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from DNS rebinding attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, November 2007.

[26] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th International World Wide Web Conference (WWW)*, May 2006.

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

[28] Aaron Johnson. The Referer header, intranets and privacy, February 2007. http://cephas.net/blog/2007/02/06/the-referer-header-intranets-and-privacy/.

[29] Paul Johnston and Richard Moore. Multiple browser cookie injection vulnerabilities, September 2004. http://www.westpoint.ltd.uk/advisories/wp-04-0001.txt.

[30] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2006.

[31] Chris Karlof, Umesh Shankar, J. D. Tygar, and David Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, November 2007.

[32] Amit Klein. Exploiting the XMLHttpRequest object in IE—Referrer spoofing and a lot more. . . , September 2005. http://www.cgisecurity.com/lib/XmlHTTPRequest.shtml.

[33] Peter-Paul Koch. Frame busting. http://www.quirksmode.org/js/framebust.html.

[34] David Kristol and Lou Montulli. HTTP State Management Mechanism. RFC 2965, October 2000.

[35] David Kristol and Lou Montulli. HTTP State Management Mechanism. RFC 2109, February 1997.

[36] V. T. Lam, Spiros Antonatos, P. Akritidis, and Kostas G. Anagnostakis. Puppetnets: Misusing web browsers as a distributed attack infrastructure. In *Proceedings of the 13th ACM Conference on Computer*

and Communication Security (CCS), October 2006.

[37] PHP Manual. Session handling functions. http://www.phpbuilder.com/manual/en/ref.session.php.

[38] Chris Masone, Kwang-Hyun Baek, and Sean Smith. WSKE: Web server key enabled cookies. In *Proceedings of Usable Security 2007 (USEC '07)*.

[39] Microsoft. XDomainRequest object. http://msdn2.microsoft.com/en-us/library/cc288060(VS.85).aspx.

[40] Netscape. Persistent client state: HTTP cookies. http://wp.netscape.com/newsref/std/cookie_spec.html.

[41] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *InfoScale '06: Proceedings of the 1st International Conference on Scalable Information Systems*, 2006.

[42] Petko D. Petkov. Google Gmail e-mail hijack technique, September 2007. http://www.gnucitizen.org/blog/google-gmail-e-mail-hijack-technique/.

[43] Yngve Pettersen. HTTP state management mechanism v2. IETF Internet Draft, February 2008. http://www.ietf.org/internet-drafts/draft-pettersen-cookie-v2-02.txt.

[44] phpBB. http://phpbb.com/.

[45] Prototype JavaScript framework. http://www.prototypejs.org/.

[46] Ruby on rails. http://www.rubyonrails.org/.

[47] Secunia. Microsoft Internet Explorer "XMLHTTP" HTTP request injection, September 2005. http://secunia.com/advisories/16942/.

[48] Eric Sheridan. OWASP CSRFGuard Project, 2008. http://www.owasp.org/index.php/CSRF_Guard.

[49] Trac. http://trac.edgewall.org/.

[50] Anne van Kesteren et al. Access control for cross-site requests. http://www.w3.org/TR/access-control/.

[51] Luis von Ahn, Nick Hopper Manuel Blum, and John Langford. CAPTCHA: Using hard AI problems for security. In *Eurocrypt 2003*.

[52] Weilin Zhong. Session Fixation, 2008. http://www.owasp.org/index.php/Session_Fixation.