Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection

Abstract. A cross site request forgery (CSRF) attack occurs when a user's web browser is instructed by a malicious webpage to send a request to a vulnerable web site, resulting in the vulnerable web site performing actions not intended by the user. CSRF vulnerabilities are very common, and consequences of such attacks are most serious with financial websites. We recognize that CSRF attacks are an example of the confused deputy problem, in which the browser is viewed by websites as the deputy of the user, but may be tricked into sending requests that violate the user's intention. We propose Browser-Enforced Authenticity Protection (BEAP), a browser-based mechanism to defend against CSRF attacks. BEAP infers whether a request reflects the user's intention and whether an authentication token is sensitive, and stripes sensitive authentication tokens from any request that may not reflect the user's intention. The inference is based on the information about the request (e.g., how the request is triggered and crafted) and heuristics derived from analyzing real-world web applications. We have implemented BEAP as a Firefox browser extension, and show that BEAP can effectively defend against the CSRF attacks and does not break the existing web applications.

Key words: Cross-Site Request Forgery, Web Security, Browser Security

1 Introduction

Cross-site request forgery, also known as one-click attack or session riding and abbreviated as CSRF or XSRF, is an attack against web applications [24, 17, 18]. In a CSRF attack, a malicious web page instructs a victim user's browser to send a request to a target website. If the victim user is currently logged into the target website, the browser will append authentication tokens such as cookies to the request, authenticating the malicious request as if it is issued by the user. Consequences of CSRF attacks are most serious with financial websites, as an attacker can use CSRF attacks to perform financial transactions with the victim user's account, such as sending a check to the attacker, purchasing a stock, purchasing products and shipping to the attacker.

A CSRF attack does not exploit any browser vulnerability. As long as a user is logged into the vulnerable web site, simply browsing a malicious web page can lead to unintended operations performed on the vulnerable web site. Launching such CSRF attacks is possible in practice because many users browse multiple sites in parallel, and users often do not explicitly log out when they finish using a web site. A CSRF attack can also be carried out without a user visiting a malicious webpage. In a recent CSRF attack against residential ADSL

routers in Mexico, an e-mail with a malicious IMG tag was sent to victims. By viewing the email message, the user initiated an HTTP request, which sent a router command to change the DNS entry of a leading Mexican bank, making any subsequent access by a user to the bank go through the attacker's server [2].

CSRF appeared in the Open Web Application Security Project (OWASP) top 10 web application threats in 2007 (ranked at 5) [15]. Several CSRF vulnerabilities against real-world web applications have been discovered [23, 19, 20]. In 2007, a serious CSRF vulnerability in Gmail was reported [21]. It allowed a malicious website to surreptitiously add a filter to a victim user's Gmail account that forwards emails to a third party address. CSRF vulnerabilities are very common. The potential damage of CSRF attacks, however, has not been fully realized yet. We quote the following from an online article [8],

> Security researchers say it's only a matter of time before someone awakens the "sleeping giant" and does some major damage with it – like wiping out a user's bank account or booking a flight on behalf of a user without his knowledge.
> "There are simply too many [CSRF-vulnerable Websites] to count," says rsnake, founder of ha.ckers.org. "The sites that are more likely to be attacked are community websites or sites that have high dollar value accounts associated with them – banks, bill pay services, etc."

Several defense mechanisms have been proposed and used for CSRF attacks. However, they suffer from various limitations (see Section 2.3).

In this paper, we study browser-based defense against CSRF attacks, which is orthogonal to server-side defenses. The websites should follow the best practice to defend against the CSRF attacks before browser-side defenses are universally adopted. One crucial advantage of a browser-based solution compared with a server-side solution is that a user who started using the protected browser will immediately have all his web browsing protected, even when visiting websites that have CSRF vulnerabilities. Furthermore, because the number of major browsers is small, deploying protection at the browser end can be achieved more easily, compared with deploying server-side defenses at all websites.

We recognize that CSRF attacks are an example of the confused deputy problem. The current web design assumes that the browser is the deputy of the user and that any HTTP request sent by the browser reflects the user's intention. This assumption is not true as many HTTP requests are under the control of the web pages and do not necessarily reflect the user's intention. This becomes a security concern for HTTP requests that have sensitive consequences (such as financial consequences).

Our solution to this problem is to enhance web browsers with a mechanism ensuring that *all sensitive requests sent by the browser should reflect the user's intention*. We achieve that by inferring whether an HTTP request reflects the intention of the user and whether an authentication token is sensitive, and striping all sensitive authentication tokens from the HTTP requests that may not reflect the user's intention. We call it *Browser-Enforced Authenticity Protection*.

We have implemented a prototype of BEAP as a Firefox browser extension. The implementation consists of about 800 lines of Javascript. An extension without modifying the browser core enables easy initial deployment. The full benefit of BEAP will be achieved if it is implemented in major web browsers. We use theoretical analysis and experiments to show that BEAP can effectively defend against the CSRF attacks and it does not break the existing web applications.

In Section 2, we describe the background, the related work, and the CSRF vulnerabilities we found in real-world web applications. We describe our proposal and the prototype implementation in Section 3. In Section 4, we analyze the effectiveness and compatibility of our proposal. Finally, we conclude in Section 5.

2 Understanding CSRF Attacks and Existing Defenses

CSRF attacks exploit existing authenticated sessions. Two common approaches for maintaining authenticated web sessions are cookies and HTTP authentication credentials, which we call *authentication tokens*.

Cookies [13] are pieces of text data sent by the web server to the browser. The browser stores the cookies locally and sends them along with every further request to the original web site who sets them. After a web site has authenticated a user, for example, by validating the user name and password entered by the user, the web site can send back a cookie containing a "session ID" that uniquely identifies the session, which is referred to as *authentication cookie*. If the web server relies only on cookies for user authentication, every request that has a valid authentication cookie is interpreted as an intended request issued by the authenticated user who owns the session. When sending a cookie to a browser, the website can specify an optional attribute expires among other three attributes. The expires field takes the value of a date that indicates how long the cookie is valid. After the date passes, the browser deletes the cookie. If the expires field is omitted, then the cookie is called a *session cookie* and should be deleted when user closes the web browser. Cookies with an expires field are called *persistent cookies*. Most financial websites and sensitive services specify the authentication cookie as a session cookie, because the session cookies are removed when the browser is closed and won't be abused by others who may share the same computer and browser.

HTTP authentication [4], an authentication mechanism defined in the HTTP protocol [6], is widely used within Intranet environments. In the mechanism, when accessing a webpage that requires authentication, the browser will popup a dialog asking for the username and password. After entering the information, the credential is encoded and sent to the web server via the Authorization request header. The browser remembers the credential until the browser is closed. When later the user visiting the webpages in the same authentication realm, the browser automatically includes the credential in the request via the Authorization header.

CSRF attacks use HTTP requests that have lasting observable effects at the web site. Two request methods are used in real-world HTTP requests: GET and POST. According to the HTTP/1.1 RFC document [6], the GET method, which is known as a "safe" method, is used to retrieve objects. The GET requests

should not have any lasting observable effect (e.g., modification of a database). The operations that have lasting observable effects should be requested using the method POST. The POST requests have a request body and are typically used to submit forms. However, there exist web applications that do not follow the standard and use GET for requests that have lasting side effects.

Visiting web pages in one site may result in HTTP requests to another site; these are called cross-site requests. More precisely, in a cross-site request, the link of the request is provided by a website that is different from the destination website of the request. Cross-site requests are common. For example, a webpage may include images, scripts, style files and sub-frames from a third-party website. When the user clicks a hyper-link or a button contained in a webpage, the linked URL may be addressing a third-party website.

2.1 The CSRF Attack

The general class of cross site request forgery (CSRF) attacks was first introduced in a posting to the BugTraq mailing list [24], and has been discussed by web application developers [17, 18]. CSFR attacks use cross-site requests for malicious purposes. For example, suppose that the online banking application of `bank.com` provides a "pay bills" service using an HTML form. The user asks the bank to send a check to a payee by completing the form and clicking the "Sumbit" button. Upon the user clicking the button, a POST request is sent to the server, together with the authentication cookie. When the web server receives this HTTP request, it processes the request and sends a check to the payee identified in the request.

A CSRF attack works as follows.While accessing the bank account, the user simultaneously browses some other web sites. One of these sites, evil.org, contains a hidden form and a piece of JavaScript. As soon as the user visits the web page, the browser silently submit the hidden form to `bank.com`. The format and content of the request is exactly the same as the request triggered by the user clicking the submit button in the "pay bill" form provided by the bank. On sending the request, the user's browser automatically attaches the authentication cookies to the request. Since the session is still active in the server, the request will be processed by the server as issued by user. As illustrated in this example, POST requests can be forged by a hidden form. If the bank uses GET request for the pay bill service, the request can be easily forged by using various HTML elements, such as ⟨img⟩, ⟨script⟩, ⟨iframe⟩, ⟨a⟩ (hyper-link) and so on.

We note that as long as a user is logged in to a vulnerable web site, a single mouse click or just browsing a page under the attacker's control can easily lead to unintended operations performed on the vulnerable web site.

CSRF vs. XSS. CSRF vulnerabilities should not be confused with XSS vulnerabilities. In XSS exploits, an attacker injects malicious scripts into an HTML document hosted by the victim web site, typically through submitting text embedded with code which is to be displayed on the page, such as a blog post. Most XSS attacks are due to vulnerabilities in web applications which fail in sanitizing untrustworthy inputs which might in turn be displayed to users. CSRF attacks do not rely on the execution and injection of malicious JavaScript code. CSRF

vulnerabilities are due to the use of cookies or HTTP authentication as the authentication mechanism. A web site that does not have XSS vulnerabilities may contain CSRF vulnerabilities.

2.2 Real-world CSRF vulnerabilities

In order to understand how commonly the CSRF vulnerability exists in the real-world web applications, one of the authors of the paper examined about a dozen web sites for which he has an account and usually visits. As a result, we found four of them are vulnerable to CSRF attacks as shown in Table 1. We verified all the attacks with Firefox 2.0.

Vulnerable web site	**Targeted sensitive operation**
A university credit union site	Money transfer between accounts; adding a new account
A university web mail	Deleting all emails in the Inbox
An online forum for HTML development	Posting a message; updating user profile
Department portal site	Editing biography information

Table 1. The CSRF vulnerabilities discovered in real world websites.

The university credit union site relies on session cookies for authentication. Some services provided in the online banking are vulnerable to the CSRF attack. In particular, adding new accounts and transferring money between accounts are vulnerable. In the experiment, we conducted a benign attack that transfers $0.01 from the victim's checking account to the saving account. We also successfully launch an attack to add an external account. Combining these two enables the adversary to transfer money from the victim's account to an arbitrary external account. Fortunately, the bank requires contacting the help-desk personally to confirm the operation of adding an external account. And also the bill paying service is not vulnerable.

The university web mail uses session cookies for authentication. Most sensitive operations (e.g., sending an email, changing the password) are protected against the CSRF attacks using secret token validation (see Section 2.3). However, the feature of "managing folders" is vulnerable, and a CSRF attack can be launched to remove all emails in the victim's Inbox.

In an online forum for HTML development, all operations are vulnerable to the CSRF attack. The attacker is able to impersonate the victim user to send a posting, update the user profile, and so on. The vulnerable forum is created using phpBB [16], which is the most widely used open source forum solution. All forums created using phpBB 2.0.21 or earlier are vulnerable to the CSRF attack [22]. This is a well-known vulnerability and there are CSRF attack generators for phpBB forums available online. Many public forums have upgraded to phpBB 2.0.22 or later, but there are still many forums using the vulnerable versions.

In the departmental portal site, a CSRF attack is able to edit the biography information of the victim shown on the webpage.

We have reported the vulnerabilities to the websites of the university credit union and the university web mail; we did not expose the name of those websites here because they have not fixed the vulnerabilities yet. These examples of vulnerabilities demonstrate that there exist a considerable amount of web services vulnerable to the CSRF attacks and the potential damage could be severe.

2.3 Existing CSRF Defenses

Several defense mechanisms have been proposed for CSFR attacks, we now discuss their limitations.

Filtering authentication tokens from cross-site requests. Johns et al. [10] proposed a client-side proxy solution, which stripes all authentication tokens from a cross-site request. The proxy intercepts web pages before they reach the browser and appends a secret random value to all URLs in the web page. Then the proxy removes the authentication tokens from the requests that do not have a correct random value. The solution breaks the auto-login feature and content sharing websites (such as Digg, Facebook, etc.) because it does not distinguish legitimate cross-site requests from malicious cross-site requests. In addition, it does not support HTML dynamically created in the browser and cannot work with SSL connections.

Authenticating web forms. The most popular CSRF defense is to authenticate the web form from which an HTTP request is generated. This is achieved by having a shared random secret, called a as a *secret validation token*, between the web form and the web server. If a web form provides a sensitive service, the web server embeds a secret validation token in an invisible field or the POST action URL of the form. Whenever form data is submitted, the request is processed only if it contains the correct secret value. Not knowing the secret, the adversary cannot forge a valid request. One drawback of this approach is it requires nontrivial changes to the web applications. Moreover, as pointed out by Barth et al. [3], although there exist several variants of this technique they are generally complicated to implement correctly. Many frameworks accidentally leak the secret token to other websites. For example, NoForge proposed in [11] leaks the token to other websites through the URL and the HTTP Referer header.

Referer-checking. In many cases, when the browser issues an HTTP request, it includes a Referer header that indicates which URL initiated the request. A web application can defend itself against CSRF attacks by rejecting the sensitive requests with a Referer of a different website. A major limitation with this approach is that some requests do not have a Referer header. There does not exist a standard specification on when to and when not to send the Referer header. Different browser vendors behave differently. Johns and Winter [10] give a summary on when browsers do not send the Referer header in major browsers. As a result, both a legitimate request and a malicious request may lack the Referer header. The adversary can easily construct a request lacking the Referer header. Moreover, because the Referer header may contains sensitive information that impinges on the privacy of web users, some users prohibit their browsers to send

Referer header and some network proxies and routers suppress the Referer headers. As a result, simply rejecting the requests lacking a Referer header incurs a compatibility penalty. Barth et al. [3] suggested a new Origin header that includes only the hostname part of the Referer header, to alleviate the privacy concern. It remains to be seen whether this will be adopted. In conclusion, using a server-side referer-checking to defeat the CSRF attacks has a dilemma in handling the requests that lack a Referer header.

2.4 A variant of CSRF attack

All existing CSRF defenses fail when facing a variant of CSRF attacks mentioned in [7]. We use the Facebook as an example to illustrate the attack. Facebook allows the users to post an article or a video from any website to the user's own profile. For example, the user can post a video from `Youtube.com` to his Facebook profile by clicking "Share – Facebook" under the video. When clicking the link, the following GET request is sent to the Facebook: `http://www.facebook.com/sharer.php?u=http://www.youtube.com/watch?v=VIDEO_ID&t=VIDEO_TITLE`. This request loads a confirmation page (Fig. 1(A)) which asks the user the click a "Post" button to complete the transaction. After the user clicking the "Post" button, a POST request is sent to `http://www.facebook.com/ajax/share.php` to confirm the posting operation.

An attacker is able to launch a CSRF attack that posts anything to the victim user's profile. On the malicious webpage, the attacker includes an iframe linking to the posting confirmation page (Fig. 1(A)). In addition, the attacker is able to auto-scroll the iframe to the "Post" button and hide other parts of the page by using two nested iframes and manipulating the sizes of the iframes. The sample code of the attack with Firefox 2.0 is given in Appendix A. As a result, what is shown in the browser looks like Fig. 1(B). The user can be easily tricked to click the "Post" button without knowing that he is posting something to his own Facebook profile.

Facebook.com uses secret validation token to defend against CSRF attacks. However, because the request is sent by user clicking the "Post" button in the confirmation page provided by Facebook the request will include a correct validation token. Using a referer-checking would also fail because the final posting request has a Referer header of Facebook.com.

This attack is traditionally defended using "frame busting", in which the target webpage includes a piece of JavaScript to force itself to be displayed in a top-level frame [12]. However, this defense can be defeated if the attacker disables the JavaScript in the sub-frame that links to the target webpage [9].

3 Browser-Enforced Authenticity Protection (BEAP)

CSRF attacks are particularly difficult to defend because cross-site requests are a feature of the web. Many web sites use legitimate cross-site requests, and some of these usages require the attachment of cookies to cross-site requests to work properly (e.g., posting a video from Youtube to Facebook in the above example).

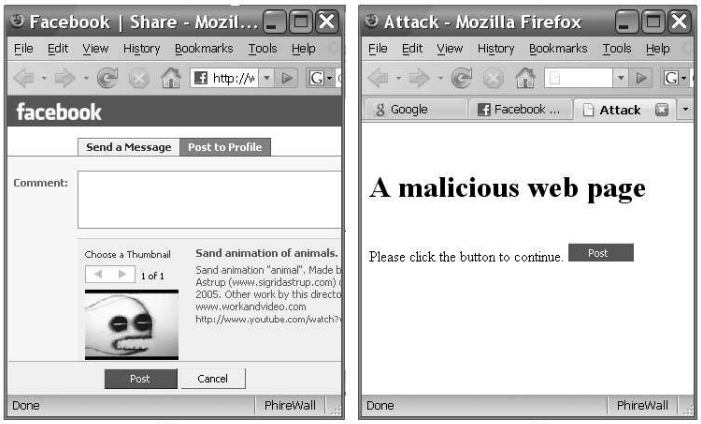

Fig. 1. (A): The confirmation page that posts a video from Youtube.com to the Facebook profile; (B): A malicious page that includes (A) as an iframe and tries to trick the user click the button without seeing other parts of (A);

To effectively defend against CSRF attacks, one needs as much information about an HTTP request as possible, in particular, how the request is triggered and crafted. Such information is available only within the browser. Existing defenses suffer from the fact that they do not have enough information about HTTP requests. They either have to change the web application to enhance the information they have or to use unreliable source of information (such as Referer header). Even when such information is available, it is still insufficient. For example, they cannot defend against the attack in Section 2.4 because while they can tell the request is coming from their web form, they do not know that the web form is actually embedded in a page controlled by the attacker.

We focus on browser-based defense against CSRF attacks. It is well known that CSRF is a confused deputy attack against the browser. The current web design assumes that the browser is *always* the deputy of the user and that any HTTP request sent by the browser reflects the user's intention. This assumption is not true as many HTTP requests are under the control of the web pages and do not necessarily reflect the user's intention. This confusion causes no harm when these requests have no sensitive consequences, and merely retrieve web pages from the web server. However, when these requests have sensitive consequences (such as financial consequences), it becomes a severe security concern. Because such requests occur in authenticated sessions, these requests have authentication tokens attached. The fundamental nature of the CSRF attack is that the user's browser is easily tricked into sending a sensitive request that does not reflect the user's intention.

Our solution to this problem is to directly address the confused deputy problem of the browser. More specifically, we propose Browser-Enforced Authenticity

Protection (BEAP), which enhances web browsers with a mechanism ensuring that *all sensitive requests sent by the browser reflect the user's intention*. BEAP achieves this through the following. First, BEAP infers whether an HTTP request reflects the intention of the user. Second, BEAP infers whether authentication tokens associated with the HTTP request are sensitive. An authentication token is sensitive if attaching the token to the HTTP request could have sensitive consequences. Third, if BEAP concludes that an HTTP request reflects the user's intention, the request is allowed to be sent with authentication tokens attached. If BEAP concludes that an HTTP request may not reflects the user's intention, it strips all sensitive authentication tokens from the HTTP request. In this rest of this section, we describe BEAP in details.

3.1 Inferring the User's Intention

In inferring whether an HTTP request reflects the user's intention, we classify the requests into two types depending on the source of the request. Type-1 requests are caused by the webpages hosted in the browser. When displaying a webpage, the browser may send additional requests to retrieve the resources included in the web page, such as images, scripts and so on. These resources may come from the same website or a third-party website. Similarly, when the user clicks a hyper-link or a button contained in a webpage, requests are sent by the browser. In addition, the Javascripts contained in the webpages may send requests as well. In all these cases, the URLs and contents of the requests are determined by the source webpage. Whether such a request reflects the user's intention is inferred by *browser-enforced Source-set checking*, which we will explain soon.

Type-2 requests are not associated with a source webpage. For example, when the user clicks an URL embedded in an email, the URL is passed to the browser as a startup argument, resulting in an HTTP request that is not associated with any webpage already hosted in the browser. We use the following *user-interface intention heuristics* to infer whether a type-2 request reflects the user's intention.

1. *Address-bar-entering.* When the user types in a URL in the address bar and hits enter, the request sent by the browser is considered as intended, because we can assure that the user intends to visit the URL she typed in.
   Note that we distinguish between typing in by keyboard and pasting from the clipboard. The adversary may send the victim an email, which contains a URL that links to a CSRF attack. Instead of providing a hyper-link for the user to click, the email can ask the user to copy and paste the URL to the browser's address-bar. To defeat this trick, only when the URL is typed in to the address-bar by the keyboard, the request is intended. If the URL is pasted from the clipboard, the request is not considered to be intended.
2. *Bookmark-clicking.* When the user selects a link from the bookmarks, the request is considered as intended, because users are usually careful in maintaining the bookmarks.
3. *Default-homepage.* When the browser displays the default home page either when it starts or when user clicks the "homepage" button, the request is

considered intended, because the configuration of default homepage is set by the user and cannot be easily modified by malicious web sites.

All other type-2 requests are not considered to be intended. For example, when the user clicks a link from the history, or when the user clicks a link outside the browser (e.g., in an email or a word document), the requests are not considered as intended. When performing those actions, users normally do not have a clear idea about which web site they are going to. The history and the links outside the browser may contain malicious contents that could launch CSRF attacks. Note that these requests are still allowed to proceed, we will only strip sensitive authentication tokens from them.

Browser-enforced *Source-set* Checking. To determine whether a type-1 request reflects the user's intention, we borrow the idea from the server-side referer-checking technique. Our approach has two significant differences. First, the enforcement is done by the browser rather than the web application. In this way, the Referer header does not need to be sent to the web server. This addresses the privacy concerns caused by sending out the Referer header, and it is compatible to the browsers and network devices that block the Referer header. In addition, the browser is able to check the Referer for all requests whose links are provided by a webpage (type-2 requests); so it avoids the dilemma in the server-side referer-checking with the requests that lack a Referer header. Second, we extend the notion of Referer to *Source-set* by taking into account the visual relationships among webpages in the browser. As a result, we can defeat the CSRF attack against Facebook mentioned in Section 2.4. *Source-set* checking can only be done in the browser.

Intuitively, the *Source-set* of a request includes all web pages that can potentially affect the request. We define the *Source-set* as follows.

Definition 1. *The* referer *of a request is the webpage that provides the link to the request. The* Source-set *of a request includes its referer and all webpages hosted in ancestor frames of the referer.*

For example, in Fig. 1, when the user clicks the "Post" button in the last tab, a request is sent to Facebook.com. The referer of the request is the innermost iframe that links to `http://www.facebook.com/sharer.php`. The *Source-set* includes the referer and its two ancestor webpages that are from the malicious website (In the attack, the malicious webpage includes an iframe linking to another malicious webpage, which further includes an iframe linking to Facebook. See Appendix A for the sample code of the attack).

The rationale for including all ancestors of the referer page in the *Source-set* of a request is because all ancestor webpages can potentially affect the request. Users are typically unaware of the existence of the frame hierarchy, and they assume they are visiting the website hosted in the top-level frame with the URL shown in the address-bar. The parent frame is able to manipulate the URL, size, position and scrolling of child frame, to fool the user. As a result, when the user performs some actions in the child frame, those actions may not reflect the user's

intention. Therefore, the referer and all its ancestor webpages are considered to be in the *Source-set* of a request.

Given a type-1 request, we consider it reflect the user's intention if all webpages in the *Source-set* are from the same website as the destination of the request. This is based on the following assumption: a request sent by a website to itself reflects the user's intention. In other words, a website won't launch a CSRF attack against itself.

3.2 Inferring the Sensitive Authentication Tokens

We have introduced a mechanism to infer whether an HTTP request reflects the user's intention. A simple way to defend against the CSRF attacks is to stripe all cookies and other authentication tokens from all requests that may not reflect the user's intention. However, such a policy would break some existing web applications. In particular, it would disable the legitimate cross-site requests that need to carry authentication tokens. An important observation is that although legitimate cross-site requests may need to carry an authentication token, legitimate cross-site requests typically do not lead to sensitive consequence, because sensitive operations typically require an explicit confirmation that is done in the target website. Based on this observation, we further infer whether an authentication token is sensitive or not for a request, and stripe only sensitive authentication tokens from requests that may not reflect the user's intension.

We use heuristics derived from analyzing the real-world web applications to determine whether an authentication token is sensitive or not for a request, based on the following information: (1) Whether the request is GET or POST. (2) Whether the token is a session cookie, a persistent cookie or an HTTP authorization header. (3) Whether the communication channel is HTTP or HTTPS. Our heuristics are summarized in Table 2 and are explained below.

	GET		POST
	HTTP	HTTPS	Sensitive
Session Cookies	Not Sensitive	Sensitive	
Persistent Cookies	Not Sensitive		
HTTP Authorization Header	Sensitive		

Table 2. The default policy enforced by the browser

The HTTP authorization headers are always sensitive. The HTTP authorization headers are typically used in the home/enterprise network. The services using the authorization headers for authentication are typically sensitive, e.g., home router administration, enterprise network services. In addition, it would be severe if a malicious website in the Internet is able to launch a CSRF attack against a service inside the Intranet.

For cookies we distinguish between the two request methods. All cookies that are attached to the POST requests are sensitive for two reasons. First, according

to the HTTP/1.1 RFC document, all the operations that have lasting observable effects should be requested using the method POST. Second, the POST requests are used to submit forms and forms are mostly submitted to the same website as that provides the form. So to stripe authentication tokens from the cross-site POST requests will protect all web applications that follow the RFC standard, and won't affect the existing web applications.

However, there exist some web applications that do not follow the standard and use GET requests for sensitive operations. We would like to protect those web applications against the CSRF attacks as well. For the cookies with GET requests, the policy further distinguishes between the session cookies and persistent cookies. The persistent cookies (those that have an expiration date) with GET requests are not sensitive. The persistent cookies are commonly used by the websites to provide personalized services without asking the user to explicitly log in. For example, `Amazon.com` displays recommendations based on the user's history activities. This is achieved by storing the user's identity and related information in persistent cookies. If the user links to `Amazon.com` from a third party website (e.g., a search engine), the request should carry the persistent cookies so that `Amazon.com` is able to recognize the user and provides a personalized service. Therefore, there exists legitimate cross-site GET requests that need to carry persistent cookies. On the other hand, most sensitive web applications (especially financial websites such as banks) use session cookies (those that does not have an expiration date and will be deleted when the browser is closed) as the authentication token for sensitive operations. For example, the persistent cookies are not enough for a user to place an order in `Amazon.com`, he needs to type in his password to obtain a session cookie to place an order. Some financial websites provide a "Remember me" option with the login form, but typically that is used to remember the user's username, the user still needs to type in the password to obtain a session cookie in order to access his account. Furthermore, using persistent cookies for sensitive operations is a bad practice, because the users may access their accounts from public computers (e.g., in an Internet Cafe). Using persistent cookies for authenticating sensitive operations would allow the persons who use the same computer following the user to impersonate the user.

It is a bit complicated for the session cookies with GET requests. We observe some websites issue legitimate cross-site GET requests that need to carry session cookies. In particular, the content sharing websites, such as Digg, Facebook, etc., allow people to discover and share contents from anywhere on the Internet, by submitting links and stories. Many webpages include links to the submission pages of those websites, so that the users can easily post the current article or video to their accounts. For example, as shown in Figure 2, `Youtube.com` provides links to various content sharing websites under each video. When clicking the Facebook link, a GET request is sent from Youtube.com to Facebook.com. If the user already logs in to Facebook.com, the request will carry the session cookie and the user can be directly linked to the submission page (Fig. 1(A)) without logging in again. To preserve this functionality of the content sharing websites, the policy treats the session cookies with GET requests using the HTTP protocol

as not sensitive. In contract, the session cookies with GET requests using the HTTPS protocol are sensitive, because the sensitive services are typically served over HTTPS.

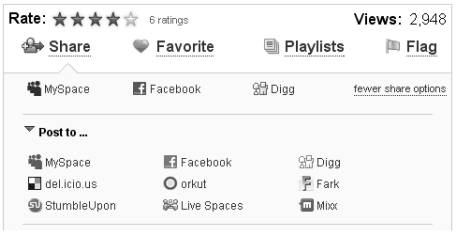

Fig. 2. Youtube provides links to various content sharing websites under the video player.

In conclusion, we infer whether an authentication token is sensitive as summarized in Table 2. To defend against the CSRF attack, we stripe the sensitive authentication tokens from the requests that may not reflect the user's intention.

3.3 Implementation

We have implemented a prototype of our proposal as a Firefox browser extension. It consists of about 800 lines of Javascript code. The extension intercepts each request when it is going to be sent, and removes the cookies and HTTP authorization headers that are not allowed to be attached according to the policy. The user interface in Firefox is implemented using XUL (XML User Interface Language), which is an XML user interface markup language. The XUL is flexible and extensible. To implement the user intention heuristics for type-1 requests, the extension hooks onto the events corresponding to those actions and overloads the event-handlers. To compute the *Source-set* of each request, the extension first identified the referer of the request, and then computes the source-set based on the frame hierarchy. The overhead introduced by our implementation is minimal. See Appendix B for details.

4 Evaluation and Discussions

Effectiveness of BEAP. How effective is BEAP for defending against CSRF attacks? In other words, how effective dose BEAP achieves "all sensitive requests sent by the browser reflect the user's intention"? We now answer these questions by analyzing under what assumptions the two inferences work correctly.

We observe that, under three assumptions, a CSRF attack always results in a request that BEAP considers to not reflect the user's intention. First, the browser has not been compromised. BEAP is not designed to defend against attacks that exploit vulnerabilities in browsers to take over the browser or the operating

system. BEAP defends against CSRF attacks, which exploit web browsers' design feature of allowing cross-site requests. Defending against browser exploitation is orthogonal to our work. Second, a user will not type in a CSRF attack URL in the address bar, or include a CSRF attack page in the bookmark, or use it as the default homepage. Under these two assumptions, type-2 requests that are considered as intended are not CSRF attacks. Third, a website does not include CSRF attacks against itself. This ensures that any CSRF attack via type-1 requests will be correctly classified. The third assumption means that we cannot defend against CSRF attacks that are injected into the target website. For example, the attacker may be able to inject a CSRF attack into a forum via a posting, which sends a posting on the victim's behalf. In this case, the malicious request is actually not a cross-site request, and will be treated as intended. Such an attack cannot be defeated by a pure client-side defense, because the browser cannot tell which requests in a webpage are legitimately added by the web site and which ones are maliciously added by user postings. The problem should be addressed by having the web application sanitize the user input to be displayed in the website, similar to defending against XSS attacks.

Second, BEAP allows non-sensitive cookies to be sent with requests that are not intended. This causes no harm when these requests do not have sensitive consequences. This is true assuming that websites do not contain sensitive operations that (1) use GET requests and rely on persistent cookies for authentication, or (2) use GET requests over HTTP and rely on session cookies for authentication. We would like to point out that these are all bad practices and are vulnerable to attacks other than CSRF attacks. First, using GET for requests that have sensitive consequence violates the HTTP/1.1 standard [6]. Second, when using persistent cookies for authenticating sensitive services, the accounts can be easily stolen if the user access the account in a public computer. Third, serving sensitive service over HTTP enables the network attacker to launch session injection attack. In particular, we did not observe any financial websites violate these assumptions; they are all hosted over HTTPS and relying on session cookies for authentication.

We have also experimentally evaluated our implementation, by verifying that it successfully defends against all attacks we have found in Section 2.2.

Compatibility of BEAP. BEAP will stripe cookies and HTTP authentication headers from some requests. Would this affect the existing web applications and change the user's browsing experiences? We now show that the answer is no.

First, we point out that cookie blocking has already been used for other purposes. Cookies, such as those set by `doubleclick.com`, can be used to track users' browsing behavior and violate user' privacy. Because of this, Internet Explorer 6 and later versions protect the user's privacy with respect to cookies [14]. In particular, IE requires web sites to deploy policies as defined by P3P (Platform for Privacy Preferences) [1]. When a website does not provide a P3P policy or the policy does not satisfy the user's preference, IE performs cookie filtering against the website. The approach applied by IE's cookie filtering has similarities with our defense against the CSRF attacks, but it aims at protecting privacy while

we aim at protecting authenticity. The cookie filtering infers whether a cookie may violate the user privacy based on the type of the cookie and the heuristics derived from real-world web applications. When the focus is privacy rather than authenticity, persistent cookies are considered more sensitive than session cookies, and a persistent cookie with no associated P3P policy is "leashed", and will not be attached to requests downloading third-party content.

We tested the compatibility of our implementation against 19 popular websites ranging 6 categories as shown in Table 3. On each website, we logged in the account and tried the major functionalities provided by the website and the operations that normal users would perform. For some of them, we created a new user account. Everything worked well, and all the browsing experiences remained unchanged. We did not use a crawler or an automatic tool to perform a large-scale compatibility testing, because testing the compatibility is possible only when we have an account on a website and log into the account to perform authenticated operations. In particular, creating web accounts on financial websites typically require having physical accounts.

Categories	**Web sites**	**Operations**
Email	Gmail, Hotmail	Check emails, send emails, change settings
Social network	MySpace, Facebook, Blogger	create accounts, add friends, modify the profiles
Online shopping	Amazon, ebay	place bids, buy items, update the profiles
Financial sites	PayPal, Chase, Citi Cards, American Express, Fidelity, Discover Cards, a credit union	add a bank account, money transfer, pay bills
Personal desktop	iGoogle, Windows Live	Setup a personal desktop
Internet portal	Yahoo!	Check emails, write a movie review, join a group

Table 3. The web applications used for compatibility evaluation.

Finally, we note that while we have not encountered web sites that use cross-site requests in a way affected by BEAP's policy, it is certainly plausible for such sites to exist. However, we note that the functionalities provided by these web sites are not disabled. When cookies are striped, the worst case is that the user needs to re-enter the password in order to perform certain operations.

5 Conclusions

CSRF vulnerabilities are common in real-world web applications, and the consequences of such attacks are most severe with financial websites. We have proposed a browser-based mechanism called BEAP to defend against the CSRF attacks. It infers whether a request sent by the browser is sensitive and whether an authentication token is sensitive, and stripes sensitive authentication tokens from any request that may not reflect the user's intention. We have implemented BEAP as a browser extension for Firefox, and have shown that BEAP can effectively defend against the CSRF attacks, and does not break the existing web applications.

References

1. The platform for privacy preferences project (p3p). *http://www.w3.org/TR/P3P*.
2. The web hacking incidents database, 2008. *http://www.webappsec.org/projects/whid/byid_id_2008-05.shtml*.
3. A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008.
4. J. Franks, P. Hallam-Baker, J.Hostetler, S.Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication. RFC 2617, June 1999. *http://www.ietf.org/rfc/rfc2617.txt*.
5. Google. Load time analyzer 1.5, firefox add-on, Mar. 2007. *https://addons.mozilla.org/en-US/firefox/addon/3371*.
6. N. W. Group. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999. *http://www.ietf.org/rfc/rfc2616.txt*.
7. R. Hansen and T. Stracener. Xploiting google gadgets: Gmalware and beyond, Aug. 2008.
8. K. J. Higgins. CSRF vulnerability: A 'sleeping giant', 2006.
9. C. Jackson. Defeating frame busting techniques, 2005. *http://www.crypto.stanford.edu/framebust/*.
10. M. Johns and J. Winter. RequestRodeo: Client side protetion against session riding. In *Proceedings of the OWASP Europe 2006 Conference*, 2006.
11. N. Jovanvoic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Proceedings of the Second IEEE Conference on Security and Privacy in Communication Networks*, September 2006.
12. P. Koch. Frame busting. *http://www.quirksmode.org/js/framebust.html*.
13. D. Kristol and L. Montulli. HTTP state management mechanism. RFC 2965, Oct. 2000. *http://www.ietf.org/rfc/rfc2965.txt*.
14. MSDN. Privacy in internet explorer 6. *http://msdn.microsoft.com/en-us/library/ms537343(VS.85).aspx*.
15. OWASP. Top ten most critical web application security vulnerabilties. Whitepaper, 2007. *http://www.owasp.org/index.php/Top_10_2007*.
16. phpBB. Create communities worldwide. *http://www.phpbb.com*.
17. C. Shiflett. Foiling cross-site attacks, Oct. 2001. *http://shiflett.org/articles/foiling-cross-site-attacks*.
18. C. Shiflett. Security corner: Cross-site request forgeries, Dec. 2004. *http://shiflett.org/articles/cross-site-request-forgeries*.
19. US-CERT. Cross-site request forgery (CSRF) vulnerability in @mail webmail 4.51. CVE-2006-6701, Dec. 2006. *http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-6701*.
20. US-CERT. Multiple cross-site request forgery (CSRF) vulnerabilities in phpmyadmin before 2.9.1. CVE-2006-5116, Oct. 2006. *http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-5116*.
21. US-CERT. Google gmail cross-site request forgery vulnerability. Vulnerability Note 571584, Oct. 2007. *http://www.kb.cert.org/vuls/id/571584*.
22. US-CERT. Cross-site request forgery (CSRF) vulnerability in privmsg.php in phpbb 2.0.22. CVE-2008-0471, Jan. 2008. *http://nvd.nist.gov/nvd.cfm?cvename=CVE-2008-0471*.
23. US-CERT. Cross-site request forgery (CSRF) vulnerability in the Linksys wrt54gl wireless-g broadband router. CVE-2008-0228, Jan. 2008. *http://nvd.nist.gov/nvd.cfm?cvename=CVE-2008-0228*.
24. P. W. Cross-site request forgery, 2001. *http://www.tux.org/p̃eterw/csrf.txt*.

Appendix

A The Attack Code of the Facebook Example

The following code is constructed for Firefox 2.0.

1. The top-level frame of the malicious webpage.

```
<html>
    <head>
        <title>Attack</title>
    </head>
    <body>
        <br><h1>A malicious web page</h1></br>
        Please click the button to continue.
        <iframe src ="inner.html" width=70 marginwidth="25%" height=20
        scrolling="no" frameborder="0" class="iframe"></iframe>

    </body>
</html>
```

2. The mid-level frame "inner.html".

```
<html>
    <body onload="window.scrollTo(1440, 980);">
        <iframe src="http://www.facebook.com/sharer.php?u=
        http%3A//www.youtube.com/watch%3Fv%3DnQSZOri6Pj0&
        t=Sand%20animation%20of%20animals."
        width=3000 height=1000 frameborder=0></iframe>
    </body>
</html>
```

B Performance Evaluation

We evaluated the performance overhead introduced by the browser extension. The experiment was carried out on a 2.19GHz Intel Core 2 Duo with 2GB of memory , running the Windows operating system. We used Firefox 2.0.0.13 as a base for performance comparison. We compared the page loading times for account login on a few common web sites. The page loading times are measured using the Load Time Analyzer extension [5]. Each page is loaded 5 times, and the loading times are averaged. The results is shown in Table 4. The performance overhead is less than 8%, with an average of 2%.

Web sites	MySpace	iGoogle	Paypal	Yahoo!	Ebay
Page loading times for login (base)	2629	1352	6422	1094	1387
Page loading times for login (upgraded)	2733	1464	6484	1125	1399

Table 4. The comparison of the page loading times for login.