Preventing Cross Site Request Forgery Attacks

Nenad Jovanovic, Engin Kirda, and Christopher Kruegel
Secure Systems Lab
Technical University of Vienna
Email: {enji,ek,chris}@seclab.tuwien.ac.at

Abstract

The web has become an indispensable part of our lives. Unfortunately, as our dependency on the web increases, so does the interest of attackers in exploiting web applications and web-based information systems. Previous work in the field of web application security has mainly focused on the mitigation of Cross Site Scripting (XSS) and SQL injection attacks. In contrast, Cross Site Request Forgery (XSRF) attacks have not received much attention. In an XSRF attack, the trust of a web application in its authenticated users is exploited by letting the attacker make arbitrary HTTP requests on behalf of a victim user. The problem is that web applications typically act upon such requests without verifying that the performed actions are indeed intentional. Because XSRF is a relatively new security problem, it is largely unknown by web application developers. As a result, there exist many web applications that are vulnerable to XSRF. Unfortunately, existing mitigation approaches are time-consuming and error-prone, as they require manual effort to integrate defense techniques into existing systems. In this paper, we present a solution that provides a completely automatic protection from XSRF attacks. More precisely, our approach is based on a server-side proxy that detects and prevents XSRF attacks in a way that is transparent to users as well as to the web application itself. We provide experimental results that demonstrate that we can use our prototype to secure a number of popular open-source web applications, without negatively affecting their behavior.

1 Introduction

Cross site request forgery [18, 20, 23] (abbreviated XSRF or CSRF, sometimes also called "Session Riding"), denotes a relatively new class of attack against web application users. By launching a successful XSRF attack against a user, an adversary is able to initiate arbitrary HTTP requests from that user to the vulnerable web application. Thus, if the victim is authenticated, a successful XSRF attack effectively bypasses the underlying authentication mechanism. Depending on the web application, the attacker could, for instance, post messages or send mails in the name of the victim, or even change the victim's login name and password. Furthermore, the damage caused by such attacks can be severe. In contrast to the well-known web security problems such as SQL injection and XSS, cross site request forgery (XSRF) appears to be a problem that is little known by web application developers and the academic community. As a result, only few mitigation solutions exist. Unfortunately, these solutions do not offer complete protection against XSRF or require significant modifications to each individual web application that should be protected.

In this paper, we present a solution that provides protection from XSRF attacks. More precisely, our approach is based on a server-side proxy that detects and prevents XSRF attacks in a way that is transparent to users as well as to the web application itself. One important advantage of our solution is that there is only minimal manual effort required to protect existing applications. Our experimental results demonstrate that we can use our prototype to secure a number of popular open-source web applications against XSRF attacks, without negatively affecting the applications' behavior. An expanded version of this paper containing additional details can be found on our web site [6].

2 Cross Site Request Forgery

In this section, we introduce the concepts and mechanisms behind XSRF attacks in more detail.

2.1 User Authentication in Web Applications

HTTP is a stateless protocol that is not able to recognize when a number of requests all belong to a particular user. This is cumbersome when applications have to support user authentication, as there is no straightforward mechanism to identify requests of a user that has already performed a successful login. One way to overcome this problem is to preserve user-specific state in client-side cookies [12]. By inserting a `Set-Cookie` HTTP header into the server's reply, a web application can instruct the client browser to create a cookie with a given name and value. In all subsequent requests to the server, the browser automatically includes this cookie information, using the `Cookie` HTTP header. Based on this cookie information, a web application can then associate requests with certain clients.

Of course, using cookies to store information is only suitable for data that may be freely modified by the client. Because data is stored on the client's machine, it is under the user's direct control. For some web applications, information must not be modified between requests. In other cases, the amount of data that is associated with a certain user is too large to constantly exchange it between the client and the server. To address these issues, web applications typically make use of *sessions*.

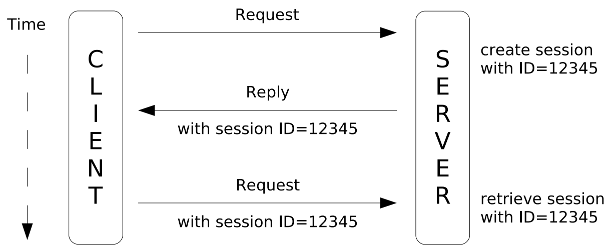


Figure 1. Using sessions for server-side state.

A session is established to recognize requests that belong together, and to associate these requests with (session) data stored at the server. To this end, each session is assigned a unique identifier (the session ID), and the client is only provided with this identifier. With each request, the client provides its ID (see Figure 1), which the web application can subsequently use to retrieve the appropriate session data.

There are two possibilities to have the client attach the session IDs to each request. The first possibility is to perform *URL rewriting*. In this case, hyperlinks and other request triggers (such as HTML forms) are augmented with an additional parameter that contains the session ID. For instance, the hyperlink with the relative target location `./index.php` might be extended into `./index.php?sessid=12345` to store the session ID with the value "12345". URL rewriting can be implemented by the application. However, many application run-time and development environments (for example, PHP [13]) already provide an automatic rewriting mechanism to ease the task for web developers. The second possibility to include session information is to set cookies, which are automatically sent by the user browser with each request.

A convenient side-effect of sessions is that they can be used to track the authentication state of a user. For instance, after a successful authentication of the client, a web application could store a boolean value `auth=true` for future reference. When a user continues the current session by sending subsequent requests, the web application can easily determine whether that user is already logged in by consulting this boolean value. As a result, the user is able to perform privileged actions without the need to explicitly submit a password each time. Instead, the authentication occurs implicitly by the underlying session mechanism. That is, the session ID serves as an implicit authentication token.

2.2 Exploiting Session Mechanisms

The presented concepts behind web application sessions imply that the session ID temporarily has the same significance as the user's original credentials. That is, as long as the session has not expired, a web application treats requests with valid session IDs as requests of the user who initially started the session. If an attacker manages to obtain the session ID of an authenticated user, it is possible to issue requests with the same privileges as this user. As a result, the session ID has become a primary target for web application attacks. For instance, one of the goals of cross site scripting (XSS) attacks is to inject malicious JavaScript code into the reply of a vulnerable application with the aim to leak the session ID to the attacker. In such attacks, the attacker abuses the fact that web applications cannot reliably distinguish between requests with session IDs originating from the legitimate user, and requests with stolen session IDs originating from an attacker.

In contrast, *cross site request forgery* (XSRF) is a relatively unknown form of attack that is *not* motivated by the attempt to steal the session ID. Instead, XSRF attacks abuse the fact that most web applications cannot distinguish between *intended* user requests, and requests that the user issued because she was tricked to do so. For instance, assume that the online banking application of `www.bigbank.com` receives the following request from an authenticated user:

```
GET /transfer.php?amount=10000&to=7777
```

The application interprets this as a request to transfer 10,000 USD from the user's bank account to the account with number 7,777. Since BigBank's web application does not take into account the possibility of XSRF attacks (unfortunately, this problem is present in many other web applications), it optimistically assumes that the request indeed originated from the HTML form designated for this purpose (shown in Figure 2), and faithfully carries out the transaction. In reality, however, the GET request was generated in the following way: After paying an invoice via online banking, the user forgets to log out and proceeds by surfing to some other web sites. One of these sites, `evilxsrf.org`, contains the following hyperlink:

```
<a href='www.bigbank.com/transfer.php?
   amount=10000?to=7777'>Click here</a>
for something really interesting.
```

```
<form action="transfer.php" method="get">
 To: <input type="text" name="to"/>
 Amount: <input type="text" name="amount"/>
 <input type="submit" value="Submit"/>
</form>
```

Figure 2. Legitimate money transaction form of `www.bigbank.com`.

As soon as the user clicks on this link, the previously presented GET request is sent to `www.bigbank.com`. Since the user forgot to log out, the session has not been invalidated yet and the cookie with the session ID still exists. As a result, the user's browser automatically appends the cookie to the request, which is successfully authenticated by the banking application. Without intending so, the user has just transferred a considerable amount of money to some unknown bank account.

The described, simple attack will probably work only against users that are not security-aware and have limited knowledge about the mechanisms used in web applications. For instance, the value of the `href` attribute will appear in the browser's status bar as soon as the user moves the mouse pointer above the link (although this could possibly be avoided by using JavaScript code that hides the status bar). Also, users become increasingly aware of the security implications of clicking on links in mails. However, the critical request can also be performed through the following `src` attribute of an image tag:

```
<img src='www.bigbank.com/transfer.php?
   amount=10000?to=7777'>
```

When the user visits the page containing this tag, the browser immediately attempts to retrieve the image by sending the appropriate GET request to `www.bigbank.com`. Compared to the previous case, the user did not even have to actively follow any link, which clearly makes the attack more dangerous. Moreover, XSRF attacks are not limited to GET requests. Figure 3 demonstrates how an equivalent POST request can be assembled through an HTML form and automatically submitted by a short piece of JavaScript code. Once again, visiting the malicious HTML page is sufficient for the attack to succeed. Note that although disabling JavaScript would prevent the automatic submission of the form in this case, this measure is not suitable as general cure against XSRF attacks. This underlines the fact that XSRF problems are independent of XSS vulnerabilities and do not rely on the execution or injection of malicious JavaScript code.

The analysis of the mechanisms behind XSRF attacks leads to the following observation: As long as a user is logged in to a web application, she is vulnerable. A single mouse click or just browsing a page under the attacker's control can easily lead to unintended requests. Most web applications are not aware of this fact, leaving their users in danger.

```
<form action="http://www.bigbank.com/transfer.php"
 method="post">
 <input type="hidden" name="to" value="7777"/>
 <input type="hidden" name="amount" value="10000"/>
 <input type="submit"/>
</form>
<script type="text/javascript">
 document.forms[0].submit();
</script>
```

Figure 3. Malicious XSRF page for POST parameters.

3 Existing Mitigation Techniques

A common advice for mitigating the XSRF threat that appears frequently in the web development community is to use POST instead of GET parameters. However, as we demonstrated in the previous section, this approach is not adequate for preventing XSRF attacks. It only raises the bar for the attacker, as it closes certain attack vectors such as the use of image tags. In addition, completely removing the use of GET parameters is sometimes not possible when it would result in applications that are more cumbersome for users to navigate and more difficult for developers to implement.

Checking the HTTP `Referer` header would be an effective countermeasure if the web application could rely on its correctness. In the previous example, the request that is generated by clicking the malicious link would contain a referrer to `evilxsrf.org`. By maintaining a white-list of accepted referrers, the banking application could deduce that this request was initiated due to an XSRF attack, and refuse to perform the transaction. Unfortunately, modern browsers can be configured to send empty or even arbitrary values for this header. Moreover, sending the referrer header is discouraged, as it may result in leaking sensitive information to third parties (as mentioned in RFC 2616 [17]). This leads to the question of how to treat *empty* referrer headers. When classifying requests with an empty referrer header as valid, it would become impossible to detect attacks against users who follow the recommendation and disable the transmission of the referrer header. On the other hand, when regarding such requests as XSRF attacks, *all* requests of these users would be rejected. This dilemma is further aggravated by the fact that an attacker can make use of several browser-specific tricks to trigger an XSRF request with an empty referrer [10].

From the previous explanation, it should become clear that XSRF attacks only work when a cookie is used to store the session ID. The reason is that the browser *automatically* includes cookies into requests, even when a user clicks on a simple link. In case of URL rewriting, on the other hand, the session ID has to be embedded into the request trigger (e.g., a hyperlink or a form) explicitly. Thus, when the attacker attempts to create a page with a hyperlink that performs the XSRF request, this link will not contain the proper session ID and thus, will not result in a successful attack. Of course, the adversary cannot prepare the link with a correct session ID, because he has no knowledge about this identifier; otherwise he could use this ID directly to impersonate the authenticated user.

The problem is that cookie-based session management is much more popular and wide-spread for a number of reasons, some of which are even security-related [5, 14, 15]. For example, in URL-based solutions, the session ID appears in the browser's location bar. One implication is that a user might bookmark a page together with the session ID. When visiting the web site via this bookmark, the web server might again associate the session with this ID (this type of session management is called *permissive* and is present, for example, in PHP). As a result, one session ID is used for multiple sessions, increasing the chances for an attacker to successfully steal and exploit the ID. Another possibility is that an attacker could simply peek over a victim's shoulder to steal the session ID (e.g., in a public Internet cafe).

The best solution proposed so far is the use of a *shared secret* (or *token*) between the client and the server to identify the actual origin of a request. For instance, the example banking application from the previous section could be adapted such that the form shown in Figure 2 contains an additional, hidden token field. This token must be generated by the application (such that it is not easily guessable by an attacker) and associated with the current session. Requests for financial transactions are only processed if they contain the correct token. The drawback of this approach is the considerable amount of manual work that it involves. Many current web applications have evolved into large and complex systems, and retrofitting them with the mechanisms necessary for token management would require detailed application-specific knowledge and considerable modifications to the application source code. Even more important, there is no guarantee that the modified code is indeed free of XSRF vulnerabilities, as developers tend to make errors and omissions.

XSRF attacks are still relatively unknown to web developers and attackers. Nevertheless, we believe that the attention paid to this class of attacks will reach that of more traditional XSS attacks in the near future as the attack becomes better known and understood. Unfortunately, current mitigation techniques have shortcomings that limit their general applicability. To address this problem, the following section presents a novel and automatic approach for XSRF protection.

4 A Proxy-Based Solution

To be useful in practice, a mitigation technique for XSRF attacks has to satisfy two properties. First, it has to be effective in detecting and preventing XSRF attacks with a very low false negative and false positive rate. Second, it should be generic and spare web site administrators and programmers from application-specific modifications. Unfortunately, all existing approaches presented in the previous section fail in at least one of the two aspects.

Our solution to the XSRF problem is to decouple the necessary security mechanisms from the application and to provide a separate module that can be plugged into existing systems with minimal effort. More precisely, we propose a *proxy* that is placed on the server side between the web server and the target application. This proxy is able to inspect and modify client requests as well as the application's replies (output) to automatically and transparently extend applications with the previously sketched *shared secret* technique. In particular, the proxy has to

- ensure that replies to an authenticated user are modified in such a way that future requests originating from this document (i.e, through hyperlinks and forms) will contain a valid *token*, and
- take countermeasures against requests from authenticated users that do not contain a valid token.

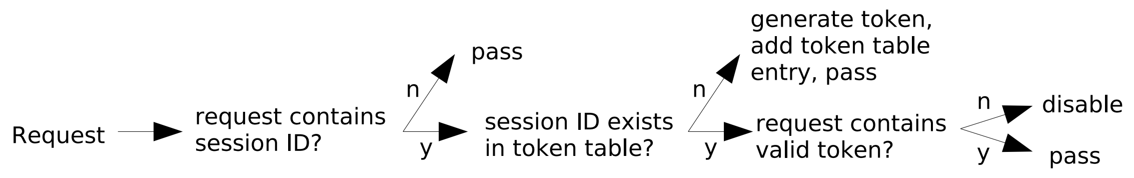
An essential prerequisite for this mechanism is the proxy's ability to associate a user's session with a valid token. To this end, the proxy maintains a *token table* with entries that map session IDs to tokens.

By decoupling the proxy from the actual application, the XSRF protection can be offered transparently for (virtually) all applications. Note that, alternatively, our proxy could also be located between the client and the web server. However, this case could lead to problems in combination with SSL connections. With our proposed architecture, SSL issues are directly handled by the web server, which eases the tasks that are to be performed by the proxy.

In the following sections, we present a more detailed description of how requests to and replies from the web application are handled, along with illustrative examples.

4.1 Request Processing

Figure 4 provides an overview of the steps that the proxy has to take during request processing. As a first step, we check whether the request contains a session ID or not. If there is no session ID in the request, it is classified as benign. The reason is that since the request does not refer to an existing, authenticated session, it is not able to perform any privileged actions. Thus, we can safely pass the request to the target application.



Request
request contains session ID?
n
y
pass
session ID exists in token table?
n
y
generate token, add token table entry, pass
request contains valid token?
n
y
disable
pass

Figure 4. Request processing.

If the request does contain a session ID, we consult the token table to check whether there already exists an entry with a corresponding token. If there is such an entry, we require that the request also contains this token. A request that fails to satisfy this condition is classified as an XSRF attack. This is because legitimate requests, originating from a document generated by the protected application, are guaranteed to *always* contain a token when they use a session ID. The reason is that the documents produced by the application are modified such that this token will be present (the exact mechanism to achieve this is described in detail in Section 4.2).

The action to be taken when an XSRF request is detected is configurable by the site administrator. In our experiments, we generated a warning message to inform the victim about the attack, together with a (correctly tokenized) link to the application's main page. Note that there is no need to terminate the user's current session when an XSRF attack is detected. After following the link provided in the generated warning message, the user can continue her work normally. An even more convenient, but less educational, alternative would be to instantly redirect the user to the main page, without the need for any additional interaction.

In the case when the request contains a session ID that does not exist in the token table, we have to assume that a new session was established. The proxy generates a new, random token and inserts the token, together with the session ID, into the token table. In addition, the request is passed to the target application.

4.2 Reply Processing

As discussed briefly in the previous section, the task of the reply processing step is to extend the output of a web application such that a subsequent request of the user contains the correct token. This is achieved in a fashion similar to URL rewriting. Assume that the proxy has to process an output page of the target application containing the following relative hyperlink:

```
<a href='index.php?action=logout'>
  LogOut
</a>
```

Assume further that the proxy has already determined that the client is authenticated, and that a certain session ID is in use. In this case, it is necessary to rewrite the hyperlink's URL such that it contains the token associated with this session ID:

```
<a href='index.php?action=logout
  &token=99'>LogOut
</a>
```

When the user follows this link, the mechanism has ensured that the proper token is transmitted.

The name of the parameter that stores the token ("token" in this example) can be chosen arbitrarily, but must not interfere with the names of other parameters used by the target application. The token's value ("99") is retrieved from the token table that the proxy maintains.

At this point, an important question is the following: How can the proxy determine whether a client is authenticated or not? For our purposes, we treat the state "a client is authenticated" as equal to "a client has an active session." This is a safe assumption, because XSRF attacks cannot succeed when there is no session information that can be exploited to force the victim into performing privileged actions (that is, actions which require previous authentication) on behalf of the attacker.

The next question is how to determine whether a user has an active session or not. Programming languages such as PHP provide a built-in session infrastructure that could be consulted about whether there exists such a session. However, many applications make use of custom session management techniques. Sometimes, session information is even stored in a back-end database. In such cases, the target application could be instrumented with functions that enable the proxy to issue appropriate queries about the session state. Unfortunately, this would lead to the undesirable necessity to perform application-specific modifications.

We solve the problem of determining whether a session exists in the following way. Basically, there are two cases that have to be distinguished, depending on whether the application sets a session cookie while processing a client's request or not. We can check this by searching the application's reply for an HTTP `Set-Cookie` header. Of course, this approach requires our system to distinguish between session cookies (i.e., cookies that store session information) and cookies that are set for other purposes. While it might be possible to use heuristics to automatically identify session cookies, we currently require the administrator of the system to manually specify their names. Typically, this is straightforward, as many applications make use of

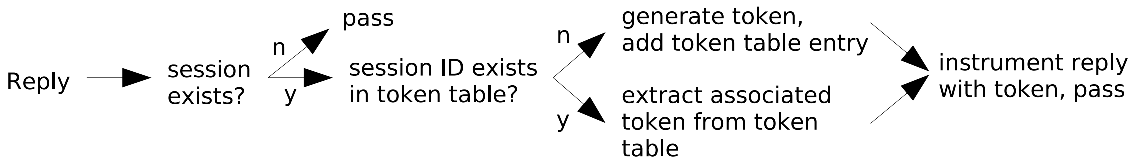

Reply
session exists?
n
y
pass
session ID exists in token table?
n
y
generate token, add token table entry
extract associated token from token table
instrument reply with token, pass

Figure 5. Reply processing.

the built-in session infrastructure provided by the run-time environment. For example, when PHP is used, the name of the session cookie defaults to "PHPSESSID." If a session cookie *is* set in the application's reply, we assume that there exists a session, and this session has an ID equal to the session cookie's value. If a session cookie is *not* set in the reply, we further investigate the client's request that corresponds to the reply. If this request contains a session ID, we conclude again that there exists a session. Such a situation arises regularly when a client is already logged in, and her browser automatically sends the authentication cookie to the server along with each request.

At this point, note that our approach is safe (i.e., it does not miss any attacks). If there exists no session, although the proxy assumes that there is one, tokens are included into the applications' documents, but its regular behavior is not affected. On the other hand, if we miss an active session, the reply would not be instrumented with the token. Subsequently, this would lead to a false XSRF alarm for the next user request.

After determining that there exists an active session, we query the token table for an associated token. If there is no such entry, it means that the session has been newly created. Hence, we generate a random token and add a corresponding entry to the token table. Finally, the reply is instrumented with the token before returning it to the client. The following fields have to be modified:

- `href` attributes of `a` tags.
- `action` attributes of `form` tags.
- `src` attributes of `frame` and `iframe` tags.
- `onclick` attributes of `button` tags.
- `refresh` attributes of `button` tags.
- `url` attributes of `refresh` meta tags.

During our experiments, we did not encounter any other fields that required rewriting. However, extending the rewriting engine to take into account more fields would be straightforward. An overview of the complete reply processing step is given in Figure 5.

4.3 Token Table Cleanup

The token table should be freed from stale entries regularly to save memory and CPU time. To this end, we extended the table by a third column that holds timestamps, which indicate the point in time when the corresponding entry was last used. When the time that passed since this point is longer than a configurable session life-time (that defaults to 24 minutes, in accordance with PHP's default session life-time), the entry is removed.

4.4 Discussion of Attacks against the System

Our proxy cannot prevent XSRF attacks if the target application fails to defend against certain other types of attacks. For instance, if insufficient measures are taken against cross site scripting (XSS), the adversary could inject malicious JavaScript into the application that steals the user's cookies (containing the session ID). This underlines once more that a reasonable level of security can only be achieved by preparing against a wide range of possible attack vectors.

Our proxy treats all request triggers sent by the target application as legitimate. Hence, all these triggers are automatically instrumented with valid tokens. This implies that if an attacker managed to inject an XSRF hyperlink into the reply of a protected application, our proxy would automatically augment this hyperlink with a valid token as well. This would allow the adversary to induce requests to pages that accept GET parameters. Using POST instead of GET requests would raise the bar for the attacker in this case. However, the attacker could also point the injected link to a site under his control. When the victim's browser requests a page on this site, the `Referer` header could be used to extract the token, allowing the attacker to create valid POST requests as well. To mitigate these problems, the attacker should not be allowed to inject request triggers into an application, a requirement that should be fulfilled by security-aware systems in any case. In addition, clients can decrease their exposure by disallowing the transmission of `Referer` headers (e.g., by configuring their browser appropriately, or by using a firewall).

Given the design of our system, an attacker can obtain the token for any session ID that he presents to the proxy. To this end, the attacker can simply send a request with a particular session ID to the web application and afterwards simulate an XSRF attack against this session. Then, he can extract the associated token from the generated reply (for instance, from the link that is provided along with our warning message). Fortunately, this ability is of no use to the attacker because he has no knowledge about the session ID that a client uses. Otherwise, there would be no need to perform an XSRF attack. Instead, the attacker could directly impersonate the victim using the session ID. Of course, the range of possible session IDs must be large enough to thwart any brute-force guessing attempts.

Finally, the attacker could launch a denial-of-service attack against the proxy. Recall that the token table is extended by a new entry whenever an incoming request contains a session ID that does not exist in the token table. Thus, the adversary could flood the token table with a large number of session IDs with the intention of significantly degrading the proxy's performance. However, the same attack could be launched already against most web servers that track user sessions. The reason is that many applications start to issue session IDs *at the first visit* of a client, and not after the client has logged in. This corresponds exactly to the behavior of our proxy. Hence, for these systems, the possibility for such denial-of-service attacks is not originally introduced by the proxy.

4.5 Eliminating State

As mentioned previously, the token table associates session IDs with tokens. This explicit, stateful mapping can be replaced by a mapping that requires no state at all to be stored by the proxy. To this end, a token is computed by applying a hash function based on a secret server key to a session ID. As this approach does not require the explicit storage of session ID to token mappings (the token value can always be computed from the session ID), it eliminates the DoS attack vector sketched in Section 4.4. Another major difference compared to the token table approach is that the mappings remain constant over time. When using a token table, mappings change as they are introduced with randomly generated tokens and removed when they expire.

At first glance, static mappings might appear dangerous from a security point of view. An attacker could, in theory, construct an explicit representation of the hash function that is used by sending *all* possible session IDs to the server and writing down the observed tokens in return. By inverting this mapping, the attacker could deduce the session ID from the user's token (modulo hash collisions). This would turn the token into a piece of information worth stealing, increasing the attacker's opportunities. However, reconstructing even small parts of the hash function is not feasible in practice due to the vast number of possible session IDs. In PHP, for example, the space for session IDs comprises 62^{32} entries (32 digits or case-sensitive characters from a to z). For harvesting just 0.01% of all possible mappings in one year, an attacker would have to issue $7 * 10^{45}$ requests per second. An even higher level of security could be achieved by changing the hash function each year, invalidating an attacker's previous harvesting efforts. The resulting convenience penalty for users with active sessions would be minimal, as users with incorrect tokens are immediately provided with a link containing the correct token.

```
<select onChange="top.location.href=this.options[this.selectedIndex].value.
concat('&xsrf_token='+document.getElementsByTagName('html')[0].
getAttribute('xsrf_token'))">
```

Figure 6. JavaScript snippet from PhpNuke with modifications (in boldface).

4.6 Limitations

The presented concepts rely on the assumption that all request triggers (such as hyperlinks and `action` attributes of forms) are directly available in the output generated by the target application. If this is not the case, reply processing misses certain request triggers, which can result in subsequent false XSRF alarms. For instance, assume a client-side JavaScript is responsible for constructing a hyperlink that points back into the target application. At the time of request processing, this hyperlink is not detected. Hence, the link will eventually lack the necessary token, and result in a false XSRF warning when the client follows it. Fortunately, this is a minor issue for a number of reasons. First, such cases are very rare in practice. During our experiments, we observed this effect in not more than four select boxes. Moreover, adjusting the involved JavaScript such that it also integrates the token into the generated link is a trivial task that does not require any application-specific knowledge. For instance, Figure 6 shows such a JavaScript snippet from PhpNuke after our one-line modification (represented by the passage in boldface). The modified code simply extracts the token from the `xsrf_token` attribute of the document's `html` tag and appends it to the constructed URL. The token was previously embedded into the `html` tag by the proxy's rewriting engine. By adapting the JavaScript code at all offending locations, which took less then ten minutes for each application, we successfully eliminated all false warnings. Finally, note that this issue does *not* represent a gap in our security measures. There is no way for the attacker to exploit this issue and launch a successful XSRF attack.

5 Implementation

To demonstrate the feasibility of our concepts, we implemented *NoForge*, a server-side proxy that is able to defend PHP applications against XSRF attacks. As explained previously, this proxy is located between the web server and the protected web applications. To realize this in a straightforward fashion, we decided to implement the proxy as wrapper functions around those PHP applications that we intend to protect. These wrapper functions check the input and output of the application and perform the necessary request and reply processing.

Table 1. Tested Applications

Application	Version	Downloads	Exploits
phpBB	2.0.19	10,483,075	Delete postings.
			Send postings.
phpMyAdmin	2.8.0.2	9,494,550	Delete databases.
			Create databases.
Gallery	2.0.4	3,937,352	None found.
XOOPS	2.0.13.2	3,448,408	None found.
phpNuke	7.0	2,727,943	Delete messages.
			Add messages.
Coppermine Photo Gallery	1.4.4	1,981,777	Modify user accounts.
			Make existing albums world-writable.
Squirrelmail	1.4.6	1,905,277	Change user information.
			Send mails.

In our test environment, we added simple alias rules to the configuration of the Apache web server that match requests to protected applications and redirect these requests to the proxy wrapper functions. After an incoming request is processed, control is passed to the target application. To be able to process and modify the output generated by the target application before this output is returned to the client, we make use of the output buffering mechanism supplied by PHP. This way, all data generated by the target application is redirected into a buffer that can then be used for the required post-processing step. Note that PHP's output buffers are stackable, which means that this solution works even if the target application performs output buffering itself. During reply processing, the task of instrumenting the output with a token is taken over by a Java program based on the HTMLParser [2] package (a package for parsing and modifying HTML code).

During the implementation of the proxy wrapper routines, we encountered the following problem: The detection of an active session requires that the HTTP headers returned to the client are inspected. Unfortunately, PHP offers no direct mechanisms to access these headers. The output that is written into the proxy's output buffer only contains the message body. We solved this problem by equipping the proxy with additional wrappers around those PHP functions that are responsible for generating the interesting headers (such as `header()`, `setcookie()`, or `session_start()`). Since it is not possible to overwrite built-in PHP functions, we had to write a simple *sed* script that converts calls to these built-in functions into calls to our wrapper functions. For instance, a call to `header()` is automatically rewritten into a call to a wrapper function called `_xsrf_header()`. Note that this trivial modification to the target application is a one-time, fully automatic effort and highly reliable due to its simplicity. Alternatively, the proxy could also be implemented on the network level, where it has full access to the complete HTTP stream.

Another implementation issue we encountered was that if the target application halted program execution using the `exit()` or `die()` function, execution of our proxy stopped as well. As a result, no reply processing was performed. In this case, we successfully applied the same solution as before. By providing additional wrappers to the offending functions, we were able to catch such calls and initiate proper processing of the reply.

To summarize, here are the steps that are necessary for protecting a web application with our prototype implementation:

1. Add an appropriate alias to the Apache configuration.
2. Execute the `sed` script on the target application to enable the proxy's wrapper functions.
3. Specify the cookie names that the target application uses to store session IDs (typically, this defaults to "PHPSESSID").
4. Specify the page that the user shall be redirected to in case of an XSRF attack.

These steps typically take less that five minutes for each application. Clearly, this process is far less time-consuming than manually adapting the whole target application to prevent XSRF attacks.

6 Experimental Results

To test our implementation, we chose the current stable releases of the seven largest (in terms of all-time downloads) open-source PHP web applications from SourceForge [22]. The high number of downloads (see Table 1) indicates that these applications are popular and wide-spread. Despite the fact that these applications are popular and well-maintained, we quickly discovered XSRF vulnerabilities in five of the applications. The two remaining ones, Gallery 2.0.4 and XOOPS 2.0.13.2, appeared to be immune to XSRF attacks, although we did not conduct exhaustive tests or source code reviews in pursue of XSRF vulnerabilities.

For the five vulnerable applications, we constructed a number of XSRF exploits that modify important data by abusing the privileges of an authenticated user. For instance, we managed to post and delete messages from a forum in the name of the victim, send mail, or even change the user name and password of the current user, resulting in identity theft. A comprehensive list of the created exploits is provided in Table 1.

Our first test evaluated the proxy's ability to protect vulnerable applications. After verifying that our exploits were working properly, we installed the proxy and repeated the attacks. This time, every XSRF attempt was correctly detected.

Apart from protecting applications, a central requirement that the proxy has to fulfill is to not interfere with the normal application behavior (both regular behavior as well as behavior in case of errors). To test this property, we can observe and compare the application's behavior without the proxy's protection to the behavior with enabled XSRF protection. If the two results are identical, the proxy succeeded in performing its task transparently. Hence, we applied the following test procedure for each application:

1. Log into the application.
2. On the following page, if there is a request trigger (e.g., a hyperlink) that was not activated yet:
   (a) Activate the next unvisited request trigger. If the trigger is a form, test it with correct as well as with incorrect input.
   (b) Hit the browser's "back" button.
   (c) Continue with Step 2.
3. Log out.

Note that these tests also cover the correct behavior of the browser's "back" button, which is a widely used convenience feature that must not be broken by XSRF countermeasures. Also note that the use cases that were targets of our demonstration exploits have already been tested in the previous stage (i.e., while verifying that the XSRF protection works). In addition to this systematic procedure, we also chose some random work flows typical for the application in question. Altogether, we are confident that the coverage of our tests is large enough to give representative results. There was not a single case in which we observed deviant behavior caused by the presence of the proxy.

As far as performance is concerned, we observed no noticeable delay when interacting with the applications protected by our proxy. This is satisfying, as we implemented our prototype without performance in mind, and it still represents many opportunities for optimization. Also, by implementing the proxy and the rewriting engine in a language such as C or C++ instead of in PHP and Java, an additional boost in performance can be expected. Another alternative implementation would be to turn the proxy's application logic into a module for the Apache web server, or integrate it into the already available `mod_security` [11] module.

7 Related Work

The detection of web-based attacks has received considerable attention because of the increasingly critical role that web-based services are playing on the Internet. This includes web application firewalls [19] to protect applications from malicious requests as well as intrusion detection systems that attempt to identify attacks against web servers and their applications [1, 9]. Also, code analysis tools were proposed that check applications for the existence of bugs that can lead to security vulnerabilities [4, 7].

In particular, cross site scripting (XSS) attacks have received much interest, and both server-side and client-side solutions were proposed. For example, in [3], the use of a variety of software-testing techniques (including dynamic analysis, black-box testing, fault injection and behavior monitoring) are suggested to identify XSS vulnerabilities. Alternatively, dynamic techniques on the server side [16, 24] can be used to track non-validated user input while it is processed by the application. This can help to detect and mitigate XSS flaws. Finally, in previous work, we implemented a client-side solution [8] to protect users from XSS attempts. Unfortunately, these solutions cannot be applied to the problem of cross site request forgery, because XSRF attacks are not due to input validation problems.

The general class of cross site request forgery (XSRF) attacks was first introduced by Peter W. in a posting [23] to the BugTraq mailing list, and has since been picked up by web application developers [21]. However, it appears to be a little known problem in the academic community and, as a result, has only received little attention (which is one of the reasons why we believe that this paper might be interesting). The mitigation mechanisms for XSRF that were proposed so far (discussed in more detail in Section 3) either provide only partial protection (such as replacing GET requests by POST requests, or relying on the information in the Referer header of HTTP requests) or require significant modifications to each individual web application that should be protected (when embedding shared secrets into the application's output). Our solution, on the other hand, attempts to retain the advantage of a solution based on shared secrets while removing the need to modify application source code. That is, by using a web proxy, we can transparently embed secret tokens into the output of web applications.

In concurrent and independent work [10], an orthogonal proxy-based solution on the client side was presented. It also builds upon the token approach, and additionally proposes the use of an outside entity for detecting IP-based authentication. For cases in which JavaScript code initiates HTTP requests, this code is altered automatically to contain the token. Without evaluation, the reliability of this technique (which requires a certain extent of program understanding) is difficult to assess. Also, we believe that a manual treatment of these rare cases on the server side provides a more stable and efficient solution. Besides, due to the usual difficulties with client-side proxies, this implementation does not support SSL connections yet.

8 Conclusion

In a cross site request forgery (XSRF) attack, the trust of a web application in its authenticated users is exploited, allowing an attacker to make arbitrary HTTP requests in the victim's name. Unfortunately, current XSRF mitigation techniques have shortcomings that limit their general applicability. To address this problem, this paper presents a solution that provides a completely automatic protection from XSRF attacks. Our approach is based on a server-side proxy that detects and prevents XSRF attacks in a way that is transparent to users as well as to the web application itself.

We have successfully used our prototype to secure a number of popular open-source web applications that were vulnerable to XSRF. Our experimental results demonstrate that the solution is viable, and that we can secure existing web applications without adversely affecting their behavior.

Currently, XSRF attacks are relatively unknown to both web developers and attackers that are on the hunt for easy targets. However, we expect the attention paid to this class of attacks to soon reach that of more traditional web security problems (such as XSS or SQL injections), and we hope that our solution will prove useful in protecting vulnerable web applications.

Acknowledgments

This work was supported by the Austrian Science Foundation (FWF) under grants P18368 (Omnis) and P18764 (Web-Defense), and by the Secure Business Austria competence center.

References

[1] M. Almgren, H. Debar, and M. Dacier. A lightweight tool for detecting web server attacks. In *ISOC Symposium on Network and Distributed Systems Security (NDSS)*, 2000.

[2] HTMLParser. `http://htmlparser.sourceforge.net/`, 2006.

[3] Y.-W. Huang, S.-K. Huang, and T.-P. Lin. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *12th International World Wide Web Conference (WWW)*, 2003.

[4] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *13th International World Wide Web Conference*, 2004.

[5] Java Q & A - Session State in the Client Tier. `http://java.sun.com/blueprints/qanda/client_tier/session_state.html`, 2006.

[6] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing Cross Site Request Forgery Attacks. Technical report.

[7] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[8] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *21st ACM Symposium on Applied Computing (SAC)*, 2006.

[9] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *10th ACM Conference on Computer and Communication Security (CCS)*, 2003.

[10] Martin Johns and Justus Winter. RequestRodeo: Client Side Protection against Session Riding. `{OWASPAppSec2006Europe}`, 2006.

[11] ModSecurity. `http://www.modsecurity.org/`.

[12] Persistent Client State: HTTP Cookies. `http://wp.netscape.com/newsref/std/cookie\_spec.html`, 1999.

[13] PHP: Hypertext Preprocessor. `http://www.php.net`.

[14] PHP Manual. `http://www.php.net/manual/en`.

[15] PHP Session Security. `http://www.webkreator.com/php/configuration/php-session-security.html`, 2002.

[16] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[17] RFC 2616, Security Considerations. `http://www.w3.org/Protocols/rfc2616/rfc2616-sec15.html`, 1999.

[18] T. Schreiber. Session Riding: A Widespread Vulnerability in Today's Web Applications. `http://www.securenet.de/papers/Session\_Riding.pdf`, 2001.

[19] D. Scott and R. Sharp. Abstracting Application-Level Web Security. In *11th International World Wide Web Conference (WWW)*, 2002.

[20] C. Shiflett. Foiling Cross-Site Attacks. `http://www.securityfocus.com/archive/1/191390`, 2001.

[21] C. Shiflett. PHP Security. In *O'Reilly Open Source Convention*, 2004.

[22] SourceForge. `http://sourceforge.net/`, 2006.

[23] P. W. Cross-Site Request Forgeries. `http://www.securityfocus.com/archive/1/191390`, 2001.

[24] L. Wall, T. Christiansen, R. Schwartz, and S. Potter. *Programming Perl (2nd ed.)*. O'Reilly & Associates, Inc., 1996.