Cross-Site Request Forgery (CSRF)

From OWASP

> *This is an **Attack**. To view all attacks, please see the Attack Category page.*

Last revision (mm/dd/yy): **06/20/2017**

Overview

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

Related Security Activities

How to Review Code for CSRF Vulnerabilities

See the OWASP Code Review Guide article on how to review code for CSRF vulnerabilities.

How to Test for CSRF Vulnerabilities

See the OWASP Testing Guide article on how to test for CSRF vulnerabilities.

How to Prevent CSRF Vulnerabilities

See the CSRF Prevention Cheat Sheet for prevention measures.

Listen to the OWASP Top Ten CSRF Podcast (http://www.owasp.org/download/jmanico/owasp_podcast_69.mp3).

Most frameworks have built-in CSRF support such as Joomla (http://docs.joomla.org/How_to_add_CSRF_anti-spoofing_to_forms), Spring (http://blog.eyallupu.com/2012/04/csrf-defense-in-spring-mvc-31.html), Struts (http://web.securityinnovation.com/appsec-weekly/blog/bid/84318/Cross-Site-Request-Forgery-CSRF-Prevention-Using-Struts-2), Ruby on Rails (http://guides.rubyonrails.org/security.html#cross-site-request-forgery-csrf), .NET (http://www.troyhunt.com/2010/11/owasp-top-10-for-net-developers-part-5.html) and others.

Use OWASP CSRF Guard to add CSRF protection to your Java applications. You can use CSRFProtector Project to protect your php applications or any project deployed using Apache Server. There is a .Net CSRF Guard at OWASP as well, but it's old and doesn't look complete.

John Melton also has an excellent blog post (http://www.jtmelton.com/2010/05/16/the-owasp-top-ten-and-esapi-part-6-cross-site-request-forgery-csrf/) describing how to use the native anti-CSRF functionality of the OWASP ESAPI (http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API).

Description

CSRF is an attack that tricks the victim into submitting a malicious request. It inherits the identity and privileges of the victim to perform an undesired function on the victim's behalf. For most sites, browser requests automatically include any credentials associated with the site, such as the user's session cookie, IP address, Windows domain credentials, and so forth. Therefore, if the user is currently authenticated to the site, the site will have no way to distinguish between the forged request sent by the victim and a legitimate request sent by the victim.

CSRF attacks target functionality that causes a state change on the server, such as changing the victim's email address or password, or purchasing something. Forcing the victim to retrieve data doesn't benefit an attacker because the attacker doesn't receive the response, the victim does. As such, CSRF attacks target state-changing requests.

It's sometimes possible to store the CSRF attack on the vulnerable site itself. Such vulnerabilities are called "stored CSRF flaws". This can be accomplished by simply storing an IMG or IFRAME tag in a field that accepts HTML, or by a more complex cross-site scripting attack. If the attack can store a CSRF attack in the site, the severity of the attack is amplified. In particular, the likelihood is increased because the victim is more likely to view the page containing the attack than some random page on the Internet. The likelihood is also increased because the victim is sure to be authenticated to the site already.

Synonyms

CSRF attacks are also known by a number of other names, including XSRF, "Sea Surf", Session Riding, Cross-Site Reference Forgery, and Hostile Linking. Microsoft refers to this type of attack as a One-Click attack in their threat modeling process and many places in their online documentation.

Prevention measures that do NOT work

Using a secret cookie

Remember that all cookies, even the *secret* ones, will be submitted with every request. All authentication tokens will be submitted regardless of whether or not the end-user was tricked into submitting the request. Furthermore, session identifiers are simply used by the application container to associate the request with a specific session object. The session identifier does not verify that the end-user intended to submit the request.

Only accepting POST requests

Applications can be developed to only accept POST requests for the execution of business logic. The misconception is that since the attacker cannot construct a malicious link, a CSRF attack cannot be executed. Unfortunately, this logic is incorrect. There are numerous methods in which an attacker can trick a victim into submitting a forged POST request, such as a simple

form hosted in an attacker's Website with hidden values. This form can be triggered automatically by JavaScript or can be triggered by the victim who thinks the form will do something else.

A number of flawed ideas for defending against CSRF attacks have been developed over time. Here are a few that we recommend you avoid.

Multi-Step Transactions

Multi-Step transactions are not an adequate prevention of CSRF. As long as an attacker can predict or deduce each step of the completed transaction, then CSRF is possible.

URL Rewriting

This might be seen as a useful CSRF prevention technique as the attacker cannot guess the victim's session ID. However, the user's session ID is exposed in the URL. We don't recommend fixing one security flaw by introducing another.

HTTPS

HTTPS does nothing to defend against CSRF.

Examples

How does the attack work?

There are numerous ways in which an end user can be tricked into loading information from or submitting information to a web application. In order to execute an attack, we must first understand how to generate a valid malicious request for our victim to execute. Let us consider the following example: Alice wishes to transfer $100 to Bob using the *bank.com* web application that is vulnerable to CSRF. Maria, an attacker, wants to trick Alice into sending the money to her instead. The attack will comprise the following steps:

1. building an exploit URL or script
2. tricking Alice into executing the action with social engineering

GET scenario

If the application was designed to primarily use GET requests to transfer parameters and execute actions, the money transfer operation might be reduced to a request like:

```
GET http://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1
```

Maria now decides to exploit this web application vulnerability using Alice as her victim. Maria first constructs the following exploit URL which will transfer $100,000 from Alice's account to her account. She takes the original command URL and replaces the beneficiary name with herself, raising the transfer amount significantly at the same time:

```
http://bank.com/transfer.do?acct=MARIA&amount=100000
```

The social engineering aspect of the attack tricks Alice into loading this URL when she's logged into the bank application. This is usually done with one of the following techniques:

- sending an unsolicited email with HTML content
- planting an exploit URL or script on pages that are likely to be visited by the victim while they are also doing online banking

The exploit URL can be disguised as an ordinary link, encouraging the victim to click it:

```
<a href="http://bank.com/transfer.do?acct=MARIA&amount=100000">View my Pictures!</a>
```

Or as a 0x0 fake image:

```
<img src="http://bank.com/transfer.do?acct=MARIA&amount=100000" width="0" height="0" border="0">
```

If this image tag were included in the email, Alice wouldn't see anything. However, the browser *will still* submit the request to bank.com without any visual indication that the transfer has taken place.

A real life example of CSRF attack on an application using GET was a uTorrent exploit (https://www.ghacks.net/2008/01/17/dos-vulnerability-in-utorrent-and-bittorrent/) from 2008 that was used on a mass scale to download malware.

POST scenario

The only difference between GET and POST attacks is how the attack is being executed by the victim. Let's assume the bank now uses POST and the vulnerable request looks like this:

```
POST http://bank.com/transfer.do HTTP/1.1

acct=BOB&amount=100
```

Such a request cannot be delivered using standard A or IMG tags, but can be delivered using a FORM tag:

```
<form action="http://bank.com/transfer.do" method="POST">
<input type="hidden" name="acct" value="MARIA"/>
<input type="hidden" name="amount" value="100000"/>
<input type="submit" value="View my pictures"/>
</form>
```

This form will require the user to click on the submit button, but this can be also executed automatically using JavaScript:

```
<body onload="document.forms[0].submit()">
<form...
```

Other HTTP methods

Modern web application APIs frequently use other HTTP methods, such as PUT or DELETE. Let's assume the vulnerable bank uses PUT that takes a JSON block as an argument:

```
PUT http://bank.com/transfer.do HTTP/1.1

{ "acct":"BOB", "amount":100 }
```

Such requests can be executed with JavaScript embedded into an exploit page:

```
<script>
function put() {
        var x = new XMLHttpRequest();
        x.open("PUT","http://bank.com/transfer.do",true);
        x.setRequestHeader("Content-Type", "application/json");
        x.send(JSON.stringify({"acct":"BOB", "amount":100}));
}
</script>
<body onload="put()">
```

Fortunately, this request will **not** be executed by modern web browsers thanks to same-origin policy restrictions. This restriction is enabled by default unless the target web site explicitly opens up cross-origin requests from the attacker's (or everyone's) origin by using CORS with the following header:

```
Access-Control-Allow-Origin: *
```

Related Attacks

- Cross-site Scripting (XSS)
- Cross Site History Manipulation (XSHM)

Related Controls

- Add a per-request nonce to the URL and all forms in addition to the standard session. This is also referred to as "form keys". Many frameworks (e.g., Drupal.org 4.7.4+) either have or are starting to include this type of protection "built-in" to every form so the programmer does not need to code this protection manually.
- Add a hash (session id, function name, server-side secret) to all forms.
- For .NET, add a session identifier to ViewState with MAC (described in detail in the CSRF Prevention Cheat Sheet).
- Checking the referrer header in the client's HTTP request can prevent CSRF attacks. Ensuring that the HTTP request has come from the original site means that attacks from other sites will not function. It is very common to see referrer header checks used on embedded network hardware due to memory limitations.
    - XSS can be used to bypass both referrer and token based checks simultaneously. For instance, the Samy worm (http://en.wikipedia.org/wiki/Samy_%28computer_worm%29) used an XHR to obtain the CSRF token to forge requests.
- "Although CSRF is fundamentally a problem with the web application, not the user, users can help protect their accounts at poorly designed sites by logging off the site before visiting another, or clearing their browser's cookies at the end of each browser session." --http://en.wikipedia.org/wiki/Cross-site_request_forgery#_note-1

- Tokenizing

References

- OWASP Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet
- The Cross-Site Request Forgery (CSRF/XSRF) FAQ (http://www.cgisecurity.com/articles/csrf-faq.shtml)

  quote: "This paper serves as a living document for Cross-Site Request Forgery issues. This document will serve as a repository of information from existing papers, talks, and mailing list postings and will be updated as new information is discovered."
- Testing for CSRF

  CSRF (aka Session riding) paper from the OWASP Testing Guide project.
- CSRF Vulnerability: A 'Sleeping Giant' (http://www.darkreading.com/document.asp?doc_id=107651&WT.svl=news1_2)

  Overview Paper
- Client Side Protection against Session Riding (http://www.owasp.org/index.php/Image:RequestRodeo-MartinJohns.pdf)

  Martin Johns and Justus Winter's interesting paper and presentation for the 4th OWASP AppSec Conference which described potential techniques that browsers could adopt to automatically provide CSRF protection – PDF paper (http://www.owasp.org/index.php/Image:RequestRodeo-MartinJohns.pdf)
- OWASP CSRF Guard

  J2EE, .NET, and PHP Filters which append a unique request token to each form and link in the HTML response in order to provide universal coverage against CSRF throughout your entire application.
- OWASP CSRF Protector

  Anti CSRF method to mitigate CSRF in web applications. Currently implemented as a PHP library & Apache 2.x.x module
- A Most-Neglected Fact About Cross Site Request Forgery (CSRF) (http://yehg.net/lab/pr0js/view.php/A_Most-Neglected_Fact_About_CSRF.pdf)

  Aung Khant, http://yehg.net, explained the danger and impact of CSRF with imperiling scenarios.
- OWASP CSRF Tester

  The OWASP CSRFTester gives developers the ability to test their applications for CSRF flaws.

- Pinata-CSRF-Tool: CSRF POC tool (http://code.google.com/p/pinata-csrf-tool/)

  Pinata makes it easy to create Proof of Concept CSRF pages. Assists in Application Vulnerability Assessment.