Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet

From OWASP

Contents

Introduction

Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious Web site, email, blog, instant message, or program causes a user’s Web browser to perform an unwanted action on a trusted site for which the user is currently authenticated. The impact of a successful cross-site request forgery attack is limited to the capabilities exposed by the vulnerable application. For example, this attack could result in a transfer of funds, changing a password, or purchasing an item in the user's context. In affect, CSRF attacks are used by an attacker to make a target system perform a function (funds Transfer, form submission etc.) via the target's browser without knowledge of the target user, at least until the unauthorized function has been committed.

Impacts of successful CSRF exploits vary greatly based on the role of the victim. When targeting a normal user, a successful CSRF attack can compromise end-user data and their associated functions. If the targeted end user is an administrator account, a CSRF attack can compromise the entire Web application. The sites that are more likely to be attacked are community Websites (social networking, email) or sites that have high dollar value accounts associated with them (banks, stock brokerages, bill pay services). This attack can happen even if the user is logged into a Web site using strong encryption (HTTPS). Utilizing social engineering, an attacker will embed malicious HTML or JavaScript code into an email or Website to request a specific 'task url'. The task then executes with or without the user's knowledge, either directly or by utilizing a Cross-site Scripting flaw (ex: Samy MySpace Worm).

For more information on CSRF, please see the OWASP Cross-Site Request Forgery (CSRF) page.

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.

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 can not guess the victim's session ID. However, the user’s credential is exposed over the URL.

General Recommendation: Synchronizer Token Pattern

In order to facilitate a "transparent but visible" CSRF solution, developers are encouraged to adopt the Synchronizer Token Pattern (http://www.corej2eepatterns.com/Design/PresoDesign.htm). The synchronizer token pattern requires the generating of random "challenge" tokens that are associated with the user's current session. These challenge tokens are the inserted within the HTML forms and links associated with sensitive server-side operations. When the user wishes to invoke these sensitive operations, the HTTP request should include this challenge token. It is then the responsibility of the server application to verify the existence and correctness of this token. By including a challenge token with each request, the developer has a strong control to verify that the user actually intended to submit the desired requests. Inclusion of a required security token in HTTP requests associated with sensitive business functions helps mitigate CSRF attacks as successful exploitation assumes the attacker knows the randomly generated token for the target victim's session. This is analogous to the attacker being able to guess the target victim's session identifier. The following synopsis describes a general approach to incorporate challenge tokens within the request.

When a Web application formulates a request (by generating a link or form that causes a request when submitted or clicked by the user), the application should include a hidden input parameter with a common name such as "CSRFToken". The value of this token must be randomly generated such that it cannot be guessed by an attacker. Consider leveraging the java.security.SecureRandom class for Java applications to generate a sufficiently long random token. Alternative generation algorithms include the use of 256-bit BASE64 encoded hashes. Developers that choose this generation algorithm must make sure that there is randomness and uniqueness utilized in the data that is hashed to generate the random token.

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken" value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWEwYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZjMTVi
MGYwMGEwOA==">
…
</form>
```

In general, developers need only generate this token once for the current session. After initial generation of this token, the value is stored in the session and is utilized for each subsequent request until the session expires. When a request is issued by the end-user, the server-side component must verify the existence and validity of the token in the request as compared to the token found in the session. If the token was not found within the request or the value provided does not match the value within the session, then the request should be aborted and the event logged as a potential CSRF attack in progress.

To further enhance the security of this proposed design, consider randomizing the CSRF token parameter name and or value for each request. Implementing this approach results in the generation of per-request tokens as opposed to per-session tokens. Note, however, that this may result in usability concerns. For example, the "Back" button browser capability is often hindered as the previous page may contain a token that is no longer valid. Interaction with this previous page will result in a CSRF false positive security event at the server. Regardless of the approach taken, developers are encouraged to protect the CSRF token the same way they protect authenticated session identifiers, such as the use of SSLv3/TLS.

Disclosure of Token in URL

Many implementations of this control include the challenge token in GET (URL) requests as well as POST requests. This often implemented as a result of sensitive server-side operations being invoked as a result of embedded links in the page or other general design patterns. These patterns are often implemented without knowledge of CSRF and an understanding of CSRF prevention design strategies. While this control does help mitigate the risk of CSRF attacks, the unique per-session token is being exposed for GET requests. CSRF tokens in GET requests are potentially leaked at several locations: browser history, HTTP log files, network appliances that make a point to log the first line of an HTTP request, and Referrer headers if the protected site links to an external site. The ideal solution is to only include the CSRF token in POST requests and modify server-side actions that have state changing affect to only respond to POST requests. If sensitive server-side actions are guaranteed to only ever respond to POST requests, then there is no need to include the token in GET requests. In most JavaEE web applications, however, HTTP method scoping is rarely ever utilized when retrieving HTTP parameters from a request. Calls to "HttpServletRequest.getParameter" will return a parameter value regardless if it was a GET or POST. This is not to say HTTP method scoping cannot be enforced. It can be achieved if a developer explicitly overrides doPost() in the HttpServlet class or leverages framework specific capabilities such as the AbstractFormController class in Spring. While this is the ideal approach, attempting to retrofit this pattern in existing applications requires significant development time and cost.

In order for an attacker to successfully carry out a valid CSRF attack against an application that has similar characteristics as described above (i.e. CSRF tokens on GET and POST requests, no explicitly enforcing HTTP method scoping), several actions are typically required:

1. The victim must be interacting with the application such that a valid CSRF token is generated.
2. The CSRF token must be transmitted in a GET request by interacting with the web page
3. A component with insight into (at least) the first line of the HTTP GET request must process the request at some point between (and including) the browser and the server
4. This same component must capture the token in a repository that is accessible to an attacker (ex: server log files, browser history, etc.)
5. The attacker must parse this repository and retrieve the token.

6. The attacker needs to somehow know what user owns this exposed CSRF token
7. The attacker needs to trick the target victim (i.e. the one owning the CSRF token) into interacting with an HTML document.
8. The victim's session that holds the exposed token must still be valid (i.e. not invalidated, expired, idle/absolute timed-out, etc.).

While placing CSRF tokens in GET requests does present a potential risk as described above, it is very minimal. The likelihood of successfully discovering and exploiting this vulnerability is significantly low. Coupled with strong session management practices (i.e. timeouts), frequent use of SSLv3/TLS, and network based services that do not log this mechanism, there is little risk to the CSRF token being exposed to an attacker. Even if the CSRF token is exposed and the attacker is somehow able to figure how the associated user, the token is only valid for the lifetime of the session. Ultimately, the acceptance of this risk as opposed to the cost of significant architecture design is up to the business.

Viewstate (ASP.NET)

ASP.NET has an option to maintain your ViewState. The ViewState indicates the status of a page when submitted to the server. The status is defined through a hidden field placed on each page with a <form runat="server"> control. Viewstate can be used as a CSRF defense, as it is difficult for an attacker to forge a valid Viewstate. It is not impossible to forge a valid Viewstate since it is feasible that parameter values could be obtained or guessed by the attacker. However, if the current session ID is added to the ViewState, it then makes each Viewstate unique, and thus immune to CSRF.

To use the ViewStateUserKey property within the Viewstate to protect against spoofed post backs. Add the following in the OnInit virtual method of the Page-derived class (This property must be set in the Page.Init event)

```
protected override OnInit(EventArgs e) {
   base.OnInit(e);
   if (User.Identity.IsAuthenticated)
      ViewStateUserKey = Session.SessionID; }
```

The following keys the Viewstate to an individual using a unique value of your choice.

```
 (Page.ViewStateUserKey)
```

This must be applied in Page_Init because the key has to be provided to ASP.NET before Viewstate is loaded. This option has been available since ASP.NET 1.1.

However, there are limitations (http://keepitlocked.net/archive/2008/05/29/viewstateuserkey-doesn-t-prevent-cross-site-request-forgery.aspx) on this mechanism. Such as, ViewState MACs are only checked on POSTback, so any other application requests not using postbacks will happily allow CSRF.

Double Submit Cookies

Double submitting cookies is defined as sending the session ID cookie in two different ways for every form request. First as a traditional header value, and again as a hidden form value. When a user visits a site, the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine. This is typically referred to as the session ID. The site should require every form submission to include this pseudorandom value as a hidden form value and also as a cookie value. When a POST request is sent to the site, the request should only be considered valid if the form value and the cookie value are the same. When an attacker submits a form on behalf of a user, he can only modify the values of the form. An attacker cannot read any data sent from the server or modify cookie values, per the same-origin policy. This means that while an attacker can send any value he wants with the form, the attacker will be unable to modify or read the value stored in the cookie. Since the cookie value and the form value must be the same, the attacker will be unable to successfully submit a form unless he is able to guess the session ID value.

While this approach is effective in mitigating the risk of cross-site request forgery, including authenticated session identifiers in HTTP parameters may increase the overall risk of session hijacking. Architects and developers must ensure that no network appliances or custom application code or modules explicitly log or otherwise disclose HTTP POST parameters. An attacker that is able to obtain access to repositories or channels that leak HTTP POST parameters will be able to replay the tokens and perform session hijacking attacks. Note, however, that transparently logging all HTTP POST parameters is a rare occurrence across network systems and web applications as doing so will expose significant sensitive data aside from session identifiers including passwords, credit card numbers, and or social security numbers. Inclusion of the session identifier within HTML can also be leveraged by cross-site scripting attacks to bypass HTTPOnly protections. Most modern browsers prevent client-side script from accessing HTTPOnly cookies. However, this protection is lost if HTTPOnly session identifiers are placed within HTML as client-side script can easily traverse and extract the identifier from the DOM. Developers are still encouraged to implement the synchronizer token pattern as described in this article.

Direct Web Remoting (DWR) (http://directwebremoting.org) Java library version 2.0 has CSRF protection built in as it implements the double cookie submission transparently.

Prevention Frameworks

There are CSRF prevention modules available for J2EE, .Net, and PHP.

OWASP CSRF Guard (For Java)

The OWASP CSRFGuard Project makes use of a unique per-session token verification pattern using a JavaEE filter to mitigate the risk of CSRF attacks. When an HttpSession is first instantiated, CSRFGuard will generate a cryptographically random token using the SecureRandom class to be stored in the session.

Similar Projects

CSRFGuard has been implemented in other languages besides Java. They are:

- PHP CSRF Guard
- .Net CSRF Guard

Challenge-Response

Challenge-Response is another defense option for CSRF. The following are some examples of challenge-response options.

- CAPTCHA
- Re-Authentication (password)
- One-time Token

While challenge-response is a very strong defense to CSRF (assuming proper implementation), it does impact user experience. For applications in need of high security, tokens (transparent) and challenge-response should be used on high risk functions.

Checking Referer Header

The Checking of the http Referer can be used to prevent CSRF. This method is desirable for securing embedded network hardware such as modems, routers, and printers because it does not increase memory requirements. The major limitation is that browsers will omit the Referer in the http header when they are being used over HTTPS. This restriction does not apply for embedded network hardware because HTTPS is not used. There are two known methods of bypass a Referer check. A CRLF vulnerability (http://secunia.com/advisories/22467/) in the http client could allow an attacker manipulate the HTTP request header and thus spoof the referer. The other method is using XSS to write a form to a page on the target domain and then automatically submit the request.

Client/User Prevention

Since CSRF vulnerabilities are reportedly widespread, it is recommended to follow best practices to mitigate risk. Some mitigating include:

- Logoff immediately after using a Web application
- Do not allow your browser to save username/passwords, and do not allow sites to “remember” your login
- Do not use the same browser to access sensitive applications and to surf the Internet freely (tabbed browsing).
- The use of plugins such as No-Script makes POST based CSRF vulnerabilities difficult to exploit. This is because JavaScript is used to automatically submit the form when the exploit is loaded. Without JavaScript the attacker would have to trick the user into submitting the form manually.

Integrated HTML-enabled mail/browser and newsreader/browser environments pose additional risks since simply viewing a mail message or a news message might lead to the execution of an attack.

No Cross-Site Scripting (XSS) Vulnerabilities

Cross-Site Scripting is not necessary for CSRF to work. However, all stored cross-site scripting attacks and special case reflected cross-site scripting attacks can be used to defeat token based CSRF defenses, since a malicious XSS script can simply read the site generated token from the response, and include that token with a forged request. This technique is exactly how the MySpace (Samy) worm (http://en.wikipedia.org/wiki/Samy_(XSS)) defeated MySpace's anti CSRF defenses in 2005, which enabled the worm to propagate. XSS cannot defeat challenge-response defenses such as Captcha, re-authentication or one-time passwords. It is imperative that no XSS vulnerabilities are present to ensure that CSRF defenses can't be circumvented. Please see the OWASP XSS Prevention Cheat Sheet for detailed guidance on how to prevent XSS flaws.

Related Articles

Cross-Site Request Forgery (CSRF)

For more information on CSRF please see the OWASP Cross-Site Request Forgery (CSRF) page.

How to Review Code for CSRF Vulnerabilities

See the OWASP Code Review Guide article on how to Reviewing code for Cross-Site Request Forgery issues.

How to Test for CSRF Vulnerabilities

See the OWASP Testing Guide article on how to Test for CSRF Vulnerabilities.

CSRF Testing Tool

Check out the OWASP CSRF Tester tool which allows you to test for CSRF vulnerabilities. This tool is also written in Java.

Other Articles in the OWASP Prevention Cheat Sheet Series

- Authentication Cheat Sheet
- **Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet**
- Forgot Password Cheat Sheet
- Cryptographic Storage Cheat Sheet
- SQL Injection Prevention Cheat Sheet
- Transport Layer Protection Cheat Sheet
- XSS (Cross Site Scripting) Prevention Cheat Sheet
- DOM based XSS Prevention Cheat Sheet

References

Cross Site Reference Forgery: An introduction to a common web application weakness (http://www.isecpartners.com/files/XSRF_Paper_0.pdf)

Cross-Site Request Forgeries: Exploitation and Prevention (http://www.freedom-to-tinker.com/sites/default/files/csrf.pdf)

Authors and Primary Editors

Paul Petefish - paulpetefish[at]solutionary.com

Eric Sheridan - eric.sheridan[at]aspectsecurity.com

Dave Wichers - dave.wichers[at]aspectsecurity.com