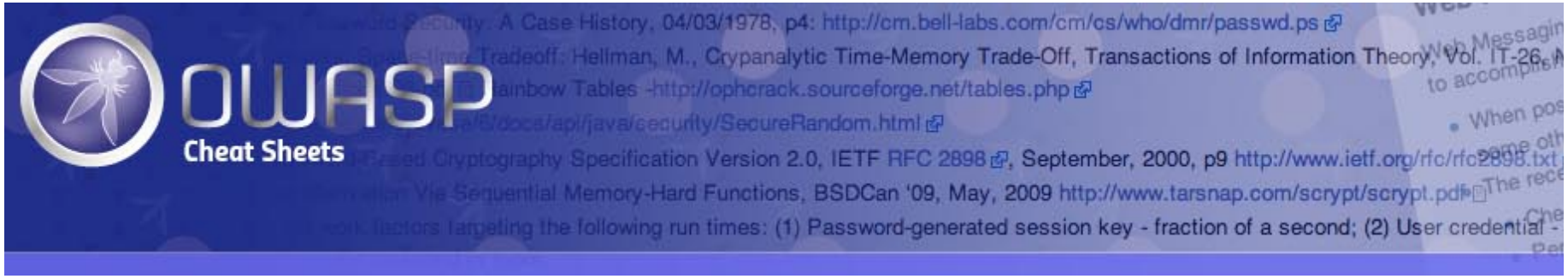
Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet

From OWASP



Last revision (mm/dd/yy): **08/14/2014**

Introduction

Other Cheatsheets

Developer Cheat Sheets (Builder)

- Authentication Cheat Sheet
- Choosing and Using Security Questions Cheat Sheet
- Clickjacking Defense Cheat Sheet
- C-Based Toolchain Hardening Cheat Sheet
- **Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet**
- Cryptographic Storage Cheat

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 effect, 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

Sheet
- DOM based XSS Prevention Cheat Sheet
- Forgot Password Cheat Sheet
- HTML5 Security Cheat Sheet
- Input Validation Cheat Sheet
- JAAS Cheat Sheet
- Logging Cheat Sheet
- .NET Security Cheat Sheet
- Password Storage Cheat Sheet
- Pinning Cheat Sheet
- Query Parameterization Cheat Sheet
- Ruby on Rails Cheatsheet
- REST Security Cheat Sheet
- Session Management Cheat Sheet
- SQL Injection Prevention Cheat Sheet
- Transport Layer Protection Cheat Sheet
- Unvalidated Redirects and Forwards Cheat Sheet
- User Privacy Protection Cheat Sheet
- Web Service Security Cheat Sheet
- XSS (Cross Site Scripting) Prevention Cheat Sheet

Assessment Cheat Sheets (Breaker)

- Attack Surface Analysis Cheat

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

Sheet
- XSS Filter Evasion Cheat Sheet
- REST Assessment Cheat Sheet

Mobile Cheat Sheets

- IOS Developer Cheat Sheet
- Mobile Jailbreaking Cheat Sheet

OpSec Cheat Sheets (Defender)

- Virtual Patching Cheat Sheet

Draft Cheat Sheets

- OWASP Top Ten Cheat Sheet
- Access Control Cheat Sheet
- Application Security Architecture Cheat Sheet
- Business Logic Security Cheat Sheet
- PHP Security Cheat Sheet
- Secure Coding Cheat Sheet
- Secure SDLC Cheat Sheet
- Threat Modeling Cheat Sheet
- Web Application Security Testing Cheat Sheet
- Grails Secure Code Review Cheat Sheet
- IOS Application Security Testing Cheat Sheet
- Key Management Cheat Sheet
- Insecure Direct Object Reference Prevention Cheat Sheet

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

- Content Security Policy Cheat Sheet

the user's current session. These challenge tokens are then 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="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWE...
```

```
wYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZ...
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, token should be reset 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 is 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 Referer headers if the protected site links to an external site.

In the latter case (leaked CSRF token due to the Referer header being parsed by a linked site), it is trivially easy for the linked site to launch a CSRF attack on the protected site, and they will be able to target this attack very effectively, since the Referer header tells them the site as well as the CSRF token. The attack could be run entirely from javascript, so that a simple addition of a script tag to the HTML of a site can launch an attack (whether on an originally malicious site or on a hacked site). This attack scenario is easy to prevent, the referer will be omitted if the origin of the request is HTTPS. Therefore this attack does not affect web applications that are HTTPS only.

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. This is in fact what the RFC 2616 (http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.1.1) requires for GET 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.

For these cases, attempting to retrofit this pattern in existing applications requires significant development time and cost, and as a temporary measure it may be better to pass CSRF tokens in the URL. Once the application has been fixed to respond to HTTP GET and POST verbs correctly, CSRF tokens for GET requests should be turned off.

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 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 a random value in both a cookie and as a request parameter, with the server verifying if the cookie value and request value are equal.

When a user authenticates to a site, the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine separate from the session id. The site does not have to save this value in any way. The site should then require every sensitive submission to include this random value as a hidden form value (or other request parameter) and also as a cookie value. 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 a malicious CSRF request, the attacker will be unable to modify or read the value stored in the cookie. Since the cookie value and the request parameter or form value must be the same, the attacker will be unable to successfully submit a form unless he is able to guess the random CSRF value.

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.

Encrypted Token Pattern

Overview

The Encrypted Token Pattern leverages an encryption, rather than comparison, method of Token-validation. After successful authentication, the server generates a unique Token comprised of the user's ID, a timestamp value and a nonce (http://en.wikipedia.org/wiki/Cryptographic_nonce), using a unique key available only on the server. This Token is returned to the client and embedded in a hidden field. Subsequent AJAX requests include this Token in the request-header, in a similar manner to the Double-Submit pattern. Non-AJAX form-based requests will implicitly persist the Token in its hidden field, although I recommend persisting this data in a custom HTTP header in such cases. On receipt of this request, the server reads and decrypts the Token value with the same key used to create the Token.

Validation

On successful Token-decryption, the server has access to parsed

values, ideally in the form of claims (http://en.wikipedia.org/wiki/Claims-based_identity). These claims are processed by comparing the UserId claim to any potentially stored UserId (in a Cookie or Session variable, if the site already contains a means of authentication). The Timestamp is validated against the current time, preventing replay attacks. Alternatively, in the case of a CSRF attack, the server will be unable to decrypt the poisoned Token, and can block and log the attack.

This pattern exists primarily to allow developers and architects protect against CSRF without session-dependency. It also addresses some of the shortfalls in other stateless approaches, such as the need to store data in a Cookie, circumnavigating the Cookie-subdomain and HTTPONLY issues.

CSRF Prevention without a Synchronizer Token

CSRF can be prevented in a number of ways. Using a Synchronizer Token is one way that an application can rely upon the Same-Origin Policy to prevent CSRF by maintaining a secret token to authenticate requests. This section details other ways that an application can prevent CSRF by relying upon similar rules that CSRF exploits can never break.

Checking The Referer Header

Although it is trivial to spoof the referer header on your own browser, it is impossible to do so in a CSRF attack. Checking the referer is a commonly used method of preventing CSRF on embedded network devices because it does not require a

per-user state. This makes a referer a useful method of CSRF prevention when memory is scarce. This method of CSRF mitigation is also commonly used with unauthenticated requests, such as requests made prior to establishing a session state which is required to keep track of a synchronization token.

However, checking the referer is considered to be a weaker from of CSRF protection. For example, open redirect vulnerabilities can be used to exploit GET-based requests that are protected with a referer check and some organizations or browser tools remove referrer headers as a form of data protection. There are also common implementation mistakes with referer checks. For example if the CSRF attack originates from an HTTPS domain then the referer will be omitted. In this case the lack of a referer should be considered to be an attack when the request is performing a state change. Also note that the attacker has limited influence over the referer. For example, if the victim's domain is "site.com" then an attacker have the CSRF exploit originate from "site.com.attacker.com" which may fool a broken referer check implementation. XSS can be used to bypass a referer check.

In short, referer checking is a reasonable form of CSRF intrusion detection and prevention even though it is not a complete protection. Referer checking can detect some attacks but not stop all attacks. For example, if you HTTP referrer is from a different domain and you are expecting requests from your domain only, you can safely block that request.

Checking The Origin Header

The Origin HTTP Header (https://wiki.mozilla.org/Security/Origin) standard was introduced as a method of defending against CSRF and other Cross-Domain attacks. Unlike the referer, the origin will

be present in HTTP request that originates from an HTTPS url.

If the origin header is present, then it should be checked for consistency.

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.

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, any cross-site scripting vulnerability can be used to defeat token, Double-Submit cookie, referer and origin based CSRF defenses. This is because an XSS payload can simply read any page on the site using a XMLHttpRequest and obtain the 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.

Authors and Primary Editors

Paul Petefish - paulpetefish[at]solutionary.com
Eric Sheridan - eric.sheridan[at]owasp.org
Dave Wichers - dave.wichers[at]owasp.org

Categories: Cheatsheets | Popular
