Cross Site Reference Forgery

An introduction to a common web application weakness

Jesse Burns
©2005, Information Security Partners, LLC.
http://www.isecpartners.com

Version 1.1

Introduction

Cross Site Reference Forgery (XSRF) is a class of attack that affects web based applications with a predictable structure for invocation[1]. The attack's name is abbreviated differently but the most common form is XSRF, the acronym CSRF is also in common use. XSRF attacks are also known as "Hostile Linking" attacks, and have in some form been known about and exploited since before the turn of the millennium. The CSRF name was given to them by Peter Watkins (peterw@usa.net) in a June 2001 posting to the Bugtraq mailing list. Applications that are susceptible to XSRF attacks are said to have an XSRF flaw or to be susceptible to XSRF.

XSRF flaws exist in web applications with a predictable action structure and which use cookies, browser authentication or client side certificates to authenticate users. The basic idea of XSRF is simple; an attacker tricks the user into performing an action of the attackers choosing by directing the victim's actions on the target application with a link or other content. This is easiest to understand in the example of a HTTP GET.

For example the link http://www.google.com/search?q=iSEC+Partners causes anyone who clicks it to search Google for "iSEC Partners", this is both harmless, and by design. But a link like this one http://www.isecpartners.com/EditProfile?action=set&key=emailAddress&value=evil@isecpartners.com could tell an application which authenticated users only by cookie, browser authentication or certificate to edit a user's profile, and change their email address.

Links can be easily obfuscated so they appear to go elsewhere, and to conceal words that would disclose their actual function. XSRF attacks effect applications that use either HTTP GET or HTTP POST to call their actions, although actions invoked with HTTP GET are often easier to exploit.

An example of XSRF exploitation — Goat Chat

Goat Chat is a hypothetical web application that offers messaging between users. Upon login Goat Chat sets a large, unpredictable session ID cookie which is used to authenticate further requests by users without them including the users name or password. One of the features of Goat Chat is that users can send each other links inside their text messages. Goat Chat is hypothetically deployed at: https://goatchat.isecpartners.com/ and uses HTTPS to keep messages, credentials, and session identifiers secret from network eavesdroppers. The application has effective cross site scripting filters, blocking HTML content of any type.

Goat Chat supports the following user actions:

- Login
- Send a message
- Check for new messages
- Logout

An "incoming messages" frame is displayed to users who have logged in. This frame uses Javascript or a refresh tag to execute the "Check for new messages" action every 10 seconds on behalf of the user. New messages appear in this frame, and include the name of the sender and the text of the message. Text that is formatted as an HTTP or HTTPS url is automatically converted into a link. The "Send a message" action

[1] Invocation means the calling of an action. An example action might be changing a setting, posting a search, or logging out of an application.

takes two parameters, destination (a user name or "all"), and the message itself which is a short string. The "Logout" action requires no parameters.

To determine if this application is susceptible to XSRF, we need to look at the "Send a message" action. When we do this we find the following simple HTML form is submitted to send messages:

```
<form action="GoatChatMessageSender" method="GET">
Send To:<br>
<INPUT type="radio" name="Destination" value="Bob">Bob<BR>
<INPUT type="radio" name="Destination" value="Alice">Alice<BR>
<INPUT type="radio" name="Destination" value="Malory">Malory<BR>
<INPUT type="radio" name="Destination" value="All">All<BR>
Message: <input type="text" name="message" value="" \>
<br><input type="submit" name="Send" value="Send Message" \>
</form>
```

Figure 1 Form for sending a message

Here is what it looks like in its frame:

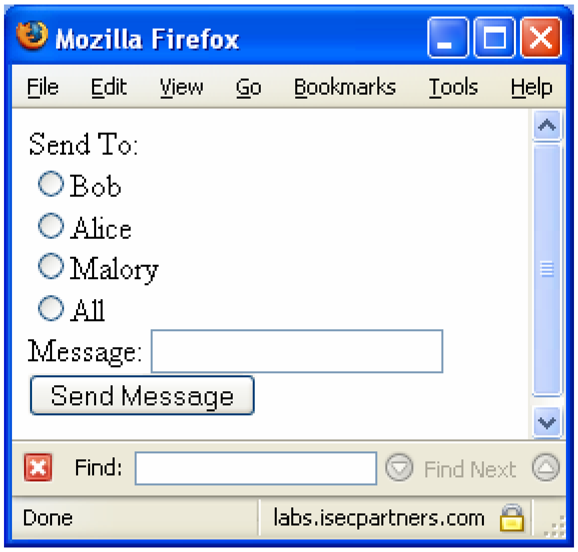

Figure 2 Rendering of Figure 1

From looking at this form we can figure out that when users wish to send the message "Hi Alice" to Alice, the following URL will be fetched when the user clicks Send Message.

```
https://goatchat.isecpartners.com/GoatChatMessageSender?Destination=Alice&messa
ge=Hi+Alice&Send=Send+Message
```

URL 1 An URL that sends "Hi Alice" to Alice when fetched by a logged in user

When the user performs an HTTP GET for URL 1, their browser includes the cookies appropriate for the goatchat.isecpartners.com site. These cookies are sent if the URL is typed in manually, if it is followed as part of loading a frame, clicking a link, due to an image request, or by submitting a form, even if it is loaded as the result of a 302 redirect or a meta-refresh tag. The only requirement for the cookie to be sent is that the logged in user is the one making a request. Attackers can exploit this.

If the logged in user were to visit a third party site run by devious hackers, and that site had the following image tag in it, URL 1 would be fetched by the victims browser and executed by the application server, as if the user submitted a message to Alice saying "Hi Alice".

```
<img src =
"https://goatchat.isecpartners.com/GoatChatMessageSender?Destination=Alice&mess
age=Hi+Alice&Send=Send+Message" \>
```

Figure 3 HTML which causes users to visit URL 1

An attacker could alternatively send the user a link in an email, or via IM. The attacker could obfuscate it with a link to a site like http://tinurl.com/ which would then redirect the victim to the target site. Many sites have redirection pages that can be used to redirect victims to any other page, the parameters are often subject to obfuscation via URL encoding or other tricks.

If this example had used the method POST rather than GET, then exploitation by link would be done differently. The attacker would send the victim a link that directed the victim to an attacker controlled site, the site would contain (possibly within an iframe) a form pointing to the target site, but with hidden fields. The form could be submitted automatically with ecmascript, or when the user clicked on it. More sophisticated variations allow for multi-stage forms.

The Figure 4 example is of the exploitation of a system which allows password changes without the entering of a user's old password. In this example the target servlet requires an HTTP post, and the attacker creates a self submitting form to fulfill this requirement. Note that this exploit is not as reliable as image based request in Figure 3 because the user's browser (or at least the tiny frame this exploit is placed in) is actually directed to the targeted site. Users with disabled browser scripting won't be exploited, and depending on the user's browser and configuration form submissions to other sites may result in a security popup box.

```
<HTML><BODY>
<form method="POST" id="evil" name="evil"
action="https://www.isecpartners.com/VictimApp/PasswordChange">
<input type=hidden name="newpass" value="badguy">
</form>
<script>document.evil.submit()</script>
</BODY></HTML>
```

Figure 4 Hidden form based exploit

Even if scripting is disabled however a "close this window" link that is actually a submit button may trick a user into submitting the form on the attackers behalf.

Reflected vs. Stored XSRF

Similarly to Cross Site Scripting (XSS) vulnerabilities, XSRF vulnerabilities can be divided into two major categories, Stored and Reflected.

A stored XSRF vulnerability is one where the attacker can use the application itself to provide the victim the exploit link or other content which directs the victim's browser back into the application, and causes attacker controlled actions to be executed as the victim. Stored XSRF vulnerabilities are more likely to succeed as the user who receives the exploit content is almost certainly currently authenticated to perform actions. Stored XSRF vulnerabilities also have a more obvious trail, which may lead back to the attacker.

In a reflected XSRF vulnerability the attacker uses a system outside the application to expose the victim to the exploit link or content. This can be done using a blog, an email message, an instant message, a message board posting, or even a flyer posted in a public place with an URL that a victim types in. Reflected XSRF attacks will frequently fail, as users may not be currently logged into the target system when the exploits are tried. The trail from a reflected XSRF attack may be under the control of the attacker, however and could be deleted once the exploit was completed.

Easy exploitation scenarios

Some applications are easier to attack with XSRF than others. Certainly applications with stored XSRF vulnerabilities are reliably exploitable, but other decisions developers make affect exposure.

Many applications direct HTTP GET calls to the same handler that is used for HTTP POST. In servlet's for example the doGet() method simply calls the doPost() method, redirecting the parameters. This makes simpler image or link based exploits possible, and eases the exploitation of XSRF flaws.

Some applications, particularly intranet sites and administrative consoles in switches, access points, bridges, and other network devices use integrated browser authentication. This type of authentication doesn't expire, and the credentials remain available until the browser is closed. For someone who works with their browser all day, this can be hours or even days. This is a very wide window for attack, and this design decision increases the effectiveness of reflected XSRF attacks.

Many applications have extremely cookie lives, allowing users to return to the site without re-authenticating. Long session lives can expose users to the risk of XSRF hours, or even days after using a site, popular sites with this configuration are of course even worse as attackers can guess that a large number of people selected at random are users of those services.

A few applications allow users to change their passwords without entering their old password, if these password change mechanisms are vulnerable to XSRF, then attackers may target this feature.

Distinguishing between XSRF and XSS

XSRF and XSS, especially reflected XSS are related security risks, and the similarity can be confusing. Many people find themselves trying to determine if an attack they have uncovered is exploiting an XSS weakness or an XSRF weakness. The distinction often easier to make by considering the solution to the weakness you have identified. In the case of an XSS flaw, an attacker exploits a lack of input and / or output filtering. If a change to the application that filters out dangerous characters like <, >, ", ', &, ;, or # could resolve the flaw, then it is not an XSRF issue but a XSS issue. XSRF is about the predictability of the structure of the application. XSS is related to the application performing insufficient data validation.

XSS flaws may allow bypassing of any of XSRF protections by leaking valid values of the tokens, allowing referrer's to appear to be the application itself, or by hosting hostile HTML elements right in the target application. Therefore resolving XSS flaws should be given priority over XSRF weaknesses.

Myths about XSRF

Myth: XSRF is just a special case of XSS.
Fact: XSRF is a separate vulnerability from XSS, with a different solution. XSS protections won't stop XSRF attacks, although XSS are important to solve and should be prioritized.

Myth: Applications aren't vulnerable to XSRF if they use multi-page forms to perform actions
Fact: While multi-page forms certainly make exploitation harder, attackers can usually exploit them. iSEC frequently uses multiple iframes when demonstrating multi-page form XSRF exploits for customers.

Myth: Applications that use HTTP POST aren't vulnerable to XSRF
Fact: While POSTs can be more difficult to exploit, they certainly are exploitable. Form's can mislead users about what they are sending and to where, and scripting can lead to automatic submission.

Myth: XSRF attacks are the user's fault
Fact: Users are usually not at fault for XSRF exploits, applications have to be designed secure in the hands of actual people, not web application security experts.

Myth: XSRF can be prevented by filtering based on the Referer header
Fact: This is very unreliable, attackers can easily block the sending of the Referer header, and the HTTP RFC's make it clear the this header is optional at users discretion. Browsers also omit the referer header when they are being used over SSL.

Myth: XSRF weaknesses are low risk
Fact: XSRF weaknesses can be used by attackers to perform any action an authorized user of the application can perform! This could include changing passwords, buying merchandise, or transferring money depending on the domain of your application.

Myth: My firewall / SSL server / or the .NET / Struts framework protect me from XSRF.
Fact: If you haven't specifically addressed the need to protect your application about XSRF, you are almost certainly vulnerable.

Terminology Definitions

- Action: Some behavior in the web application such as setting an account parameter, executing a trade, creating, updating, or deleting an object. An action is associated with a HTTP request that is usually a GET or POST, and some number of query parameters.
- Action Formulator: The source of the HTML elements that resulted in the creation of the request. Under normal circumstances a web application is the only action formulator for its actions, in the case of an XSRF attack, an unauthorized Action formulator is present.
- Action Name: A unique name for each action, this could be a human readable name like "Execute_Trade" or a unique number like 42. Action names can be implied from the URI they are located at, for example the name of the servlet that implements the action could be it's effective Action Name.
- Session Identifier: an unpredictable value used to uniquely identify a users session. In a J2EE environment this it typically called a JSESSIONID, in an ASP environment it is commonly called the ASPSESSIONID. Many other web application frameworks have an unpredictable session identifier stored in a cookie, and used to identify user sessions and store server side information about the user.
- HMAC_sha1(x, y): A keyed cryptographic hash based on SHA-1[2] and using the keying technique described in RFC 2104[3]. The output of this function needs to be encoded so that it will not contain

[2] http://www.itl.nist.gov/fipspubs/fip180-1.htm Secure hash algorithm 1, while recent cryptanalysis results suggest this algorithm is theoretically imperfect, it is the current standard for secure cryptographic hashes.
[3] http://www.faqs.org/rfcs/rfc2104.html Keyed-Hashing for Message Authentication. This RFC defines a standard way of creating cryptographic hashes that can only be verified with knowledge of the key.

invalid characters, for example with base 64 encoding[4]. The java cryptography extensions[5] and the OpenSSL[6] and RSA BSAFE[7] libraries all support HMACs with SHA-1.

- Query Parameter: a name value pair, associated with a GET or POST request. Query parameters can contain values provided by the user, or values from the server generating the form or link the query was generated by. A query parameter can also come from a form input.

Protection approaches

Approach 1: Use cryptographic tokens to prove the Action Formulator knows a session specific secret.
Level of protection: Very High **Recommended by iSEC**

Description:
When the web application formulates an Action (by generating a link or form that causes an Action when submitted or clicked by the user) the application includes as a query parameter (usually as an "Input" tag of type "hidden") a name value pair with a name like: XSRFPreventionToken, and a value that is an HMAC_sha1(Action_Name + Secret, SessionID).

When an action is performed by the user, before the action is executed the XSRFPreventionToken has it's value verified by comparing the value of the provided token to a calculation of HMAC_sha1_(Reqested_action_name + Secret, User_SessionID). If the values do not match, then the Action Formulator is not the application, the Action should be aborted and the event can be logged as a potential security incident.

Note the action name should be different for each action, although the "Secret" can remain constant. From time to time XSRFPreventionTokens may leak out from the application, for example an application that doesn't use SSL will result in XSRFPreventionTokens being sent to other sites in the referer field when users click on links to those sites. By keeping action_names unique the application minimizes the impact on the application of a potentially hostile third party learning an XSRFPreventionToken (the attacker could now formulate only the actions with the same name as that which referred the victim to the attacker's site). Using SSL is almost always necessary for secure web applications.

Advantages: Very strong protection, no additional memory requirements per user session.

Disadvantages: Requires the dynamic generation of all Actions. This widespread change can be eased through integration with a thin client framework. The approach also requires a small amount of computation on each request formulation, and upon Action verification.

[4] http://en.wikipedia.org/wiki/Base64 A common web application encoding format for binary data.
[5] http://java.sun.com/products/jce/ The standard for java crytpto, with many provider implementations including one from Sun.
[6] http://www.openssl.org/ A common, free cryptography library. This library has had numerous security flaws, and should be kept up to date.
[7] http://www.rsasecurity.com/node.asp?id=1202 A commercial, somewhat expensive cryptography provider for Java and C.

Approach 2: Use secret tokens to prove the Action Formulator knew an Action and user specific secret.
Level of protection: Very High **Recommended** by iSEC

Description:
When the web application formulates an Action (by generating a link or form that causes an Action when submitted or clicked by the user) the application includes as a query parameter (usually as an "Input" tag of type "hidden") a name value pair with a name like: XSRFPreventionToken, and a value that is a randomly generated 128 bit value that has been base 64 encoded. For each action this token need only be generated randomly once, after that the token value is stored in a session specific table mapping Action names to validation tokens.

When an action is performed by the user, before the action is executed the submitted XSRFPreventionToken has its value verified by comparing the provided token to the value stored in the mapping table for this Action. If there is no value in the table for this action name, or if the value provided does not match the value in the table exactly then the Action Formulator is not the application, the Action should be aborted and the event can be logged as a potential security incident.

Similarly to approach 1, the action name should be different for each action. From time to time XSRFPreventionTokens may leak out from the application, for example an application that doesn't use SSL will result in XSRFPreventionTokens being sent to other sites in the referer field when users click on links to those sites. By keeping action_names unique the application minimizes the impact on the application of a potentially hostile third party learning an XSRFPreventionToken (the attacker could now formulate only the actions with the same name as that which referred the victim to the attacker's site). Using SSL is almost always necessary for secure web applications.

Advantages: Very strong protection, extremely little additional computation per Action.

Disadvantages: Requires the dynamic generation of all Actions. This widespread change can be eased through integration with a thin client framework. Requires additional memory equivalent a 128 bit constant times the number of actions per session.

Approach 3: Use the optional HTTP referer[sic] header[8] to verify Action Formulators.
Level of protection: Medium

Description:
Upon invocation of an Action, verify that the referer header in the HTTP request is from a source that is authorized to invoke the application. Disallow access if the referer header is not present, or if it's value points to a page or site that should not be formulating the action. If the header isn't from the allowed set, the action should be aborted and the event can be logged as a potential security incident.

Advantages: Simple to implement.

Disadvantages: The header is optional and may not be present, some browsers disable this header and it is not available when interactions occur between HTTPS and HTTP served pages. The risk of header spoofing exists, and tracking the valid sources of invocations may be difficult in some applications.

[8] http://www.w3.org/Protocols/HTTP/HTRQ_Headers.html#z14 this header was misspelled referer in the HTTP\1.1 specification, and now must be given with the spelling seen here.

Approach 4: Require changes to application state to be done only with HTTP POST operations.
Level of protection: Very Low

Description:
Only allow HTTP POST operations to perform changes to the state of the application such as creating, updating, or deleting of objects. This makes exploitation through images ineffective, and helps reduce exploitability of allowing image links to be shared inside an application.
In a low risk environment, this may encourage attackers to target other sites rather than those with this limited protection.

Advantages: Simple to implement.

Disadvantages: Attackers can adjust their attacks to be form based like XSRF examples 1, and 2. Submit forms automatically, or though tricking users by making huge, mislabeled submit buttons.

Approach 5: Use a simplified XSRFPreventionToken.
Level of protection: Very High if done correctly, to Medium

Description:
When the web application formulates an Action (by generating a link or form that causes an Action when submitted or clicked by the user) the application includes as a query parameter (usually as an "Input" tag of type "hidden") a name value pair with a name like: XSRFPreventionToken, and a value that is a randomly generated 128 bit value that has been base 64 encoded. The XSRFPreventionToken is generated upon user login, is stored in the session, and never changes. This token is reused for every action, and because of this actions do not have to be "named".

When an action is performed by the user, before the action is executed the submitted XSRFPreventionToken has its value verified by comparing the provided token to the value for this session. If the value provided does not match the value in the session exactly then the Action Formulator was not the application, the Action should be aborted and the event can be logged as a potential security incident.

Note avoid disclosing the XSRFPreventionToken to the user by using forms with the POST method. This helps avoid people snooping the XSRFPrevention token off the screen, or learning it from seeing referer headers or web server logs.

Advantages: Simpler to implement than approach 1 or 2.

Disadvantages: Requires changes to all action formulations in the application, requires the application to be completely under SSL, or to use HTTP POST only for any operation that required an XSRFPreventionToken in its formulation to avoid token leakage through the referer header.