Y.Ramesh, T. Naresh, T. Chalapathi / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com
Vol. 2, Issue 1,Jan-Feb 2012, pp.1150-1156

Breakdown the Session Riding Attacks (XSRF) with Cryptographic NONCE

Y.Ramesh*, T. Naresh, T. Chalapathi*****

*(Assoc. Prof., CSE Department)
**(Asst. Prof., CSE Department)
***(Asst. Prof., CSE Department)
(Aditya Institute of Technology And Management,Tekkali-532201, A.P.)

Session Riding(XSRF)[1] is an attack outlined in the OWASP [2]Top 10 whereby a malicious website will send a request to a web application that a user is already authenticated against from a different website. This way an attacker can access functionality in a target web application via the victim's already authenticated browser. Targets include web applications like social media, in browser email clients, online banking, and web interfaces for network devices. We propose Browser-Enforced Cryptographic Nonces, a browser-based mechanism to defend against Session Riding (XSRF) attacks and infers whether a request reflects the user's intention and whether an Cryptographic Nonce is sensitive, and strips sensitive authentication tokens from any request that may not reflect the user's intention.
Keywords: web security, browser security, web server security

Abbreviations: OWASP- Open Web Application Security Project, XSRF- cross site request forgery

I. Introduction

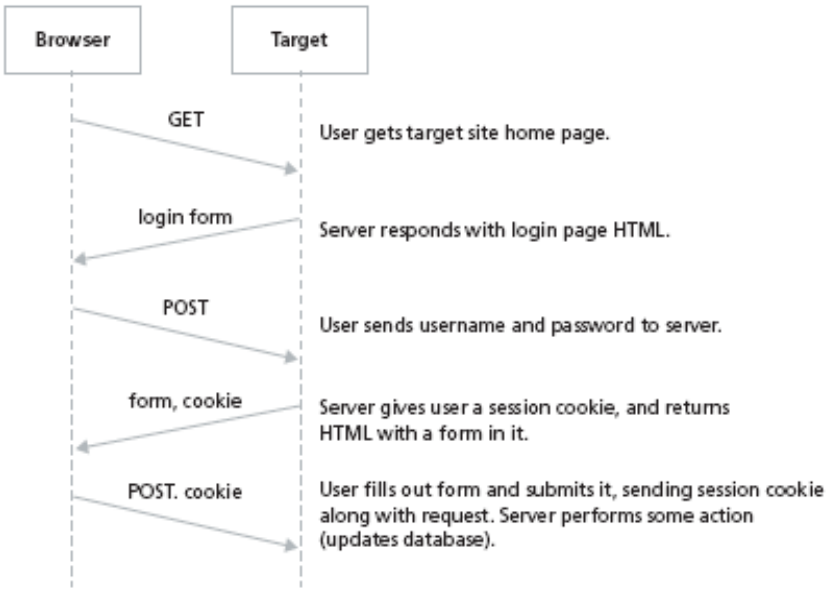
A few years ago, Cross Site Request Forgery was not taken as a serious bug. It wasn't even taken as bug at all. But today, web is about lots of money and many non-IT users manage important websites and that's why this kind of bug is very popular these days. An important condition for a successful attack is that a user must click the attacker's link.

A browser typically uses two ways of requesting web applications – sending data via URL parameters where HTTP GET request is used, and sending data via forms where HTTP POST is used. The application typically does some action – inserts a new user into a table, deletes a forum post, etc. Nothing strange? Yes, but … but there is one problem – the web application typically doesn't check if requests are generated by the web application itself (= user clicks a link or sends a filled form). Still seems okay? Let's continue. What if the attacker creates a link for some action and sends it to the user? The user clicks the link and the action is performed without the user even noticing. And this is called Cross Site Request Forgery.

We already know that users have to click on the attacker's link or fill their form. Another condition is that the user must be logged on to the vulnerable web, but these days, almost every application provides the "keep me logged in" functionality. ASP.NET complicates a successful attack because of ViewState. If ViewState is turned on, you cannot send tampered POST requests to an ASP.NET application because validation of ViewState fails. So, many developers think that an ASP.NET application is bulletproof against XSRF. But there are always a few catches:

- GET requests can still cause XSRF.
- ViewState can be generated outside an application if you are not use machine keys as keys for ViewState encoding.
- Even if you are using machine keys to encrypt your ViewState, you are not 100% safe. ASP.NET doesn't take form values from Request.Form but from Request.Params. This is the reason why it is possible to perform something called a "One click attack". It is a special case of XSRF. You simply send ViewState and values of form fields via GET. The trick is that you can use ViewState generated by ASP.NET after post, change values of fields and validation still succeeds.

figure-1 shows what happens when a user visits a website figure-2 shows what Here's what happens in a XSRF attack



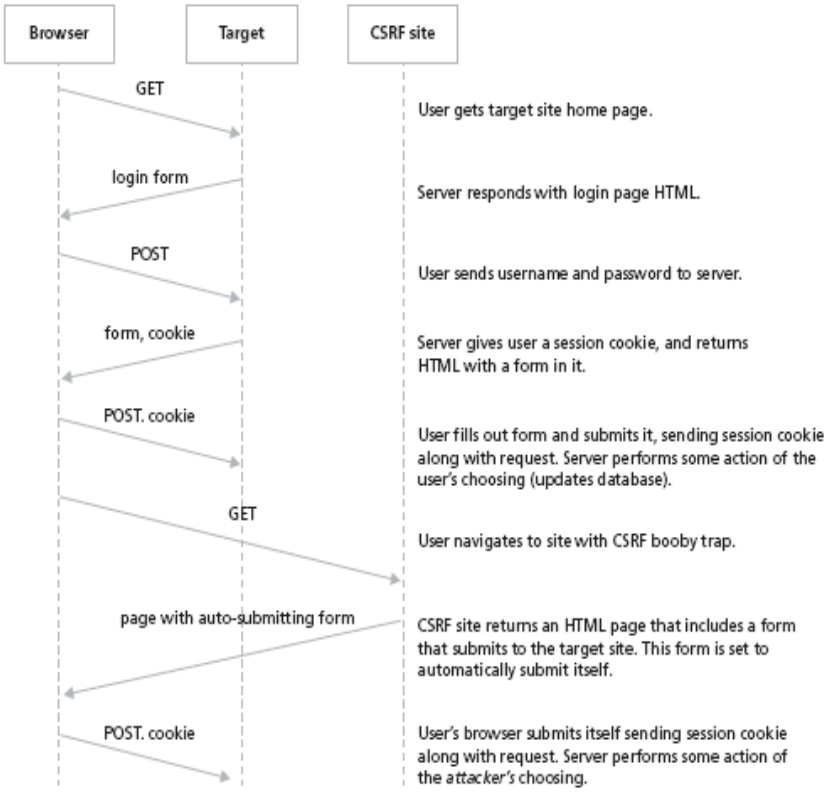


Fig:1 user visiting website

Fig:2 Launching XSRF attack

1.1 Example of Session Riding

Example shows a Session Riding also called as one-click attack. Let's have a simple page with a textbox and a button. The code below handles the Onclick action of the button: the figure 3 & 4
Shows the implementation of On click action of the Button

Figure 3 :user interface of On click Button

```
protected void btnSend_Click(object sender, EventArgs e)

{
Response.Write(txtUserID.Text);
}
```

Users typically insert a value into the txtUserID textbox and click the button. But the attacker can forge a link similar to this one to the user:
http://localhost:1326/WebSite2/default.aspx?__VIEWSTATE=%2FwEPDwUKMTkwNjc4NTIwMWRkBqEaTzfhaAEOn00zsI7zRz%2Fohdk%3D&txtUserID=this+is+naresh&Button1=show&__EVENTVALIDATION=%2FwEWAwKD%2FtmSAwLT8dy8BQKM54rGBl9krlLYLC%2B828tJxnX3AWyeazou

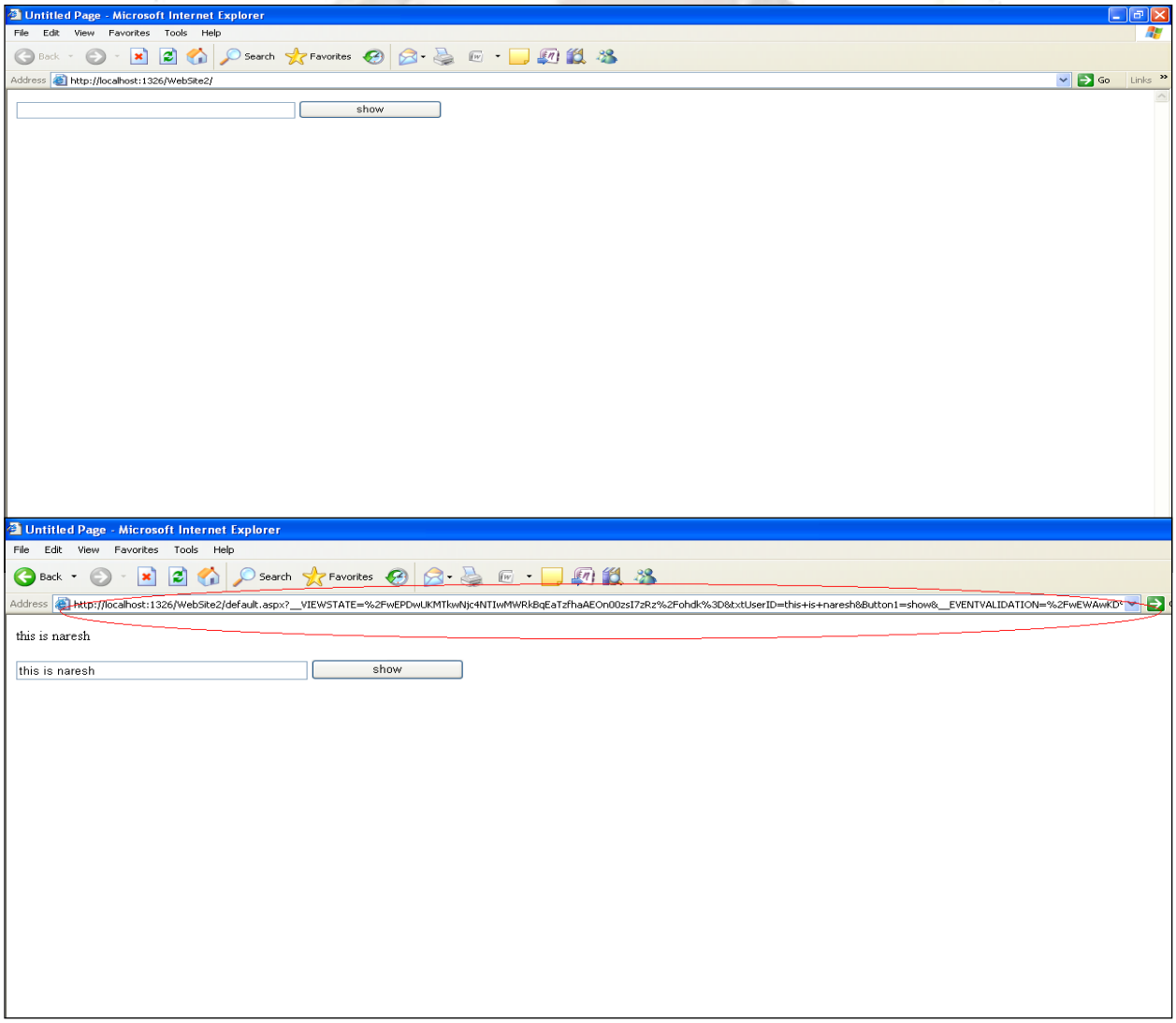
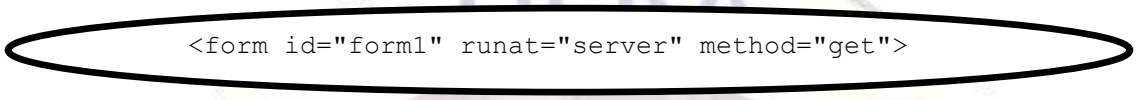


Figure 4: result of On click Button

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
```




```
</head>
<body>
      <div>

<asp:TextBox ID="txtUserID" runat="server" Width="300px"></asp:TextBox>
      <asp:Button ID="Button1" runat="server" OnClick="Button1_Click" Text="show" Width="157px" /></div>
   </form>
</body>
</html>
```

The **<anything>** macro can be changed to anything else by the attacker. ViewState is taken from the page that can be generated after postback on that page. Validation is successful.
let's say I happen to visit *hakersite.com*. It just so happens that this site is trying to attack people who bank with *mybank.com* and have setup a XSRF attack on their site. The attack will transfer $5000.00 to their account, which is account number 990099009. Somewhere on *hakersite.com* attackers have added this line of code:

*<iframe src="http://mybank.com/app/transferFunds?amount=5000&destinationAccount=*990099009*" >*

Upon loading that iframe, my browser will send that request to *mybank.com* which my browser has already logged in as me. The request will be processed and send $5000.00 to account 990099009 the attack consists in submitting a malicious HTTP form to a page that expects a form. Reasonably, this page will be consuming posted data to perform some sensitive operation. Reasonably, the attacker knows exactly how each field will be used and can come up with some spoofed values to reach his goal. It's usually a targeted attack, and it is also hard to track back because of the triangular trade that it establishes—the hacker induces a victim to click a link on the hacker's site, which in turn will post the bad code to a third site.

1.2 What Can Be Done by Session Riding

It depends on how badly an application is written. If it is very bad and the administrator of the web doesn't take care of the server properly (for example, encoding of ViewState based on machine key being turned off), then the attacker can do anything that the victim of the attack could normally do.

Characteristics are common to XSRF:

- Involve sites that rely on a user's identity
- Exploit the site's trust in that identity
- Trick the user's browser into sending HTTP requests to a target site
- Involve HTTP requests that have side effects

1.3 How to find Session Riding Bug

If you find any page/control/etc that does an action on GET request, there is a possibility of a XSRF bug. For example, try to find the following strings in your source code:

- QueryHelper.GetString("action")

Try some other similar strings. You can also search for strings:

- EnableViewState="false"
- EnableViewStateMac="false"

If you find these strings in the **<%@ page** directive, it means that a developer turned off ViewState validation (first case) or machine keys for ViewState encoding (second case). You already know that ViewState validation helps a lot to avoid POST XSRF.

II. Literature Survey

vulnerabilities have been known and in some cases exploited since 2001.[2] Because it is carried out from the user's IP address, some website logs might not have evidence of Session Riding.[3]Exploits are under-reported, at least publicly, and as of 2007[4] there are few well-documented examples. About 18

million users of eBay's Internet Auction Co. at Auction.co.kr in Korea lost personal information in February 2008. Customers of a bank in Mexico were attacked in early 2008 with an image tag in email. The link in the image tag changed the DNS entry for the bank in their ADSL router to point to a malicious website, impersonating the bank.[5]

2.1 Real-world XSRF vulnerabilities

In order to understand how commonly the XSRF vulnerability exists in the real-world web applications, one of the authors of the paper examined about a dozen web sites for which he has an account and usually visits. As a result, we found four of them are vulnerable to XSRF attacks as shown in following Table . We veri¯ed all the attacks with Firefox 2.0.

Vulnerable web site	Targeted sensitive operation
A university credit union site	Money transfer between accounts; adding a new account
A university web mail	Deleting all emails in the Inbox
An online forum for HTML development	Posting a message; updating user profile
Department portal site	Editing biography information

Table 1. The XSRF vulnerabilities discovered in real world websites.

2.2 Session Riding (XSRF) VS cross-site scripting (XSS)

Cross-site request forgery, also known as a **one-click attack** or **session riding** and abbreviated as **XSRF**, is a type of malicious exploit of a website whereby unauthorized commands are transmitted from a user that the website trusts.[6]Unlike cross-site scripting (XSS)[7], which exploits the trust a user has for a particular site, XSRF exploits the trust that a site has in a user's browser. XSRF vulnerabilities should not be confused with XSS vulnerabilities. In XSS exploits, an attacker injects malicious scripts into an HTML document hosted by the victim web site, typically through submitting text embedded with code which is to be displayed on the page, such as a blog post. Most XSS attacks are due to vulnerabilities in web applications which fail in sanitizing untrustworthy inputs which might in turn be displayed to users. XSRF attacks do not rely on the execution and injection of malicious JavaScript code. XSRF vulnerabilities are due to the use of cookies or HTTP authentication as the authentication mechanism. A web site that does not have XSS vulnerabilities may contain XSRF vulnerabilities.

2.3 Existing XSRF Defenses

Web sites have various XSRF countermeasures available:

- Requiring a secret, user-specific token in all form submissions and side-effect URLs prevents XSRF; the attacker's site cannot put the right token in its submissions[1]
- Requiring the client to provide authentication data in the same HTTP Request used to perform any operation with security implications (money transfer, etc.)
- Limiting the lifetime of session cookies
- Checking the HTTP Referer header
- Ensuring that there is no clientaccesspolicy.xml file granting unintended access to Silverlight controls[8]
- Ensuring that there is no crossdomain.xml file granting unintended access to Flash movies[9]
- Verifying that the request's header contains a X-Requested-With. Used by Ruby on Rails (before v2.0) and Django (before v1.2.5). This protection has been proven unsecure[10] under a combination of browser plugins and redirects which can allow an attacker to provide custom HTTP headers on a request to any website, hence allow a forged request.

III. Browser Enforced Cryptographic NONCE (BECN)

Now that we've run through some common non-working solutions to XSRF vulnerabilities, we'll discuss some solutions that work. All of them are effective enough to reduce the XSRF threat to a negligible concern, but all have costs. Some are easier to implement than others, some incur heavy burdens on users, and some are more secure than others. Which one is right for you depends on your application and the circumstances of your development cycle and user base.

What is a "nonce"?

Many of these solutions involve the use of a nonce. "Nonce" is a shortened form of "cryptographic number used only once," a one-time token used in a transaction[5]. The requirements for a nonce used for XSRF protection are significantly lower than one used for cryptography. Attackers will be limited in the number of requests that they can cause their victim to send, so the nonce only needs to be somewhat difficult to predict. While there is no reason not to use a high-quality random number, such as 128 bits of cryptographically random data, or a GUID, it is acceptable to simply use a hash of two or more non-cryptographically random numbers and a static secret.

3.1 Single per-page nonce

The simplest method of XSRF protection to implement is to insert a nonce into each form and also into a special slot in the server session, and then to compare the values of these two variables when the form is submitted. Here is a pseudo-code example:

```
<%
nonce = generate_nonce()
session.nonce = nonce
%>
<form>
<input name="field1"><br>
<input name="field2"><br>
<input type="submit">
<input name="nonce" type="hidden" value="<%= nonce %>">
</form>
```

When the form is submitted, the following is executed:

```
if (post.nonce != session.nonce) {
log_XSRF_attack()
error_and_exit()
}
// normal form handling here
```

What this code does is verify that each request to process a request has been preceded by a request for the associated form. In other words, for each form submission, the form has actually been loaded. Since, due to the DOM security model, the attacker cannot read data from another site, the attacker cannot load the form and read the nonce value. Although this approach provides excellent protection against XSRF, it is not without problems. The problems with this approach lie in the realm of breaking expected web behavior, rather than in security.

3.2 Per-session nonce

To overcome the usability weaknesses of the section 3.1, a per-session token can be used. In this case, a single token is created at the beginning of the session and is used throughout the session. In this pseudo-code example, the following would be in some global application file:

```
<%
function session_initiate(first_name, last_name /* etc */) {
session.fisrt_name = first_name
session.last_name = last_name
/* etc */
session.form_token = generate_form_token()
}

%>
```

Then, in the page code:

```
<%
<form>
<input name="field1"><br>
<input name="field2"><br>
<input type="submit">
<input name="form_token" type="hidden" value="<%= session.form_token %>">
</form>
```

When the form is submitted, the following is executed:

```
if (post.form_token != session.form_token) {
log_XSRF_attack()
error_and_exit()
}
// normal form handling here
```

The primary advantage of this method is that multiple browser windows, page caching, and other functions will not cause false positives in XSRF detection. However, it is a rather fragile solution. Since the form token has a long lifespan, it must be protected from leakage. If an attacker were to be able to recover the target's form token, they would be able to issue valid requests so long as the target's session was active.

Token Security

Fortunately, the techniques that must be used to protect the token are well understood and are part of longstanding secure web development practices. The token must be secure in transport; communications should be protected via SSL[10],

Conclusion

awareness of XSRF has greatly increased, and many libraries7 are available to help developers protect their websites. However, the overwhelming majority

of sites on the Internet remain completely vulnerable. It is my hope that this paper will help in raising awareness of the issue and the available countermeasures.

Referneces

[1] Shiflett, Chris (December 13, 2004). "Security Corner: Cross-Site Request Forgeries". php|architect (via shiflett.org). http://shiflett.org/articles/cross-site-request-forgeries. Retrieved 2008-07-03.

[2] Burns, Jesse (2005). "Cross Site Request Forgery: An Introduction To A Common Web Weakness". Information Security Partners, LLC. http://www.isecpartners.com/files/XSRF_Paper_0.pdf. Retrieved 2011-10-0

[3] Ristic, Ivan (2005). *Apache Security*. O'Reilly Media. p. 280. ISBN 0-596-00724-8.

[4] Christey, Steve and Martin, Robert A. (May 22, 2007). "Vulnerability Type Distributions in CVE (version 1.1)". MITRE Corporation. http://cwe.mitre.org/documents/vuln-trends/index.html. Retrieved 2008-06-07

[5] "List of incidents for which Attack Method is Cross Site Request Forgery (XSRF)". Web Application Security Consortium. February 2008. http://www.webappsec.org/projects/whid/byclass_class_attack_method_value_cross_site_request_forgery_(XSRF).shtml. Retrieved 2008-07-04.

[6] Ristic, Ivan (2005). *Apache Security*. O'Reilly Media. p. 280. ISBN 0-596-00724-8.

[7] SWAP: Mitigating XSS Attacks using a Reverse Proxy,Peter Wurzinger, Christian Platzer, Christian Ludl, Engin Kirda, and Christopher Kruegel

[8] http://msdn.microsoft.com/ens/library/cc197955.aspx,Client access policy file to allow cross-domain access by Silverlight , controls,http://msdn.microsoft.com/en-us/library/cc197955.aspx

[9]http://www.adobe.com/devnet/flashplayer/articles/cross_domain_policy.html,Cross-domain policy file usage recommendations for Flash Player,

[10] The SSL Protocol: Version 3.0 Netscape's final SSL 3.0 draft (November 18, 1996)