Website Security Threats

Authors

Nathan Hamilton, Matthew Hopkins, Forrest Wallace

Introduction

Website security is a commonly discussed topic throughout the computer science industry. Virtually every software company utilizes some form of internet communication within their product, thus website is of great concern. Within this paper, we will be discussing three of the most common, and dangerous, forms of website attacks—cross-site scripting, cross-site request forgery, and SQL injection.

Cross-Site Scripting

Overview

One of the most common forms of attacks that tends to be ranked highly on OWASP's reports and most likely will continue to hold this position is a method called Cross-Site-Scripting. Cross-Site-Scripting, referenced as XSS now, is a type of injection attack that allows corrupt scripts to be implemented into a supposedly trusted website and be executed within the browser. These attacks can typically occur anywhere that input from a user is used within an output without generating or encoding it within the web server or backend database. Put in layman's terms this allows the data input to be read as executable code.

Consequences

Most of the time, when these flaws are brought to light they are reported as alert boxes. Which then causes employers to push it back as something to take care of later as if it is not a pressing matter. This could not be farther from the truth. By having XSS vulnerabilities within your website you are potentially giving an attacker full control, which allows them to rewrite the page, allow different events to happen such as redirecting the page, or even hiding all current content and display something entirely different. Even further, this can allow an attacker to do a multitude of things, such as setting up a keystroke reader to read in usernames and passwords entered by users, stealing and receiving session IDs or cookies. Therefore, XSS can be the entryway into your own network. For instance, if you have an enterprise, suppose a user is tricked by a phishing scheme to click on a link and execute XSS on their machine, this is a foothold to take over the machine. Then an attacker can scan the network as if they are on the network. Anything that user can do now the attacker can do, bypassing firewall and other rule sets. Essentially, because of XSS your system and network are now compromised.

Types

There are three main forms of XSS, Reflected, Persisted and DOM based scripting. Let us first start with Reflected XSS.

In Reflected XSS, typically set up in a Query string in URL, the web application takes a value out of the URL/Query string and returns it back down to the browser. For instance, passing up a User Name, and it pulls from the Query string not from database. If an attacker can manipulate that value, they can run XSS a great amount of JavaScript attacks and now they has

access. This attack is more of a I want it to happen right now to this person, more of a targeted attack. Reflected is more seen with phishing attacks and sending out the link obfuscate the link so that you cannot see there is malicious code within the Query string they click the link and the code will execute.

Next is Persisted XSS, in which you store it for later. This is not the attacker storing it for later, but web server or the backend storing mechanism such as a Database or HTML File stores the payload. Then when an unsuspecting user comes across the page it loads the value and executes in the browser. For instance, think of a site such as Quora. Posting a comment with XSS and waiting for anyone to come in to the post and then they will have the code executed on the browser. This works particularly well with trying to get administrator information, an attacker waits for an administrator to view the post the program steals session cookie and then it allows them to log in as an administrator. The main idea behind the Persisted XSS is that an attacker sticks the code out there and it will continuously hit anyone who comes across it while taking full advantage of them.

Finally, there is DOM based XSS, which is much like reflective. Except DOM does not rely on the browser directly to take that value and execute it as JavaScript, but instead relies on the Document Object Model within the browser to parse the value and execute it that way.

All the above have different signatures and techniques that they can be employed. Context of HTML does not work in a DOM based situation. In DOM you can write value to screen and nothing happens does not get executed right away, but if you have a JS function on the page that calls the value, at that point the value gets called and executed in which the XSS

attack occurs. A little bit different and you don't see it as much, but DOM is there and can be tricky to find.

Context

There is a bundle of different context that can be brought up when we talk about XSS. You have HTML context, so if I am going to attack an HTML element, the html tag <html>, the span , div tags, that is where I am putting in a script tag. There is also Attribute based Context, referring to image tag or link tag, and setting the value that the user setup as an alt attribute or some other attribute. The thought is I can break out of that attribute and start writing my own events such as on click do refer my webpage. Another thought with the image tag, can I write or add something into the image tag and say on error do this. Then if my source of the image is supposed to be there and I put in some source that will not exist then I put whatever I need character wise to break out of that attribute, on error alert XSS command. As said above, often in reports you'll see alert boxes. It is the easiest way to display it and has the least consequences caused by XSS vulnerabilities, but as now known not the only consequence.

As an attacker, they need to understand the context to be able to understand how they are going to attack that specific instance. They will use different characters and approaches for the different context. Which makes it difficult for the defenders, because defenders can't just say for HTML encoding just encode the < > symbols and the code is all protected. When you talk about attribute, quote, double quote, and space are all of concern because those are delimiters, whereas greater than and less than symbols aren't of any concern in that context.

This makes it far more difficult, so defenders want to use libraries that are dedicated to help such as .NET anti-XSS library, which is now the web protection library that is available. Defenders can use this, and it has an array of different encoding functions to allow a developer to encode properly. Similarly, via SAPID library for Java and other languages, defenders follow the same setup importing the library and all the encoding functions are available for them. Defenders need to properly encode, stating that user input is data, and to not treat the data as a delimiter or as an actual opening bracket closing bracket. Just display it on the screen, no more, no less. In some cases, defenders will need to chain two contexts together which allows them to ensure that XSS is not possible.

Prevention

In most cases, it's simple to eradicate XSS, it is just a matter of effort on the developers to be able to go in there and understand when taking data and displaying to screen along with how to encode it properly. There are tools to also help developers with this, view engines such as Razor, which is used for .NET, this tool takes most everything that is created within here is encoded automatically. By using view engines this causes it to be very difficult to find an XSS vulnerability for attackers.

Cross-Site Request Forgery

Overview

Cross-Site Request Forgery (CSRF), also known as "sea serf", one-click attack, and session riding, [7] is a malicious attack on a website or web service by tricking an authenticated

session into performing actions that are unwanted. Attackers gain access to account functionality without the owner's knowledge, allowing the attacker to initiate a variety of state changing actions. A CSRF attack works a bit different than most website security exploits. It is unique in that it works exclusively on the client or browser side of a website, with no access to the server side. This allows for the attacker to make changes to an account, such as changing login information, transferring funds, or altering the contents of a shopping cart. It does not, however, expose personal information to the attacker. Furthermore, implementing a CSRF attacks relies on the attacker's ability to lure a victim into executing it themselves. Because of this, these attacks tend to be lesser known as opposed to attacks like SQL injection and cross-site scripting, as a fewer number of people are affected. [9] The damages, however, can be just as problematic.

Requirements

To pull off a CSRF attack successfully, four things must happen. [7]

1. The target website must not check the referer header on submitted forms, or the victim must have a browser plugin that allows referer spoofing.
2. The target website must have a form submission or URL input that changes the state of some data element.
3. The attacker must determine the correct value(s) for all form or URL inputs.
4. The attacker must lure a victim into visiting a website with malicious code that executes the CSRF attack.

Referer Header

An HTTP referer is an HTTP header field that is attached to server requests. When a user clicks on some hyperlink or redirect, it sends a request to the server to redirect the user's browser to the new webpage. If the new webpage has referer headers enabled, the source webpage is included as a referer. This allows the new webpage to verify that the user was redirected by a trusted source. If the referer is not a trusted source, the webpage can deny form submissions or access altogether. So why a website would disable referer headers in the first place? Web servers typically log all HTTP traffic, including the referer (if it is present). This raises privacy concerns for many web hosts, and in today's day and age, people take their privacy very seriously. In response, many websites will utilize services that alter referer headers. The proper way to handle this situation is to simply blank that referer field. However, some services change the referer to a false address instead. [8] This is known as referer spoofing. To a CSRF attacker, spoofing the referer header is very common practice to get around a target website that checks the referer header. Therefore, checking the referer is not a completely effective way to prevent CSRF attacks, though it does guard against your more amateurish attacker.

Implementation

CSRF attacks are implemented two different ways, based on how the target website sends and receives requests from the server. In the instance of GET requests, the attacker is in luck. CSRF vulnerabilities within a website using GET requests are easily exploitable. For

instance, let's say a website validates a login through email and password. Discovering the name of the email variable within the HTML is as simple as examining the page source. Let's say the email is saved in the variable *loginEmail*. The attacker then creates a malicious URL and embeds this within the *src* property of an HTML image tag, which is placed in some webpage (See example 2.1). The image does not have to be visible (width and height can be set to 1 so it only covers 1 pixel, making it invisible to the eye). All that is needed is for the image to be loaded for the URL to take effect. Once the malicious URL is executed, the login email is changed to the email specified in the URL.

The second implementation is through POST requests. POST requests make CSRF attacks a bit harder to implement, but contrary to popular belief, it does not mitigate the threat entirely. POST requests transfer data to the server via hidden value fields, rather than through the URL. In this case, an attacker must build an HTML page with a form. In that form is an input value that changes *loginEmail* to the email specified by the attacker. Then, a JavaScript function is called to submit the form to the server. (See example 2.2)

Social Engineering

Once the false webpage is created, the attacker is faced with the hardest part of the attack—luring a victim into visiting the site while they are simultaneously logged in to the target site. This is where a bit of social engineering comes in. A link to the false webpage can be sent via email or instant message to a variety of people. If any of these users click on the link while running an authenticated session on the target website, the attack is successful. In the case of a GET CSRF attack, the attacker can simply include the HTML code for the image in the email

itself. Therefore, most email servers block the rendering of email images until the user enables it (and the user should only do this if the email is from a trusted source). A savvy attacker may even couple this CSRF attack with another that spreads the email to everyone in the victim's contact list.

After the attack is executed and the login email is changed to that of the attacker's choosing, he or she can visit the target website, and use the "forgot password" functionality. Once the password is recovered and changed, voila, the account now belongs to the attacker.

Prevention

There are multiple steps that users can take to mitigate the threat of CSRF attacks on their online accounts. One such step has been iterated to us throughout our entire lives—do not click on links you do not trust, especially if they are abnormal-looking. If you are only being redirected to websites you trust, there should be no issues. It is also a good idea to log out of any online account when you are no longer using it. This prevents an attacker from executing changes on you and your account's behalf. Online banking and other sensitive online applications will log you out after a very brief period of inactivity, but most websites do not. And finally, some web applications may give you the option to keep you logged in after closing the application. DO NOT select this option. This keeps your account logged in on your web browser until you physically log out yourself. This allows an attacker to execute a CSRF attack on that website at any time. Despite how convenient it is, you should also prevent your web browser from remembering your login credentials.

Website developers can also take steps to guard against CSRF vulnerabilities. As mentioned before, referer headers and POST should be used for all request. Since referer spoofing is an effective workaround, and POST is still exploitable, another safeguard should be used in conjunction. That safeguard is the use of a "canary" value. This value is a hidden token that can be attached to either the URL or a form. The value is also encrypted within the user's session cookie that is generated when he or she successfully logs in. When a request is sent to the server, it validates the request by comparing the value submitted with he form or URL to the encrypted value in the session cookie. [9] If the validation is approved, then everything continues as normal. If not, the server denies the request. These "canary" values are effective because they are randomly generated, making it virtually impossible for the attacker to replicate.

As effective as "canary" values can be, it is worth noting that if the website is prone to a cross-site scripting (XSS) attack, it renders this technique useless. An XSS payload can be used to read the HTML of a web page, exposing the randomly generated value. [10]

Conclusion

Awareness for cross-site request forgery has risen significantly in recent years. Typically, only websites that handle sensitive transactions and changes are targeted, and most of these websites are well-fortified against CSRF attacks. When visiting a website, or developing one, actions to prevent CSRF attacks should always be taken to ensure the safety and security of online accounts.

SQL Injection

Overview

SQL Injection uses vulnerabilities in code to send SQL attacks; it was one of the most common attacks that affected just about every website owner. In 2012, the average web application received four attacks per month and retailers received two times as many attacks.[11] In 2017, OWASP ranked SQL Injection as the number one web attack.[18]

SQL Injection (SQLI) can accomplish many goals. It can spoof the identity of a user, tamper or delete existing data, get complete disclosure of all data, and worst of all, become an admin of a server. There are many subclasses of SQLI this paper will cover: Classic SQLI, Blind or Inference SQL Injection, and Compound SQLI.

Types

Classic SQLI is an attack allowed by incorrectly filtered escape characters. The code or the database must sanitize user input before being put into the database. The SQL query that could be passed into a username field is “SELECT * FROM Users WHERE Username=’1’ or ‘1’=’1’”.[12] This SQL query will return the username. This can be used to gain passwords from the database as well.

Blind SQLI is an attack that will ask the database true or false questions and will decide an answer based on the websites response.[17] This attack is frequently used when the website shows generic error messages but does not mitigate the code that is vulnerable to SQLI. There are two main versions of Blind SQLI: Content based, and Time based.

Content based relies on the content of the website changing based on the code. A tip for checking if a website is vulnerable is if the URL has an id in it (i.e. https://www.newspaper.com/items.php?id=2). Since there is an id in this URL it can be exploited. So now we send a query “SELECT title, desc, body FROM items WHERE id =2” and inject in the URL “.com/items.php?id=2 and 1=2”. If the website returns nothing the page is vulnerable.[17] This attack, when combined with the username exploit code, can discover the username and password over time using the true or false results to see if there is a username that starts with certain characters.

Time-based relies on the database pausing for a specified or noticeable amount of time, then returning results. If it does then it is vulnerable to SQLI. Different SQL servers have different versions to cause a delay. For Microsoft SQL, the URL would look like “http://www.site.com/uul.php?id=1wait_for delay 100:00:10’--". This is a built-in function to cause a delay. In MySQL; however, it is more complicated. It must call “Benchmark(50000000, Encode (‘msg’, by 5 seconds))”.[17] This should take just a moment, but a high amount is needed for a noticeable delay. Example 3.1 shows that a union select is needed but it works by saying if the first character in the username is “x” run the delay if not then there will be no delay. This delay or no delay can be used to enumerate the username and password. There are tools to assist in this time-based attack such as SQL Map. These tools, however, are sensitive and need certain parameters met.

Compound SQLI is combining SQLI with another form of attack. One of the most useful compound SQLI is SQLI + DDOS. This type of attack is used to exhaust a server’s resources or to take down a server. A version of this attack, shown in example 3.2, creates a table of 500 rows.

These 500 rows have text of 500 bytes per column.[15] To help make these attacks less likely to be stopped, the sleep command can be used to make connections that last just long enough so the task will finish but not get caught. This version of DDOS is useful because it requires fewer bots to accomplish a successful attack.

Prevention

SQLI is a common and dangerous attack, but because of this there is a lot of information on how to defend code and databases. There are four primary defenses: use of prepared statements, use of stored procedures, white list input validation, and escaping all user supplied input.

Prepared Statements has the SQL code defined first then having the parameters passed in later, which lets the database recognize code vs data.[16] Stored Procedures are not a perfect fix, but some are able to have the same effect of parameterized queries. The difference between prepared statements and stored procedures is that the SQL code for stored procedures is stored in the database then called in the application. Stored procedures can increase risk, for example on MS SQL server it is possible to gain full rights where a similar attack using prepared statements would only get read rights.[16]

White List Input Validation is used when the design is flawed. It scrubs the input for table names or column names in the database. In addition, if the user input can be converted to a non-String like a date, numeric, Boolean, or enumerated type, it will ensure safety.[16] Escaping all user-supplied input is a last resort defense. This technique will escape user input before putting it in a query. This technique escapes all input matching the escape scheme for

the database that is being used. The database will then read the data no longer as a query but as a string.[16]

There are two additional defenses that should be considered to prevent and reduce damage from SQLI. Least privilege is reducing the privileges assigned to every database account. If an account only needs to read a limited scope, then only give it the minimum scope needed. Whitelisting input validation can be a primary defense or a secondary defense. It creates a white list of approved input and if the value is not on the list it is not allowed into the database.[16]

Appendix

Code Examples

Cross-Site Scripting

Example 1.1: Cookie Grabber

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

Example 1.2: Error Page Injection of Alert Code

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

Example 1.3: Database Query

```
<%...
        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 %>
```

Cross-Site Request Forgery

Example 2.1: CSRF with GET

```
<html>
<body>
<H1>Header</H1>
<img
src=”http://targetwebsite.com/MyAccount?emailAddress=attacker@site.com”
width=”1” height=”1” />
</body>
</html>
```

Example 2.2: CSRF with POST

```
<html>
<body>
<form name=”CSRF” method=”post”
action=”http://targetwebsite.com/MyAccount”>
      <input type=”hidden” name=”emailAddress”
      value=”attacker@site.com”>
</form>
<script type=”text/javascript”>document.CSRF.submit()</script>
</body>
</html>
```

SQL Injection

Example 3.1: SQL Injection time-based

```
1 UNION SELECT IF(SUBSTRING(user_password,1,1) =
CHAR(50),BENCHMARK(5000000,ENCODE('MSG','by 5 seconds')),null)
FROM users
WHERE user_id = 1;
```

Example 3.2: SQL Injection DDOS

```
select tab1
from (select decode(encode(convert(compress(post) using
latin1),concat(post,post,post,post)),sha1(concat(post,post,post,post)))
as tab1 from table_1)a;
```

References

[1] https://securityinfive.libsyn.com/episode-102-owasp-top-10-a3-cross-site-scripting

[2] https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[3] https://www.youtube.com/watch?v=SHmQ3sQFeLE

[4] https://www.youtube.com/watch?v=486KmQOcwWg

[5] https://www.youtube.com/watch?v=vRBihr41JTo

[6] https://www.computerweekly.com/tip/Cross-site-request-forgery-Lessons-from-a-CSRF-attack-example

[7] https://en.wikipedia.org/wiki/Cross-site_request_forgery

[8] https://en.wikipedia.org/wiki/HTTP_referer

[9] https://haacked.com/archive/2009/04/02/anatomy-of-csrf-attack.aspx/

[10] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

[11] Imperva (July 2012). "Imperva Web Application Attack Report" (PDF). Archived (PDF) from the original on September 7, 2013. Retrieved August 4, 2013.

[12] https://www.wordfence.com/learn/how-to-prevent-sql-injection-attacks/

[13] https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)

[14] https://www.ijcsmc.com/docs/papers/August2017/V6I82017.pdf

[15] https://www.securityidiots.com/Web-Pentest/SQL-Injection/ddos-website-with-sqli-siddos.html

[16]https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_1:_Prepared_Statements_.28with_Parameterized_Queries.29

[17] https://www.owasp.org/index.php/Blind_SQL_Injection

[18] https://www.owasp.org/index.php/Top_10-2017_Top_10