Ryan Vacca

Julia Vrooman

Mathew Moltzau

Homework 5 - Exploitation

**Question 1 (3 points).** The following Web app provides a 6-levels of XSS attacks: [https://xss-game.appspot.com/ (Links to an external site.)](https://xss-game.appspot.com/)

*The web app also provides enough hints and guidelines for how to do each of these attacks. Do all 6 XSS attacks. For each of them, explain two things: How the XSS attack was done, and how it could have been prevented.*

LEVEL 1 : User input is directly included in a page without proper escaping.

How was done: The vulnerability was exploited by entering a JavaScript alert(), in between JavaScript tags <script>alert()</script> within the search text field provided for user input to render a search.

How could prevent: This could be prevented by properly escaping the user entered input since user input should never be trust.

LEVEL 2: Content of Status messages are not escaped.

How was done: The vulnerability was exploited by embedding a JavaScript alert() into a set of tags <> and setting post-content = to the alert embedded in the tags.

post-content=<'alert("Attack")'>.

How could prevent: This could be prevented by escaping the status messages since we were able to update them directly from user input.

LEVEL 3: JS functions being execution sinks, causing the browser to execute any scripts inside the input which is hidden by higher level API’s.

How was done: The vulnerability was exploited by taking advantage of the window.location object and sneaking in a “onerror” action again but this time into the URL since this is the only point of user input into the web app. ' onerror='alert("Attack")'>

How could prevent: This could be prevented by escaping the “onerror” action once again, which was able to be exploited since the browser wont load a script from the URL once the page has been displayed. We needed to place the script inside a status message.

LEVEL 4: Context Matters

How was done: This was done by escaping the startTimer() function in the timer.html file by entering the following string into the timer user input field.

timer=');alert('Attack

How could prevent: This could have been prevented by escaping the timer input values.

LEVEL 5: Context Matters

How was done: This attack was done by first going transitioning to the signup page of the web application since the hint stated that the URL parameter in the source code is handled poorly. When we inject “next=javascript:alert('Attack')” in the URL from this page we can execute the alert by clicking the next button onside the frame.

How could prevent: This could have been prevented by not allowing the browser to interpret the URL as a scripting URI.

LEVEL 6: Follow the Rabbit

How was done: This attack was accomplished by adding the string data:text/plain,alert('Attack') into the URL. Once again using the URL to load the alert() but this time altering the character set

How could prevent: We could prevent this attack by not allowing the browser to be forced to compute the inappropriate responses of character set sniffing by including the appropriate headers

**Question 2 (1 point).**Describe second-order SQL injection with an example. Your description must explain how it is done through a complete example.

The second order SQL injection is intuitively what you may already be assuming it to be. It is still an SQL injection, modeling the same features of exploitation but is triggered in a slightly different way than first order SQL injections.

The most common example of SQL injection in the first order is when concatenating input from a user input field and directly adding the input into the query, without using a best practice prepared statement. This type of injection exploits the target immediately where as a second order SQL injection chooses by design to delay the attack until a secondary query makes the trigger case. So, the injection lies dormant until the secondary query awakens the vulnerability.

Think about this kind of an attack as a foreign spy, working for the adversaries government. The spy makes her way into the building “database”, which to her co-workers, she is trusted because she obviously made I through security, why would anyone think otherwise? Same scenario for the second order attack because developers typically trust data that is coming from the database.

The example we are going to use is a simple online web application that allows users to sign up, then later on change their password.

First, we sign up with the below account.

Username – “CyberBros”

Password – “ABC123”

Then, we make another account but this time with the SQL injection payload in the username.

Username – “CyberBros’ –“

Password – “doRaMe”

Next, we login using the second account and traverse to the portion of the web application to change our password. Now we construct the SQL query to awaken or payload to do some damage by changing the first accounts password even though we are logged into the second account.

UPDATE users

SET password='somethingElse'

WHERE username=' CyberBros '--' and password=' doRaMe '

Then after the query portion in the WHERE clause – is discarded for comments we are left with,

UPDATE users

SET password=’somethingElse’

WHERE username=’CyberBros’

So, we just successfully performed a second order SQL injection and updating the First account we created rather than the second, which is all we should have had access to manipulate.

**Question 3 (2 points)**. CWE (Common Weakness Enumeration) database includes a universal classification of different types of vulnerabilities or weaknesses that could happen in software codes. For example, CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') describes the XSS attack.

*For the following types of vulnerabilities, find out what CWE classes relates to them (There could be more than one). For each, explain one mitigation strategy that is discussed for preventing them from the CWE guidelines.*

*Sources for each attack below come from < https://cwe.mitre.org/data/definitions>*

Cross-Site Scripting:

CWE-79: Improper Neutralization of Input During Web Page Generation

The software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.

Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

CSRF:

CWE-352: Cross-Site Request Forgery, The web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.

Ensure that the application is free of cross-site scripting issues ([CWE-79](https://cwe.mitre.org/data/definitions/79.html)), because most CSRF defenses can be bypassed using attacker-controlled script

Identify especially dangerous operations. When the user performs a dangerous operation, send a separate confirmation request to ensure that the user intended to perform that operation.

Use the "double-submitted cookie" method as described by Felten and Zeller:

When a user visits a site, the site should generate a pseudorandom value and set it as a cookie on the user's machine. The site should require every form submission to include this value as a form value and also as a cookie value. When a POST request is sent to the site, the request should only be considered valid if the form value and the cookie value are the same.

Because of the same-origin policy, an attacker cannot read or modify the value stored in the cookie. To successfully submit a form on behalf of the user, the attacker would have to correctly guess the pseudorandom value. If the pseudorandom value is cryptographically strong, this will be prohibitively difficult.

Path Traversal:

CWE-23: Relative Path Traversal, the software uses external input to construct a pathname that should be within a restricted directory, but it does not properly neutralize sequences such as ".." that can resolve to a location that is outside of that directory.

**We did this in lab10** !!

Do not rely exclusively on a filtering mechanism that removes potentially dangerous characters. This is equivalent to a blacklist, which may be incomplete ([CWE-184](https://cwe.mitre.org/data/definitions/184.html)). For example, filtering "/" is insufficient protection if the filesystem also supports the use of "\" as a directory separator. Another possible error could occur when the filtering is applied in a way that still produces dangerous data ([CWE-182](https://cwe.mitre.org/data/definitions/182.html)). For example, if "../" sequences are removed from the ".../...//" string in a sequential fashion, two instances of "../" would be removed from the original string, but the remaining characters would still form the "../" string.

SQL Injection:

CWE-89: Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’),

Libraries / Frameworks : Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Consider using persistence layers such as Hibernate or Enterprise Java Beans, which can provide significant protection against SQL injection if used properly.

Input Validation : Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does.