Project 3 Writeup

Guilherme Gomes Haetinger, John Markham
COMPSCI 161 - Computer Security
UC Berkeley

December 12, 2019

1 Weaponize Vulnerability

Test 5's vulnerability can be easily weaponized. By modifying the **Session ID** cookie, we can manipulate the following **SQL Query**:

```
SELECT username FROM sessions WHERE id='{}'
```

Even though we can't fully exploit this **SQL Injection** vulnerable code because the *Javascript cookie value* takes semicolons as a break, we can still change the returned *username* to be whatever we want. This way, inside the post web page, we can set the *cookie* with the following **Javascript** code:

```
document.cookie = "SESSION_ID=' OR username='dirks"
```

so when the server tries to get the username that's logged in, it will be **dirks** and will proceed the action as him. Free tuition for everyone?

2 Vulnerability Writeup

2.1 Test 1

1. **How the vulnerability works** The page rendered by the **/wall/username** path echoes back **username** when there is not a user with the input username. Though the server correctly escapes full-bodied, closed HTML tags, like `<script></script>`, but misses those of the form `<img onevent="">`. This is a reflected XSS attack since the malicious Javascript code is echoed back onto a page on the client after injection.

2. **The line of code where the vulnerability works** Line 6 of the nowall.html file is where the malicious username is echoed back to the client. In server.py, lines 95-102 do not contain any HTML escaping of the username when preparing to render the /wall/username page in the case where otherusername does not exist.

3. **The impact an attacker can achieve (on the site, other users, etc.) by exploiting this vulnerability** With arbitrary JS code execution, they can steal cookies and impersonate users, and send malicious requests to the server that will alter the state of the application (database, server, etc.), etc.

4. **How to fix this vulnerability in this server (if applicable, feel free to provide source code)** First, line 102 should contain an escapehtml call on otherusername. Second, the server itself needs to implement more advanced methods of HTML escaping.

5. **How to avoid this vulnerability in general (techniques, tools, etc.)** One big approach to stopping XSS attacks is using a strict CSP, which can limit and even forbid all inline Javascript from running. We can modify the script-src attribute of a CSP (usually specified by a <meta> HTML tag) to be as restrictive or flexible as we would like. If no Javascript can run on the page, all forms of XSS become infeasible.

   More specifically, to prevent HTML attributes leading to arbitrary code execution, we need to force the attribute to a non-executable type. A good approach to this would be similar to using prepared statements in SQL, where we will build the displayed tag in the resulting HTML one step at a time. The approach (outlined from https://gomakethings.com/preventing-cross-site-scripting-attacks-when-using-innerhtml-in-vanilla-javascript/) is as follows:

   For every place for which user input is reflected back onto the page, in addition to sanitizing the basic HTML characters, we can build a temporary <div> element and force the user input to be interpreted as a string by passing in the user's input as textContent. In this way, we will force any potentially executable JS to a string before we display it on the client, rendering it non-executable by an attacker.

   In addition, we could also use white listing or black listing of characters to approach the problem. By disallowing some characters, we essentially stop the possibility of XSS attacks.

2.2 Test 5

1. **How the vulnerability works** We have the ability to change the **SESSION_ID** cookie by manipulating an SQL injection vulnerability. The power of this attacker is described in detail in Test 1. Essentially, they have the power to impersonate the user dirks.

2. **The line of code where the vulnerability works** Line 20 in authhelper.py has the code:

   ```
   SELECT username FROM sessions WHERE id='{}'
   ```

   which is where the vulnerability is weaponized.

3. **The impact an attacker can achieve (on the site, other users, etc.) by exploiting this vulnerability** The attacker is able to alter the experience of all other users through the power of dirks. The attacker can impersonate the user dirks. First and foremost, this could essentially destroy the site from a reputation standpoint – think Twitter if @realDonaldTrump was compromised. Second, because a user can exploit the user dirks, they can alter the application's database with posts that might lead to general discontent by any other user of the site.

4. **How to fix this vulnerability in this server (if applicable, feel free to provewide source code)** It would make it more difficult to exploit this vulnerability if the server were to implement SQL escaping in the cookie, though if a user retrieves the session id of user U, then can trivially impersonate U on the site.

   Additionally, we could add an HTTPOnly flag to the cookies to prevent this exploit, since this would make it impossible for any of an attacker's malicious Javascript code to even read the cookie in the first place.

5. **How to avoid this vulnerability in general (techniques, tools, etc.)** Prepared statements is the main protection against general SQL injection attacks. Another interesting approach listed on OWASP is the idea of whitelisting user input when appropriate. In this way, any user input that interacts with the database that we don't consider valid will automatically lead to an Exception being thrown. Since literally no one wants SQL injection attacks, this could be a good way to defend against invalid user inputs.

2.3 Test 6

1. **How the vulnerability works** This vulnerability is a more trivial version of Test 1. Users have the ability to close to add *onevent* attributes inside the img tags for their avatar. This will lead to arbitrary JS code execution. However, since the avatar is stored in the database and Javascript is only executed when a user visits a page where the avatar and malicious code are pulled from the database, this is a stored XSS attack.

2. **The line of code where the vulnerability works** Line 127 in server.py is vulnerable because of the failure to scrub the avatar thoroughly in the lines before it in the profile method. Then, line 12 in wall.html is vulnerable to malicious input (i.e. a non-escaped <img onerror="">).

3. **The impact an attacker can achieve (on the site, other users, etc.) by exploiting this vulnerability** As in Test 1, arbitrary JS execution essentially gives an attacker control over the application's state, and hence the experience of all other users of the site.

4. **How to fix this vulnerability in this server (if applicable, feel free to provide source code)** From lines 123-130, there needs to be further scrubbing of the avatar, since the escapehtml filter again fails to clean all entrypoints of Javsacript. Similar methods as those stated in Test 1 can also be employed to avoid any Javascript from being interpreted as code (i.e. creating a temporary div for user input, forcing all user input to textContent).

   Additionally, it would be beneficial to add quotes to the functionality of the HTML escaping that is currently present. At the least, this hardens SQL exploits in general, and will invalid the the exploit in this Test.

5. **How to avoid this vulnerability in general (techniques, tools, etc.)** As in Test 1, more advanced methods of escaping user input, as well as CSP's can prevent against general XSS attacks.

3 Other Issues

- **Lack of Frame Busting** Clickjacking makes it trivial for an attacker to fake actions on this site by overlaying it under an intriguing game or something of the sort. An attacker can simply overlay their game over a transparent window of Snapitterbook's website and force a user, through a series of carefully placed keystrokes and clicks, to unkowingly post something embarrassing or delete their account. The most common defense against clickjacking is using a strict CSP that specifies where frames of Snapitterbook are allowed to appear. In addition to CSP, the X-Frame-Options header accomplishes a similar goal, allowing Snapitterbook to specify which pages are allowed to embed Snapitterbook in a page.

- **No CSRF Prevention Token** This website is vulnerable to a classic CSRF hijacking attack. If a user is logged in, the application maintains this state. So, a malicious user can trick a target user into clicking a hard coded link that performs arbitrary actions: makes a fake post on their account or even deletes their account. The solution to this is to require a CSRF token. In this way, an attacker cannot actually form this hardcoded link, becuase they cannot guess the CSRF token generated by the server (assuming proper implementation of the CSRF token). Here, the server will send the client a token which the attacker will not know unless they can literally see and inspect that user's session (at which point its trivial to pwn them anyways), making it impossible to "fake" a request by a user.

  Interestingly, maintaining a Referer flag is typically a strong defense against attacks of this type. But in this case, we cannot rely on the Referer flag to mitigate all CSRF attacks since the malicious URL can come from a post within the website, etc.

- **No Salt in Password Hash** When there isn't a salt mixed in with a password hash, it becomes trivial to steal passwords via a rainbow (dictionary) attack on an application. That is, an attacker can simply obtain a list of the top trillion passwords used and just brute force a guess for every one of them. If users aren't password savvy, this is trivially trivial to do. And even if they are, many passwords stringed together is still relatively trivial to brute force. Thus, adding a salt to the password hash and storage mechanism mitigates the possibility that someone could guess a user's password.