Web Application Security

Aleksandar Mitic, Mohammad-Ali Omer

April 2020

1 Regular Expression Denial of Service (ReDoS)

With a ReDos attack we wanted to find someway, using regular expressions, to overload the server by running a regex computes for a long time. Trying to do this we quickly found that when creating an account there is a check to see if the password contains the username. Here the inputted username is uses a regex pattern against the password without any sanitation.

To exploit this we entered the username as *^(a+)+$* and the password as *aaaaaaaaaaaaaaaaaaaa!*. When the regex pattern is run against the password it will first *(aaaaaaaaaaaaaaaaaaaa)* and then look for the ending character *$* which will not match. It then matches *(aaaaaaaaaaaaaaaaaa)(aa)* which will also fail on the ending character. Then it matches *(aaaaaaaaaaaaaaaaaa)(a)(a)* which also fails on the ending character. As you can see here this backtracking will go on for many more combinations and with each *a* we add the number of combinations grows exponentially. This takes up a lot of processing power on the server and can cause a denial of service.

One easy way to mitigate this to escape all the special regex characters like *(,), [,], +, ** and so on by adding *//* in front of them. This will prevent backtracking as the regex engine will look for the absolute value of the input instead of interpreting is as a regex. However as with sanitation in general it is easy to miss some cases. Another way to prevent redos would be to also limit the execution time of the matching. With this solution you might limit some non malicious case however in general it will be very few cases if the time limit is set reasonably.

We choose to only use the escape solution as it was enough in this case. *MDN web docs* provides a function for this here that we used.

2 Timing attack

For our timing attack we wanted to find a protected input which might involve some timing. So naturally we looked at the password field since the website

needs to compare the inputted password to the stored password which is vulnerable to timing attacks of the comparison is implemented badly. We did find that there was a timing difference if we inputted the correct password contrary to inputting a password with the wrong first character. By testing it further we found that the comparison took longer for each new correct character since the comparison traverses a longer part of the password.

With all this information we started to implement the timing attack. Our attack strategy started off with getting the password length since that is the first thing that is compared between the two passwords. So initialise an empty string and add an arbitrary character to the string until we see a big difference in time, this difference is acquired through testing and we found that if the time doubled then we had the right password length in this case. When we have the right length we then go on to setting each possible character in the first position, in this case it was only the digits 0-9. In a more extensive attack we would iterate over the 256 extended ASCII characters. If we found a significant difference between an input and the one previous we knew that we had found the right character for that position. We did this for each position in the password and by the end we had constructed the right password.

A mitigation strategy is to do constant time comparison. Instead of returning false you have different checks you always run for each comparison and change a boolean to false if any of the checks fail. I.e if the password length don't match you set the boolean to false. If any of the characters don't match you set the boolean to false. You also iterate over the max length allowed for a password of the site and wrap around. Then in the end if you return the boolean which will be true only if all of the checks passed and false if at least one of the checks failed. Since you always run the same amount of checks and the comparison is dependent on the max length allowed the comparison will be constant for all passwords.

3 Cross Site Scripting (XSS)

Here we were tasked with trying to find a place where forumerly was vulnerable to a XSS attack.

We started looking for this vulnerability by just looking around on the app and try to find where it allowed input data. One place where this was found was in the feature of posting threads where the user can enter the body of the post. By looking at the code we could then see that the body is the directly injected into the HTML with very poor filtering. For example the filter would remove strings like $<script$, however this could easily be bypassed by entering something like $<scri<scriptpt$ which after the filter is applied results in $<script$.

To exploit this attack we simply created a post containing with the malicious

code which in our case was *<scri<scriptpt>alert("XSS")</script>*. This code does not do anything harmful however it does illustrate that we are able to run javascript in the victims browser. When this is posted, depending the type of malicious code, we either wait for users to find this post or maybe mail a link to it if we are target a specific victim. When the victims load this post the body will be injected into the HTML and the code will run.

With an vulnerability like this the number of attacks are many. For example we could change the design of the website to cause a popup to appear prompting the users to enter their credentials because their session has timed out. This could then be mailed back to us. Another possible attack is to steal their cookie by mailing it to us.

Preventing these kinds of attacks has many pitfalls. The app we used had some filtering however we found ways to get around it. One way to improve the filtering is to use some standard libraries for HTML sanitizing instead of writing your own, for example *DOMPurify* in the case of javascript. These libraries cover most of the pitfalls that there are when filtering HTML.

4 Privilege Escalation via XSS

Firstly we wanted see if the session was reset every time the user logged in or if the same session was kept. Indeed the session did not change and we gathered that if we could change the cookie of another user so that it had our session id we could perform a session fixation and would have access to their profile.

This is where XSS came in. As we noticed in the previous task the discussion threads were vulnerable to XSS attacks and we could input our own script. We knew that this could have a big impact since we could input our own javascript-code changing all sorts of properties like the cookie. So we inputted our own script changing the session id for the login path in the cookie. So each user that would open our discussion thread with the hidden script and then login would login with out session id. We could then use the session id to send requests on behalf of that user. We tried it out by only updating the bio but now you have access to all of the other users profile such as deleting user, posting threads on their behalf etc.

As the session is kept in an HttpOnly cookie we were not able to overwrite it and instead had to add another same-name cookie with a more specific path. This way when the victim would login our cookie would have a more specific path and get sent first.

Regarding the question if we can steal a session cookie, since HttpOnly is set on the session cookie they are not visible to any javascript-code. So we can not use any XSS to steal a cookie of an administrator and login with all of their

privileges.

To mitigate this exploit we changed the server to regenerate the session cookie every time a user logs in. This way session fixation attacks are not possible as even if the attacker set the session cookie for a victim it will become invalid when the victim logs in.

5 Contributions

As the previous lab we wanted to get familiar with the code together to make sure we both had a good understanding to start off. So we sat together to do the first task, ReDoS. When we were both familiar with the source code we split up to take on one task each. Mohammad did the second task, the timing attack, and Aleksandar did the third task, XSS. We still communicated continuously with each other while working separately so that both of us knew exactly what the other one was doing. Lastly we did the last task, Privilege Escalation via XSS, together.