CS5331 Final Project (Group 18)

Study on ReDoS Attack

Huang Kun (A0194962N), Miao Anbang (A0091818X),

Xiao Yikai (A0212237R), Zhang Xinyue (A0212223B)

I. Introduction of ReDoS

Denial of Service (DoS) attack is one of the most well known attacks on the web and dates back to the twentieth century. It is often used to overload a system so that it cannot serve the intended users. Also it can be used as a distraction for other security breaches. Most of such attacks are distributed and thus they are often referred as Distributed Denial of Service (DDoS) attacks. While most of the DDoS attacks are at the network level, application level DDoS attacks, which accounts for 20% of such attacks, is becoming more prevalent [1].

Regular Expression (RegEx) is a character sequence that defines a matching pattern. It is a widely used technique to validate input and detect intrusion, particularly in web technologies. Currently there are two major kinds of RegEx implementations, one uses Non-deterministic Finite Automaton (NFA) while the other one uses Deterministic Finite Automaton (DFA). Commonly used programming languages, such as Java, Python, PHP, Ruby, adopt the NFA implementation due to its better support of more expressive patterns. However, the default greedy mode of NFA could potentially cause severe performance drawbacks.

Regular Expression Denial of Service (ReDoS) attack, which exploits the NFA potential risks, is an application level DoS attack that sends specially designed strings to the target server with imperfect RegEx implementation (Evil RegEx) , in order to cause the system to respond very slowly or even become temporarily unavailable to intended users. In this project, we explore the theory behind the NFA vulnerability, design and implement an experiment to demonstrate the effects of ReDoS attacks and propose two solutions to prevent such attacks.

The rest of the report is organized as follows: Section II gives a detailed explanation of the ReDoS attacks, with a commonly used example. Section III proposes two solutions in order to prevent the ReDoS attacks. Section IV illustrates the experiment that was setup to demonstrate the effects of the attack on a vulnerable server and the solutions to mitigate ReDoS. Section V presents the results from the experiments. Section VI concludes the result with the direction of further research.

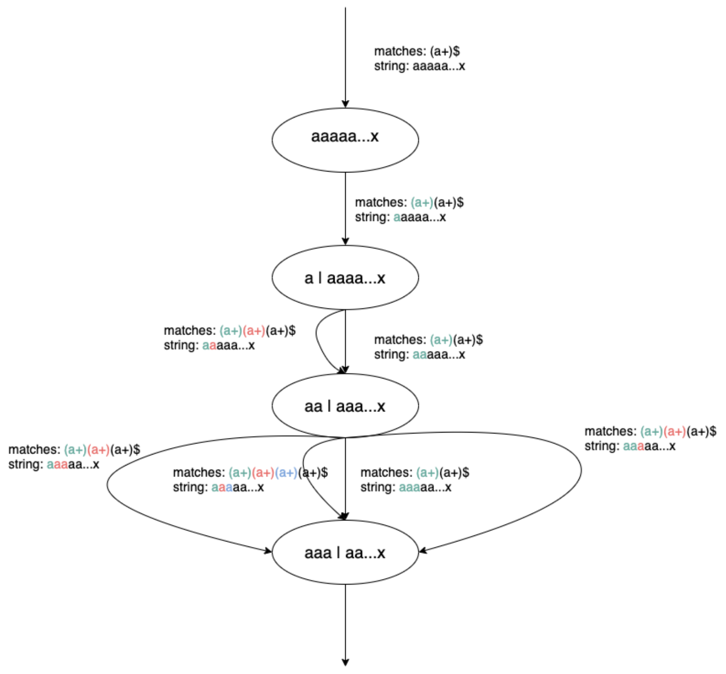
II. Theory behind ReDoS

To illustrate the ReDoS attack theory, two topics are discussed here, RegEx logic (NFA) and attack logic.

1. RegEx logic (NFA):

As mentioned in the introduction, the algorithm which leads to ReDoS is Non-deterministic Finite Automaton (NFA). NFA defines a flow, where in a finite state machine there are many states, and for each state there are multiple possible paths to the next states [2]. NFA will try all those possible paths until a match is found.

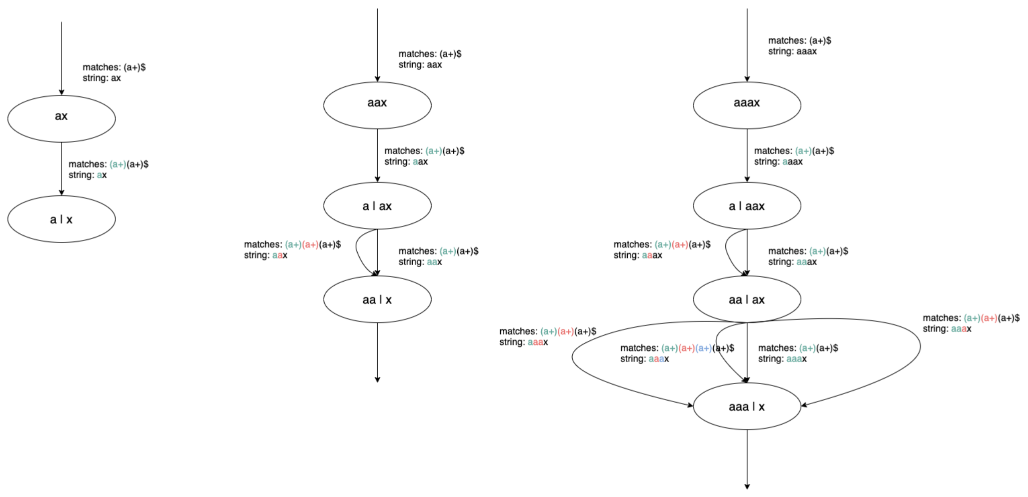
Taken pattern string “aaaaa...x” and pattern “/(a+)+$/” as example, the sequential flow is shown as following:



[Figure 1: NFA RegEx matching flow]

As shown in Figure 1, a matching process is performed for each path to each state until there is a match. For the top state, there will be 1 path to the second state. For the second state, there will be 2 paths to the third state. For the third state, there will be 4 paths to its next, etc. The path quantity is increasing exponentially if no match is found for every path.

By applying the above, the case of string “ax”, “aax”, “aaax” trying to match “/(a+)+$/” will have NFA diagrams as shown below:



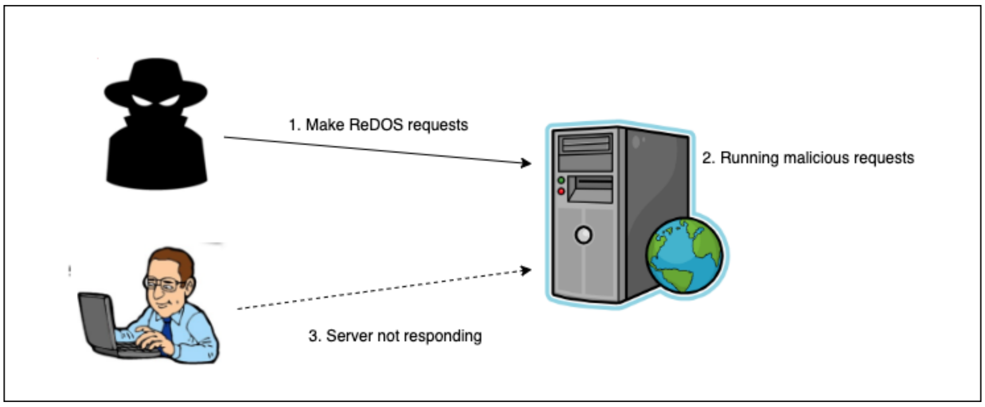
[Figure 2: NFA with for RegEx matching cases]

[left: case for “ax”; center: case for “aax”; right: case for “aaax”]

For “ax”, failure attempt count is 2, “aax” has 4, “aaax” has 8. Thus, conclusion can be made that for a string with pattern “aa...ax” to match pattern “/(a+)+$/” takes 2^n failure attempts to reach the final result.

2. Attack logic

Attackers can utilize this exponentially increasing computanio time of RegEx matching as the attacking logic.



[Figure 3: ReDoS Attack logic]

The attacker can keep sending large amounts of requests to the server with a long evil string. For each request from the attacker, the server will take a long time to process it. As a result, by sending a large quantity of requests, the CPU resource of the server is abnormally occupied and it might not be able to handle requests from intended users.

III. Solution Proposal

Based on the theory above, two solutions are proposed as shown below:

1. Flaw checking

It is not possible to avoid using RegEx, but it's possible to avoid using vulnerable RegEx. ReDoS can only attack on servers having "Evil RegEx", which has patterns like "/(*X*+)+$/". However, the purpose of "/(*X*+)+$/" is to match duplication of "*X*", which can be fulfilled by simply using RegEx of "/(*X*)+$/". Thus, during server-side development, identifying those Evil RegEx and replacing them by safe RegEx can be another solution. Possible identifying methods can be static analysis. However in this project, only the effect of the solution will be discussed.

2. Input validation

Since the computation time is exponential to the length of the input string, if the input length is restricted by the server, the computation time is subsequently restricted. As normally RegEx is utilized in fast checking features such as login rather than documentation scan, this restriction is easy to implement and practical to satisfy industrial needs.

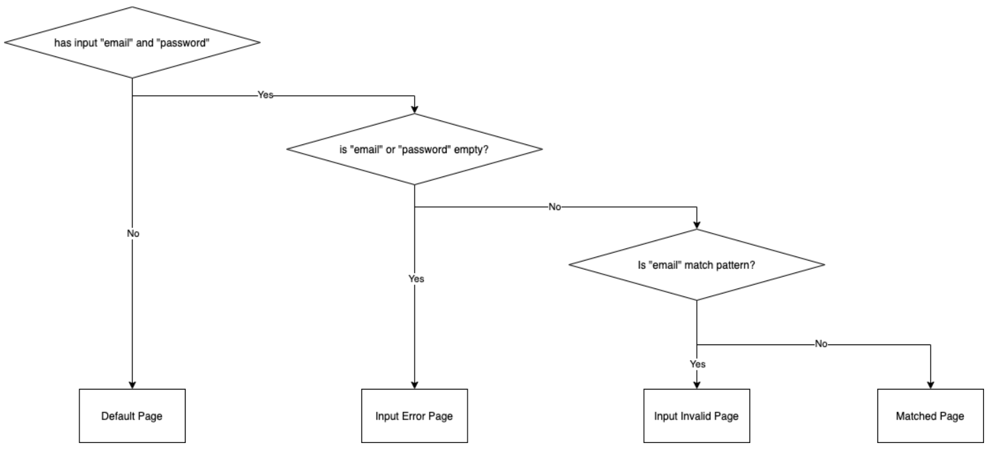
IV. Experiment Design & Setup

To verify the above theory and solutions, an experiment is designed as follows: three parties are required, namely server, attacker and intended user; the server contains three PHP instance pages, including vulnerable server, solution 1 server, solution 2 server.

The setups of the different components in the experiment are:

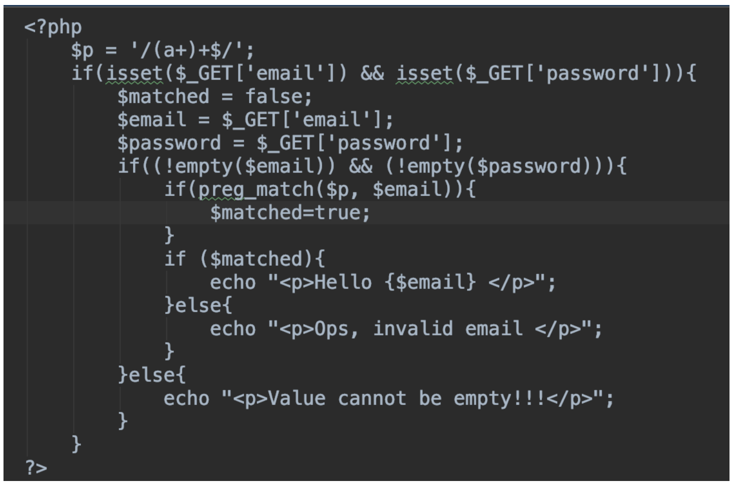
1. Vulnerable server: PHP server with Evil RegEx

The server simulates a scenario of user login, including input fields of "email" and "password". The server has the following user login logic:


has input "email" and "password"
Yes
No
is "email" or "password" empty?
No
Yes
Is "email" match pattern?
No
Yes
Default Page
Input Error Page
Input Invalid Page
Matched Page

[Figure 4: Login server logic]

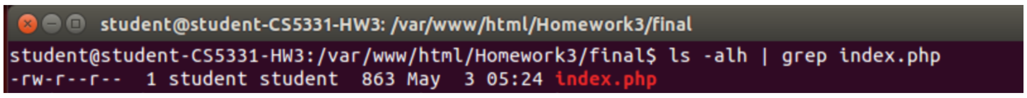
With the above logic, server code with Evil RegEx is shown below:



```php
<?php
    $p = '/(a+)+$/';
    if(isset($_GET['email']) && isset($_GET['password'])){
        $matched = false;
        $email = $_GET['email'];
        $password = $_GET['password'];
        if((!empty($email)) && (!empty($password))){
            if(preg_match($p, $email)){
                $matched=true;
            }
            if ($matched){
                echo "<p>Hello {$email} </p>";
            }else{
                echo "<p>Ops, invalid email </p>";
            }
        }else{
            echo "<p>Value cannot be empty!!!</p>";
        }
    }
?>
```

[Figure 5: Vulnerable server code]

The PHP file is deployed to Apache web server.

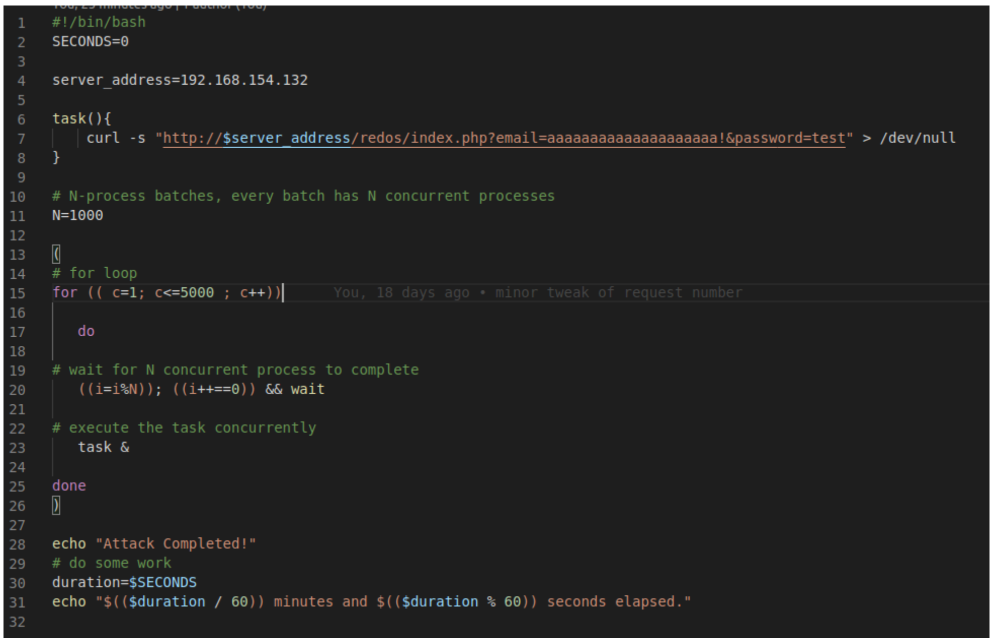


```
student@student-CS5331-HW3: /var/www/html/Homework3/final
student@student-CS5331-HW3:/var/www/html/Homework3/final$ ls -alh | grep index.php
-rw-r--r--  1 student student  863 May  3 05:24 index.php
```

[Figure 6: PHP file location]

2. Attacker: a bash script sending concurrent evil requests in groups

This attacker continuously sends requests to the vulnerable server. The requests are sent concurrently in groups with a fixed group size. After one group finishes sending, another group will be sent out immediately. Implementation of the attacker script is shown as following:



```bash
#!/bin/bash
SECONDS=0

server_address=192.168.154.132

task(){
    curl -s "http://$server_address/redos/index.php?email=aaaaaaaaaaaaaaaaaaaa!&password=test" > /dev/null
}

# N-process batches, every batch has N concurrent processes
N=1000

(
# for loop
for (( c=1; c<=5000 ; c++))       You, 18 days ago • minor tweak of request number

   do

# wait for N concurrent process to complete
   ((i=i%N)); ((i++==0)) && wait

# execute the task concurrently
   task &

done
)

echo "Attack Completed!"
# do some work
duration=$SECONDS
echo "$(($duration / 60)) minutes and $(($duration % 60)) seconds elapsed."
```

[Figure 7: Attacking template]

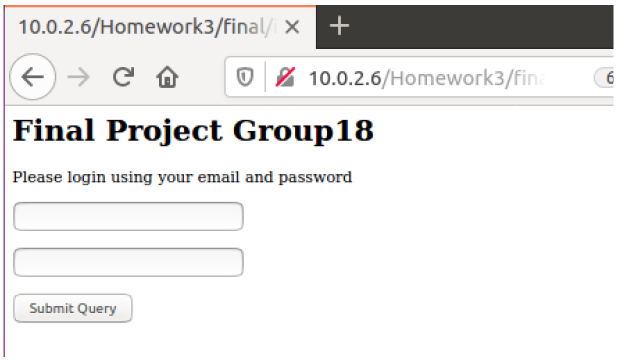
Different request group sizes are tested against the same server setting.

	Group Size	VM Server Setting
1	100	2 Core + 2GB RAM
2	1000	2 Core + 2GB RAM
3	2000	2 Core + 2GB RAM

[Table 1: Vulnerable server test list]

3. Intended user: web browser

The intended user for this lab is a web browser, namely Firefox. The request is inspected using developer tools of Firefox and the response times are to be observed.



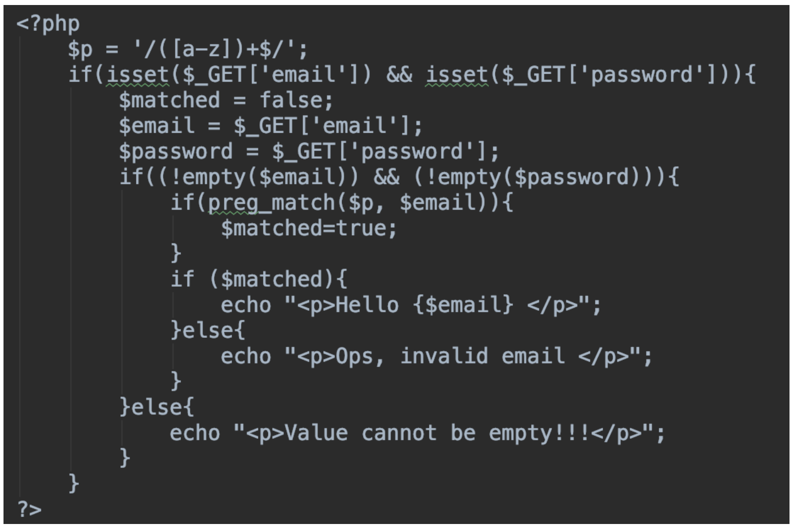
Final Project Group18

Please login using your email and password

[Figure 8: Firefox browsing server]

4. Solution server 1: PHP server without Evil RegEx (Flaw checking)

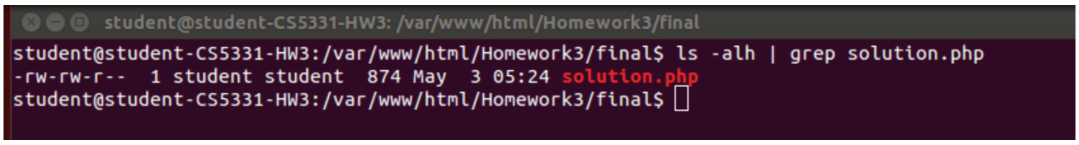
Solution server 1 is an implementation of solution proposal 1. It is designed to use the same user login logic as the vulnerable server, but replacing the Evil RegEx pattern with”/([a-z])+$/”. Code is shown as below:



```php
<?php
    $p = '/([a-z])+$/';
    if(isset($_GET['email']) && isset($_GET['password'])){
        $matched = false;
        $email = $_GET['email'];
        $password = $_GET['password'];
        if((!empty($email)) && (!empty($password))){
            if(preg_match($p, $email)){
                $matched=true;
            }
            if ($matched){
                echo "<p>Hello {$email} </p>";
            }else{
                echo "<p>Ops, invalid email </p>";
            }
        }else{
            echo "<p>Value cannot be empty!!!</p>";
        }
    }
?>
```

[Figure 9: Solution of Flaw checking: fix Evil RegEx]

The solution PHP file is deployed to the same Apache web server as the vulnerable server.



```
student@student-CS5331-HW3: /var/www/html/Homework3/final
student@student-CS5331-HW3:/var/www/html/Homework3/final$ ls -alh | grep solution.php
-rw-rw-r--  1 student student  874 May  3 05:24 solution.php
student@student-CS5331-HW3:/var/www/html/Homework3/final$ 
```

[Figure 10: PHP file location]

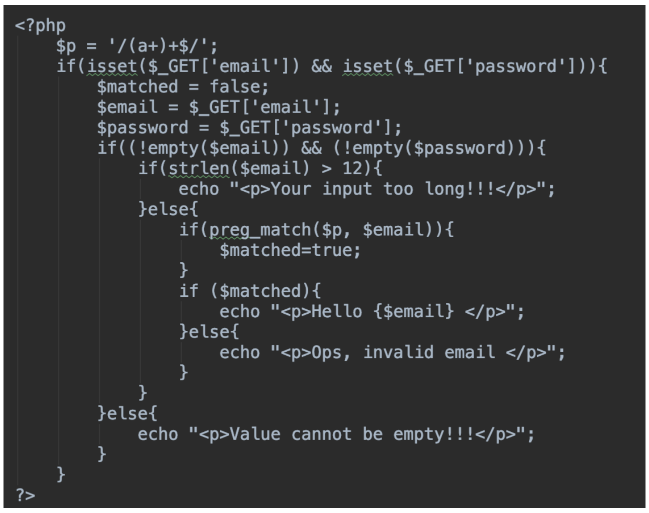
The following setup of Bash script attack is performed to verify the effectiveness of this solution.

	Group Size	VM Server Setting
1	1000	2 Core + 2GB RAM

[Table 2: Solution server 1 test list]

5. Solution server 2: PHP server with length limitation (Input validation)

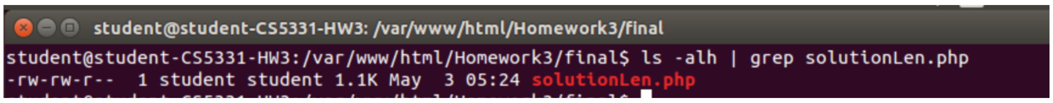
Solution server 2 is an implementation of solution proposal 2. It is designed to use the same logic of the vulnerable server, but restricting the input length to 12. Code is shown as below:



```php
<?php
    $p = '/(a+)+$/';
    if(isset($_GET['email']) && isset($_GET['password'])){
        $matched = false;
        $email = $_GET['email'];
        $password = $_GET['password'];
        if((!empty($email)) && (!empty($password))){
            if(strlen($email) > 12){
                echo "<p>Your input too long!!!</p>";
            }else{
                if(preg_match($p, $email)){
                    $matched=true;
                }
                if ($matched){
                    echo "<p>Hello {$email} </p>";
                }else{
                    echo "<p>Ops, invalid email </p>";
                }
            }
        }else{
            echo "<p>Value cannot be empty!!!</p>";
        }
    }
?>
```

[Figure 11: Solution of Input checking: fix input length]

The solutionLen PHP file is deployed to the same Apache web server as the vulnerable server.



```
student@student-CS5331-HW3: /var/www/html/Homework3/final
student@student-CS5331-HW3:/var/www/html/Homework3/final$ ls -alh | grep solutionLen.php
-rw-rw-r--  1 student student 1.1K May  3 05:24 solutionLen.php
```

[Figure 12: PHP file location]

The following setup of Bash script attack is performed to verify the effectiveness of this solution.

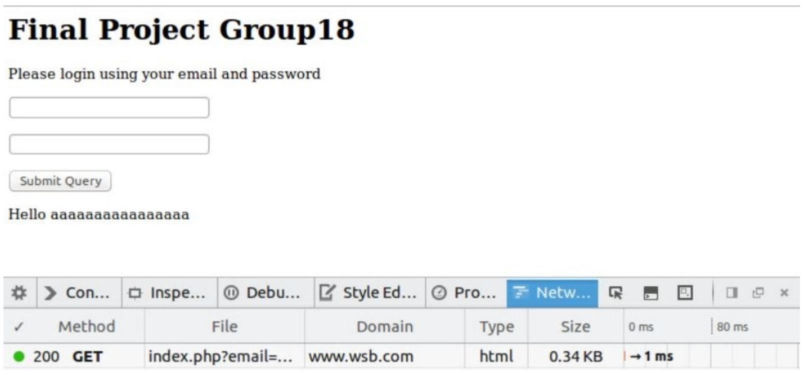
	Group Size	VM Server Setting
1	1000	2 Core + 2GB RAM

[Table 3: Solution server 2 test list]

V. Results

1. This session illustrates how the server response time is impacted by sending the well-crafted input from UI. We took two input strings, both with 16 characters. The first string will match the RegEx, while the second one is a well-crafted input which will make the server run out of combinations to find a match.

- Email input aaaaaaaaaaaaaaaa (16*a) which matches the RegEx



Final Project Group18

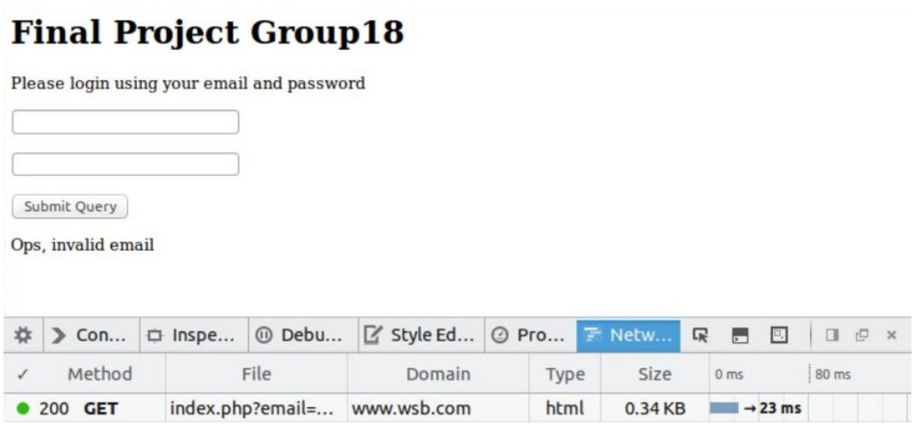
Please login using your email and password

Hello aaaaaaaaaaaaaaaa

Method	File	Domain	Type	Size	0 ms
● 200 GET	index.php?email=...	www.wsb.com	html	0.34 KB	→1 ms

[Figure 13: Response time of the web page while sending matched string]

- Email input aaaaaaaaaaaaaaa! (15*a+!) which does not match the RegEx



Final Project Group18

Please login using your email and password

Ops, invalid email

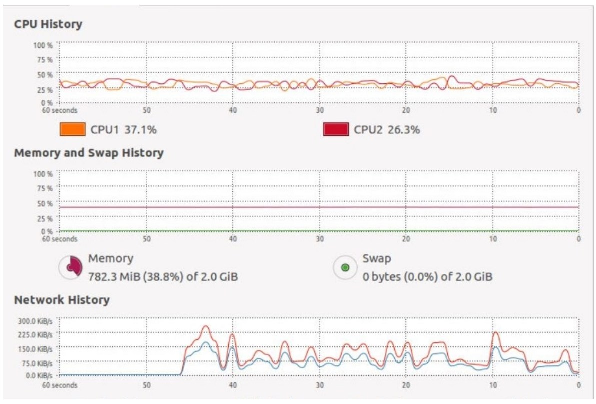
Method	File	Domain	Type	Size	0 ms
● 200 GET	index.php?email=...	www.wsb.com	html	0.34 KB	→23 ms

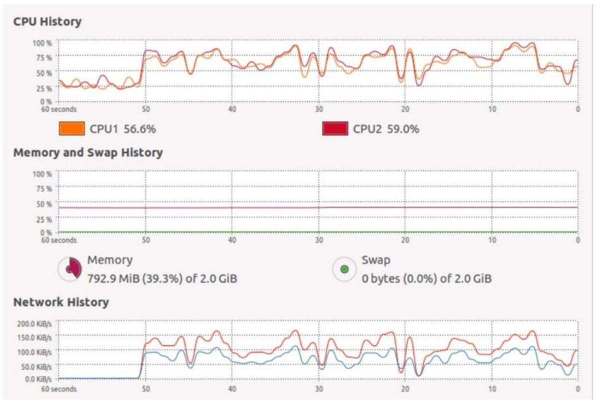
[Figure 14: Response time of the web page while sending well-crafted string]

Above two screenshots demonstrated that the response time was significantly higher when the user sent a well-crafted input to the server which contains an Evil RegEx. The response time of the server was about 23ms when the user sent a well-crafted string, compared to 1ms when the user sent a matched string.

2. This session illustrates how the CPU usage of the server is impacted by sending groups of requests to the server using bash attack scripts. We have conducted comparative tests on the vulnerable server by controlling the input string length and the groups of requests sent to the server. We used two bash scripts, one contained the matched string, while the other one contained the well-crafted input with the same length to send to the server. The CPU usage and network usage were captured while we were sending the concurrent requests to the server.

- 100 batches each containing 100 concurrent requests and the string length is 16

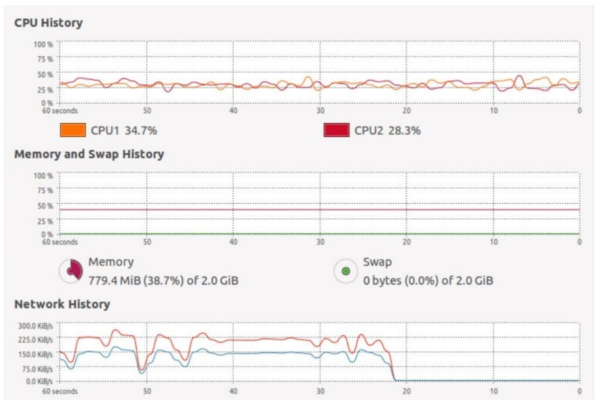


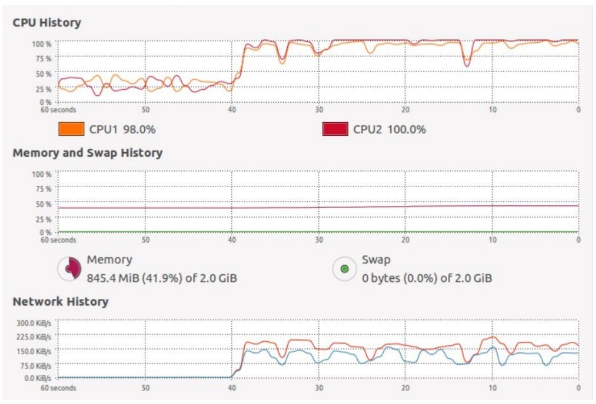


[Figure 15: Input aaaaaaaaaaaaaaaa] [Figure 16: Input aaaaaaaaaaaaaaa!]

We found that with the same length (16 characters) of string, there was not much impact on the server when the input string matched the RegEx. However, there was a significant increase in the CPU usage when we were sending the well-crafted inputs to the servers with 100 requests in a group. At this stage we could see that the CPU had not yet reached full usage.

- 100 batches each containing 1000 concurrent requests and the string length is 16

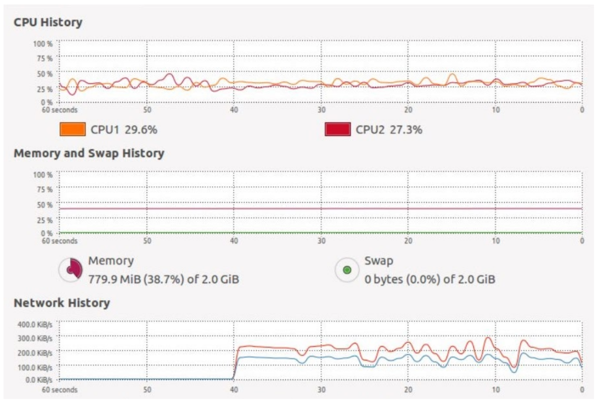


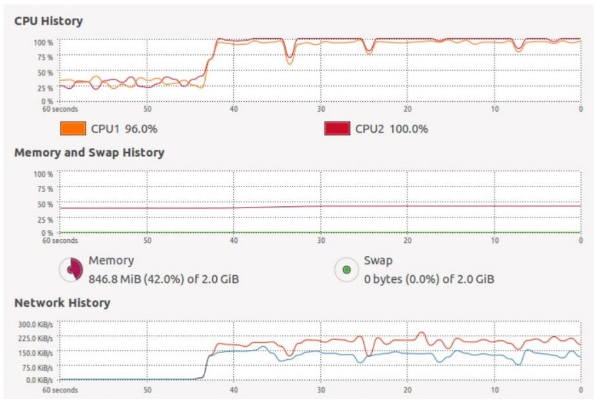


[Figure 17: Input aaaaaaaaaaaaaaaa] [Figure 18: Input aaaaaaaaaaaaaaa!]

We found that with the same length (16 characters) of string, there was not much impact on the server when the input string matched the RegEx even though the number of requests in a group increased 10 times compared to previous. CPU usage reached 100% this time when we were sending the well-crafted inputs to the server with 1000 requests in a group. CPU usage dropped a few times in between only but it increased to the full right after the drop.

- 100 batches each containing 2000 concurrent requests and the string length is 16





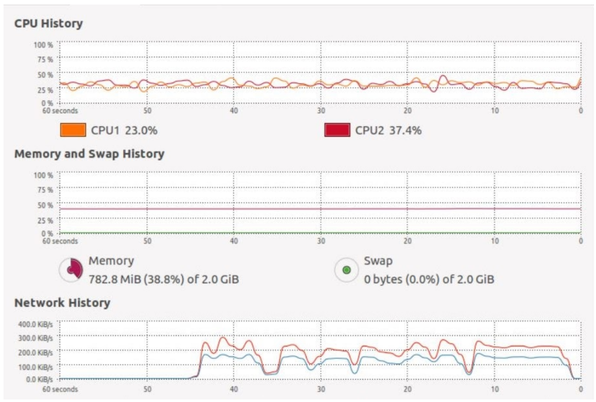
[Figure 19: Input aaaaaaaaaaaaaaaa]

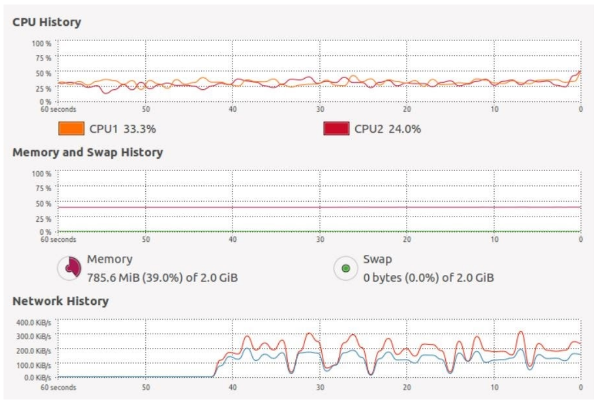
[Figure 20: Input aaaaaaaaaaaaaaa!]

We found that with the same length (16 characters) of string, still there was not much impact on the server when the input string matched the RegEx even though the number of requests in a group increased to 2000. However, both CPU cores reached 100% usage immediately after we triggered the requests in a group of 2000 with the well-crafted inputs. Both CPU cores usage were always stuck to 100% while the intended requests were hitting the server.

3. This session illustrates how our two proposed solutions worked with the same strings we sent to the server. We captured the CPU usage while we were sending the requests in groups of 1000, one with the matched string, the other one with the well-crafted string to the server.

- Proposed Solution 1 - PHP server without Evil RegEx



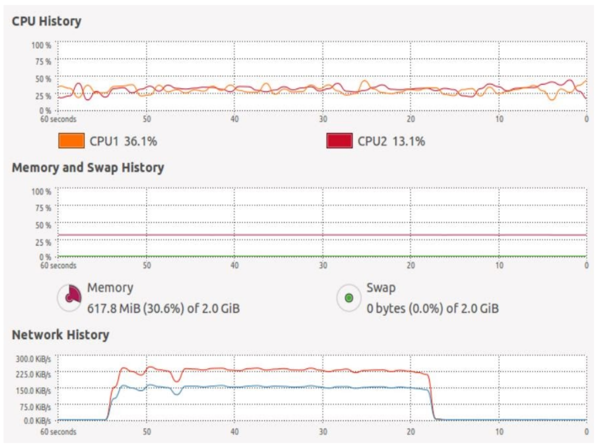


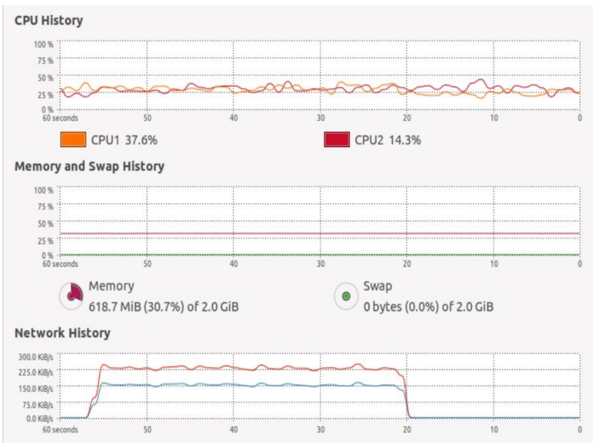
[Figure 21: Input aaaaaaaaaaaaaaaa]

[Figure 22: Input aaaaaaaaaaaaaaa!]

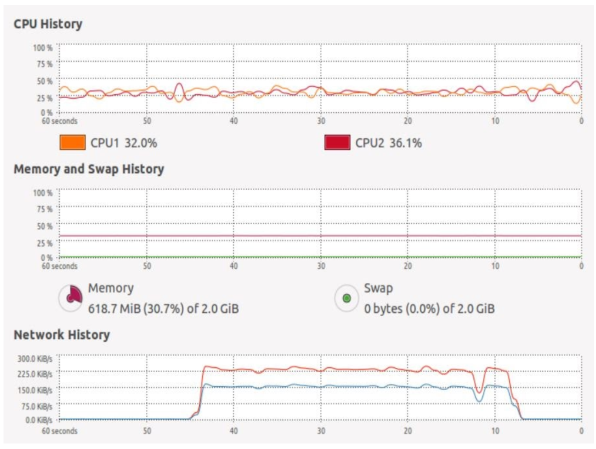
As per our proposal, if we refrain from using the Evil RegEx in the PHP server, there should not be a noticeable increase in the CPU usage even if we send requests with the well-crafted inputs to the server. To prove our solution, we used the same requests, one contained the match string of 16 characters and the other one contained the well-crafted input with the same length. It was observed that there was not much impact on the CPU usage no matter which requests we sent. Both CPUs were running between 20% to 40% usage while the requests reached the server.

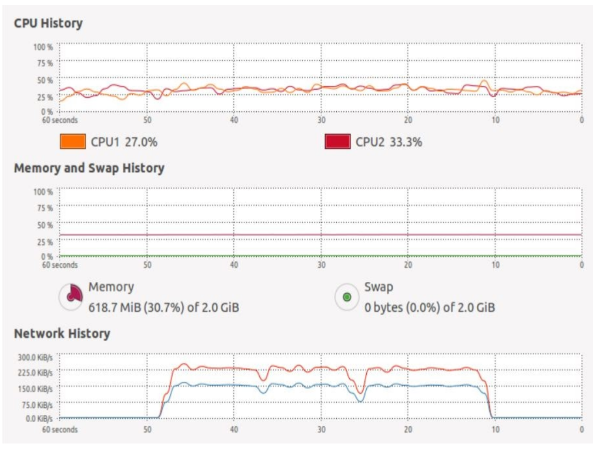
- Proposed Solution 2 - PHP server with Evil RegEx but limiting the length of the input string to 12 character





[Figure 23: Input aaaaaaaaaaaaaaaa] [Figure 24: Input aaaaaaaaaaaaaaa!]





[Figure 25: Input aaaaaaaaaaaa] [Figure 26: Input aaaaaaaaaaa!]

As per our proposal, if we limit the length of the input string to 12 characters or less, the strings which are longer than 12 characters will be filtered out without trying to find a match with the RegEx. Therefore, there should not be a noticeable increase in the CPU usage while the groups of requests containing the well-crafted inputs reach the server. To prove our solution, we used four different requests, two requests were the same one which we used for the first solution. The other two contained the 12-character length string, one holding the matched string while the other one holding the well-crafted string.

While we were sending requests as groups of 1000 with the 16-character length string to the server, we didn't witness an increase of the CPU usage in the server. Both CPUs were running between 20% to 45% usage while the requests reached the server.

While we were sending requests as groups of 1000 with the 12-character length string to the server, it was observed that the CPU usage was slightly higher while we were sending the well-crafted input compared with the matched string but the impact to the CPU was very minimal. But the CPU usage was still within reasonable range. Both CPUs were running between 20% to 50% usage while the requests reached the server.

VI. Conclusion

In this project, a very specific type of Denial of Service attack(DoS attack): Regular Expression Denial of Service (ReDoS) attack is examined and tested. ReDoS exploits the flaw of Regular Expression (RegEx) implementation using Non-deterministic Finite Automaton (NFA), which causes an exponential increase of computation time by sending carefully drafted strings to vulnerable web servers with Evil RegEx.

Besides going through the fundamental theories of NFA implementation of RegEx, a sample attack on a php web server is also designed to showcase the power of ReDoS. There are two php solutions proposed in the experiments, namely flaw checking and input validation.

In the experiment, one vulnerable server with Evil RegEx and two solution servers using different protection techniques are created. All the servers have the exact same specifications.

First, server resource utilizations and client browser response time when web servers are almost idle are noted down. Then ReDoS attacks are simulated using a bash script to send non-stop concurrent requests in groups to the different web servers. Server resource utilizations and client browser response time under attacks are noted down as well .

Comparing the results shown above, it is observed that under very light load, Evil RegEx on the vulnerable server is already causing noticeable delays in terms of client response time compared to those protected servers. While under high load, the vulnerable server with Evil RegEx is overwhelmed by incoming requests. The client response time is significantly higher than those of protected servers. There is no obvious increase in terms of server resource utilizations or client response time of the two protected servers under DoS attacks.

Based on the experiment, it is clear that ReDoS is a very practical threat to web servers using PHP and RegEx expressions. Two simple solutions to ReDoS are proposed and their feasibility is verified.

Further in this topic, there are some improvements that can be done regarding the ReDoS experiment. First, even though the vulnerable server is under high load and response time from the client side is very large compared to normal requests, the server is never really suffocated to death. It is totally doable to launch attacks from multiple machines in a distributed way and bring down the server completely. Second, in our experiments, only PHP pages are tested. It is also possible that other popular web languages such as Java, Python that use NFA engines might suffer the same fate given similar setup.

VII. Reference

[1] Ginovsky, John (27 January 2014). "What you should know about worsening DDoS attacks". *ABA Banking Journal*. Archived from the original on 2014-02-09.)

[2] *Regular expression Denial of Service - ReDoS*. Retrieved May, 2020, from https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS

VIII. Appendix

Code is available at: https://github.com/ZhangXinyue018/cs5331_group_project