Bachelor Degree Project

Performance comparison between Apache and NGINX under slow rate DoS attacks

Authors: Josef Al-Saydali
Mahdi Al-Saydali
Supervisor: Morgan Ericsson
Semester: VT 2020
Subject: Computer Science

Abstract

One of the novel threats to the internet is the slow HTTP Denial of Service (DoS) attack on the application level targeting web server software. The slow HTTP attack can leave a high impact on web server availability to normal users, and it is affordable to be established compared to other types of attacks, which makes it one of the most feasible attacks against web servers. This project investigates the slow HTTP attack impact on the Apache and Nginx servers comparably, and review the available configurations for mitigating such attack. The performance of the Apache and NGINX servers against slow HTTP attack has been compared, as these two servers are the most globally used web server software. Identifying the most resilient web server software against this attack and knowing the suitable configurations to defeat it play a key role in securing web servers from one of the major threats on the internet. From comparing the results of the experiments that have been conducted on the two web servers, it has been found that NGINX performs better than the Apache server under slow rate DoS attack without using any configured defense mechanism. However, when defense mechanisms have been applied to both servers, the Apache server acted similarly to NGINX and was successful to defeat the slow rate DoS attack.

Keywords: Apache, NGINX, slow HTTP, low rate HTTP, RUDY, slow rate DoS, Slowloris

Preface

We would like to thank our supervisor Morgan Ericsson for his valuable feedback and support throughout the work on this project. We would also like to thank our Network security program coordinator Ola Flygt for his continuous guidance and support. Finally, we would like to thank our family for their great support during our study.

Contents

1 Introduction

Slowloris, RUDY, and slow read are DoS attacks that have two special characteristics; they are slow in terms of sending rate and small in terms of bandwidth consumption. They are difficult to be detected by intrusion detection systems and they have been eye-catching to hackers. They target web servers mainly, therefore it is important to configure and equip web servers with defense mechanisms for defeating and mitigating these attacks.

This chapter presents background information on the subject of slow rate DoS attack, the problem to be investigated, and the aimed objective of the project. It also provides the motivation behind the research, related works in this field, and the target group.

1.1 Background

Denial of service attack

Denial of service attack or as commonly known as DoS attack is a cyberattack that targets a computer machine and intends to make it unavailable for legitimate users. The attacker usually sends as much as possible data to the server in a way that makes the server busy with these illegitimate data making it unavailable to the normal users [1][2]. There are many types of DoS attacks, one of them is the application-level attack where the attacker tries to bring a web server down [3]. The focus in this paper is on slow rate DoS attack which is an application layer attack [3].

TCP protocol

To be able to fully understand the attack we should refer to the TCP protocol. Though the attack is targeting the application layer, it uses the TCP protocol in the transport layer for establishing the connection to the WEB application. HTTP uses TCP protocol as its underlying communication channel protocol, so the HTTP data are delivered and encapsulated through TCP packets. TCP is a reliable protocol, meaning the recipient notifies the sender of the received data chunk. TCP connection is established by the three-way handshake: SYN request stand for synchronize sent by the client. SYN-ACK sent by the server once the SYN from the client received. ACK stands for acknowledge. The client acknowledges the receiving of SYN-ACK from the server by send an ACK. Figure 1.1 illustrates TCP three-way handshake.

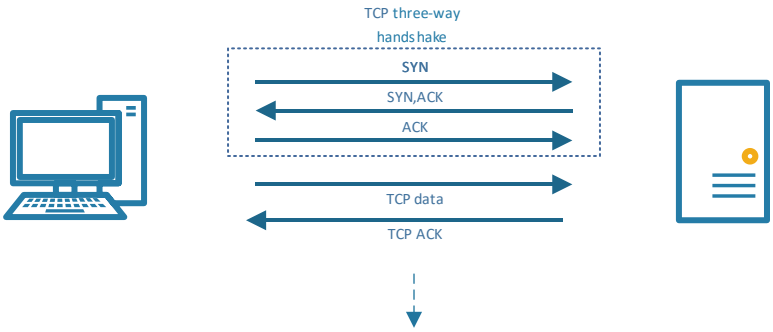

Figure 1.1 TCP three-way handshake

Slow header attack

Slow header attack, also known as slowloris attack, is based on the GET HTTP request. The attacker sends as many as possible incomplete GET requests to the server in order to make all its resources busy. They send the requests at a slow rate so it is not detected by the server's firewall or intrusion detection system. The attack overcomes the issue of the large bandwidth as the requests are in a low size [4]. Usually, web servers have a connection timeout so if the connection is idle for a period of time, it will be closed. Starting from this point, the attacker sends small HTTP headers to the server in order to keep the connection alive. As a result, the server capacity will be full and normal users will not be able to reach the server [5] [8]. Figure 1.2 illustrates slowloris attack.

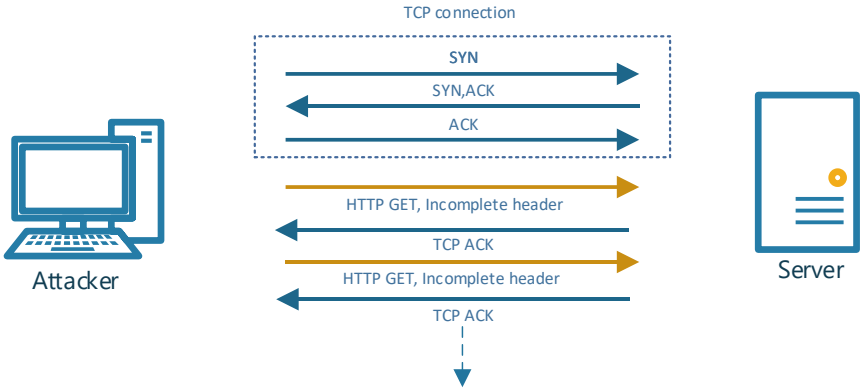

Figure 1.2 Slow HTTP header

Slow body attack

Slow body or RUDY (acronym for R U Dead Yet) attack is similar to slowloris given that the traffic is sent in small size and slowly. However, slow body relies on the POST HTTP request for bringing down the server. First, the attacker sends the head of the POST request completely to the server containing the content-length field which is the size of the body that the client intends to send to the server, usually, is set to a high number in order to exhaust the server. Then, he/she sends the payload which is the body in small chunks at a slow rate to make the connection alive so that it is not terminated by the server [22]. The attacker opens as many as possible connections with the intention of seizing the server's availability thus blocking legitimate users from reaching it [4] [5]. Figure 1.3 illustrates RUDY attack.

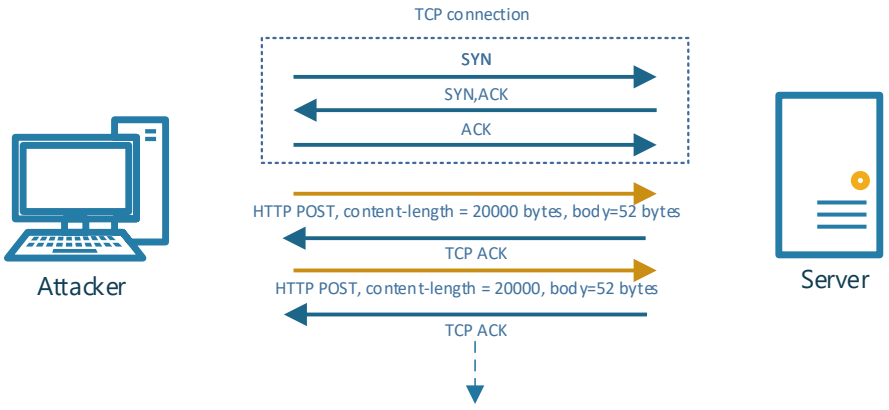

Figure 1.3 Slow HTTP Body

Slow read attack

In the previous two attacks, the request message is not sent completely, but only parts of it are sent. Conversely, in this type of attack, the attacker sends a complete GET request message to the victim server without deceleration. However, the attacker introduces a small window size for receiving data from the server [5]. They proceed in receiving the response in very small chunks and at a slow rate. To make the attack more effective the intruder request a large file such as a large image in order to prolong the connection. The attacker opens as many as possible connections with the purpose of blocking the server from receiving any new requests. As a result legitimate users would not be able to reach the server [4] [23]. Figure 1.4 illustrates slow read attack.

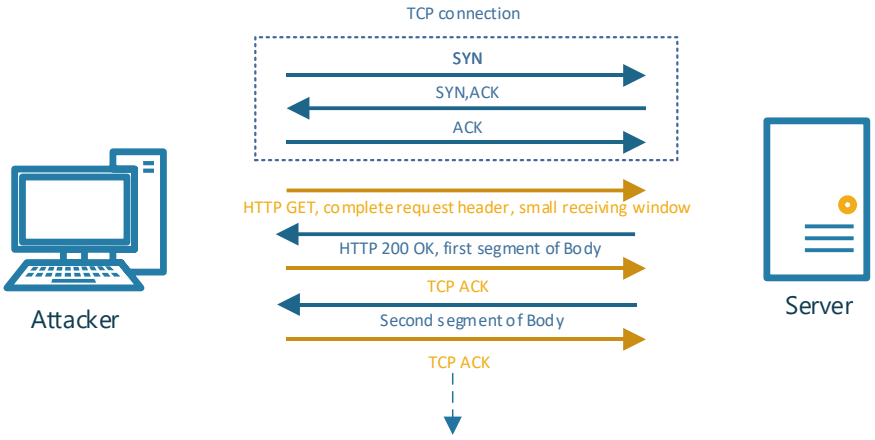

Figure 1.4 Slow HTTP read

1.2 Related work

We have reviewed several works regarding the defense mechanism that is feasible against slow rate attack. One of these articles is [5] written by Suroto, which explains some mitigation methods on several web server applications. The article suggests using some additional modules or WAF (Web application firewall) for Linux OS. One of the most used WAFs on Linux is mod_security, which we have excluded from our experiment since this mitigating tool is configured to be used on the operating system level regardless of the running web server application and that does not serve our goal of comparing the Apache and NGINX servers. Moreover, Suroto does not conduct any experiments in his article.

The article [6] states that the DoS attacks are serious and can be cumbersome for companies and governments. The researchers provide accurate taxonomy by analyzing the slow DoS attacks against web applications and gathering them into type groups based on the common characteristics. The DoS attacks has been categorized into three types: Pending requests DoS like Slowloris and Slow HTTP POST, Long responses and Multilayer DoS. The article is helpful for understanding the mechanism of the different types of dos attacks but there is an absence on describing the performance and the efficiency of those attacks.

Researchers in article [7] developed a defense technique that is useful when detection tools are inadequate since DoS attacks are hard to differentiate from legitimate requests. They begin by characterizing the Dos attacks that targeting the application-layer servers according to the parameters they exploit: Flooding DoS that uses the request flooding technique to overwhelm the server application. Asymmetric DoS attack uses requests that demand high computations on the server side. A DDoS-Shield has been developed with different policy that allow or deny a request. The work showed an enhancement of the application layer victim performance.

Another research [9] has an empirical study on using an attack detection approach for triggering filtering against the traffic of slow HTTP attack. The researchers are finding patterns of slow HTTP attack traffic, if any of them is found while sniffing the traffic to the server, a new firewall will be imposed to limit the attack. The implemented IPS (Intrusion prevention system) is designed to be used on Linux based operating systems, and it has been tested on the Apache server.

While we have found many articles regarding the slow rate attack and the defense techniques, we have noticed a lack in researching the performance of the different web servers against slow HTTP attack, and in comparing the resilience of different web servers against the slow HTTP attack.

1.3 Problem formulation

Many factors play a role while choosing a web server software like cost, performance, and security. In this paper, the concentration will be on the security aspect which can be critical for start-up companies. Specifically, a novel threat that has been one of the largest threats in computer networks; the low rate DoS attack that targets Internet web servers [21]. In this research, an actual low rate DoS attack will be carried upon web servers using virtual machines. The low rate attack will be performed against two types of software: Apache and NGINX. The purpose of the experiment, first, is to test the chosen defense configuration and tools against

slow rate attack and find the suitable configuration for each web server software. The second part is to compare the performance of these two types of servers under slow rate DoS attack in two phases: default configurations and when using mitigation tools and configurations.

1.4 Motivation

In our daily life on the internet we use services that are provided by servers and those servers are vulnerable to many attacks that can make the delivered services unreachable. Many of these services are critical such as airlines flights booking systems, critical hospital services, online payments, and more. Denial of service (aka DoS) attacks affect such systems and render them unreachable to the users. One type of DoS attack is the low rate DoS attack which has been one of the topmost threats in internet security and it is difficult to detect as it makes use of legitimate requests to the server [21]. It targets web servers mainly whereas the attacker sends as many as possible requests to the server at a low rate within a long period of time, chunking up all available connections [4] [5]. Therefore, identifying the most resilient web server software against this type of attack and knowing the suitable configurations to defeat it, play a key role in securing web servers from one of the major threats on the internet [21].

1.5 Objectives

The study will address the following objectives:

O1	Carry out a literature review about the current situation of low rate DoS attack
O2	Implement a low rate DoS on Apache and NGINX servers
O3	Measure the impact of the attack on the Apache and NGINX servers
O4	Analyze the results from comparing the two servers performance

The expectation of this experiment is to come out with a clear comparison between Apache and Nginx with and without mitigation. There is a probability regarding that Nginx will perform better than Apache against the low rate DoS attack. The reason for this anticipation is due to the different architecture of limiting the maximum number of clients.

1.6 Scope/Limitation

The study is limited to low rate DoS attack, meaning that other types of DoS attacks will not be investigated in this paper. In addition, the setup of the experiment is limited to the virtual solution, thus physical devices will not be part of the setup environment.

1.7 Target group

The target group of this paper is cybersecurity professionals and web managers. Besides, the study might be interesting for computer technicians in general. The study will help to choose between the apache and Nginx servers when configuring a fresh server or replacing an old one. Moreover, it can target server administrators when facing slow rate HTTP attack and guide them to configure or replace the server with the aspect of slow rate HTTP tolerance.

2 Method

In order to address the first objective, a literature review regarding low rate DoS attacks has been carried out and presented in chapter 1. For addressing the other research questions a controlled experiment has been conducted. In a controlled experiment a system is tested in a controlled environment that produces quantitative data that is collected and analyzed. In this study, experiments have been carried out for comparing two web servers' performance under specific types of DoS attacks. The comparison has been done in two contexts; the first one in default configurations and without using any mitigation tools and the second context is in using defense and mitigation mechanisms.

Before comparing the two web server software performance against the attacks, the effectiveness of the chosen defense mechanisms will be tested in controlled experiments. Each mechanism will be tested against three types of slow rate DoS attacks which will be stated in chapter 3. The effective mechanisms for each web server against different types of the attack will be chosen to be used in the comparing experiments. Consequently, ineffective defense mechanisms will be dropped and will not be part of the comparing experiments. The experiments will produce quantitative data that will be analyzed to draw a conclusion on which web server software is more resilient and how it behaved under each type of attack.

In each experiment, an attack will be triggered against the web server, and the server availability during the attack will be measured. The attack will be considered successful if the server is unavailable for more than 50% of the attack time. Also, a defense mechanism is considered effective when more than 50% of the attack's impact is overwhelmed.

The term performance in this paper refers to the server's availability to legitimate users. If the server process and respond to a received request within a specific period, a timeout, then the server would be recognized as available. Otherwise if the server did not respond during the timeout, it would be considered as unavailable. The performance of a server within an experiment is considered by taking into account its availability during all period of the attack. The more available the server is during the period, the higher performance it will have.

Data collection has been done by running controlled experiments that produced quantitative data which has been used to produce the results. In the experiments, when an attack is launched on the server, the server's availability has been monitored and the data was saved to a delimited text file along with other information such as the number of closed and pending connections. The data can be reproduced by using commands provided in the Appendix along with technical information presented in chapter three.

2.1 Reliability and Validity

For the experiments to be reliable, similar defense mechanisms have been used in both web server software. In addition, defense mechanisms in both software have been set to equivalent configurations. Having said that, it is not possible to have identical configurations in the two servers, as some options are missing or differently configured. Furthermore, to have a fair comparison both serves have been hosted with the same data that is meant to be served to the end-users such as website pages and images. Also, to ensure the validity of the results, each experiment was repeated five times and the mean values were taken. Additionally, all information regarding the experiments has been provided in chapter 3 such as the environment

setup, the tools that have been used, and the attack types and parameters, thus ensuring reproducibility.

The experiments have been done in a virtual environment with no real users, so there is uncertainty about how the imposed configurations would deal with the real traffic, and whether they would generate false-positive statues or not. The experiments would have been more realistic if a simulation of real users' traffic has been considered. Furthermore, the content hosted in the servers could be chosen according to common content used in web servers to simulate a real-world scenario.

3 Experiment

3.1 Experiments environment

The environment setup contains the following three machines as shown in Figure 3.1:

1. The server: which is the victim machine that will be compromised by the attacker.
2. The observer: a machine that will send legitimate requests to the server and wait for the response with the purpose of monitoring the server.
3. The attacker: a computer that will perform a low rate DoS attack by sending malicious requests to the server.

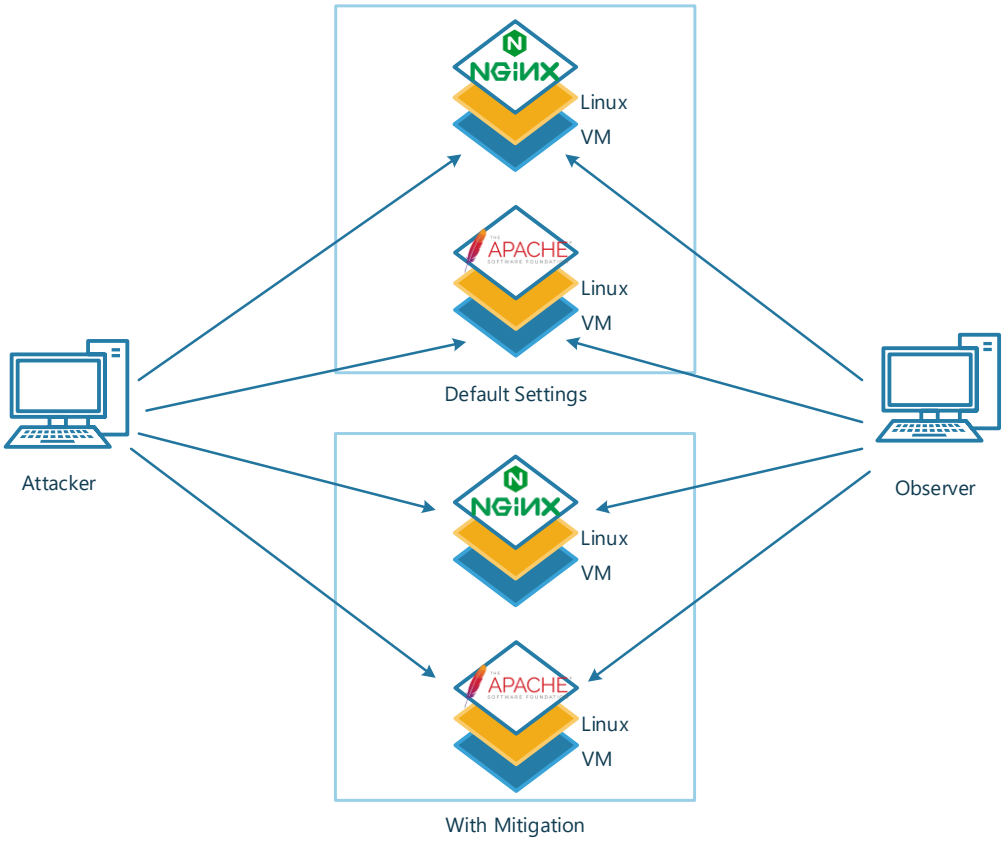

Figure 3.1 Illustration of the experiment setup

The low rate attack will be performed against two types of server software: Apache and Nginx. The purpose of the experiment is to compare the performance of these two types of servers and find the suitable configuration of each type of the two software in order to defend against the slow rate attack and mitigate its impact. Therefore, the attack will be performed using the default configurations for both servers and the impact will be measured. Next, the attack will be performed against the two servers by using the manual configurations this time.

The focus of this experiment is to use mitigation options that are provided within the software and to analyze the effectiveness of the mitigation tools comparably. For the apache server, since the software is built with the architecture of modules which gives flexibility to the software, the server can be configured or have some extra features by using additional modules. Apache modules that have been used in our experiment are: mod_reqtimeout, mod_qos, and

mod_antiloris. For the Nginx server, the options that are provided by the software will be manipulated such as body size, timeouts, and parallel connections.

The experiment has been conducted on virtual machines by using VirtualBox software. The specification for each one of the three VMs that are part of the setup is described in Table 3.1. Web server software that has been used in the experiments are Apache v4.2 and NGINX v1.8. Slowhttptest is the tool that has been used to carry out the attacks against the two web servers.

Machine specification	**Attacker**	**Observer**	**Server**
CPU	Intel core i3 2.4 GHz	Intel core i3 2.4 GHz	Intel core i3 2.4 GHz
RAM	1 GB	1 GB	2 GB
Operating system	Ubuntu 16	Ubuntu 16	Ubuntu 16

Table 3.1 VMs specifications

Server states that will be shown in the results are as the following:

- Closed: means the connection was served and closed and there is no more communications through it.
- Pending: means the incoming connections to the server are put on hold due to server being busy and the server would respond to them whenever it is available.
- Connected: refers to the connections that are still being served and there is data exchange between the server and the client.
- Service available: refers to whether the server under attack is responding or not. If the server respond to a received request within a timeout, it means the server is available. Otherwise if the server did not respond, it means it is not available.

3.2 Mitigation tools

In this section, built-in tools and configurations available in Apache and NGINX servers that will be used in the experiments will be described. The mechanism behind each tool and configuration also will be explained.

3.2.1 NGINX

client_header_timeout

This directive is used to set a maximum period for the server to wait between two consecutive headers. If the client does not send any header data within the time limit, the server will drop the request and send 408 (Request Timeout) response status code back to the client. This directive is useful to defend against slow header attack because this kind of attack relies primarily on sending small chunks of the request header to make the server busy. By activating this directive and setting the timeout to a small number, slow header attack can be defeated successfully. It can be used in both HTTP and server contexts in the configuration file [10] [11]. An example of using this directive in nginx.conf:

```
server {
    client_header_timeout 30s;
    # ...
}
```

client_body_timeout

This directive is similar to *client_header_timeout* but it is dedicated to be used against the illegitimate payload. It is used to set a maximum period for the server to wait between two successive body writes. If the client does not send any body data within the time limit, the server will drop the request and send 408 (Request Timeout) response status code back to the client. This directive is useful to defend against slow body attack because this kind of attack relies primarily on sending small chunks of the payload (body) to make the server busy. By activating this directive and setting the timeout to a small number, slow body attack can be defeated successfully. It can be used in HTTP, location, and server contexts in the configuration file [10] [11]. An example of using this directive in nginx.conf:

```
server {
    client_body_timeout 25s;
    # ...
}
```

client_max_body_size

This directive is useful to protect the server from slow body attacks. It sets the maximum allowed size of the payload of a request to be specified by servers' administrators. By setting this directive to a specific number, it prevents the client from sending requests that exceed the allowed limit. The server relies on the Content-Length field in determining the body size of a request, therefore if the Content-Length number was larger than what has been specified in the configuration a 413 (Request Entity Too Large) response status code will be sent back to the client. By activating this directive and setting the timeout to a small number, slow body attack can be defeated successfully. It can be used in HTTP, location, and server contexts in the configuration file [10]. An example of using this directive in nginx.conf:

```
http {
    client_max_body_size 2m;
    # ...
}
```

Limiting connections

By using directives provided within the `ngx_http_limit_conn_module`, the number of simultaneous connections for each IP address can be limited. When the number of parallel opened connections by a client reaches the specified limit, the server will send 503 Service Unavailable server error response code back to the client [11]. The default error message can be changed by using `limit_conn_status` directive. A shared memory zone should be defined, given a name, and set its size to a specific value by using `limit_conn_zone` directive. Then, by using `limit_conn` directive we point out to the shared memory zone which is `limitbyaddr` in the following example, and set the allowed number of simultaneous connections [10] [12].

An example of using this directive in nginx.conf:

```
http {
  limit_conn_zone $binary_remote_addr zone=limitbyaddr:20m;
  limit_conn_status 429;
  # ...
      Server {
           limit_conn limitbyaddr 40;
           # ...
      }
}
```

3.2.2 Apache

mod_reqtimeout

This module is used to set a timeout for a client request to be received by the apache sever. The module can set a time limit for SSL/TLS handshake and HTTP request. HTTP requests can be divided into HTTP header and HTTP body, so mod_reqtimeout can set a different time limit for the two HTTP parts. Also, mod_reqtimeout is used to set the allowed minimum transfer rate for data that are being received from the client-side. If the client has failed to meet the time frame limit for sending the data, the server will drop the connection data and send a 408 "REQUEST TIMEOUT" error [13]. An example configuration for setting a time limit for HTTP header:

```
RequestReadTimeout header=10-20,minrate=800
```

The directive gives the client a 10-second time frame for sending the first bytes of the HTTP request. If the client has sent the first segment of the request then check the transfer rate to be a least 800 bytes/s. If the client did not send the complete request in 10 seconds, increase the time out one second for each received 800 bytes but no more than 20 seconds [14].

The upper limit can be omitted, and we will only have a strict time limit :

```
RequestReadTimeout header=10,minrate=800
```

The same rules apply for the body portion of the HTTP :

```
RequestReadTimeout body=10,minrate=800
```

mod_qos

The module is used to impose the quality of service concept on the Apache webserver by setting rules that collaborate cumulatively to impose different priorities to the received HTTP requests. By setting different configurations using the qos module we can eliminate unnecessary connections and keep the server from being overwhelmed with too many connections, since each connection requires a process or a thread that consumes resources like CPU and ram. Too many processes will lead to the unavailability of the server from the client side "denial of service" DoS and here comes mod_qos to mitigate that impact on the apache server [15].

mod_qos can limit the number of connections that a single IP can establish and also can set the number of the IPs that can have connections to the server simultaneously. For the different server pages or resources, mod_qos can also manage them by setting restrictions for

the number of the HTTP requests to a specific resource. With mod_qos we can decide to accept or drop a request from a client that does not meet the minimum specified rate:

```
 #QS_SrvRequestRate                              120
```

Here we are requiring that the client must have a rate of 120 bytes/sec at least so their request can be processed. There are also some conditions that can take place when the connections hit the limit, like disabling the keep-alive header when the connections reach the indicated number (In our example it is 600):

```
#QS_SrvMaxConnClose                             600
```

mod_qos with default configuration:

```
<IfModule qos_module>
# minimum request rate (bytes/sec at request reading):
#QS_SrvRequestRate                              120
# limits the connections for this virtual host:
#QS_SrvMaxConn                                  100
# allows keep-alive support till the server reaches 600 connections:
#QS_SrvMaxConnClose                             600
# allows max 50 connections from a single ip address:
#QS_SrvMaxConnPerIP                              50
</IfModule>
```

mod_antiloris

This module is a mitigation tool designed to target slowLoris attack (slow HTTP header). The module mitigation concept depends on limiting the number of connections that a single user can have to the Apache server. Antiloris depends on the IP address for defining the user, so we can set configurations that will block any IP that crosses the threshold of the connection limit.. Example of antiloris configuration:

```
<IfModule antiloris_module>
    IPTotalLimit 16
    IPOtherLimit 8
    IPReadLimit  8
    IPWriteLimit 8
</IfModule>
```

IPTotalLimit means all connections in any state, while IPOtherLimit for idle connections, IPReadLimit for connections in reading mode, and IPWriteLimit for connections in write mode.[16]

3.3 Conducted experiments

3.3.1 Nginx server

In this section, experiments that were conducted on the Nginx server by configuring the built-in features in order to come up with the most suitable configuration for defeating slow rate DoS attack will be described. The configurations that has been used in this section are as follow.

Using timeouts

This technique is used to set a maximum period for the server to wait between two consecutive data. If the client does not send any data within the time limit, the server will drop the request and send 408 (Request Timeout) response status code back to the client. This is useful to defend against slow header and slow body attacks because this kind of attacks rely primarily on sending small chunks of the data to make the server busy. By activating this technique and setting the timeout to a small number, slow header and body attack can be defeated successfully.

Limiting connections

Since the number of simultaneously opened connections is an important parameter of a successful low rate attack, limiting this number can be very useful to protect the server from such attacks. Nginx software provides administrators with the ability to limit the number of simultaneous connections that are opened by each client based on their IP address by using limit_conn directive.

Limiting payload size

This method is useful to protect the server from slow body attacks. By making use of it, the maximum allowed size of the payload of a request can be specified by servers' administrators. By setting a specific number, this method prevents the client from sending requests that exceed the allowed limit. The server relies on the Content-Length field in determining the body size of a request, therefore if the Content-Length number was larger than what has been specified in the configuration a 413 (Request Entity Too Large) response status code will be sent back to the client. By activating this technique and setting the timeout to a small number, slow body attack can be defeated successfully.

Experiment 1

Using header timeout: is effective when it is used against slow header attack where the attacker sends HTTP request headers at a slow rate. Therefore, it has been tested against slow header attack only. Table 3.2 shows the parameters that have been used in the experiment. The following directive has been added to the Nginx conf file in the HTTP context which sets the client header timeout to 5 seconds, this means that if the client did not send any header related data within five seconds, the request will be dropped out.

```
client_header_timeout 5s;
```

Test parameters	
Test type	SLOW HEADERS
Number of connections	10000
Verb	GET
Content-Length header value	4096
Extra data max length	52
Interval between follow up data	10 seconds
Connections per seconds	1000
Timeout for probe connection	3
Target test duration	240 seconds
Using proxy	no proxy

Table 3.2 slow header attack 10000 connections

Limiting connections: The following configuration limits simultaneous connections from one IP address to 10 simultaneous connections. If a client tries to open more than ten parallel connections a 429 error will be sent back to the client.

limit_conn_zone $binary_remote_addr zone=addr:10m;

limit_conn addr 10;

limit_conn_status 429;

Experiment 2

Using body timeout: is effective when it is used against slow body attack where the attacker sends a complete HTTP request to the server but sends the body in small chunks at a slow rate. Therefore, it has been tested against slow body attack only. Table 3.3 shows the parameters that have been used in the experiment. The following directive has been added to the Nginx conf file in the HTTP context which sets the client body timeout to 5 seconds. This means that if the client did not send any body-related data within five seconds, the request will be dropped out.

```
client_body_timeout 5s;
```

Test parameters	
Test type	SLOW BODY
Number of connections	10000
Verb	POST
Content-Length header value	4096
Extra data max length	50
Interval between follow up data	5 seconds
Connections per seconds	100
Timeout for probe connection	3
Target test duration	240 seconds
Using proxy	no proxy

Table 3.3 slow body attack 10000 connections

Limiting connections: The following configuration limits simultaneous connections from one IP address to fifty simultaneous connections. If a client tries to open more than fifty parallel connections a 429 error will be sent back to the client.

```
limit_conn_zone $binary_remote_addr zone=addr:10m;
limit_conn addr 50;
limit_conn_status 429;
```

Limiting body size: The following configuration limits the body size of an HTML request to 3 kilobytes by checking the content-length field, thus if it was more than 3KB a 413 error will be sent back to the client telling them that the request entity is too large.

```
client_max_body_size 3k;
```

Experiment 3

Limiting connections: The following configuration limits simultaneous connections from one IP address to five simultaneous connections. If a client tries to open more than five parallel connections, a 429 error will be sent back to the client.

```
limit_conn_zone $binary_remote_addr zone=addr:10m;
limit_conn addr 5;
limit_conn_status 429;
```

Test parameters	
Test type	SLOW READ
Number of connections	3000
Receive window range	512 – 1024
Pipeline factor	3
Read rate from receive buffer	32 bytes / 5 sec
Connections per seconds	1000
Timeout for probe connection	3
Target test duration	240 seconds
Using proxy	no proxy

Table 3.4 slow read attack 3000 connections

3.3.2 Apache server

This section explains and gives details regarding the experiments that were conducted on the Apache server. The section clarifies the different configurations of the apache modules that have been used to mitigate the three types of slow rate DoS attacks.

mod_reqtimeout

Setting a timeout for the HTTP header or body to be completed can highly mitigate the slow DOS attack. In this test, we have set the time limit for the header to be received completely to 8

seconds, and for the body to 10 seconds. The sender should have a minimum speed of 800 bytes/second so it can extend the timeout to the upper limit:

```
RequestReadTimeout header=2-8,minrate=800
RequestReadTimeout body=2-10,minrate=800
```

Reqtimeout module cannot deal with slow read attacks since with this type of attack the server receives an illegitimate request with a complete HTTP header and within the time limit.

mod_qos

By this module, we have set the number of connections that the server can handle to 1000. Also, when the number of connections reaches 600 the qos module will disable the keep-alive messages. Each connection should at least have a speed of 120 bytes/second otherwise the connection will be closed. The maximum connections of every single IP have been set to 50.

Configuration snippet of mod_qos:

```
#minimum request rate (bytes/sec at request reading 120):
 QS_SrvRequestRate                                  120
# limits the connections for this virtual host:
 QS_SrvMaxConn                                      1000
# allows keep-alive support till the server reaches 600 connections:
 QS_SrvMaxConnClose                                 600
# allows max 50 connections from a single ip address:
 QS_SrvMaxConnPerIP                                  50
```

mod_antiloris

Antiloris is designed to deal with slow DOS attacks that are coming from a single IP. So this module can keep track of the number of connections opened by each IP. We have set IPTotalLimit to 16, while IPOtherLimit (idle connections), IPReadLimit (read mode), and IPWriteLimit (write mode) have been set to 8;

```
<IfModule antiloris_module>
    IPTotalLimit 16
    IPOtherLimit 8
    IPReadLimit  8
    IPWriteLimit 8
</IfModule>
```

Experiment 4

In this experiment, Apache server will be tested under slow header attack by opening 500 illegitimate connections to the server and measuring the server's response during the attack. mod_reqtimeout, mod_qos and mod_antiloris will be used individually to test their effectiveness against slow header attack. Table 3.5 shows the parameters that have been used in the experiment.

Test parameters	
Test type	SLOW HEADERS
Number of connections	500
Verb	GET
Content-Length header value	4096
Extra data max length	52
Interval between follow up data	10 seconds
Connections per seconds	200
Timeout for probe connection	3
Target test duration	240 seconds
Using proxy	no proxy

Table 3.5 slow header attack 500 connections

Experiment 5

In this experiment, Apache server will be tested under slow body attack by opening 500 illegitimate connections to the server and measuring the server's response during the attack. mod_reqtimeout, mod_qos and mod_antiloris will be used individually to test their effectiveness against slow body attack. Table 3.6 shows the parameters that have been used in the experiment.

Test parameters	
Test type	SLOW BODY
Number of connections	500
Verb	POST
Content-Length header value	4096
Extra data max length	22
Interval between follow up data	10 seconds
Connections per seconds	200
Timeout for probe connection	3
Target test duration	240 seconds
Using proxy	no proxy

Table 3.6 slow body attack 500 connections

Experiment 6

In this experiment, Apache server will be tested under slow header attack by opening 500 illegitimate connections to the server and measuring the server's response during the attack. mod_qos and mod_antiloris will be used individually to test their effectiveness against slow

header attack, while mod_reqtimeout will not be used in this experiment since the slow read attack sends a complete request so mod_reqtimeout is not feasible to stop this attack. Table 3.6 shows the parameters that have been used in the experiment.

Test parameters	
Test type	SLOW READ
Number of connections	500
Receive window range	512 – 1024
Pipeline factor	3
Read rate from receive buffer	32 bytes / 5 sec
Connections per seconds	100
Timeout for probe connection	3
Target test duration	240 seconds
Using proxy	no proxy

Table 3.7 slow read attack 500 connections

3.3.3 Comparison between Apache and NGINX

3.3.3.1 Before mitigation (default settings)

Experiment 7

In this experiment, Apache and NGINX servers will be tested under slow header attack with both small and large number of opened connections in order to compare the performance of the two servers against each other without using any defending mechanism. This is done, first, by opening 500 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.5 shows the parameters that have been used.

Second, Apache and NGINX servers will be tested under slow header attack with a large number of opened connections in order to compare the performance of the two servers against each other without using any defending mechanism. This is done by opening 10000 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.2 shows the parameters that have been used.

Experiment 8

Apache and NGINX servers will be tested under slow body attack with both small and large number of opened connections in order to compare the performance of the two servers against each other without using any defending mechanism. This is done, first, by opening 500 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.6 shows the parameters that have been used.

Then, Apache and NGINX servers will be tested under slow body attack with a large number of opened connections in order to compare the performance of the two servers against each other without using any defending mechanism. This is done by opening 10000 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.3 shows the parameters that have been used.

Experiment 9

In this experiment, Apache and NGINX servers will be tested under slow read attack with both small and large number of opened connections in order to compare the performance of the two servers against each other without using any defending mechanism. This is done, first, by opening 500 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.7 shows the parameters that have been used.

Second, Apache and NGINX servers will be tested under slow read attack with a large number of opened connections in order to compare the performance of the two servers against each other without using any defending mechanism. This is done by opening 3000 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.4 shows the parameters that have been used.

3.3.3.2 After mitigation

Nginx configuration

The following directives have been added to the Nginx conf file in the HTTP context which sets the client header and body timeout to 5 seconds. This means that if the client did not send any data within five seconds, the request will be dropped out.

```
client_header_timeout 5s;
client_body_timeout 5s;
```

The following configuration limit simultaneous connections from one IP address to fifty simultaneous connections. If a client tries to open more than fifty parallel connections a 429 error will be sent back to the client.

```
limit_conn_zone $binary_remote_addr zone=addr:10m;
limit_conn addr 50;
limit_conn_status 429;
```

The following configuration limits the body size of an HTML request to 3 kilobytes by checking the content-length field, thus if it was more than 3KB a 413 error will be sent back to the client telling them that the request entity is too large.

```
client_max_body_size 3k;
```

Apache configuration

We have enabled the three selected modules with the following configurations:

Reqtimeout module:

```
RequestReadTimeout header=2-8,minrate=800
RequestReadTimeout body=2-10,minrate=800
```

Qos module:

```
<IfModule qos_module>
  # minimum request rate (bytes/sec at request reading 120):
  QS_SrvRequestRate                                  120
  # limits the connections for this virtual host:
  QS_SrvMaxConn                                      1000
```

```
  # allows keep-alive support till the server reaches 600 connections:
  QS_SrvMaxConnClose                                600
  # allows max 50 connections from a single ip address:
  QS_SrvMaxConnPerIP                                 50
</IfModule>
```

Antiloris module:

```
<IfModule antiloris_module>
    IPTotalLimit 16
    IPOtherLimit 8
    IPReadLimit  8
    IPWriteLimit 8
</IfModule>
```

Experiment 10

The aim of this experiment is to test Apache and NGINX servers under slow header attack with both small and large number of opened connections in order to compare the performance of the two servers against each other by using a built-in defend mechanism. This is done, first, by opening 500 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.5 shows the parameters that have been used.

Then, Apache and NGINX servers will be tested under slow body attack with a large number of opened connections in order to compare the performance of the two servers against each other without using any defending mechanism. This is done by opening 10000 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.2 shows the parameters that have been used.

Experiment 11

Apache and NGINX servers will be tested under slow header attack with both small and large number of opened connections in order to compare the performance of the two servers against each other by using a built-in defend mechanism. This is done, first, by opening 500 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.6 shows the parameters that have been used.

Second, Apache and NGINX servers will be tested under slow body attack with a large number of opened connections in order to compare the performance of the two servers against each other without using any defending mechanism. This is done by opening 10000 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.3 shows the parameters that have been used.

Experiment 12

In this experiment, Apache and NGINX servers will be tested under slow header attack with both small and large number of opened connections in order to compare the performance of the two servers against each other by using a built-in defend mechanism. This is done, first, by

opening 500 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.7 shows the parameters that have been used.

Then, Apache and NGINX servers will be tested under slow body attack with a large number of opened connections in order to compare the performance of the two servers against each other without using any defending mechanism. This is done by opening 3000 illegitimate connections to the servers and measuring the server's response during the attack. Table 1.4 shows the parameters that have been used.

4 Results

4.1 Nginx configuration

In this section results of Nginx configuration experiments that has been used in order to come up with the most suitable configuration for defeating slow rate DoS attack will be presented, which are timeouts, limiting connections, and limiting size.

Experiment 1

Header timeout: The attack was successful against the Nginx server before activating the header timeout as shown in Figure 4.1. After activating the header timeout the attack was rendered unsuccessful. The following directive has been added in the Nginx conf file which sets the client_header_timeout to 5 seconds: *client_header_timeout 5s;* The attack was unsuccessful against the Nginx server after activating header timeout and the server was available during the attack as shown in Figure 4.2.

Limiting connections: Slow header attack was successful against Nginx server before applying connection limits as shown in Figure 4.1. After applying connections limits the attack was also successful. Limiting the number of simultaneous connections did not improve the defense against the slow header attack as shown in Figure 4.3.

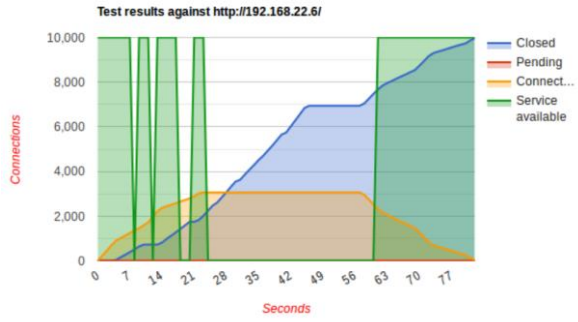

Figure 4.1 Nginx server under slow header attack in default configuration

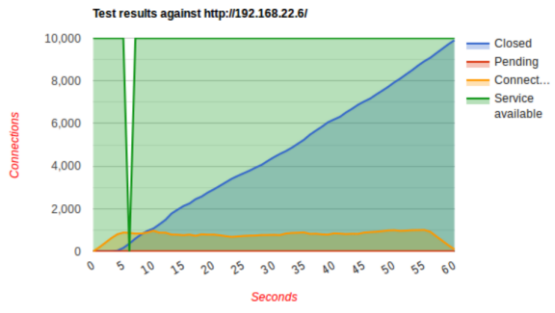

Figure 4.2 Nginx server under slow header attack after using header timeout

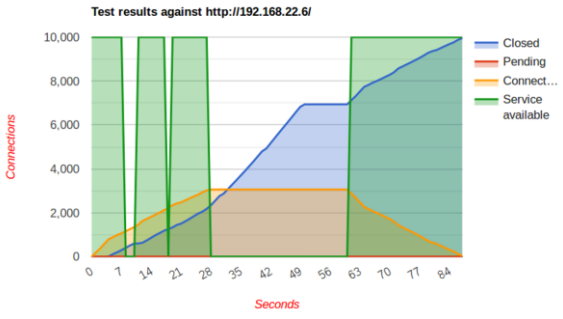

Figure 4.3 Nginx server under slow header attack after limiting connections

Experiment 2

Body timeout: The attack was successful against the Nginx server before applying mitigation as shown in Figure 4.4. After activating body timeout the attack was rendered unsuccessful. The following directive has been added in the Nginx conf file which sets the client body timeout to 5 seconds: *client_body_timeout 5s.* The attack was unsuccessful against the Nginx server after activating body timeout as shown in Figure 4.5

Limiting connections: Slow body attack was successful against Nginx server before applying connection limits as shown in Figure 4.4. After applying connections limits the attack was rendered unsuccessful. Limiting the number of simultaneous connections did improve the defense against the slow body attack as shown in Figure 4.6.

Limiting body size: Slow body attack was successful against the Nginx server before applying the body size limit as shown in Figure 4.4. After applying body size limits the attack was rendered unsuccessful. Limiting body size did improve the defense against the slow body attack as shown in Figure 4.7.

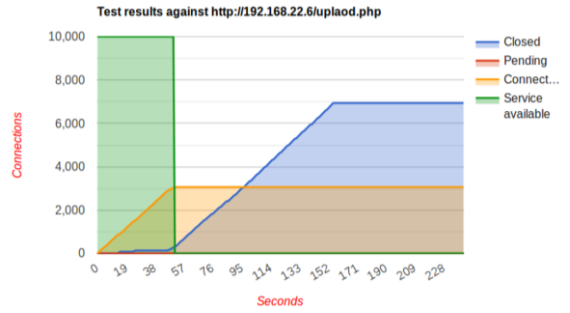

Figure 4.4 Nginx server under slow body attack in default configuration

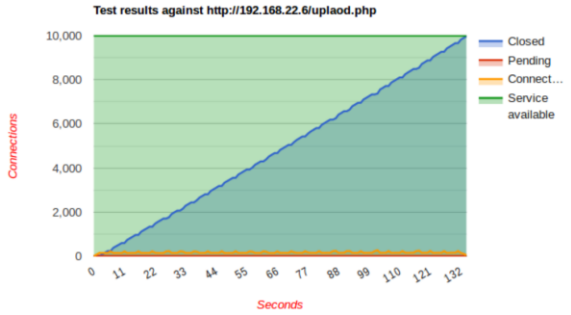

Figure 4.5 Nginx server under slow body after using body timeout

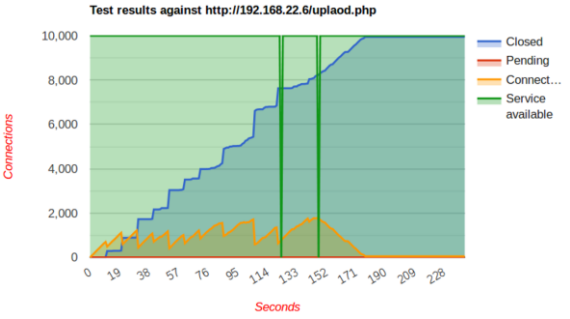

Figure 4.6 Nginx server under slow body attack after limiting connections

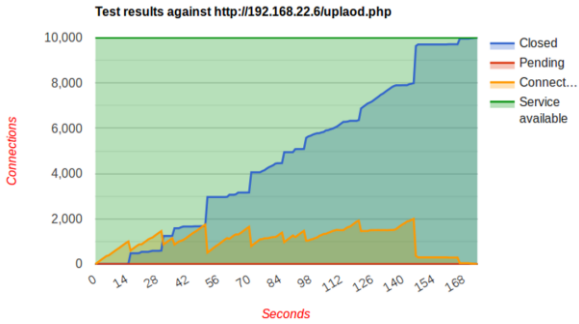

Figure 4.7 Nginx server under slow body attack after limiting body size

Experiment 3

Limiting connections: Slow read attack was successful against the Nginx server before applying connection limits as shown in Figure 4.8. After applying connections limits the attack was rendered unsuccessful. Limiting the number of simultaneous connections did improve the defense against the slow read attack as shown in Figure 4.9.

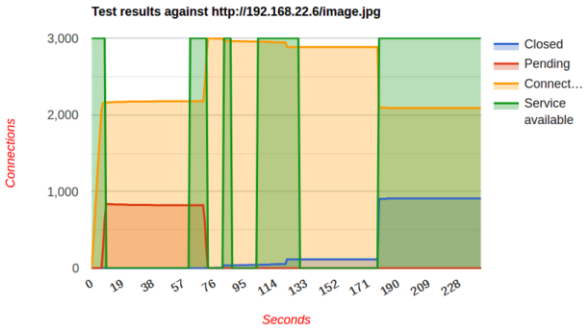

Figure 4.8 Nginx server under slow read attack in default configuration

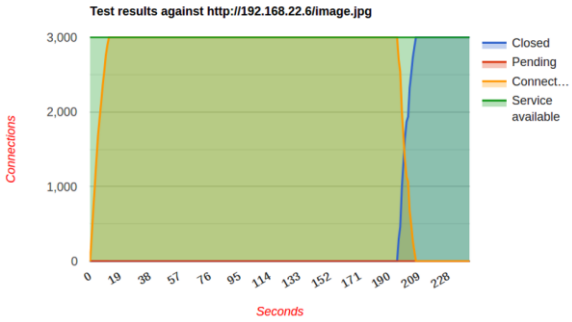

Figure 4.9 Nginx server under slow read attack after limiting connections

4.2 Apache Configuration

Experiment 4

Before enabling the reqtimeout module, the slow header attack succeeded in bringing the Apache server down so it was a successful denial of service attack. We can see in Figure 4.10 that the server was available in the first seconds of the attack, then all the connections have been occupied. With enabling the reqtimeout module, we can see enhancement in the apache server mitigation against the slow header attack, but still, we can see a denial of server between the 6^{th}- 8^{th} seconds.

We can see that the qos (quality of service) module can handle slow header attack efficiently. Also, it can be noticed that the qos module has closed 450 of the connections, since we have specified a maximum of 50 connections for each IP.

After enabling the antiloris module, it has been noticed that the impact of slow header attack has been reduced significantly. Also, it can be noticed that the antiloris module has closed most of the connections, while 8-9 connections have remained.

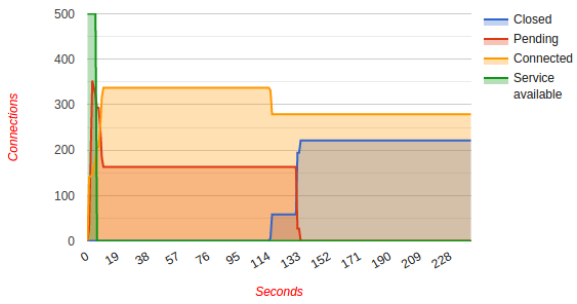

Figure 4.10 Apache server under slow header attack in default configuration

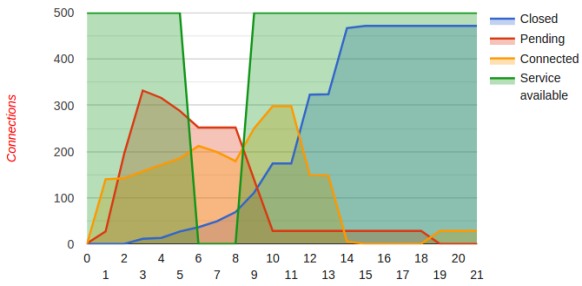

Figure 4.11 Apache server under slow header attack after enabling reqtimeout

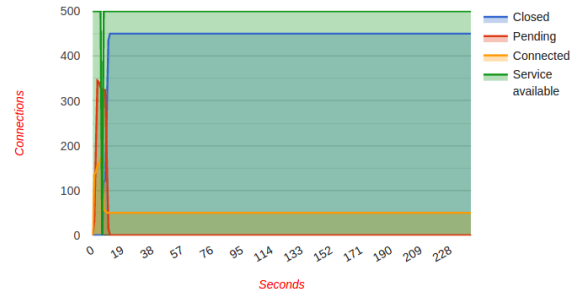

Figure 4.12 Apache server under slow header attack after enabling qos

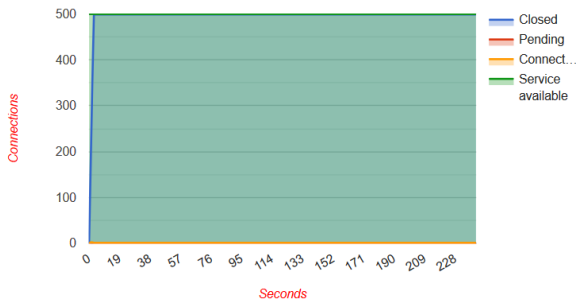

Figure 4.13 Apache server under slow header attack after enabling antiloris

Experiment 5

In this experiment, the slow body attack was successful in bringing the Apache server down before enabling the reqtimeout module, so the slow body attack was able of rendering the server to the denial of service state with only 500 hundred connections. After enabling reqtimeout module and setting a time limit for the HTTP body to be received, we can see mitigation of 45% against the slow HTTP body attack.

Here is the same case as the previous experiment, slow body attack can be mitigated by the qos (quality of service) module can handle efficiently .We have specified a maximum of 50 connections for each IP, so the qos module has closed 450 of the connections.

For this experiment, antiloris module has prevented the slow body attack from rendering the Apache server out of reach. Antiloris module has closed most of the connections, while 8-9 connections have remained and we can see the high availability of the server.

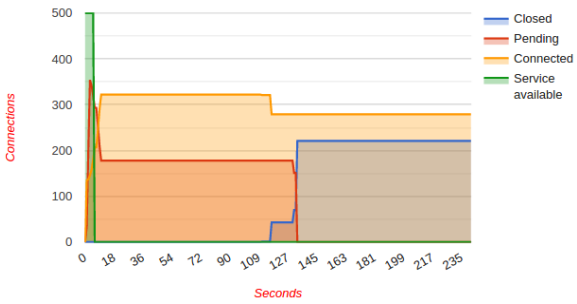

Figure 4.14 Apache server under slow body attack in default configuration

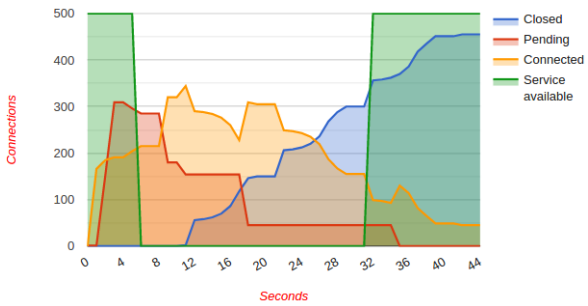

Figure 4.15 Apache server under slow body attack after enabling reqtimeout

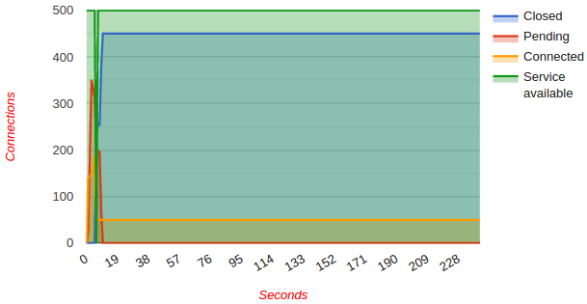

Figure 4.16 Apache server under slow body attack after enabling qos

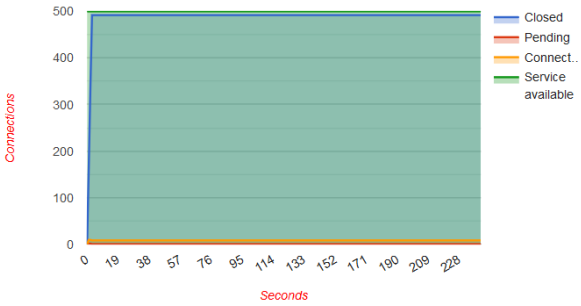

Figure 4.17 Apache server under slow body attack after enabling antiloris

Experiment 6: Slow Read

Here is the same case as the two previous experiments, the qos (quality of service) module has mitigated slow read attack efficiently. The qos module has closed 450 of the connections and 50 connections have been remained, as we have specified a maximum of 50 connections for every single IP.

After enabling the antiloris module, it has hindered the impact of slow read attack on a high extend compared to default settings Figure 4.20. Also, it can be noticed that the antiloris module has closed most of the connections, while 8-9 connections have remained.

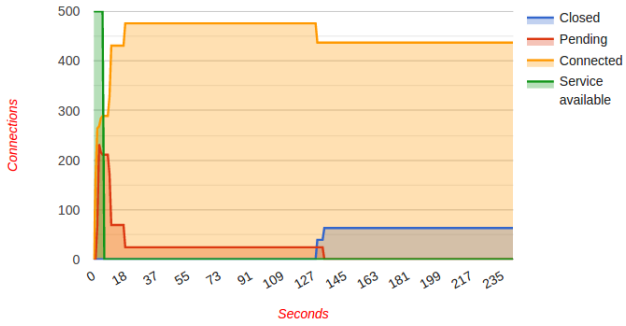

Figure 4.18 Apache server under slow read attack in default configuration

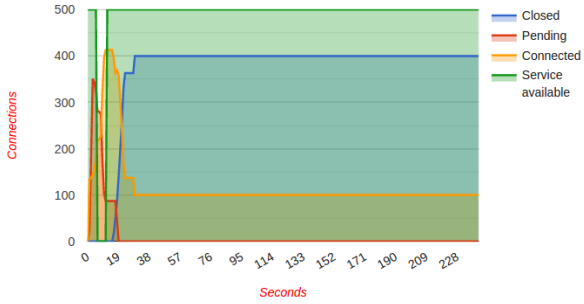

Figure 4.19 Apache server under slow read attack after enabling qos

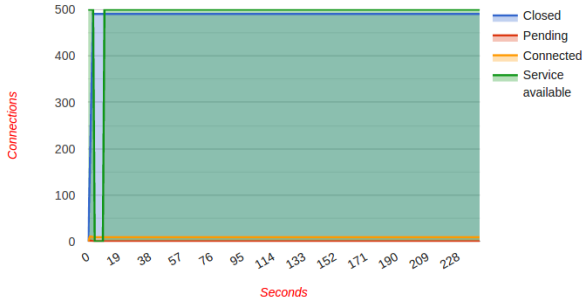

Figure 4.20 Apache server under slow read attack after enabling antiloris

4.3 Comparison between Apache and NGINX

The results of the experiments that were conducted on Apache and Nginx servers for comparing their performance will be displayed in this section.

4.3.1 Before mitigation (default settings)

This subsection presents the results of experiments on Apache and Nginx servers before applying any defense mechanism against the slow rate DoS attack.

Experiment 7

This experiment shows that the Apache server has been heavily hindered down under the slow header attack that was carried on. The attack was done by opening only 500 illegitimate connections on the server-side that rendered the Apache server unavailable as shown in Figure 4.21, while comparably NGINX has survived successfully from the slow header attack as shown in Figure 4.22.

The Apache server has been down most of the time during the slow header attack that was carried on in this experiment. The attack was done by opening a large number of connections to the server which was 10000 illegitimate connections that rendered the Apache server unavailable as shown in Figure 4.23. In this experiment unlike the previous one, the NGINX server was partly down during the slow header attack as shown in Figure 4.24. Nevertheless, the NGINX server showed better performance than the Apache server.

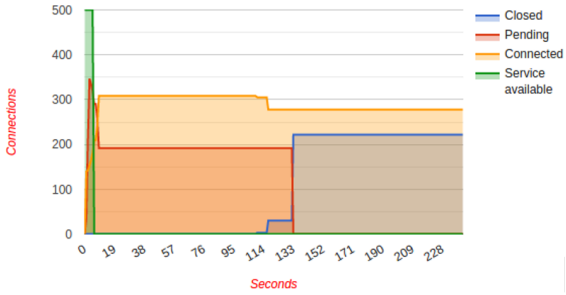

Figure 4.21 Apache server under slow header attack 500 connections in default configuration

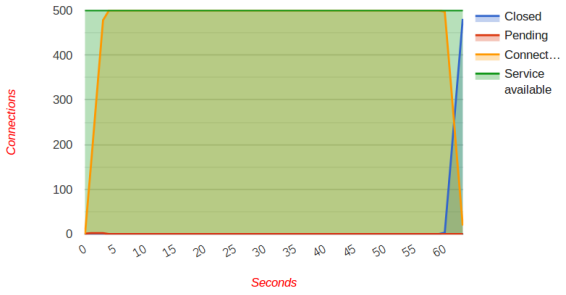

Figure 4.22 Nginx server under slow header attack 500 connections in default configuration

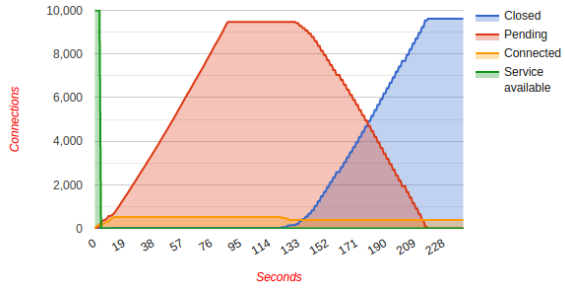

Figure 4.23 Apache server under slow header attack 10000 connections in default configuration

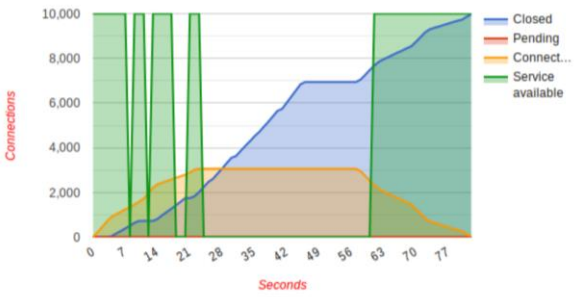

Figure 4.24 Nginx server under slow header attack 10000 connections in default configuration

Experiment 8

This experiment shows that the Apache server has been heavily hindered down under the slow header attack that was carried on. The attack was done by opening only 500 illegitimate connections on the server-side that rendered the Apache server unavailable as shown in Figure 4.25, while comparably NGINX has survived easily from the slow body attack as shown in Figure 4.26.

The Apache server has been down most of the time during the slow body attack that was carried on in this experiment. The attack was done by opening a large number of connections to the server which was 10000 illegitimate connections that rendered the Apache server unavailable as shown in Figure 4.27. In this experiment unlike the previous one, the NGINX server was down most of the time during the slow body attack as shown in Figure 4.28. However, the NGINX server showed better performance than the Apache server.

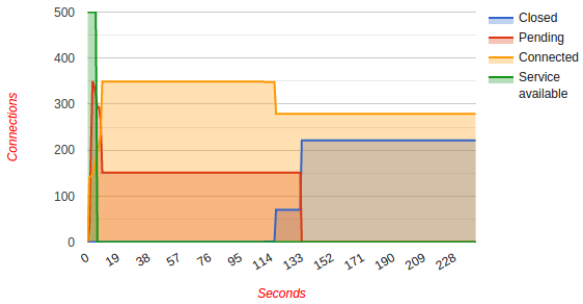

Figure 4.25 Apache server under slow body attack 500 connections in default configuration

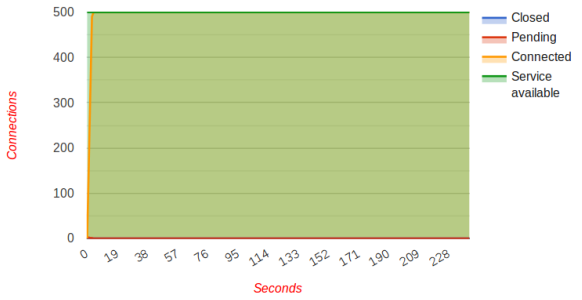

Figure 4.26 Nginx server under slow body attack 500 connections in default configuration

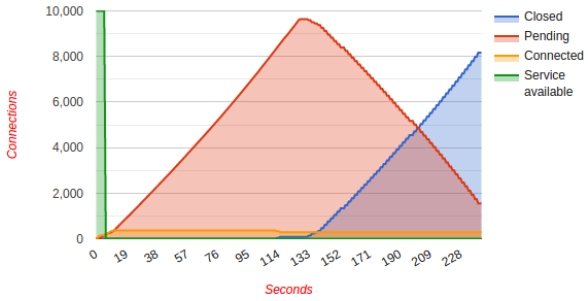

Figure 4.27 Apache server under slow body attack 10000 connections in default configuration

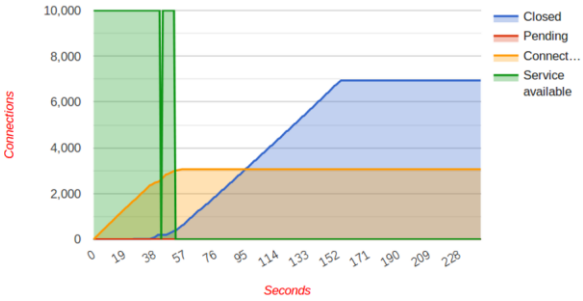

Figure 4.28 Nginx server under slow body attack 10000 connections in default configuration

Experiment 9

This experiment shows that the Apache server has been heavily hindered down under the slow read attack that was carried on. The attack was done by opening only 500 illegitimate connections on the server-side that rendered the Apache server unavailable as shown in Figure 4.29, while comparably NGINX has survived easily from the slow read attack as shown in Figure 4.30.

The Apache server has been down most of the time during the slow read attack that was carried on in this experiment. The attack was done by opening a large number of connections to the server which was 3000 illegitimate connections that rendered the Apache server unavailable as shown in Figure 3.31. In this experiment unlike the previous one, the NGINX server was down partly during the slow read attack as shown in Figure 4.32. However, the NGINX server showed better performance than the Apache server.

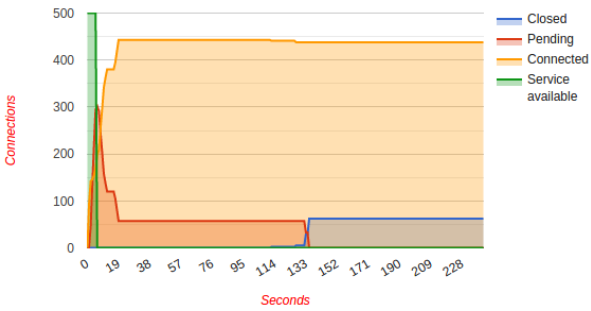

Figure 4.29 Apache server under slow read attack 500 connections in default configuration

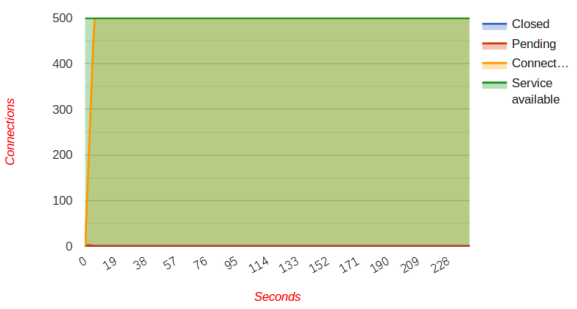

Figure 4.30 nginx server under slow read attack 500 connections in default configuration

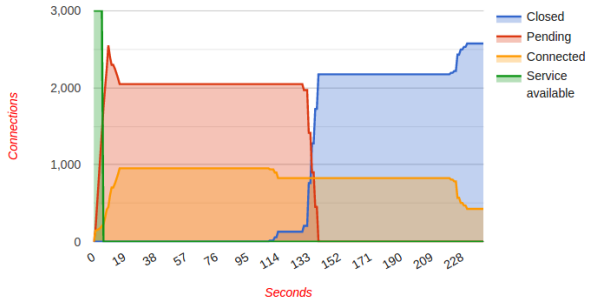

Figure 4.31 Apache server under slow read attack 3000 connections in default configuration

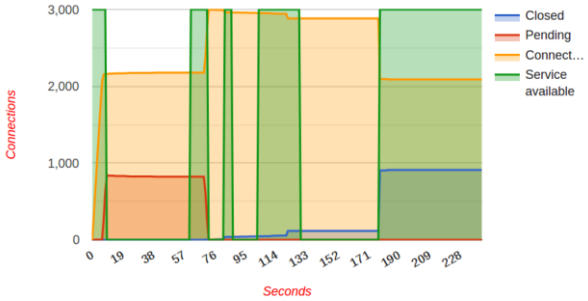

Figure 4.32 nginx server under slow read attack 3000 connections in default configuration

4.3.2 After mitigation

This subsection presents the results of experiments on Apache and Nginx servers after applying the defense mechanism against the slow rate DoS attack.

Experiment 10

Contrary to Apache's performance in experiment 7 that has been shown in subsection 4.3.1, Apache server showed a good performance against the slow header attack as can be seen in Figure 4.33. Similarly, NGINX server acted very well under the attack and it was available all the time. The attack was done by opening 500 illegitimate connections to the server but both NGINX and Apache server were available during the attack. However, in NGINX it took only 5 seconds until all connections were closed while in Apache it took more than twice the time to close the illegitimate connections.

When using 10000 illegitimate connections attack, both NGINX and Apache servers were available during the attack and showed a good performance against the slow header attack. Apache's performance showed considerable enhancement compared to experiment 7 as can be seen in Figure 4.35. Similarly, NGINX server acted very well under the attack and it was available all the time and it took only 48 seconds until all connections were closed while Apache took twice the time to close the illegitimate connections.

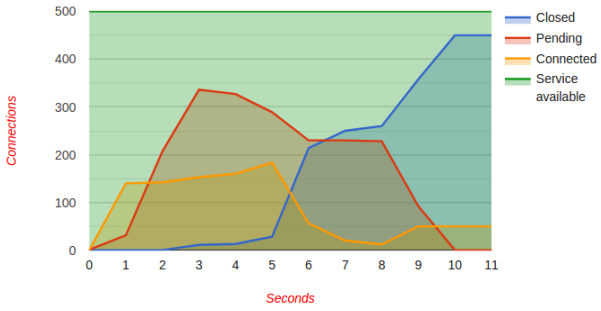

Figure 4.33 Apache server under slow header attack 500 connections in manual configuration

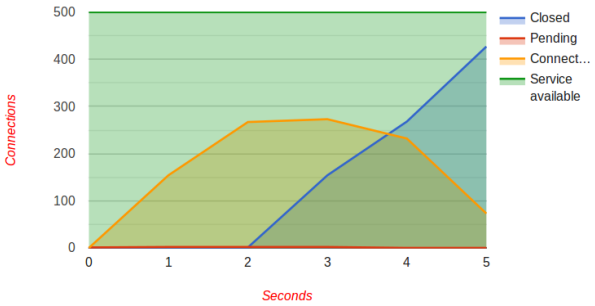

Figure 4.34 NGINX server under slow header attack 500 connections in manual configuration

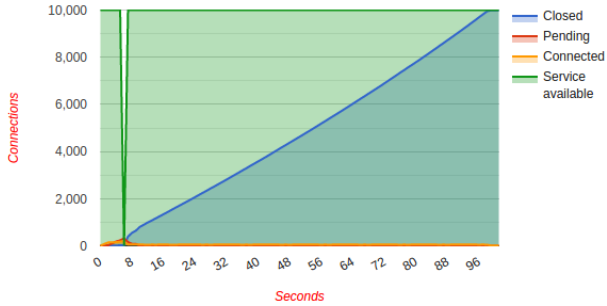

Figure 4.35 Apache server under slow header attack 10000 connections in manual configuration

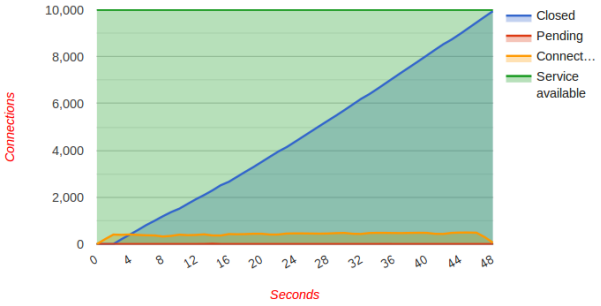

Figure 4.36 NGINX server under slow header attack 10000 connections in manual configuration

Experiment 11

Contrary to Apache's performance in experiment 8 that has been shown in subsection 4.3.1, Apache server showed a good performance against the slow header attack as can be seen in Figure 4.37. Similarly, NGINX server acted very well under the attack and it was available all the time. The attack was done by opening 500 illegitimate connections to the server but both NGINX and Apache servers were available during the attack. However, on Apache, it took only 16 seconds until all connections were closed while in NGINX it took more than twice the time to close the illegitimate connections.

When using 10000 illegitimate connections attack, both NGINX and Apache servers were available during the attack and showed a good performance against the slow body attack. Apache's showed considerable performance. Similarly, NGINX server acted very well under the attack and it was available all the time Figure 4.40

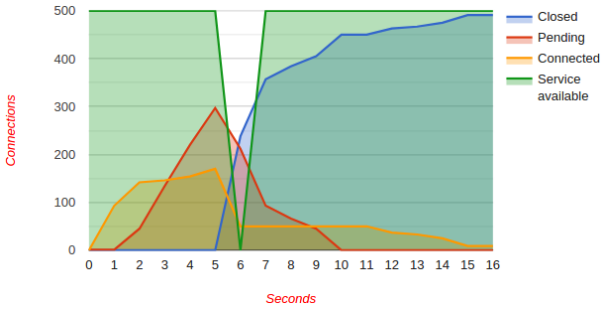

Figure 4.37 Apache server under slow body attack 500 connections in manual configuration

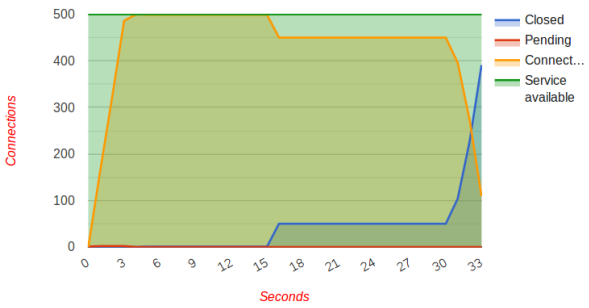

Figure 4.38 nginx server under slow body attack 500 connections in manual configuration

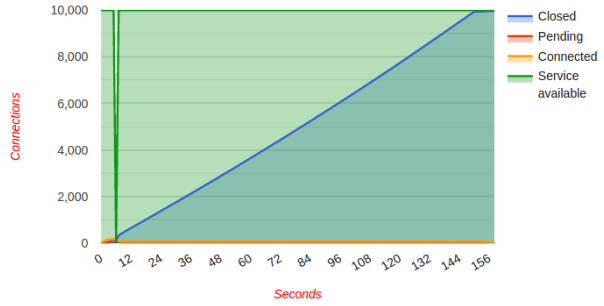

Figure 4.39 Apache server under slow body attack 10000 connections in manual configuration

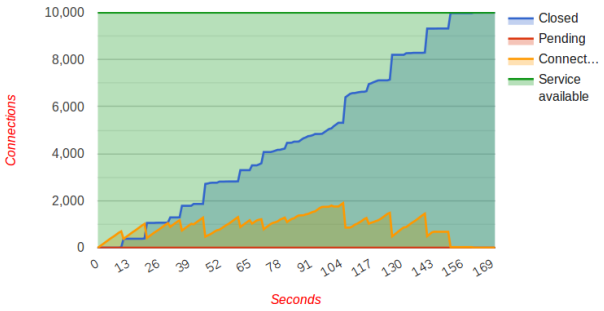

Figure 4.40 nginx server under slow body attack 10000 connections in manual configuration

Experiment 12

Contrary to Apache's performance in experiment 9 that has been shown in subsection 4.3.1, Apache server showed a good performance against the slow header attack as can be seen in Figure 4.41. Similarly, NGINX server acted very well under the attack and it was available all the time. The attack was done by opening 500 illegitimate connections to the server but both NGINX and Apache servers were available during the attack.

When using 3000 illegitimate connections attack, both NGINX and Apache servers were available during the attack and showed a good performance against the slow read attack. Apache's showed considerable performance. Similarly, NGINX server acted very well under the attack and it was available all the time Figure 4.44

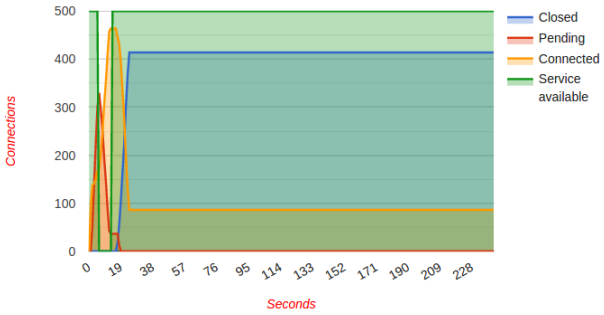

Figure 4.41 Apache server under slow read attack 500 connections in manual configuration

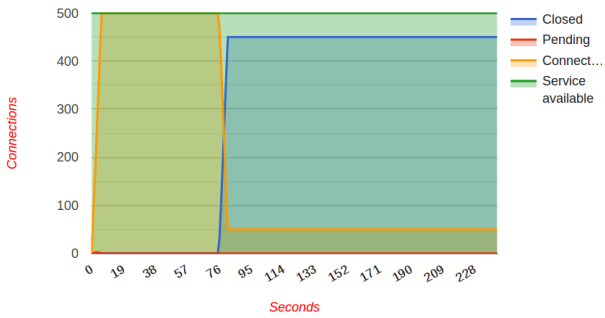

Figure 4.42 nginx server under slow read attack 500 connections in manual configuration

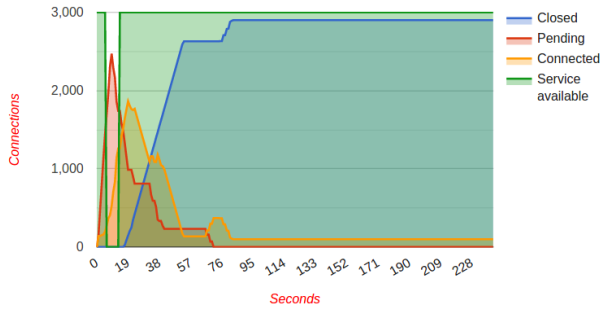

Figure 4.43 Apache server under slow read attack 3000 connections in manual configuration

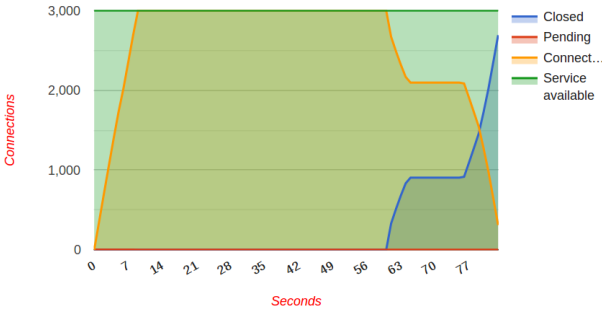

Figure 4.44 nginx server under slow read attack 3000 connections in manual configuration

5 Analysis and Discussion

In this chapter results presented in the previous chapter will be analyzed. It contains three sections, each section presents an analysis of the related experiments.

5.1 NGINX

As has been stated in chapter 3, header timeout and body timeout have been used in experiments 1 and 2 in order to mitigate slow header and slow body attacks respectively. Given the fact, that the slow header/body attack mechanism is based on sending request header/body at a slow pace to keep the server busy, so setting a timeout for receiving these data would drop the request thus rendering the attack unsuccessful. Therefore, header and body timeout has been used only against slow header and slow body attacks respectively.

In experiments 1, 2, and 3, limiting connections technique has been used to defeat slow rate DoS attack. The mechanism is based on limiting the number of simultaneous connections that can be opened by a client based on the IP address, and whereas one of the topmost important factors of a successful slow rate DoS attack is the number of parallel opened connections to the server, setting a limit for that would defeat or at least mitigate the impact of the attack. The mitigation technique was highly successful in protecting NGINX server from slow body and slow header attacks as presented in the results chapter in subsection 4.1 However, the same technique showed deficiency in the confrontation of the slow header attack as has been stated in 4.1 The reason behind this failure is that the responsible module for limiting connections which is *ngx_http_limit_conn_module* does not count connections that have incomplete requests. It counts only connections that have complete request headers and that have been read by the server [17]. And since the attacker, in the slow header attack, sends incomplete request headers, the connection would not be counted and therefore the attack will evade this protection technique.

In experiment 2, a maximum body size has been set by using *client_max_body_size* directive. As have been seen in the results in 4.1 subsection this technique made a huge difference in protecting the server from slow body attack. This is due to the fact that *client_max_body_size* directive counts on the Content-Length field in the head of the request. The server checks this field, if it was larger than specified by the directive, the request will be rejected [10]. Thus the attack will be unsuccessful which can be seen easily in the result of experiment 2.

In all experiments where mitigation mechanisms were successful it was noticeable from the results that the server was able to close larger number of connections in less time after applying defense mechanisms. While this is true, it was also conspicuous that number of connected connections after applying defense mechanisms was less compared to before. Therefore, although defense mechanisms was able to protect NGINX form the attacks they introduced a downside effect which is less number of served connections. This shows that the protection mechanism in this context is a tradeoff between security and performance.

5.2 Apache

For apache, we have used three modules that depend on setting a timeout for the HTTP header and body, and limiting the connections that a server can handle or a single IP (client) can

establish. By this aspect, we can realize that the mitigation methods on both apache and Nginx do not differ conceptually, but as a matter of fact, they rely on the same techniques for mitigation. Before using any defensive mechanism we have tested the three types of slow HTTP attack with 500 connections and we saw how the Apache server has reacted poorly against these attacks.

As we have used the reqtimout module we have not placed a test for the slow read attack, as we already knew that the slow HTTP read sends a complete header request within the time limit without segmenting it. The slow HTTP read attack, proposes a small buffer window of bytes that the sender (attacker) can read once at a second, so it keeps the server busy by sending small segments at a low rate, without the need of sending the header at low speed.

Not all of the mitigation tools worked ideally in the aspect of defeating the slow rate attacks. For example, in experiment 5 we the reqtimeout module has been used, it can be noticed that the slow body attack was able to bring the apache server down for nearly half the time of the test.

As in NGINX, applying defense mechanisms led Apache server to close the connections sooner than before applying them, which means better defense in the term of defending slow attacks. As a result the number of pending connections was very minimum also. However, the number of served connections were much less compared to default configurations. For example in experiment 5 when using mod_qos, the number of connected connections was dropped down from around 300 in default configurations to around only 50 connections. This tradeoff between security and availability should be well considered when applying defense mechanisms.

5.3 NGINX vs Apache

As can be seen in experiments 7, 8, and 9 NGINX server acted more effectively than Apache server against slow rate DoS attacks. This was obviously seen when 500 connections were opened to the servers, as Apache was unavailable most of the time during the attacks while NGINX was completely available and none of the attacks in these experiments could be able to bring the server down. Even when the number of simultaneous connections was increased to high values, NGINX showed better performance than the Apache server. The main reason behind this difference in performance is the underlying architecture of each server in processing connections.

Also, a prominent difference between Apache and NGINX in both experiments before and after applying mitigation, is the number of connected connections that each server could handle. For example, in experiments 7 and 8 when 10000 connections were opened to the servers, NGINX was able to serve approximately 3000 connections at the peak where Apache was able only to handle approximately 400 at the most. Moreover, Apache tends to put the majority of incoming connections in pending state which means they are on hold due to server being busy. While NGINX makes minimum usage of pending state as it servers the connection and close it as soon as possible.

Also, another advantage to the NGINX server, is the tendency towards closing connections very soon when compared to Apache, which is an important factor in defending against slow HTTP attacks. Because, as discussed in chapter one, slow HTTP attacks depend on sending as many as possible requests to the server, therefore dropping more connections by the server can defeat the attack and render it unsuccessful. As a result, this is counted as an important feature in NGINX's ability to defend against slow HTTP DoS attacks.

NGNINX was designed to overcome the problem of serving a high number of concurrent connections by implementing asynchronous, non-blocking, event-driven design [18]. The event-driven model that has been used in NGINX was used primarily to run-over the issues of scalability and performance that was in Apache [19]. NGINX has a master process that generates worker processes which in turn listens for incoming connections. Each worker process in NGNIX can process thousands of simultaneous connections at the same time while each worker process in Apache can only handle one connection [19][20]. This is why slow rate DoS attacks could not bring NGINX server down easily compared to the Apache server.

5.4 Discussion

The findings of the research showed that NGNIX is more resistant against slow rate DoS attacks in default configuration than Apache as were predicted before conducting the experiment. Having said that, NGINX server is also exposed to be unreachable under severe slow rate DoS attacks. However, the defense mechanisms that were chosen in this study showed high improvement in protecting both servers from the attacks.

The study experiments the behavior of the two servers, NGNIX and Apache, against slow HTTP attack, while the related works that have been reviewed in this paper do not have any evaluating of the server's performance. The objectives that have been set in this study have been achieved, as we have applied the three types of slow HTTP attacks on NGNIX and Apache and also measured their performance.

6 Conclusion

Security is always a matter of concern for web server managers as many services on the web are exposable to threats. This paper examined the built-in defense mechanism in two well-known web servers which are Apache and NGINX, against one of the novel security threats which is slow rate DoS attack that targets web servers. From the results of the experiments that have been conducted, it was shown that both servers are protectable against slow rate DoS attack by making use of the built-in tools. However, Apache is more vulnerable to this type of attack when using default configurations, compared to NGNIX server which has better performance under slow rate attack.

The project started by carrying out a literature review regarding low rate DoS attack (Objective 1) which has been done with the aid of academic search engines such as Google Scholar, Microsoft Academic and LNU's OneSearch. By achieving the first objective, we were able to write the Background subchapter (1.1), Related works 1.2, and Mitigation tools 3.2 subchapters. In addition, the literature review highly contributed in implementing the low rate DoS attack on Apache and NGINX servers (Objective 2). Also, it facilitated the preparation of the experiment environment and choosing the appropriate tools for conducting the experiments.

As presented in chapter 3, we were able to implement the attack by making use of recognized software and tools such as VirtualBox, Linux OS, Apache and NGINX software, and slowhttptest. Additionally, after achieving O1, we were able to choose suitable configurations and tools to defend against the attacks. The performance of the servers was measured by sending frequent requests to the server machine and anticipating/waiting for the response. If the server responded within the timeout frame it was tagged as available otherwise it was tagged as unavailable (Objective 3). Also, based on O1 and the information we have gathered in O3 , we were able to analyze the results by comparing the two servers taking into consideration the availability parameter in addition to pending, connected and closed parameters (Objective 4).

These findings are important to web server managers when they want to choose between Apache and NGINX taking into account the slow rate DoS attack and the downsides of using defense mechanisms. Since our experiment has been conducted on virtual machines with modest resources, it would be more realistic if we had applied the experiment on a real webserver with capable resources and measured the impact of the slow HTTP attack.

6.1 Future work

The experiments could have been conducted in a more realistic environment so it can be guaranteed that imposed configuration would not generate false-positive statues. More studies should be done to analyze the traffic on different servers and balance between serving clients and blocking intruders regarding the slow HTTP attack. Moreover, slow HTTP can be used in distributed denial of service attack DDoS, which will have a larger impact on servers, and will require different configurations and mitigation tools.

References

[1] CERT, “1997 Tech Tip: Denial of Service Attacks,” *resources.sei.cmu.edu*. [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=496599. [Accessed: 16-Sep-2020].

[2] “Denial-of-Service (DoS) Attack,” *Cloudflare*, 2019. [Online]. Available: https://www.cloudflare.com/learning/ddos/glossary/denial-of-service/. [Accessed: 17-Sep-2020].

[3] B. B. Gupta and O. P. Badve, "Taxonomy of DoS and DDoS attacks and desirable defense mechanism in a Cloud computing environment," *Neural Computing and Applications,* vol. 28, no. 12, pp. 3655-3682, 2017682, Apr. 2016.

[4] M. Haddadi and R. Beghdad, “DoS-DDoS: Taxonomies of attacks, countermeasures, and well-known defense mechanisms in cloud environment,” *EDPACS*, vol. 57, no. 5, pp. 1–26, May 2018.

[5] S. Suroto, "A Review of Defense Against Slow HTTP Attack," *JOIV : International Journal on Informatics Visualization,* vol. 1, no. 4, pp. 127-134, 2017.

[6] E. Cambiaso, G. Papaleo and M. Aiello, "Taxonomy of Slow DoS Attacks to Web Applications," *International Conference on Security in Computer Networks and Distributed Systems,* pp. 195-204, 2012.

[7] S. Ranjan, R. Swaminathan, M. Uysal and E. Knightly, "DDoS-Resilient Scheduling to Counter Application Layer Attacks Under Imperfect Detection," in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, 2006.

[8] “What is a Slowloris Attack?,” *NETSCOUT*, 2019. [Online]. Available: https://www.netscout.com/what-is-ddos/slowloris-attacks. [Accessed: 17-Sep-2020].

[9] M. Sikora, T. Gerlich and L. Malina, "On Detection and Mitigation of Slow Rate Denial of Service Attacks," in *2019 11th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, 2019.

[10] “Module ngx_http_core_module,” *nginx.org*. [Online]. Available: http://nginx.org/en/docs/http/ngx_http_core_module.html. [Accessed: 18-Sep-2020].

[11] R. Nelson, “DDos Mitigation - Using NGINX to Prevent DDoS Attacks,” *NGINX*, 02-Jul-2015. [Online]. Available: https://www.nginx.com/blog/mitigating-ddos-attacks-with-nginx-and-nginx-plus/.[Accessed: 12-Sep-2020].

[12] D. DeJonghe, *Complete NGINX Cookbook*. O’Reilly Media, Inc, 2019.

[13] I. Muscat, “How To Mitigate Slow HTTP DoS Attacks in Apache HTTP Server | Acunetix,” *Acunetix*, 04-Oct-2017. [Online]. Available: https://www.acunetix.com/blog/articles/slow-http-dos-attacks-mitigate-apache-http-server/.[Accessed: 14-Sep-2020].

[14] “mod_reqtimeout - Apache HTTP Server Version 2.5,” *httpd.apache.org*. [Online]. Available: https://httpd.apache.org/docs/trunk/mod/mod_reqtimeout.html. [Accessed: 18-Sep-2020].

[15] “mod_qos,” *mod-qos.sourceforge.net*. [Online]. Available: http://mod-qos.sourceforge.net/index.html. [Accessed: 18-Sep-2020].

[16] L. Nick, “Deltik/mod_antiloris,” *GitHub*, 24-Jul-2020. [Online]. Available: https://github.com/Deltik/mod_antiloris. [Accessed: 18-Sep-2020].

[17] “Module ngx_http_limit_conn_module,” *nginx.org*. [Online]. Available: https://nginx.org/en/docs/http/ngx_http_limit_conn_module.html#limit_conn. [Accessed: 17-Sep-2020].

[18] “Nginx vs Apache – which is the best web server?,” *Plesk*, 22-Jul-2018. [Online]. Available: https://www.plesk.com/blog/various/nginx-vs-apache-which-is-the-best-web-server/.[Accessed: 13-Sep-2020].

[19] S. Bradley, “NGINX—Server Software With Event-Driven Architecture,” *Vanseo Design*, 27-Mar-2018. [Online]. Available: https://vanseodesign.com/web-design/nginx-web-server/. [Accessed: 18-Sep-2020].

[20] “Inside NGINX: Designed for Performance & Scalability,” *NGINX*, 11-Jun-2015. [Online]. Available: https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/.[Accessed: 17-Sep-2020].

[21] W. Zhijun, L. Wenjing, L. Liang and Y. Meng, "Low-Rate DoS Attacks, Detection, Defense, and Challenges: A Survey," *IEEE Access,* vol. 8, pp. 43920-43943, 2020.

[22] “What is R.U.D.Y. (R-U-Dead-Yet?) | DDoS Tools | Imperva,” *Learning Center*. [Online]. Available: https://www.imperva.com/learn/ddos/rudy-r-u-dead-yet/. [Accessed: 17-Sep-2020].

[23] “What is a Slow Post Attack?,” *NETSCOUT*. [Online]. Available: https://www.netscout.com/what-is-ddos/slow-post-attacks. [Accessed: 17-Sep-2020].

Appendix

Commands used in each experiment

Experiment 1
slowhttptest -c 10000 -H -g -o ./output_file -i 10 -r 1000 -t GET -u http://192.168.22.6 -x 24 -p 3

Experiment 2
slowhttptest -c 10000 -B -g -o ./output_file -i 5 -r 100 -t POST –u http://192.168.22.6/uplaod.php -x 24 -p 3

Experiment 3
slowhttptest -g -o ./output_file -c 3000 -X -r 1000 -w 512 -y 1024 -n 5 -z 32 -k 3 -u http://192.168.22.6/image.jpg -p 3

Experiment 4
slowhttptest -c 500 -H -g -o ./output_file -i 10 -r 200 -t GET -u http://192.168.22.6 -x 24 -p 3

Experiment 5
slowhttptest -c 500 -B -g -o ./output_file -i 10 -r 200 -t POST -u http://192.168.22.6/a.php -x 10 -p 3

Experiment 6
slowhttptest -c 500 -X -g -o ./output_file -r 200 -w 512 -y 1024 -n 5 -z 32 -k 3 -u http://192.168.22.6/1.jpg -p 3

Experiment 7
slowhttptest -c 500 -H -g -o ./output_file -i 10 -r 200 -t GET -u http://192.168.22.6 -x 24 -p 3

slowhttptest -c 10000 -H -g -o ./output_file -i 10 -r 1000 -t GET -u http://192.168.22.6 -x 24 -p 3

Experiment 8
slowhttptest -c 500 -B -g -o ./output_file -i 10 -r 200 -t POST -u http://192.168.22.6/a.php -x 10 -p 3

slowhttptest -c 10000 -B -g -o ./output_file -i 5 -r 100 -t POST –u http://192.168.22.6/uplaod.php -x 24 -p 3

Experiment 9
slowhttptest -c 500 -X -g -o ./output_file -r 200 -w 512 -y 1024 -n 5 -z 32 -k 3 -u http://192.168.22.6/1.jpg -p 3

slowhttptest -g -o ./output_file -c 3000 -X -r 1000 -w 512 -y 1024 -n 5 -z 32 -k 3 -u http://192.168.22.6/image.jpg -p 3

Experiment 10
slowhttptest -c 500 -H -g -o ./output_file -i 10 -r 200 -t GET -u http://192.168.22.6 -x 24 -p 3

slowhttptest -c 10000 -H -g -o ./output_file -i 10 -r 1000 -t GET -u http://192.168.22.6 -x 24 -p 3

Experiment 11
slowhttptest -c 500 -B -g -o ./output_file -i 10 -r 200 -t POST -u http://192.168.22.6/a.php -x 10 -p 3

slowhttptest -c 10000 -B -g -o ./output_file -i 5 -r 100 -t POST –u http://192.168.22.6/uplaod.php -x 24 -p 3

Experiment 12

```
slowhttptest -c 500 -X -g -o ./output_file -r 200 -w 512 -y 1024 -n 5 -z 32 -k 3 -u
http://192.168.22.6/1.jpg -p 3

slowhttptest -g -o ./output_file -c 3000 -X -r 1000 -w 512 -y 1024 -n 5 -z 32 -k 3 -u
http://192.168.22.6/image.jpg -p 3
```