FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Modern slow DDoS attacks and protection against them
Student:	Lukáš Brůna
Supervisor:	Ing. Jan Fesl, Ph.D.
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

Distributed Denial of Service (DDos) attacks are the most common type of attack performed by hackers online. The mutual goal of the attacks is to deny access to a certain service from all users. One variant of DDoS attacks is the so-called slow DDoS attacks, which are precarious in that unlike usual DDoS attacks, they do not manifest in higher network traffic but rather in continuous depletion of the victims' resources.

In the bachelor thesis, focus in detail on the problem of slow DDoS (SDDoS) attacks on web servers, design and implement an algorithm allowing a reliable detection of SDDoS attacks and defense against them. In the practical part of the thesis, create a module for the Apache 2 web server, able to perform active mitigation of detected SDDoS attacks.

Thesis goals (in detail):

1) Explore the current state of research in the field of SDDoS attacks and their impact on web servers.
2) Identify the characteristics of typical SDDoS attacks.
3) Design and implement an Apache 2 web server module, capable SDDoS attack detection and mitigation.
4) Perform testing of the created module in order to verify its effectiveness and reliability.
5) Compare the results with the existent SDDoS attack defense methods and evaluate the contribution of the created module.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 26 January 2024 in Prague.

Bachelor's thesis

MODERN SLOW DDOS ATTACKS AND DEFENSE AGAINST THEM

Lukáš Brůna

Faculty of Information Technology
Katedra informační bezpečnosti
Supervisor: Ing. Jan Fesl, Ph.D.
May 16, 2024

Contents

List of Figures

This thesis would not be possible without the guidance and support of my supervisor Ing. Jan Fesl, Ph.D. and the constant encouragement and mental fortitude of my beloved girlfriend Patricie A. I also want to express my gratitude towards my family, as my brother is someone I could always look up to and my parents who always provide the support one could imagine, making sure I have someone to rely on. Without these people, this work would not be possible.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 16, 2024

Abstract

This thesis is focused on the field of Slow Distributed Denial of Service attacks and their mitigation. First, I explain how the most common types of SDDoS attacks function and what is their impact on current systems. Second section of the thesis is then focused on different possible defense solutions, how they could be implemented and if they are effective. In the final sections I choose a solution approach which I design, implement and test successfully.

Keywords SDDoS, DDoS, Slow Reads, Denial of Service, network attacks, Apache, server

Abstrakt

Tato práce je zaměřená na obor útoků pomalým odepřením služby a obrany proti nim. Vysvětluji, jak nejčastější typy těchto útoků fungují a jaký mají dopad na aktuální systémy. Velká část práce je poté zaměřena na různá obranná řesení, jak by mohla být implementována a zda-li jsou efektivní. Ve finálních částech práce si volím způsob řesení který úspěšně navrhuji, implementuji a testuji.

Klíčová slova SDDoS, DDoS, Pomalé čtení, odepření služby, síťové útoky, Apache, server

List of abbreviations

CVE	Common Vulnerabilities and Exposures
DDoS	Distributed Denial of Service
DoS	Denial of Service
FPR	False-positive ratio
HTTP	Hypertext Transfer Protocol
SDDoS	Slow Distributed Denial of Service
SDN	Software Defined Network
TCP	Transmission Control Protocol

Introduction

Distributed Denial of Service attacks are the most prominent type of attacks performed nowadays. Their goal is to deny availability of a service to all users at little cost to the attacker, only needing a botnet or a large amount of systems able to create requests towards the victim. SDDoS are a subtype of such attacks, their specificity comes from the lack of drastic increases of network traffic during the attack, opting to deplete the resources of the service instead, this also means that the attacker is not required to control a botnet as large. While the attack effectiveness is increased with the botnet size, a successful service denial can be achieved even with only one system performing the slow attack.

Such a workaround in the principles of the attack was very interesting to me, thus prompting me to pick the attack subtype as the subject of my thesis. I will focus on explaining the principles of Slow DDoS attacks and the design and implementation of a web server SDDoS defense.
SDDoS attacks, however, can vary in their techniques. As I will explain in more detail in the subsequent chapters, there are three main slow attack types, each with a different approach to ultimately achieve a denial of service. I will also discuss the efficiency and how they currently differ in their success in chapter 2.
The main goals of the thesis are to summarize the current state of research in the field of SDDoS attacks and their impact on web servers, to identify the characteristics of the three typical SDDoS attack types in chapter 2. Then in chapter 3, the goal is to design and implement an Apache2 web server module, able to detect and react to SDDoS attacks, test its efficiency and reliability. Finally, in chapter 5 compare the results to existing SDDoS defense mechanisms and evaluate its contribution.

Chapter 1

Introducing the SDDoS attack

1.1 The problematics of SDDoS

1.1.1 Specifics of Distributed Denial of Service attacks

Denial of Service is a cybersecurity threat gaining popularity among malicious parties during the last decade. As the name of the attack suggests, the main disruption caused is by denying the availability of the victim service / device to users or admins.
It usually achieves this goal by overwhelming the service by an unusually large amount of valid requests starving the victim of resources / capacity, making it unable to respond to non-malicious users.

While other attacks may be considered more serious (for example data leaks, privilege escalation due to the damage caused), the simplicity of performing the attack in comparison to the difficulty in defending against it makes it an easy choice from the hackers' point of view.

Distributed Denial of Service is a modified DoS utilizing the possibility of attacking from multiple sources while targeting one specific victim system. While this does increase the costs for an attacker, the effectiveness of attack grows exponentially.

This technique makes it impossible to resolve the attack by simply blocking the malicious actor and forces the defender to differentiate between valid user requests needing a response and attacker requests to block.

1.1.2 HTTP communication

The HTTP protocol is a request/response application-level stateless protocol [1]. Most HTTP communication is initiated by a user agent and consists of a request to be applied to a resource on some origin server. In the simplest case, this may be accomplished via a single connection between the user agent and the origin server.
The above mentioned client request begins with a method token. This token can vary according to what a client intends to do. For the purpose of this thesis, the most important methods are going to be GET and POST used to either read or upload data respectively.

For a Web Server to craft a response, it must be handled by two thread types, explained with figure 1.1.

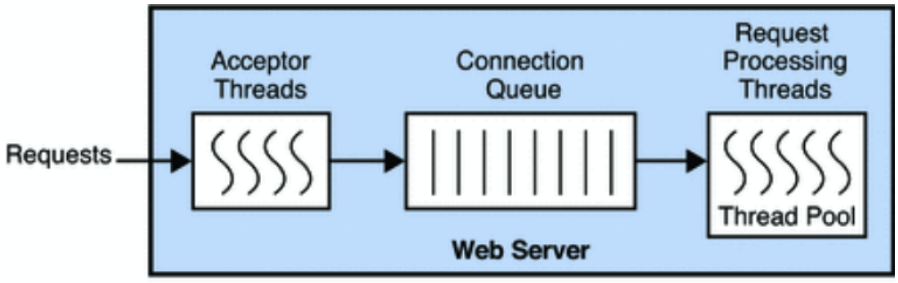

Figure 1.1 Web Server Connection Handling [2]

1.1.3 How SDDoS attacks work

The most common DDoS attacks aim to deny the availability of a service by overwhelming it with a number of requests so large, it is unable to handle it. That creates a noticeable peak in the traffic of the victim network, making it easy to detect a system being attacked. But if I compare the network traffic during DDoS (figure 1.2) and SDDoS (figure 1.3) attacks, I can see the SDDoS traffic may include a very brief initial peak, but nothing comparable to the DDoS traffic.
How do SDDoS attacks deny service availability nonetheless?
The answer is rather simple: the amount of requests does not need to be large as long as every request sent establishes a connection to the service and keeps it alive for as long as possible, thus taking up space for valid user connections. There are several ways of ensuring the server keeps the connection alive, all of which I will explain in the text below, but it is important to point out that while SDDoS is moderately detectable, it is not in fact detected by the usual monitoring solutions looking for DDoS attack peaks in their traffic data.

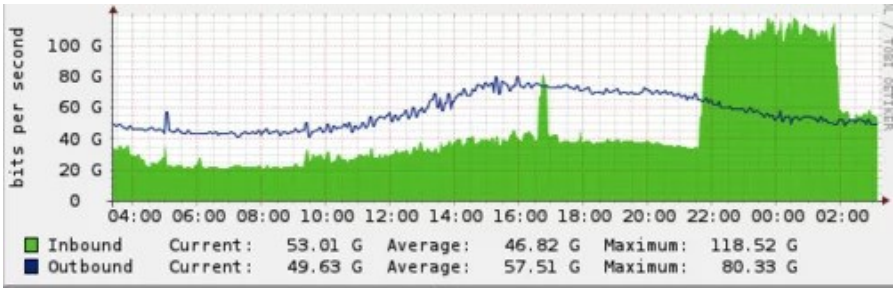

Figure 1.2 Example of network traffic during a DDoS attack. The Spamhaus Attack of 2013. [3]

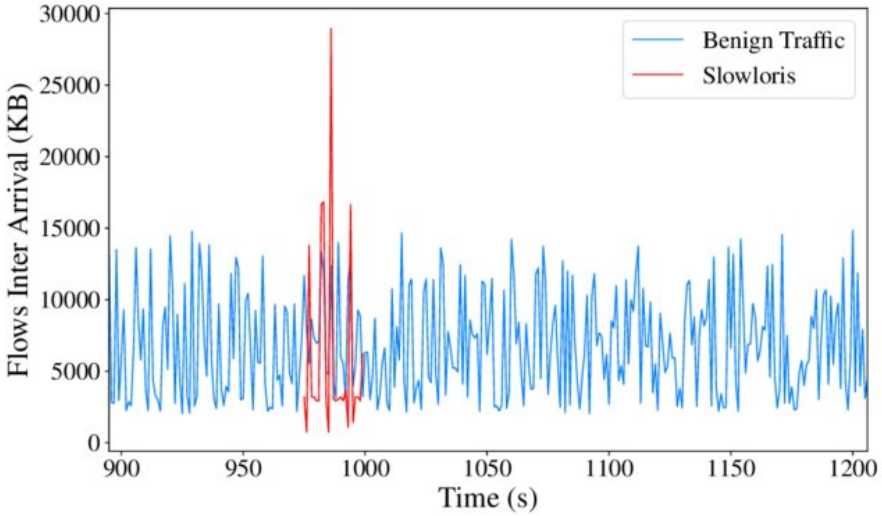

Figure 1.3 Example of network traffic during an SDDoS attack. [4]

1.1.3.1 Slow HTTP Headers attacks (also known as Slowloris [5])

The most common and one of the first detected slow HTTP attacks, successfully used against Iranian government servers in 2009 [6].
In the slow header HTTP attack, an attacker starts with a regular HTTP request[1], but not including the "end of headers"[2] flag, then waits a certain period of time before it sends an additional custom request header (e.g., "X-abcd: 1234") through the established TCP connection[3]. The attacker can send such random custom headers in repetition indefinitely, after waiting a set period of time.
According to the HTTP specification [1], all clients are allowed to add such headers. This client behaviour does not only slow down the initial request, in fact, it does not terminate it at all. It can bind servers resources for any period of time unless countermeasures like maximum request duration time and other settings are in place [7]. An example of a Slowloris attack is provided in figure 1.4.

[1] Slowloris can use both GET and POST methods [5]
[2] In the example 1.4, it would be "\r\n" that is missing. However, you can find it in 1.5, as that attack does not utilize slow header sending.
[3] This is an important factor in defending against Slowloris, see 2.2.

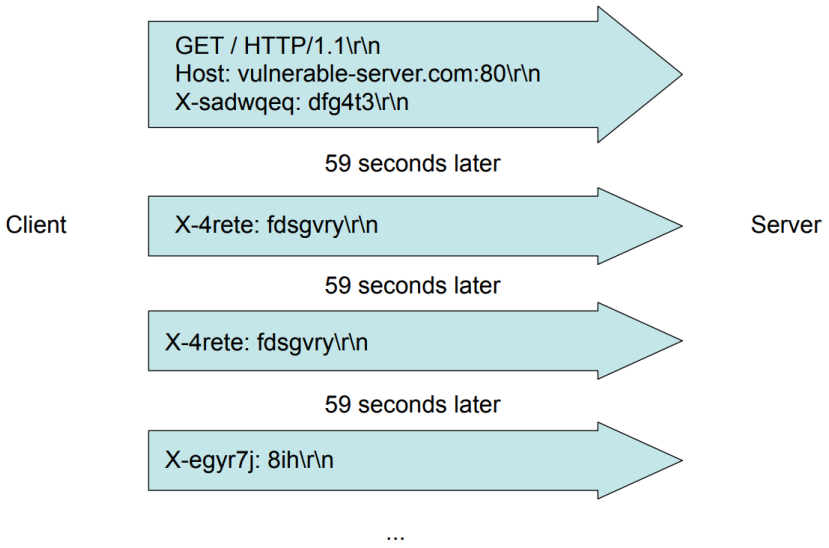

Figure 1.4 Slowloris attack example [8]

1.1.3.2 Slow HTTP Post attacks

This attack subtype is also called the slow POST attack, as it utilizes HTTP POST requests in the process, allowing the client to submit a request entity such as form data or file to be uploaded. Regular behaviour is used for the request header, however, a malicious actor slows down the transmission of the request entity, or sends a Content-Length header which is deliberately larger than the actual size of the requested entity. This requires the server to wait for additional data until timeout. Alternatively, an attacker sends slow chunks of a request entity using the chunked transfer encoding mode [7]. An example is shown in figure 1.5.

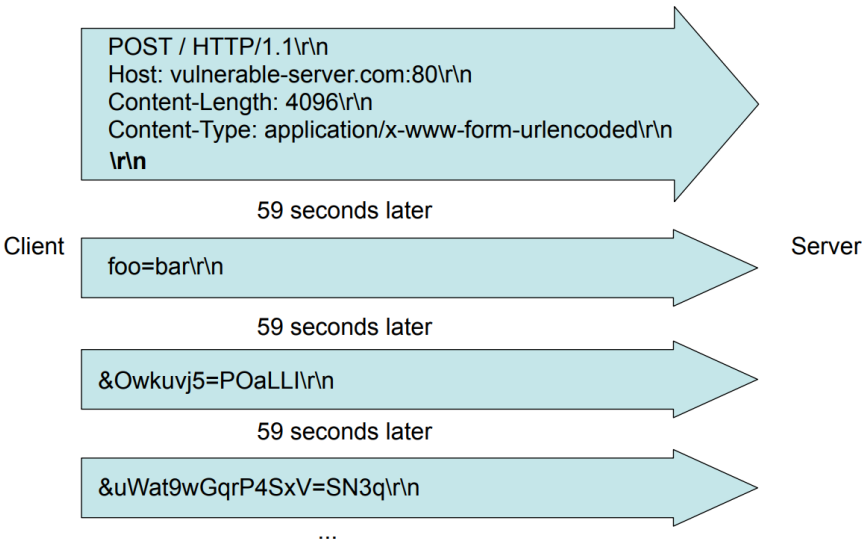

Figure 1.5 Slow HTTP Post attack example [8]

1.1.3.3 Slow Read attacks

Slow Read attacks differ from the other subtypes in that the attacker is not the one slowly sending requests / headers but rather forcing the server to slow down its own transfer. The client-server communication starts with a regular HTTP GET request to send an entity to the client. However, as soon as the server starts sending the response data, the client[4] informs it of an arbitrarily small[5] receive buffer, making the server poll its socket for write readiness indefinitely [9]. There is a prerequisite for the attack to work, the requested entity must be larger than the servers send buffer[6], otherwise the server sends the data to the *kernel send buffer* and forgets about it. While this attack subtype may appear superior, since the initial requests are indistinguishable from regular and slow client requests, it requires more attacker resources in comparison since the packets from the server need to be acknowledged and a response sent. Such issues are not present in the other attack subtypes. [7]

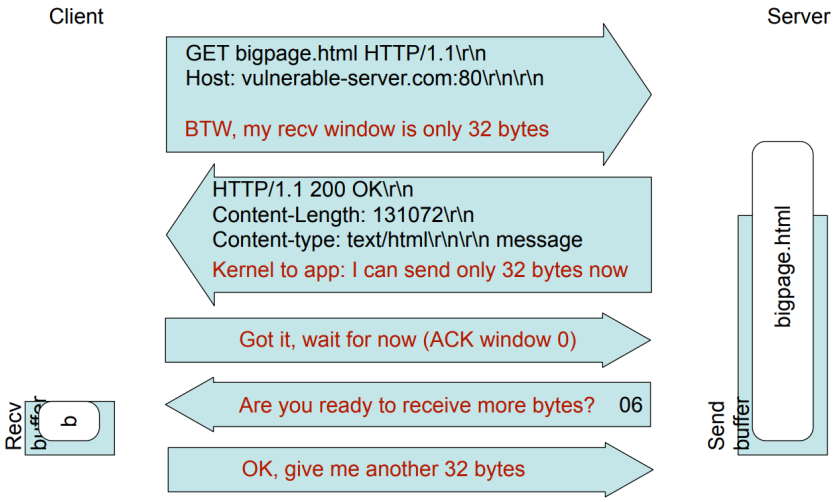

Figure 1.6 Slow HTTP Read attack example [8]

[4] in this case also the attacker
[5] in some cases the size of 0
[6] Server buffer size is usually between 65Kb and 128Kb. [9]

Chapter 2

Current state of research in the SDDoS field

2.1 SDDoS defense approaches

There are two main approaches to placing a (S)DDoS defense into the system, each coming with its positives and negatives.

One way to come around the problem is to utilize an off-site solution such as a reverse-proxy or the technology of Software Defined Networks (SDNs) to name a few. It essentially offloads the setup of defenses to the network, thus not affecting the server or its admin.
The other approach is to increase the security of the server directly on the machine, for example by setting up complex firewalls, IP blocking thresholds in server configuration or installing server modules focusing on security. Designing and implementing one such module is the primary focus of this thesis, as many of the existing modules unfortunately fall short of being effective. This approach is admittedly more work for the server administrator than just having an SDN set up, but if the module manages to provide security without slowing down traffic, it does so without creating an additional single point of failure.
Now that I have presented the two approaches, let us take a look at some examples.

2.1.1 Off-site defenses

- SDN
  Even though I mentioned some positives, SDN setups like the one shown in figure 2.1 usually create a single point of failure, the SDN Controller, which could itself become the target of a DDoS attack [10] [11]. However, looking past that disadvantage which requires an attacker to modify his attack, the results SDNs showed in protecting against SDDoS are promising [7].

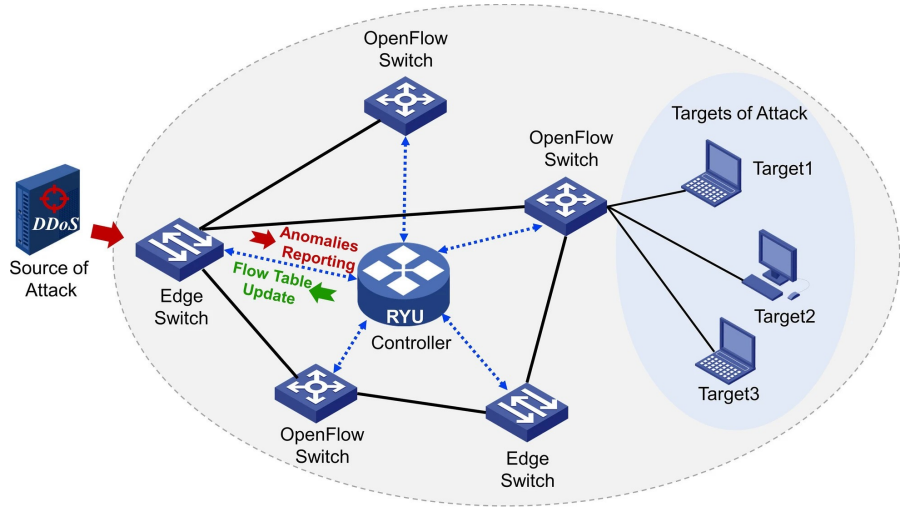

Figure 2.1 SDN architecture example [12]

- Reverse proxy servers
  While reverse proxy servers can improve a systems defense against SDDoS, it is essentially only offloading the problem to a different server. It can itself become a target of an attack. However, if the attacker does not target the proxy itself, it can prove to be a great defensive solution.

- Cloud-based protection
  Use a service that can function as a reverse proxy, protecting the origin server. This type of protection is mainly in pay-walled services[1], thus little information about how they operate is openly available. As such, I will not cover them any further.

- Load balancers and content switches
  In a similar fashion to increasing the resource capacity of a server, load balancers do improve the resistance to a smaller SDDoS attack. Nonetheless, when the attacks intensity reaches the resource limit, it is successful despite the defenses.

2.1.2 On-site defenses

- IP blocking thresholds
  Too strict of a limitation on the number of connections allowed per IP address can increase the false-positive ratio (FPR) [13], thus blocking legitimate users[2].

- physical firewalls
  They can be useful with (S)DDoS detection but can do very little to mitigate an attack. The firewall itself can become overwhelmed during an attack whether it is the target or not.

- Web application firewalls (WAF)
  WAFs often use device fingerprinting to identify malicious devices that are trying to connect. It gathers the information about new devices and decides if it is safe to continue the connection or terminate it and block the IP. [14] It can be effective against smaller attacks.

[1] There are free plans available which also provide a lot of protection.
[2] Those needing more connections or even several users using only one connection behind the same proxy

- ModSecurity
  As the name suggests, this open-source started as an Apache module, but has since been expanded to a standalone WAF. [15] It has been transferred to the custodianship of The OWASP foundation in 2024 [16].

- Server modules
  My main focus in this thesis are Apache2 modules, an on-site solution requiring a decent amount of configuration but otherwise coming with no additional costs. They are an extension to the core server software, either compiled with the server itself or dynamically during runtime. I will present a few examples below.

2.1.2.1 Examples of Apache2 modules

In the case of Apache Web servers, several modules can be employed to prevent damage from a Slowloris DDoS attack. [17] These modules include:

- Mod_limitipconn [18]
  "Allows web server administrators to limit the number of simultaneous downloads permitted from a single IP address." This module is by design ineffective against the distributed variant of SDoS.

- Mod_qos [19]
  This module provides the option of various resource management tools and configurations, giving the option of blocking unwanted requests and managing the servers resources more effectively.

- Mod_evasive [20]
  This module serves as a tool to detect and block unwanted requests based on their frequency and amount. If however the attack tool is configured to stay under such set limits, the module would not detect it as malicious activity.

- Mod_antiloris [21]
  The concept of this module is to hook into connection attempts and count the number of connections originating from the same remote IP that are in the SERVER BUSY READ state caused by GET[3] method requests. When the count exceeds a defined threshold, the connection is denied. Despite the information above, the module fails to protect against the Slowloris attack. The attacker can effortlessly bypass the protection by decreasing the value of the variable timeout to a value less than the modules expected value. The reason being that this module protects only against the SERVER BUSY READ attack. Slowloris also deploys an SERVER BUSY WRITE attack, via the POST method. [22]

- Mod_reqtimeout [23]
  It is one of the modules included by default in Apache HTTP Server v2.2.15 and up, providing a way to set timeouts and minimum data rates for receiving requests. Should a timeout occur or a data rate be to low, the corresponding connection will be closed by the server.

In research conducted by Moustis and Kotzanikolaou [22] the testing of these modules separately or in various combinations[4] showed, that server configuration techniques are not adequate by themselves to effectively protect from a dedicated adversary. If used in combination, the modules may be successful in repulsing the attack, but the delay in server response showed to be severe even with a small attack scenario of four bots, indicating that a larger attack would affect the server.

[3] explained in 1.1.2
[4] for example both Mod_limitipconn and Mod_antiloris

2.2 Impact of SDDoS on current web servers

Luckily, the current impact of the SlowLoris and Slow POST attacks is minimal, thanks to the servers' ability to recognize that the prolonging of the connection should not keep a timeout from triggering, thus an attack is denied by the KeepAliveTimeout [24] as long as the number of connections does not trigger a regular DDoS.

This is not the case with the third presented SDDoS attack type, the Slow Read attack. In figure 2.2 a stable 400 seconds long connection created by the attack tool can be seen even though the server does have timeouts configured.

Test parameters	
Test type	SLOW READ
Number of connections	1
Cookie	
Receive window range	1 - 512
Pipeline factor	1
Read rate from receive buffer	5 bytes / 1 sec
Connections per seconds	200
Timeout for probe connection	20
Target test duration	400 seconds
Using proxy	no proxy

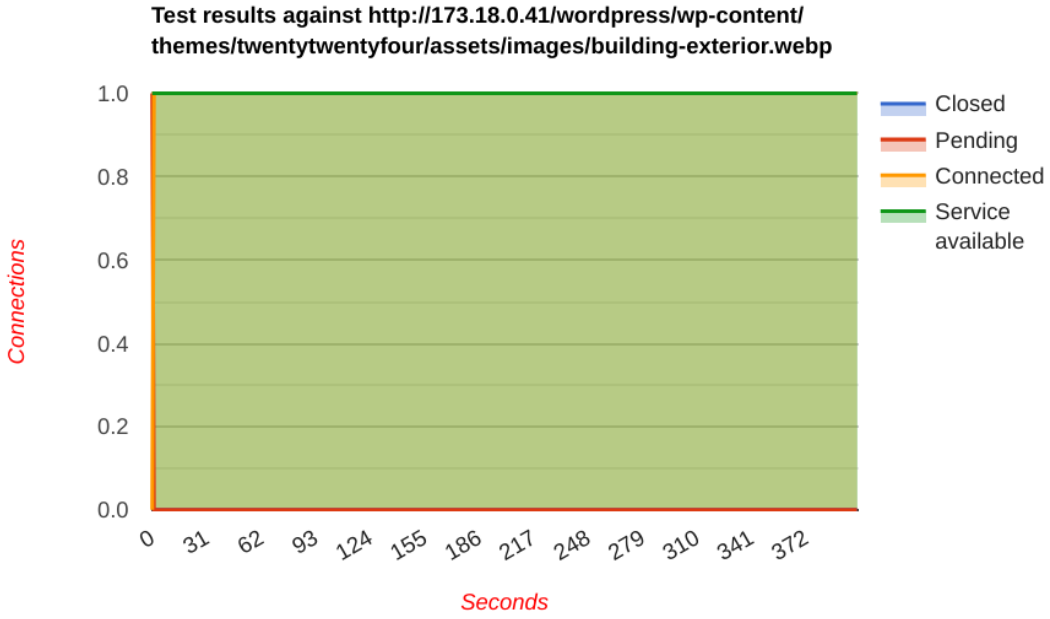

Figure 2.2 One connection connected for 400 seconds

This attack type is in a transport layer attack its core. While there is HTTP communication not found in other transport layer attacks, the connection prolonging leading to a denial of service happens purely with TCP communication, whilst no other HTTP packets are sent.

Even though SDoS is a type of attack first described in 2005 in the "Programming Model Attacks" section of Apache Security [5], the impact of Slow Read attacks prevails heavily in current

systems. Even the revision of the HTTP protocol HTTP/2 did not introduce any solutions to the problem[5] according to the large security investigation done by the Federal Communications Commission [25].

The large threat Slow Read attacks pose in comparison to the other two attack types is the reason why in the rest of the thesis I will focus solely on the Slow Read attack type.

2.3 Attack testing tools

2.3.1 slowhttptest

"SlowHTTPTest is a highly configurable tool that simulates some Application Layer Denial of Service attacks by prolonging HTTP connections in different ways." [26]
It is the main attack tool [27] available; by default installed on Kali Linux distributions. It provides four attack approaches[6]:

Test mode options	
-H	slow headers a.k.a. Slowloris (default)
-B	slow body a.k.a R-U-Dead-Yet
-X	slow read a.k.a Slow Read
-R	range attack a.k.a Apache killer

The tool indeed is highly configurable, it provides an extensive variety of options to alter the attack process.

General options			
flag	unit	description	default value
-c	connections	target number of connections	50
-i	seconds	interval between followup data in seconds	10
-l	seconds	target test length in seconds	240
-r	rate	connections per seconds	50
-s	bytes	value of Content-Length header if needed	4096
-t	verb	verb to use in request, default to GET for slow headers and response and to POST for slow body	GET / POST
-x	bytes	max length of each randomized name/value pair of followup data per tick, e.g. -x 2 generates X-xx: xx for header or &xx=xx for body, where x is random character	32
-f	content-type	value of Content-type header	application/x-www-form-urlencoded
-m	accept	value of Accept header	text/html;q=0.9, text/plain;q=0.8, image/png,*/*;q=0.5
-e	x.x.x.x:xx	address of proxy to direct all probing traffic into	

Since the first 2 attack subtypes are defended against by default timeouts, I will focus on explaining how the tool creates and operates Slow Read attacks.

[5] CVE-2016-1546, CVE-2020-9481
[6] The -R range attack a.k.a Apache killer approach is not in the scope of slow DDoS attacks as it only exploits a vulnerability of Apache servers as a DDoS attack, the vulnerability has also been patched out. [28]

Slow Read attack mode specific options			
flag	unit	description	default
-k	num	number of times to repeat same request in the connection. Use timeout multiply response size if server supports persistent connections	1
-n	seconds	interval between read operations from recv buffer in seconds	1
-w	bytes	start of the range advertised window size would be picked from	1
-y	bytes	end of the range advertised window size would be picked from	512
-z	bytes	bytes to slow read from receive buffer with single read() call	5

Even though I explained the principle of the Slow Read attack in the previous chapter, I would like to take a closer look on how the slowhttptest tool implements it.
It can be seen in the tools source code [29] and when running the tool with verbosity level 4, that it works with 2 different socket types called the "slow socket" and "probe socket". As the names suggest, the slow sockets are focused on performing the attack, usually in large numbers, their amount corresponds to the number of connected connections with the max being the target set by option -c. However, the probe socket is usually only a single socket focused on normal requests towards the server. Based on its response (or lack thereof), it determines the availability of its services.
The slow sockets, on the other hand, inform the server about their very small tcp window size value (and usually their window size scaling factor of 1) as highlighted in figure 2.3.

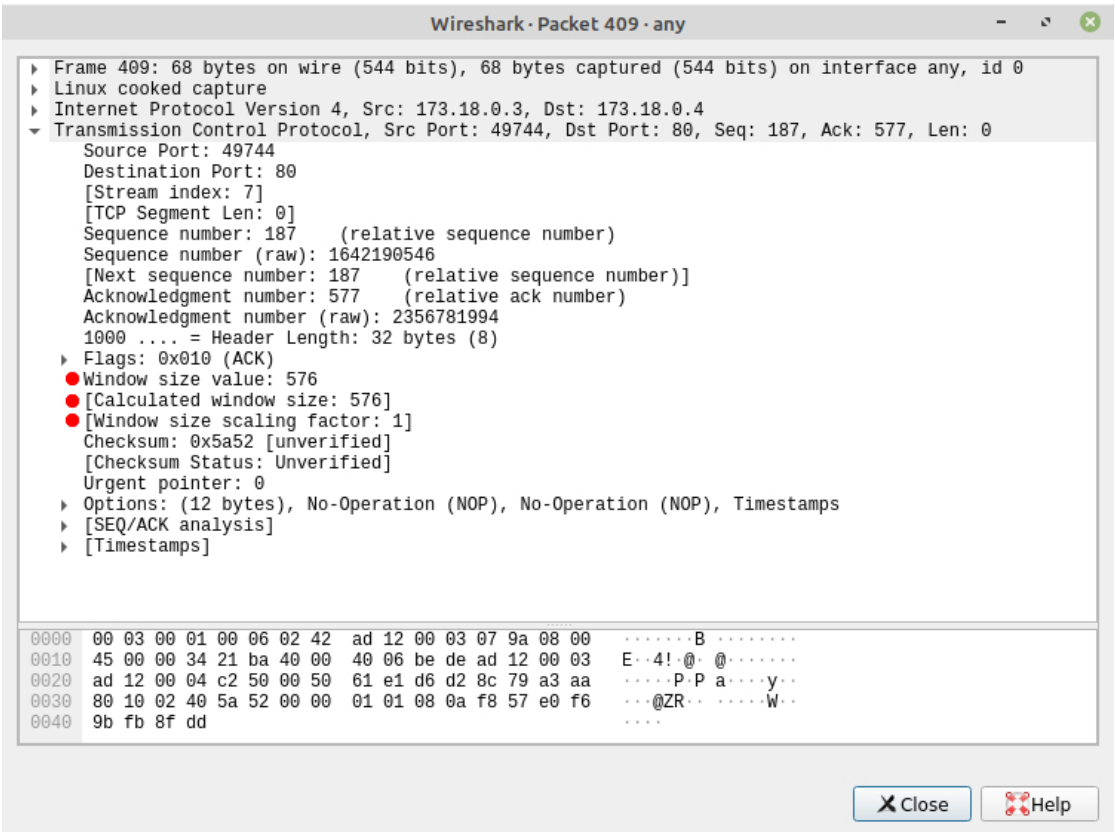

Figure 2.3 TCP packet during the attack

Sometimes the value of the window size is set to zero, but since that could lead to a simple detection mechanism for the defender, the attacking socket uses the -z option. Every -n seconds,

-z bytes of data are read from the receive buffer, allowing the attacking socket to advertise a window size slightly larger than zero but still small enough for the attack to have impact.

2.3.1.1 An example of a successful SDDoS Slow Read attack

Now I am going to show an example of the attack successfully denying the servers service using a python script to control a docker simulated botnet described in a later chapter. The slowhttptest tool generates very informative graphs about how the attack went, however, the tool was designed as a SDoS tool, not providing the possibility of distributing the attack across multiple devices while keeping the graph statistics correct. As seen in figure 2.4, the service was not available for a period of 140 seconds out of 150 seconds the attack was in progress. Which is actually correct information, however, the information about the number of pending, closed and connected connections is only corresponding to one attacking container the information was taken from. The real number of connections attempted is *number_of_attackers* * *number_of_connections_parameter* (which in this example would be 8 * 137 = 1096).

Test parameters	
Test type	SLOW READ
Number of connections	137
Cookie	
Receive window range	141 - 527
Pipeline factor	1
Read rate from receive buffer	2 bytes / 6 sec
Connections per seconds	79
Timeout for probe connection	5
Target test duration	150 seconds
Using proxy	no proxy

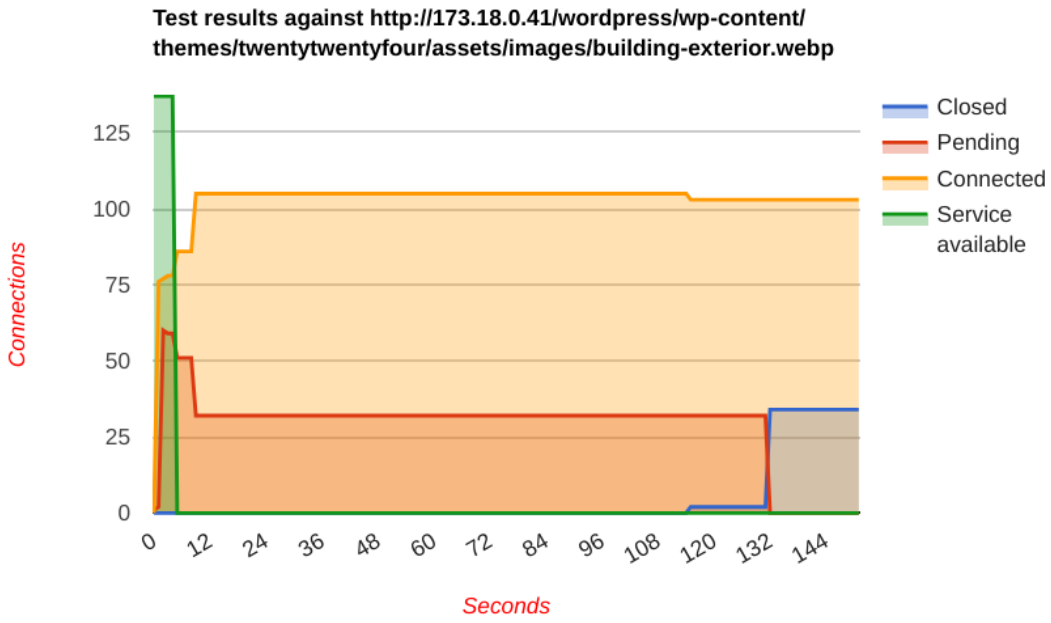

Figure 2.4 An output graph of a successful SDDoS attack using 8 bots with 137 Slow Read connections each attacking one server for 150 seconds.

2.3.2 slowloris

This tool itself is the origin of the Slowloris attack, however it is not generally used for the attack[7]. Slowhttptest itself provides an option to use the Slowloris attack type.

[7]The original website http://ha.ckers.org/slowloris/ where it was published has since shut down.

Chapter 3

WebServer Apache2 Module Architecture

3.0.1 Creating an Apache module

The main source of my knowledge how to create an Apache 2.4 module comes from the guide Developing modules for the Apache HTTP Server 2.4 [30], an official development guide provided by The Apache Software Foundation.

3.0.1.1 What an Apache module is

An Apache module is essentially included in the Apache core [31], either statically loaded in server compile-time or (if dynamic module loading is allowed) compiled separately and loaded into the server dynamically using the directive **LoadModule** [32].

The server core calls module handlers in its registry, while the modules can use the Apache API to communicate with it. As such, they gain access to important information like *request_rec* or to the memory pool of the request. [31]

Having such information and resources available to their disposal, these modules provide a variety of options to extend the features of an Apache server, whether it is for monitoring, authentication, logging or, as in my case, security.

```
module AP_MODULE_DECLARE_DATA   example_module =
{
    STANDARD20_MODULE_STUFF,
    create_dir_conf, /* Per-directory configuration handler */
    merge_dir_conf,  /* Merge handler for per-directory configurations */
    create_svr_conf, /* Per-server configuration handler */
    merge_svr_conf,  /* Merge handler for per-server configurations */
    directives,      /* Any directives we may have for httpd */
    register_hooks   /* Our hook registering function */
};
```

Figure 3.2 Module declaration [30]

As is mentioned in the guide, a module declaration is necessary in every Apache module. A module declaration includes mainly:

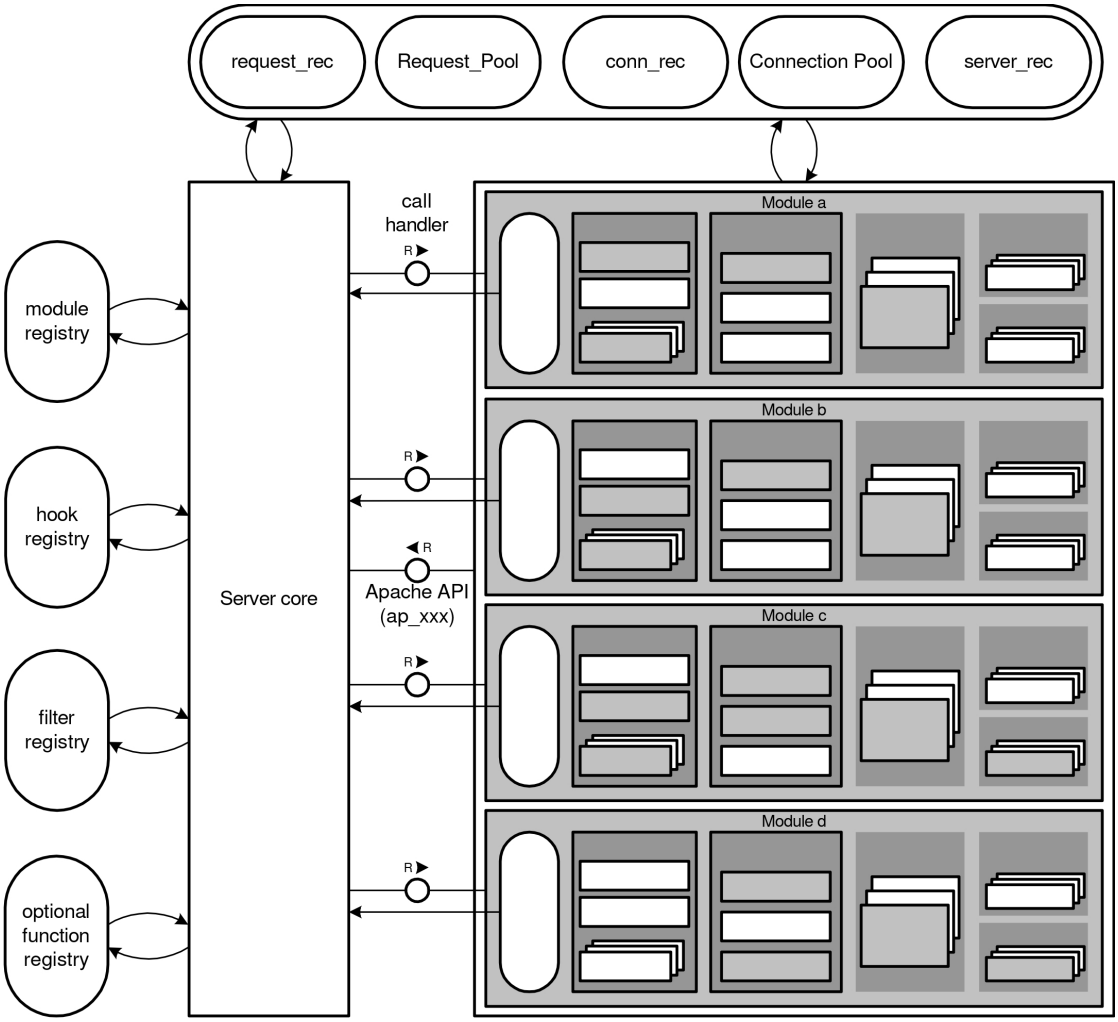

Figure 3.1 Interaction of Apache Core and Modules [31]

- Various configuration handlers
  A module can use a configuration file consisting of directives explained below. However, the configuration file sometimes needs to be context sensitive, hence why multiple handlers are present in the declaration.
- httpd directives
  Directives which are simply put a plain-text way to tell the module how to behave in certain situations. In a declaration a list of the directives to which a module responds is provided.
- hook registering function
  A function calling listed hook functions, the hooking process is explained in the subsection below.

3.0.2 Hooking into the request handling process

A hook works essentially like triggering an event which results in event handler execution. The format of a register hook function is as follows [33]:

```
ap_hook_phase_name(function_name, predecessors, successors, position);
```

There are many different phases of request receiving a hook can be called in, with the corresponding *phase_names*: *pre_config* (to do any setup required prior to processing configuration directives), *child_init* (which is called as soon as the child is started) or *quick_handler* (called before any request processing, often used by cache modules) just to name a few.
The *function_name* refers to the function called by the request hook function if a hook happens. The *position* is one of the five defined values: HOOK_REALLY_FIRST, HOOK_FIRST, HOOK_MIDDLE, HOOK_LAST, HOOK_REALLY_LAST combined with the list of predecessors and successors creates an order in which the hook functions of the same phase are called.

3.0.3 A module handler

A handler is a function receiving a callback from the hook caller inside the server, with the request record structure *request_req* as input. This structure is the most essential part of a request as it contains all of the information about it, including its memory pool, connection, the HTTP request itself and many other things [34].

An important detail of the handler is its return value, which is either general information to the server:

General response codes	
DECLINED	The request will not be handled by this module.
OK	The request has been successfully handled.
DONE	The request has been successfully handled and the thread should be closed immediately.

or specific HTTP return codes [35] which will be in the servers response.

Chapter 4

Defense design and implementation

4.1 Detection

After observing attack data and reading through the blog of the slowhttptest tool author [9], I have found that the best way to differentiate attack traffic from regular traffic is the tcp.window.size. As explained in the previous chapter, the attack forces the server to continually send data by informing it of a very small receive buffer. It does so by setting the tcp.window.size to extremely small values or to zero.
While it is possible to observe tcp.window.size being zero in normal client traffic, for example due the client being overwhelmed, it is however extremely unusual for it to appear in most of the clients responses [36].

There were multiple attempts to read the tcp.window.size in order to be able to react to it. I will go through each of the below and explain why it ultimately did or did not work in the end.

Module
First attempt to detect the tcp.window.size located in tcp headers was made in the module system of Apache2 as it is the main goal of this thesis. I tried to use the apr_network_io.h library to connect to the socket in use and extract the data, not yet knowing the data would probably not be there nevertheless, as explained later.
However, the Apache module system is limited by hooks. As explained above and shown in figure 4.1, hooks only happen in some stages of the request-response loop, but the Slow Read attack proved to be happening after any of the hooks could intercept it. The malicious communication is only detectable on the transport layer. So even after connecting to the used socket, I was unable to extract any useful communication as it was happening at different times than when I was able to listen to it.

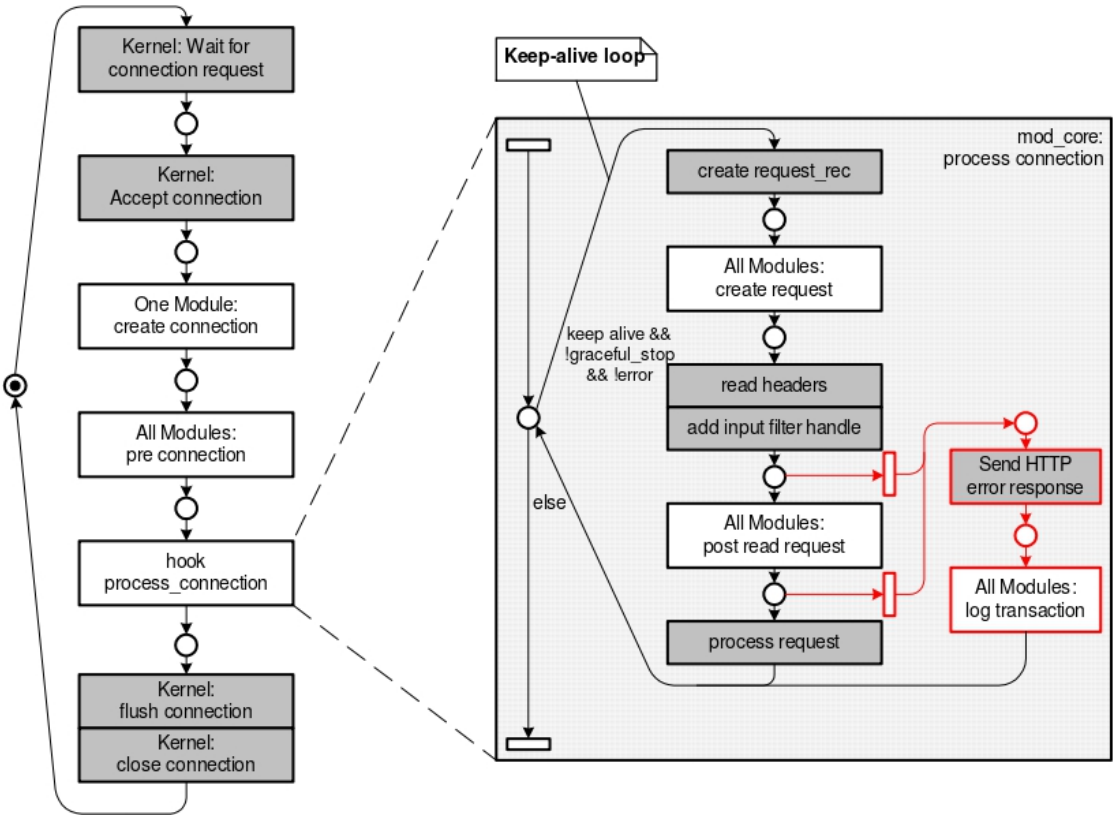

Figure 4.1 Apache request-response loop [37]

Flask framework Reverse Proxy

Because I believed time was the issue while listening to the socket, I decided to try a solution using a reverse proxy. What I hoped this would accomplish is to intercept the incoming packets, extract the TCP headers with the tcp.window.size included in them and react accordingly in the proxy itself. Using iptables as shown below, I redirected incoming traffic from port 80 to port 5000.

```
iptables -A PREROUTING -t nat -i eth0 -p tcp --dport 80 -j REDIRECT
--to-port 5000
```

At port 5000 I setup a reverse proxy [38] (the code can be found in attachments) to provide communication between the client and the service residing at port 80 without the client being able to notice a change in traffic. Even though the Flask framework [39] based on which the proxy is developed was very useful in setting up the forwarding, after using Flask to listen to the socket and read the data before passing it further, I was only able to extract http packets, stripped from any other (including TCP) information. This was because the framework is ultimately based on the application layer and it does not deal with TCP data at all.

Python socket Reverse Proxy

Keeping in mind that I need to focus on the transport layer, I stopped trying to read tcp data with the Flask framework. Initially, thanks to the online guide from Shanto Roy [40] I created a TCP proxy using the python socket module, hoping to read unprocessed data from the socket itself. With the initialization part of the code shown in figures 4.2 I unfortunately found the

socket to only read processed data in the connection, thus missing the searched for TCP headers again.

To be able to read the transport layer data I needed, I would have to create a socket of type SOCK_RAW instead of SOCK_STREAM as in figure 4.2.

```
76  if·__name__·==·"__main__":
77  ····args·=·option_check()
78  ····#·Create·a·socket·object
79  ····s·=·socket.socket(socket.AF_INET,·socket.SOCK_STREAM)
80  ····host·=·'173.18.0.41'
81  ····port·=·int(args[0])#·Reserve·a·port·for·your·service.
82  ····print("Running·the·reverse·proxy·on·port",·port)
83  ····#·Binds·to·the·port
84  ····s.bind((host,·port))
85  ····s.listen(1000)
86  ····while·True:
87  ····|···#·Establish·connection·with·client.
88  ····|···c,·addr·=·s.accept()·
89  ····|···#·lock·acquired·by·client
90  ····|···print_lock.acquire()
91  ····|···_thread.start_new_thread(on_new_client,(c,addr))
92  ····s.close()
```

Figure 4.2 TCP Proxy initialization and listen loop

I then, however, found out from MSDN documentation [41] and a helpful forum explanation [42] that to use SOCK_RAW socket type you have to give up a TCP connection because raw sockets are essentially connection-less. For my proxy to work, I need the TCP connection with the client, thus using the python socket was not an option.

TCP sniffer

I abandoned the idea of using a reverse proxy for the solution and opted to focus on the popular TCP sniffing tool **tcdump** [43] and the C/C++ network traffic capture library libpcap it uses. There is a basic packet sniffing tool available [44] which proved to be exactly what I needed, giving me the ability to read the TCP packets, their headers and therefore the tcp.window.size while not interfering with the communication. In figure 4.3 the tool packet processing code can be seen.

```
190 // Advance to the transport layer header then parse and display
191 // the fields based on the type of hearder: tcp, udp or icmp.
192 packetptr += 4*iphdr->ip_hl;
193 switch (iphdr->ip_p)
194 {
195 case IPPROTO_TCP:
196     tcphdr = (struct tcphdr*)packetptr;
197     printf("TCP  %s:%d -> %s:%d\n", srcip, ntohs(tcphdr->th_sport),
198             dstip, ntohs(tcphdr->th_dport));
199     printf("%s\n", iphdrInfo);
200     printf("%c%c%c%c%c%c Seq: 0x%x Ack: 0x%x Win: 0x%x TcpLen: %d\n",
201             (tcphdr->th_flags & TH_URG ? 'U' : '*'),
202             (tcphdr->th_flags & TH_ACK ? 'A' : '*'),
203             (tcphdr->th_flags & TH_PUSH ? 'P' : '*'),
204             (tcphdr->th_flags & TH_RST ? 'R' : '*'),
205             (tcphdr->th_flags & TH_SYN ? 'S' : '*'),
206             (tcphdr->th_flags & TH_SYN ? 'F' : '*'),
207             ntohl(tcphdr->th_seq), ntohl(tcphdr->th_ack),
208             ntohs(tcphdr->th_win), 4*tcphdr->th_off);
209     printf("+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+\n\n");
```

Figure 4.3 The code in the sniffing script processing the TCP protocol

And in figure 4.4 it can also be seen what the output of the process looks like.

```
TCP  173.18.0.41:80 -> 173.18.0.3:48476
ID:57453 TOS:0x0, TTL:64 IpLen:20 DgLen:52
*A**** Seq: 0x2d84084d Ack: 0x4665d3f6 Win: 0x1fe TcpLen: 32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.4 An example of the TCP sniffing tool outputting one packet

As already mentioned, if a client sets tcp.window.size to zero a few times, that does not necessarily mean it is an attacker [36]. The attackers, however, need to send the tcp.window.size with value zero[1] a large amount of times for the attack to be successful. I decided to then count that amount, if it exceeded a configurable threshold a regular client would not normally exceed in a certain configurable number of connections, I could confidently mark that source IP address as an attacker and respond accordingly. It is important to look at the ratio of total number of connections to the number of malicious packets, if I only set a hard limit for the number of malicious packets even a regular user could possibly reach that limit given enough time. This way I ensure that only the real attackers sending too many malicious packets in their short communication are marked. In the script code I named this value RATIO_TRESHOLD as seen in figure 4.5.

```
#define RATIO_TRESHOLD 0.2 //The ratio of malicious connections / total connections
#define TCP_WINDOW_SIZE_LIMIT 50 //How small the tcp.window.size value is considered possibly malicious
```

Figure 4.5 Configurable script values

Because the tool is written in C language and not C++, for convenience I copied a map like structure from [45] and used it to associate the amount of small tcp.window.size packets with the IP address.

4.2 Response

At first it seemed to me that a simple iptables command as such should be enough of a mitigation:

[1] or very small but non-zero as already mentioned

```
iptables -A INPUT -s <src.ip.address> -p tcp -m conntrack --ctstate ESTABLISHED
-j REJECT
```

I hoped it would force the server to refuse any other incoming packets from the attacker. However, that was not the case. While this command does block any future connections from the attacking IP address, it does not affect existing TCP connections in any way. I found that forcing a TCP connection to a close without the approval of the client party proved more difficult than I thought at first.
I ended up not only blocking future connections with the command above but also using the ss command [46] with the –kill option using the already collected source IP address of the connection:

```
ss --kill dst <src.ip.address>
```

which forcibly closes sockets, thus killing the current TCP connection.

4.2.1 Creating logs

I decided to use the Apache module system in the logging process. My script communicates with my module through a named pipe. If an attack is blocked, the script sends information to the module containing the time of the attack, attackers IP address, the number of malicious packets and the total number of packets received from the address.

4.2.2 Created logging module

Using all the above mentioned knowledge, I created a logging Apache module (mostly seen in figure 4.7) utilizing the APR library to create logs in the /var/log/apache2 directory.
The log_transaction hook is triggered every time Apache is about to log information about a request. At that point, all the available data provided by our defense script in our named pipe /var/log_attack is read. However, if this module is not running, the defense script can be left hanging, trying to write into a named pipe with a full buffer. This is a bug which should be fixed in the future.

```
root@5ecb770b7d90:/var/log/apache2# cat security.log | head -n 10
Wed May 15 13:55:00 An attack attempt from: 173.18.0.3 was blocked. # of malicious packets: 5 total # of packets: 47
-----------------------------------
Wed May 15 13:55:00 An attack attempt from: 173.18.0.3 was blocked. # of malicious packets: 6 total # of packets: 52
-----------------------------------
Wed May 15 13:55:00 An attack attempt from: 173.18.0.3 was blocked. # of malicious packets: 7 total # of packets: 57
-----------------------------------
Wed May 15 13:55:00 An attack attempt from: 173.18.0.3 was blocked. # of malicious packets: 8 total # of packets: 61
-----------------------------------
Wed May 15 13:55:00 An attack attempt from: 173.18.0.3 was blocked. # of malicious packets: 9 total # of packets: 65
-----------------------------------
root@5ecb770b7d90:/var/log/apache2#
```

Figure 4.6 Example of created log files

As seen in figure 4.6, it takes a little time before the defense script can call the *ss –kill* command and for it to successfully kill the socket with the connection. This leads to more logging of information about the same attack.

```
26  static int security_log_handler(request_rec *r);
27
28  /* register_hooks: Adds a hook to the httpd process */
29  static void register_hooks(apr_pool_t *pool)
30  {
31
32      /* Hook the request handler */
33       ap_hook_log_transaction(security_log_handler, NULL, NULL, APR_HOOK_MIDDLE);
34
35  }
36
37  static int security_log_handler(request_rec *r)
38  {
39      int fd;
40      char * myfifo = "/tmp/log_attack";
41      mkfifo(myfifo, 0666);
42      apr_size_t buffer_size = 5000;
43      char str [buffer_size];
44      apr_file_t * log_file;
45
46      fd = open(myfifo,O_RDONLY);
47      read(fd, str, buffer_size);
48
49      apr_status_t file_open = apr_file_open(
50          &log_file, // new file handle
51          "/var/log/apache2/security.log", // file name
52          APR_FOPEN_CREATE | APR_FOPEN_EXCL | APR_FOPEN_WRITE | APR_FOPEN_APPEND
53           | APR_FOPEN_XTHREAD,
54          APR_OS_DEFAULT, // permissions
55          r->pool // memory pool to use
56      );
57      apr_file_write(log_file, str, &buffer_size);
58
59      apr_file_close(log_file);
60
61      return OK;
62  }
```

Figure 4.7 Apache module code

4.3 Testing

4.3.1 Simulating a botnet in Docker

For a proper simulation of an SDDoS I needed a simulated botnet to attack a server all at once. For this I chose to create a docker network with a server, a regular user and 8 attacking containers all responding to commands from the host.
In figure 4.8 I visualised the created docker environment and the communication between the containers. The visualised **Host** is not only the host in the sense of the botnet but also the host of the entire docker environment.

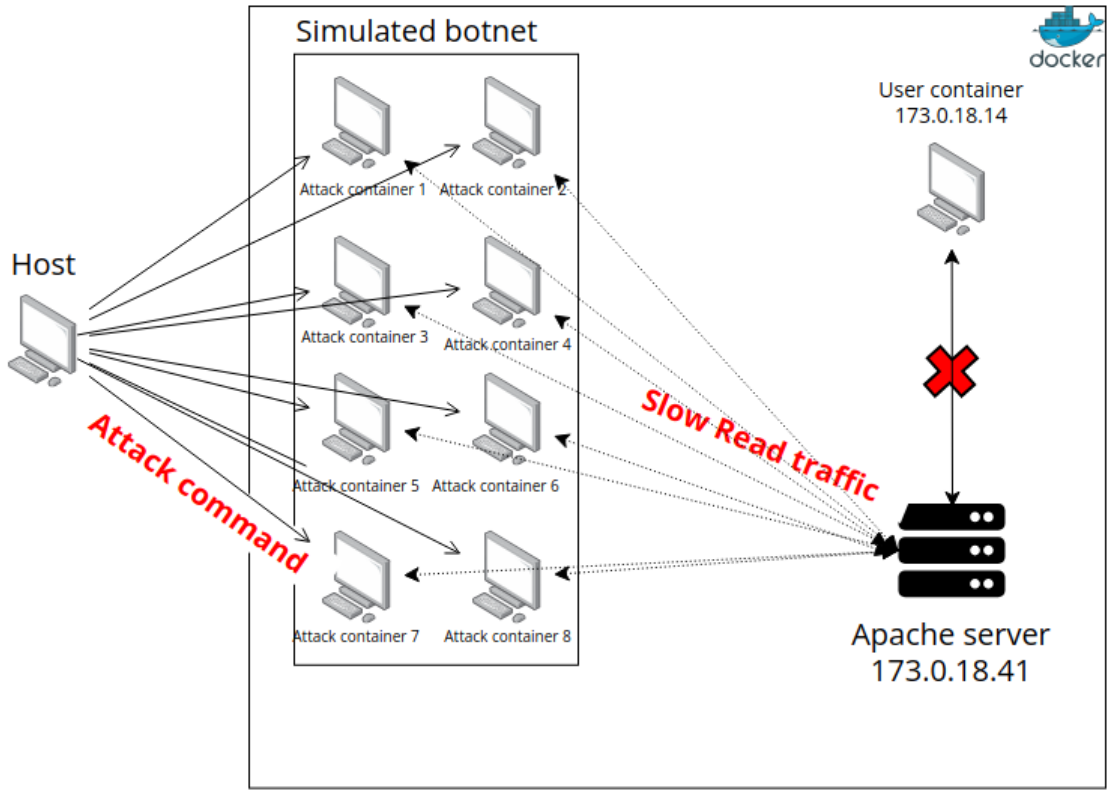

Figure 4.8 Botnet simulation

The user at 173.0.18.14
This is a regular Ubuntu Linux container making usual requests from the server, eventually used as a probing container.
The server at 173.0.18.41
This is a Ubuntu Linux container running an Apache 2 server, with a functioning Wordpress application and a MySQL database to make sure a more complex application is functioning properly even under attack.
The attacking containers at 173.0.18.5-13
These are slowhttptest containers waiting for a command from host to attack the server. I will later refer to them as **bots**

The python script to control the botnet as a simulated bot master is included in the attachments, however, the main attacking command is:

```
slowhttptest -c <#.connections> -X -g -o attack_stats -n 1 -r
200 -u <element.url>
```

distributed to the bots by the docker utility:

```
docker exec -it <container.id> sh -c '<command>'
```

Unfortunately, I was not able to make the -e option of the slowhttptest tool work, which would direct all probing traffic through a proxy, thus allowing me to check for the services availability with the script. Without it, the iptables rule to block all incoming traffic from the attackers IP address also blocks the probing request. This makes the service seem unavailable in the attack tool graphs even when it is working perfectly well.

To mitigate the tool not working properly, I created a simple bash script of my own residing at the user container, to check for the availability of the service during the attack. As you can see in figure 4.9, I used a curl command requesting a large element on the website with the option **–max-time** which is essentially just a timeout for the connection, if it times out after 4 seconds, I can confidently say the service is unavailable.

```
#!/bin/bash
date +%H:%M:%S > probing_out.txt
echo "test starting" >> probing_out.txt
for i in {1..100}
do
   echo -n $(date +%H:%M:%S) >> probing_out.txt
   OUTPUT=$(curl "173.18.0.41:80/wordpress/wp-content/themes/twentytwentyfour/assets/images/building-exterior.webp" \
 -vL --output image --max-time 4 2>&1 >/dev/null | grep ") Connection timed out after"); COUNT=$(echo $( echo $OUTPUT | wc -m))
   if [ "$COUNT" -gt "5" ]
   then
     echo " timeout" >> probing_out.txt
   else
     echo " service available" >> probing_out.txt
   fi
   sleep 5
done
```

Figure 4.9 Simple probing script

4.3.2 Having a reverse proxy

While trying to get TCP headers data, one of the approaches was to try and read it while forwarding the packets through a reverse proxy. While this did not have the desired effect, it did, however, prove to be extremely effective in defending against the attack by itself. When the reverse proxy was active, up to 18 000 connections were established to the server without the service going down, with the possibility of more connections with more resources as my hardware became the bottleneck. This solution should definitely be looked into, however, it does not fall into the scope of my thesis.

Chapter 5

Comparison of the results

5.1 Testing

I will run the test with these variable attack parameters:

- Number of bots attacking the victim at the same time
- Number of connections each bot is trying to establish
- Rate of connections per second each bot is creating
- The attacks -z option, meaning how much will the bot read from its buffer in order to make the tcp.window.size slightly larger

For this test, the defense script parameters were:

RATIO_TRESHOLD = 0.2
TCP_WINDOW_SIZE_LIMIT = 50

# bots	# conn / bot	# conn rate	-z value	service up
1	1000	50	5	100%
8	100	10	10	100%
8	1000	100	5	15%

As seen in the table, 8000 total connections proved to be too many for the defense script to handle. As soon as the amount became unbearable for the server a denial of service occurred (hence the 15% uptime before all connection have been made). Interestingly, the server did not try to close the connections at all as the threshold was never reached. That is probably because establishing all the connections to the server initially sends legitimate packets, thus lowering the total packet to malicious packet ratio. It does it enough not to be detected before the server was able to react like in the first test with only one bot. This theory is supported be the next test.

But just to show an example of the script at work with the first test, in figure 5.1 I would like to show a section of the output the script produced during the second attack of the table above. First half of the screenshot is the current state of the counting map, where each IP address shows the amount of malicious packets already sent. In the middle section, there is the output of the sniffer script which I modified, showing an incoming TCP packet from 173.8.0.7

with a tcp.window.size of zero. The lower part shows the script outputs the current ratio of total packets to malicious packets. Since that ratio is higher than the threshold set, the script kills the sockets associated to the address and bans all future connections.

```
 current small window count: 173.18.0.41: 0
173.18.0.5: 8
173.18.0.12: 17
173.18.0.11: 24
173.18.0.10: 0
173.18.0.8: 0
173.18.0.6: 37
173.18.0.3: 56
173.18.0.9: 88
173.18.0.7: 87
173.18.0.14: 2
TCP  173.18.0.7:49608 -> 173.18.0.41:80
ID:10080 TOS:0x0, TTL:64 IpLen:20 DgLen:52
*A**** Seq: 0xdb29907e Ack: 0x5c9fa2bc Win: 0x0 TcpLen: 32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Current ratio: 0.615385
running this command now: ss --kill dst 173.18.0.7 >/dev/null 2>/dev/null &
return value: 0
running this command now: iptables -A INPUT -s 173.18.0.7 -p tcp -m conntrack --ctstate ESTABLISHED -j REJECT
return value: 0
```

Figure 5.1 Test two script output

For the next test, I reduced the ratio threshold from 20% to 10% with leaving the attack parameter unchanged, in order to test my theory. The script parameters then were:

RATIO_TRESHOLD = 0.1
TCP_WINDOW_SIZE_LIMIT = 50

# bots	# conn / bot	# conn rate	-z value	service up
1	1000	50	5	100%
8	100	10	10	100%
8	1000	100	5	100%

As visible in figure 5.2 the script was able to react much faster (after 5 malicious packets already), in fact even faster than the attacking tool was to start sending packets from all the bots, we see only two bot IP addresses detected so far (with 173.18.0.14 being my probing user).

```
 current small window count: 173.18.0.14: 0
173.18.0.41: 0
173.18.0.11: 5
TCP  173.18.0.11:43166 -> 173.18.0.41:80
ID:13483 TOS:0x0, TTL:64 IpLen:20 DgLen:52
*A**** Seq: 0x7bc0f773 Ack: 0x8dbb62f3 Win: 0x0 TcpLen: 32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Current ratio: 0.115385
running this command now: ss --kill dst 173.18.0.11 >/dev/null 2>/dev/null &
return value: 0
running this command now: iptables -A INPUT -s 173.18.0.11 -p tcp -m conntrack --ctstate ESTABLISHED -j REJECT
return value: 0
```

Figure 5.2 Test six defense script output

This next test is meant to show how much the RATIO_TRESHOLD affects the ability of the server to react to the attack.
Even a single attacking bot can take the service down as the ratio never reaches 50%. Like before, the service up-time is not 0% due to the slow connection rate, thus the service keeps

responding to the probe during the initial phase of the attack.
RATIO_TRESHOLD = 0.5
TCP_WINDOW_SIZE_LIMIT = 50

# bots	# conn / bot	# conn rate	-z value	service up
1	1000	50	5	20%

5.2 Evaluating the value of the created solution

I was unable to create an Apache module which would itself be the defense solution due to the nature of the attack essentially focusing on the transport layer of the network. This means an Apache module system working primarily on the application layer was defensively ineffective, so I opted to create a logging system to the defense script.
With that in mind, I would like to evaluate the created defense script. While the script needs more testing to be done, especially testing for false positives, the tests done so far proved the script to be a viable defense solution against the Slow Read DDoS attack. With the high configurability of its sensitivity to attacks in the RATIO_THRESHOLD and TCP_WINDOW_SIZE_LIMIT parameters, I believe it to be a flexible enough solution to eventually compete with other SDDoS solutions, given enough time and work to make it safe for production deployment.

Chapter 6

Summary

The goals of this thesis were to explore in detail the current state of research in the field of SDDoS attacks and their impact on the web servers, both of which I have successfully done in chapter 2. In this chapter, I have also accomplished the second goal of the thesis which was to identify the characteristics of a typical SDDoS attack. I have explained in detail both the differences and similarities of the three most common SDDoS attack subtypes. I also explained the principles of how each of them perform their communication leading to denial of service and evaluated their current impact on servers. This led me to focus in detail on the Slow Read attack subtype in later chapters, as its impact was by far larger compared to the other two.
My next goal was to design and implement an Apache 2 web server module, capable of SDDoS attack detection and mitigation. I have not accomplished this goal as I found out the Slow Read attack is at its core a transport layer attack. This made the Apache module system - which works mostly on the application layer - very ineffective in defense solutions.
However, I have utilized the Apache module in creating a logging mechanism for a defense script. After a long process of trying to detect the attack I was successful with a script based on the C/C++ network traffic capture library libpcap, just like the very popular tool tcpdump. The script is able to successfully mark users as attackers based on the ratio of their total communication to malicious communication.
I then accomplished my next goal by performing testing of my solution with a docker container simulation including a botnet and a server running a functioning Wordpress application with a MySQL database. It proved to be an effective and flexible solution configurable by its parameters.
My final goal of the thesis was to compare the testing results with the existent SDDoS attack defense methods and evaluate the contribution of the created solution. I have not compared the testing results to other attack defense methods but I have evaluated the solution to be a success and even though further testing is needed, it could prove to be an essential tool in the defense against the Slow Read DDoS attack.

Bibliography

1. NIELSEN, Henrik; MOGUL, Jeffrey; FIELDING, Larry M Masinterand Roy T.; GETTYS, Jim; LEACH, Paul J.; BERNERS-LEE, Tim. *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1*. 1999. Available also from: `https://datatracker.ietf.org/doc/html/rfc2616`. (Accessed on 03/25/2024).
2. *Understanding Threads, Processes, and Connections (Oracle iPlanet Web Server 7.0.9 Performance Tuning, Sizing, and Scaling Guide)*. Copyright 2010 Oracle Corporation. Available also from: `https://docs.oracle.com/cd/E19146-01/821-1834/geeie/index.html`. (Accessed on 04/04/2024).
3. *Top 10 DDoS Attacks - SOCRadar® Cyber Intelligence Inc.* Copyright 2024 SOCRadar. Available also from: `https://socradar.io/top-10-ddos-attacks/`. (Accessed on 04/06/2024).
4. DEOLINDO, Vinícius M.; DALMAZO, Bruno L.; SILVA, Marcus V.B. da; OLIVEIRA, Luiz R.B. de; B. SILVA, Allan de; GRANVILLE, Lisandro Zambenedetti; GASPARY, Luciano P.; NOBRE, Jéferson Campos. Using Quadratic Discriminant Analysis by Intrusion Detection Systems for Port Scan and Slowloris Attack Classification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2021, vol. 12951 LNCS, pp. 188–200. ISBN 9783030869694. ISSN 16113349. Available from DOI: `10.1007/978-3-030-86970-0_14`.
5. RSNAKE. *GitHub - XCHADXFAQ77X/SLOWLORIS: Slowloris HTTP DoS RSnake*. 2016. Available also from: `https://github.com/XCHADXFAQ77X/SLOWLORIS/`. (Accessed on 04/04/2024).
6. ZDRNJA, Bojan. *Slowloris and Iranian DDoS attacks - SANS Internet Storm Center*. 2009. Available also from: `https://isc.sans.edu/diary/Slowloris+and+Iranian+DDoS+attacks/6622`. (Accessed on 03/25/2024).
7. LUKASEDER, Thomas; MAILE, Lisa; ERB, Benjamin; KARGL, Frank. SDN-Assisted Network-Based Mitigation of Slow DDoS Attacks. [N.d.]. Available also from: `http://uni-ulm.de/in/vs`.
8. SHEKYAN, Sergey. *Owasp_KS_slowDoS.pdf*. 2012. Available also from: `https://owasp.org/www-pdf-archive/Owasp_KS_slowDoS.pdf`. (Accessed on 04/04/2024).
9. SHEKYAN, Sergey. *Are you ready for slow reading? — Qualys Security Blog*. 2012. Available also from: `https://blog.qualys.com/vulnerabilities-threat-research/2012/01/05/slow-read`. (Accessed on 04/06/2024).

10. ELIYAN, Lubna Fayez; PIETRO, Roberto Di. DoS and DDoS attacks in Software Defined Networks: A survey of existing solutions and research challenges. *Future Generation Computer Systems*. 2021, vol. 122, pp. 149–171. ISSN 0167-739X. Available from DOI: https://doi.org/10.1016/j.future.2021.03.011.

11. MOUSAVI, Seyed Mohammad; ST-HILAIRE, Marc. Early detection of DDoS attacks against SDN controllers. In: *2015 International Conference on Computing, Networking and Communications (ICNC)*. 2015, pp. 77–81. Available from DOI: 10.1109/ICCNC.2015.7069319.

12. YU, Shanshan; ZHANG, Jicheng; LIU, Ju; ZHANG, Xiaoqing; LI, Yafeng; XU, Tianfeng. A cooperative DDoS attack detection scheme based on entropy and ensemble learning in SDN. *Eurasip Journal on Wireless Communications and Networking*. 2021, vol. 2021, pp. 1–21. ISSN 16871499. Available from DOI: 10.1186/S13638-021-01957-9/FIGURES/12.

13. HIRAKAWA, Tetsuya; TAKATA, Toyoo. The Trade-Off Between the False-Positive Ratio and the Attack Cost of Slow HTTP DoS. In: 2021, pp. 225–237. ISBN 978-3-030-57810-7. Available from DOI: 10.1007/978-3-030-57811-4_21.

14. *WAF DDoS: Why WAF and DDoS – A Perfect Prearranged Marriage*. Copyright 2024 Radware. Available also from: https://www.radware.com/cyberpedia/application-security/why-waf-and-ddos-a-perfect-prearranged-marriage/. (Accessed on 04/07/2024).

15. *GitHub - owasp-modsecurity/ModSecurity*. 2024. Available also from: https://github.com/owasp-modsecurity/ModSecurity. (Accessed on 04/03/2024).

16. BLANKENSHIP, Harold. *Trustwave Transfers ModSecurity Custodianship to OWASP — OWASP Foundation*. 2024. Available also from: https://owasp.org/blog/2024/01/09/ModSecurity.html. (Accessed on 04/03/2024).

17. *What is a Slowloris Attack? — NETSCOUT*. Copyright 2024 NETSCOUT. Available also from: https://www.netscout.com/what-is-ddos/slowloris-attacks. (Accessed on 03/28/2024).

18. JAO, David. *mod_limitipconn.c*. Copyright 2002 David Jao. Available also from: https://dominia.org/djao/limitipconn.html. (Accessed on 03/28/2024).

19. *mod_qos*. Copyright 2007-2024 Pascal Buchbinder. Available also from: https://mod-qos.sourceforge.net/. (Accessed on 03/28/2024).

20. ZDZIARSKI, Jonathan. *GitHub - jzdziarski/mod_evasive: Apache mod_evasive module*. 2017. Available also from: https://github.com/jzdziarski/mod_evasive. (Accessed on 03/28/2024).

21. *mod_antiloris download — SourceForge.net*. 2013. Available also from: https://sourceforge.net/projects/mod-antiloris/. (Accessed on 03/28/2024).

22. MOUSTIS, Dimitrios; KOTZANIKOLAOU, Panayiotis. Evaluating security controls against HTTP-based DDoS attacks. *IISA 2013 - 4th International Conference on Information, Intelligence, Systems and Applications*. 2013, pp. 165–170. ISBN 9781479907717. Available from DOI: 10.1109/IISA.2013.6623707.

23. *mod_reqtimeout - Apache HTTP Server Version 2.4*. Copyright 2024 The Apache Software Foundation. Available also from: https://httpd.apache.org/docs/2.4/mod/mod_reqtimeout.html. (Accessed on 03/28/2024).

24. *core - Apache HTTP Server Version 2.4*. Copyright The Apache Software Foundation 2024. Available also from: https://httpd.apache.org/docs/2.4/mod/core.html. (Accessed on 05/05/2024).

25. COMMISSION, Federal Communications. *fcc.gov*. 2023. Available also from: https://www.fcc.gov/sites/default/files/CSRIC8-Report-SecurityVulnerabilitiesMitigationsHTTP2-0623.docx. (Accessed 28-03-2024).

26. *slowhttptest — Kali Linux Tools*. Copyright OffSec Services Limited 2024. Available also from: https://www.kali.org/tools/slowhttptest/. (Accessed on 04/16/2024).
27. SHEKYAN, Sergey. *GitHub - shekyan/slowhttptest: Application Layer DoS attack simulator*. 2022. Available also from: https://github.com/shekyan/slowhttptest. (Accessed on 04/16/2024).
28. *Apache Killer*. Copyright 2024 Radware. Available also from: https://www.radware.com/security/ddos-knowledge-center/ddospedia/apache-killer/. (Accessed on 04/16/2024).
29. *slowhttptest/src/slowhttptest.cc at master · shekyan/slowhttptest · GitHub*. 2022. Available also from: https://github.com/shekyan/slowhttptest/blob/master/src/slowhttptest.cc. (Accessed on 05/08/2024).
30. *Developing modules for the Apache HTTP Server 2.4 - Apache HTTP Server Version 2.4*. Copyright 2024 The Apache Software Foundation. Available also from: https://httpd.apache.org/docs/2.4/developer/modguide.html. (Accessed on 05/06/2024).
31. *3.3 Extending Apache: Apache Modules (G)*. 2004. Available also from: http://www.fmc-modeling.org/category/projects/apache/amp/3_3Extending_Apache.html. (Accessed on 05/06/2024).
32. *How to use Apache2 modules — Ubuntu*. 2023. Available also from: https://ubuntu.com/server/docs/how-to-use-apache2-modules. (Accessed on 05/06/2024).
33. *Converting Modules from Apache 1.3 to Apache 2.0 - Apache HTTP Server Version 2.5*. Copyright 2023 The Apache Software Foundation. Available also from: https://httpd.apache.org/docs/trunk/developer/modules.html. (Accessed on 05/06/2024).
34. *Apache2: request_rec Struct Reference*. date unknown. Available also from: https://nightlies.apache.org/httpd/trunk/doxygen/structrequest__rec.html. (Accessed on 05/06/2024).
35. *HTTP response status codes - HTTP — MDN*. date unknown. Available also from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status. (Accessed on 05/06/2024).
36. *TCP Series #4: The TCP Receive Window and everything to know about it*. 2017. Available also from: https://accedian.com/blog/tcp-receive-window-everything-need-know/. (Accessed on 05/15/2024).
37. *3.3 Extending Apache: Apache Modules (G)*. 2004. Available also from: http://www.fmc-modeling.org/category/projects/apache/amp/3_3Extending_Apache.html. (Accessed on 05/15/2024).
38. 0XE2D0. *Flask-Reverse-Proxy/proxy.py at main · 0xe2d0/Flask-Reverse-Proxy · GitHub*. 2022. Available also from: https://github.com/0xe2d0/Flask-Reverse-Proxy/blob/main/proxy.py. (Accessed on 05/15/2024).
39. *Welcome to Flask — Flask Documentation (3.0.x)*. Copyright 2010 Pallets. Available also from: https://flask.palletsprojects.com/en/3.0.x/. (Accessed on 05/15/2024).
40. ROY, Shanto. *Write a Reverse Proxy Server in Python: Part 1 (Reverse Proxy Server) - Roy's Blog*. 2021. Available also from: https://shantoroy.com/network/write-a-reverse-proxy-server-in-python/. (Accessed on 05/15/2024).
41. *TCP/IP raw sockets - Win32 apps — Microsoft Learn*. 2022. Available also from: https://learn.microsoft.com/en-us/windows/win32/winsock/tcp-ip-raw-sockets-2?redirectedfrom=MSDN. (Accessed on 05/15/2024).
42. LEBEAU, Remy. *c++ - Connect function in raw socket? - Stack Overflow*. 2016. Available also from: https://stackoverflow.com/questions/41369086/connect-function-in-raw-socket. (Accessed on 05/15/2024).

43. *Home — TCPDUMP & LIBPCAP*. 2024 The Tcpdump Group. Available also from: `https://www.tcpdump.org/`. (Accessed on 05/15/2024).

44. HARGRAVE, Vic. *GitHub - vichargrave/sniffer: Example code from my Develop a Packet Sniffer with libpcap blog*. 2022. Available also from: `https://github.com/vichargrave/sniffer`. (Accessed on 05/15/2024).

45. *Implementation on Map or Dictionary Data Structure in C - GeeksforGeeks*. 2023. Available also from: `https://www.geeksforgeeks.org/implementation-on-map-or-dictionary-data-structure-in-c/`. (Accessed on 05/15/2024).

46. *ss(8) - Linux manual page*. 2023. Available also from: `https://man7.org/linux/man-pages/man8/ss.8.html`. (Accessed on 05/15/2024).

Attachment contents

- `readme.txt` — basic description of the content
- `script`
  - `sniffer` — The executable of the defense script implemetation
- `src`
  - `attack_gen.py` — source code of the attacking script
  - `flask_proxy.py` — source code of the flask proxy
  - `apache_module`
    - `Makefile` — apache module makefile
    - `mod_antiSDDoS.cpp` — apache module implementation source code
    - `mod_antiSDDoS.lo` — apache module implementation source code
    - `mod_antiSDDoS.la` — apache module implementation source code
    - `mod_antiSDDoS.o` — apache module implementation source code
    - `mod_antiSDDoS.slo` — apache module implementation source code
  - `defense_script`
    - `sniffer.c` — source code of the defense script implementation
    - `sniffer.o` — source code of the defense script implementation
    - `Makefile` — defense script makefile
    - `README.md` — The readmefile of the sniffing tool the defense script is based upon
  - `thesis` — zdrojová forma práce ve formátu LaTeX
- `text` — text práce
  - `thesis.pdf` — text práce ve formátu PDF