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

1.1 DDoS

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. [2] 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. [4] 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. [1]

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. [3]

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.

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

- **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 → Reporting server → Loader server → Bots. [1] More information on how the components works with some code snippets can be found in Chapter 2.

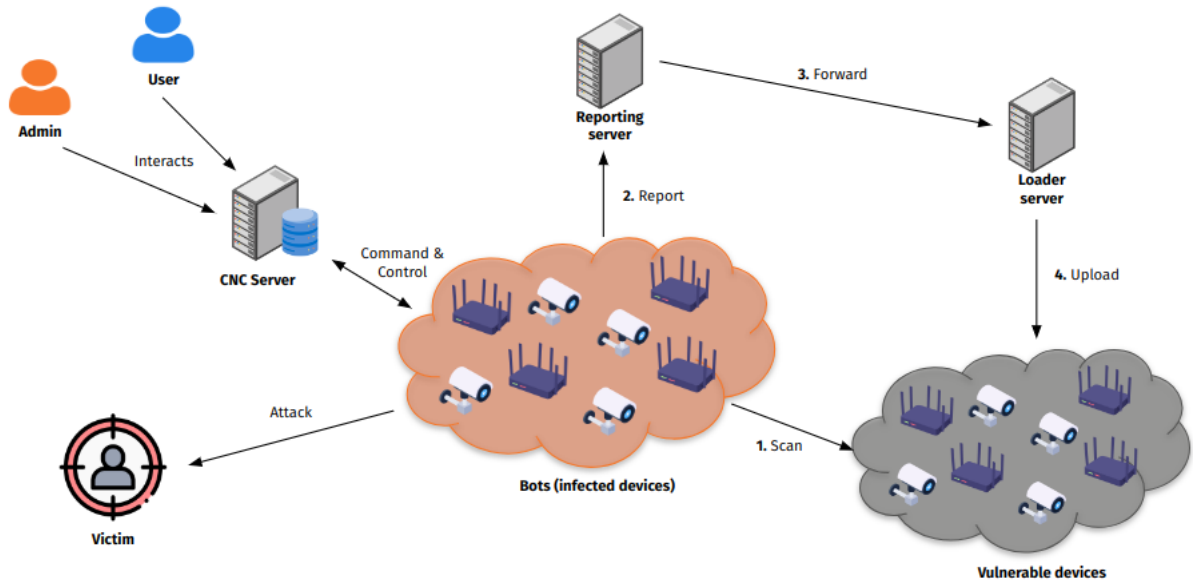

Figure 1.1: Mirai architecture

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.

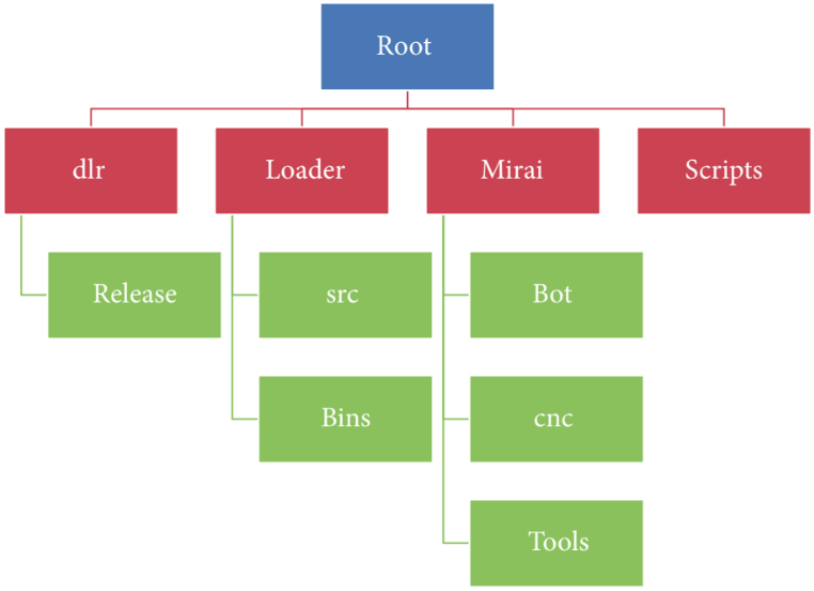

Figure 2.1: Original Mirai directory hierarchy [1]

- `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 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 2.2. 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 2.3.

```
# /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 2.2: `server_create` function

The `server_telnet_probe` is shown in Figure 2.4. 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`.

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 2.5

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 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.[1]

```
# 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 2.3: `main.c` loop

```
# 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 2.4: `server_telnet_probe` function

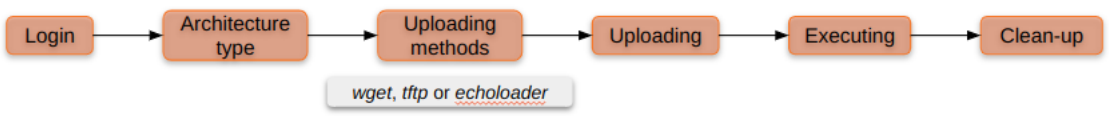

Figure 2.5: State machine of the `handle_event` function

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

5 Other malware

5.1 Hajime

5.2 Goldoon

5.3 BotenaGo

5.4 Malware comparison

6 Conclusion

Bibliography

[1] 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.

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

[3] 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.

[4] 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.