Bachelor Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Kako: Mirai's Vaccine Using Worm-Like Propagation Methods To Immunize Devices

Felix Hack
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr. Alexandra Dmitrienko
Reviewer

Peter Ten
Advisor

Submission
23. January 2024

www.uni-wuerzburg.de

Abstract

In 2016, multiple large Distributed Denial of Service (DDoS) attacks were observed and attributed to a new botnet called *Mirai* [1]. Instead of personal computers and mobile devices, Mirai targets Internet of Things (IoT) devices like routers and surveillence cameras. As these devices are often equipped with insecure credentials by default [2], Mirai features a *scanner component* to identify devices and target them with a dictionary attack of common logins to gain access to those devices. This poses a threat, as many traditional methods of securing devices are not applicable to IoT, due to their limitations in computing power and memory [3]. White-hat approaches were proposed, based on the idea of infecting devices with a benign Mirai, which protects these devices against malicious variants [4]. However, there is a potential danger that users will not be able to access their devices after being protected with a white-hat.

In this thesis, we present *Kako*, our proposed solution for a white-hat Mirai, which aims to address the issue of locking users out of their devices, while ensuring device security. *Kako* introduces a web-based dashboard that will list all vulnerable devices connected. When *Kako* is loaded onto a device with insecure credentials, they are automatically changed to secure it. The new login information can then be retrieved from the dashboard, among other data collected, such as the Medium Access Control (MAC) address and the Linux kernel version. The device manufacturer is automatically determined to make recognizing devices easier. Our approach was evaluated in a virtual environment and confirmed to be effective, as machines can no longer be infected with Mirai after being secured with *Kako*. It is also shown how Mirai's scanning component can still be used with Kako to automatically secure vulnerable devices. We believe that *Kako* can be used and extended in further research, to also protect devices from newer incarnations of Mirai and similar botnets.

Zusammenfassung

Im Jahr 2016 wurden mehrere große Distributed Denial of Service (DDoS) Angriffe einem neuen Botnetz namens *Mirai* [1] zugeordnet. Anstelle von PCs und mobilen Geräten zielt Mirai auf Geräte des Internets der Dinge (IoT), wie Router und Überwachungskameras. Da diese Geräte häufig standardmäßig mit unsicheren Anmeldedaten ausgeliefert werden [2], verfügt Mirai über eine Scannerkomponente, um diese Geräte zu identifizieren, und versucht über einen Wörterbuchangriff auf gängigen Zugangsdaten Zugang zu diesen Geräten zu erhalten. Dies stellt eine Bedrohung für IoT Geräte dar, da viele herkömmliche Methoden zur Absicherung von Computern aufgrund ihrer begrenzten Rechenleistung und Speicherkapazität nicht anwendbar sind [3]. Es wurden White-Hat-Ansätze vorgeschlagen, die auf der Idee basieren Idee, Geräte mit einem gutartigen Mirai zu infizieren, das diese Geräte vor bösartigen Varianten schützt [4]. Es besteht jedoch die potenzielle Gefahr, dass Nutzer nicht mehr auf ihre Geräte zugreifen können, nachdem sie mit einem White-Hat Mirai geschützt wurden.

In dieser Arbeit stellen wir *Kako* vor, unseren Ansatz für ein White-Hat Mirai, das darauf abzielt, Geräte zu schützen, während der Nutzer weiterhin Zugriff auf sein Gerät besitzt. *Kako* stellt ein webbasiertes Dashboard bereit, das alle verbunden Geräte auflistet, die von Mirai infizierbar sind. Sobald *Kako* auf ein verwundbares Gerät gelangt, werden die unsicheren Zugangsdaten automatisch geändert. Diese neuen Zugangsdaten können dann, zusammen mit weiteren Informationen wie der Medium Access Control (MAC) Adresse und der Linux-Kernel-Version auf dem Dashboard abgerufen werden. Zudem wird der Hersteller des Gerätes ermittelt, was das Identifizieren von Geräten erleichtert. Unser Ansatz wurde in einer virtuellen Umgebung getestet und hat sich als erfolgreich darin erwiesen, Geräte vor solchen Angriffen zu schützen. Ebenfalls wird gezeigt, dass die Scannerkomponente von Mirai erfolgreich mit *Kako* kombiniert werden kann. Wir glauben, dass *Kako* in der weiteren Forschung genutzt und erweitert werden kann, um Geräte auch vor neueren Mirai-Varianten und ähnlichen Botnetzen zu schützen.

Contents

1. Introduction

In 2016, multiple high-traffic DDoS attacks were observed, targeting various websites and online services. Compared to previous attacks, it did not originate from devices with rich resources, like PCs and mobiles, but rather involved a large amount of low-power Internet of Things (IoT) devices such as IP surveillance cameras. These devices were weaponized by a worm-like malware called *Mirai* [1, 5], named after the Japanese word for "future", that exploited the poor default security configuration of these inexpensive devices.

Mirai launched its first attacks in 2016, against websites and hosting companies, namely OVH and Dyn [1]. These attacks were very large at that time and a prime example of the sheer power a large cluster with devices of small size can have. As Mirai's source code was later leaked, many other actors used it in their own interest. For instance, at the end of 2016 Deutsche Telekom fell victim to another attack on its networks [1].

Although the first Mirai samples were spotted as early as 2016 [1, 6], they are still a threat to IoT devices to this date [7]. When looking at *MalwareBazaar*[8], there are a lot of new samples collected every day. This shows that Mirai still continues to be an active threat despite its age and competition in contrast to other botnets. Due to its publicly available source code, many threat actors use Mirai as a starting point to add their own functionality [7, 9].

Researchers and security experts started early to analyze the malware and propose defenses against it [10, 1]. One of the ideas is using Mirai's propagation techniques to create a benign worm that disinfects and immunizes devices [4]. One of the main vulnerabilities that Mirai uses are weak passwords that either the user sets or are provisioned by default. Changing them automatically to strong passwords can lock users out of their devices if they are using those passwords for authentication, which is problematic, especially if there are a large number of devices.

In this thesis, we present Kako, our proposed solution to this problem. It is named after the Japanese word for "past", in reference to Mirai's name. Fundamentally, Kako will be based on the idea of a white-hat Mirai proposed by Cao et al. [4], which secures devices by completely closing ports, while we aim to solve the resulting usability problems. Their approach uses a heavily modified Mirai bot, with all attack and scanning capabilities removed. To then be able to discover new vulnerable devices, the Command and Control (C2) server is extended with a scanner module. Once a device is found and the white-hat Mirai installed, the SSH and Telnet ports are closed, effectively preventing a real Mirai from infecting.

We will develop a dashboard for the front-end of Kako to provide an overview of vulnerable devices. This dashboard can be deployed by organizations with a broad spectrum of different devices that want to get an overview of those that are vulnerable to the attack methods of Mirai. When possible, a fix can be applied directly over the dashboard. This can be, for instance, changing the default password of the device to a new one, chosen by the user on the dashboard.

Similarly to the existing approach, we will remove all attack features to prevent the bot from being abused. When Kako is deployed to a device, instead of closing all SSH/Telnet ports, the vulnerable default password is automatically changed. This password is sent to the dashboard where it will be visible to the user. If the automatic change was successful, the user also has the possibility to set an own password. This is the major distinction of Kako, as users are in control, and the risk of locking out users is greatly reduced.

To implement Kako, we will make use of the leaked Mirai source code and modify the parts necessary to achieve our goal. The leaked code base consists of three main parts. The bot that runs on the device and the loader that plants the bot are written in C, whereas the C2 server is implemented in Golang. We create a test bench for running Mirai and our own implementation, with the help of *Vagrant* [11]. It is used to spin up a Virtual Machine (VM) for Mirai's C2 server and an arbitrary number of bot VMs. In addition, we configured an Integrated Development Environment (IDE) from *JetBrains* [12] to assist in analyzing and extending Mirai, with tools such as code completion and a debugger.

As the C2 server already implements the necessary bot management functionality and Go provides good support for libraries, a web-framework like *Gin*[13] can be used to serve the dashboard with direct access to the existing data structures of the C2 server. While the dashboard gives an overiew over vulnerable devices, the detail page allows the user to execute specific commands on the controlled device. For allowing bidirectional communication between the dashboard and the bot, some bigger changes have to be made. At first, the Mirai bot already transfers some data during the initial connection to the C2 server as part of the handshake. This can be extended to gather more information about the device, such as the MAC address or the version of the Linux kernel and software packages. Sending data to a device is already partly implemented in Mirai's attack functionality that provides sending an attack type (like DDoS) with the corresponding payload (like the target IP address). This capability can be modified so that instead of sending attack instructions, a vaccine type, e.g., changing the password, and the payload containing the new password can be sent.

Contributions.

- Creating a simplified development environment for Mirai with the help of IDEs
- Analyzing and extending the C2 protocol
- Modifying the bot component to collect extended telemetry data
- Building a dashboard for viewing devices with the ability to send commands
- Improving the useability of white-hat Mirai approaches

Outline. The remaining part of this thesis is organized as follows. Chapter 2 explains all the necessary background information and introduces the mechanics and history of Mirai. Afterward, related work on this topic is discussed in Section 3, where we discuss existing approaches for securing IoT devices and their drawbacks. The architecture of our proposed framework *Kako* is expained in Chapter 4 followed by details on the implementation in Chapter 5. The effectiveness of *Kako* is evaluated in Chapter 6. We conclude the thesis in Chapter 7 and discuss future work.

2. Background

In this section, we discuss the most important concepts that are necessary to understand the work. At first, an explanation of botnets, especially Mirai, is given, followed by an overview of IoT and defensive strategies.

2.1 Functioning of a Botnet

A typical botnet consists of two main components, being the **Command and Control (C2) server** which orchestrates all infected machines running the counterpart and the **bot** malware itself [5, 14]. In case of Mirai, there is also the Loader that, once a victim is found, connects to the device via Telnet/SSH and plants the bot. This allows one to create a large cluster of devices whose computing power can be abused.

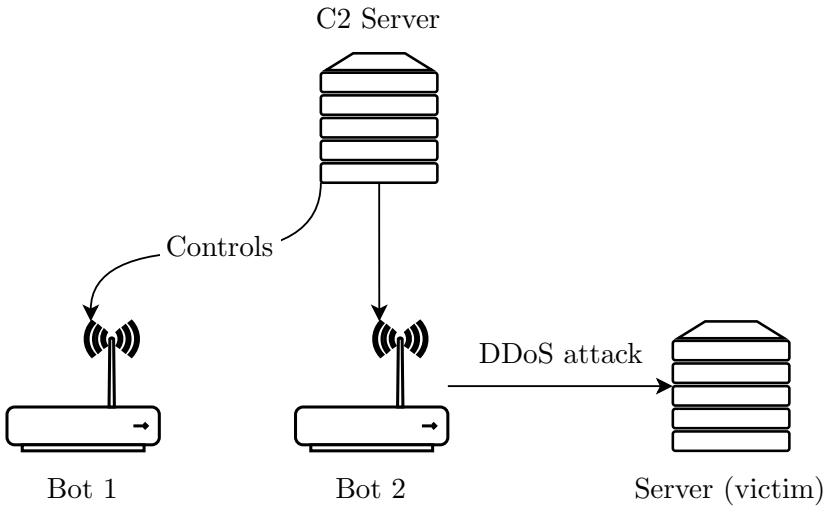

Figure 2.1: Topology of a generic botnet

Botnet components. The **bot** executable is run on the victim to gain control over it and receives commands from a C2 server. At first, it must be planted on the device, which can be done in a plethora of different ways, e.g. using vulnerabilities [15] or known default passwords [6]. Once installed, the bot registers itself at the C2 server and is from now on able to receive commands [10]. Mirai, in its original source code, uses a hard-coded list of default passwords to try logging into other devices that are running an SSH or Telnet

server [1]. This implies, that the more instances of Mirai bots are running, the faster the IPv4 address space can be scanned.

When Mirai successfully connects to another device, its IP address and credentials are reported to a report server. The report server then triggers the **loader**, that connects to the device and places the bot executable [1, 10].

Like the loader, the **C2 server** is also hosted by the botnet operators. As mentioned before, the C2 server keeps connections to all previously registered bots. In the case of Mirai, the adversary can then launch a DDoS attack by sending a command with the target and the type of DDoS attack [5, 10]. Mirai's C2 server was initially intended to run as a single centralized instance, but now there is a trend emerging that uses a decentralized approach [16]. Further architectures are derived variations, like a hybrid version.

Centralized botnets vs decentralized botnets. In centralized botnets, a single C2 server is responsible for handling connections of the bots. Although this approach is relatively simple, it can lead to some problems for the attacker. First, if the C2 server is offline due to technical issues or is blocked, it cannot control the bots anymore. Second, a single server makes it easier for IT professionals to monitor traffic or even block connections to it [16].

To address these problems, decentralizing the C2 server is an option [14, 17]. One way is to use multiple servers, creating redundancy in the event that one server fails. This can also be useful to increase the rate at which new bots can be handled. However, this does not help to overcome defensive mechanisms like blocking specific IP addresses. Another distributed approach is adopted by the *Hajime botnet*, that uses a Peer-to-Peer protocol, similar to BitTorrent. Here, each bot additionally undertakes the tasks of a C2 server. A bot can seed commands and updates to other bots and can, in return, leech those without the need for a central server [9].

Goal of botnets. When devices are infected with malware, attackers can misuse them for different malicious activities. A famous example is the TDSS malware family, that created one of the most long-lasting and complex botnets [18]. TDSS, also known as *Alureon*, is a rootkit that infects the bootloader of Windows machines, to prevent detection by Anti-virus software (AV). To monetize its operation, TDSS changes the Domain Name System (DNS) settings of the infected machines, to redirect traffic to advertisements. This generates clicks and thus profit, called click-fraud [19].

Another type of malicious use are Denial of Service (DoS) attacks. DoS attacks are mostly used to take down a system, so that legitimate persons cannot use it anymore. In case of websites, an attacker may flood the server with HTTP requests in order to congest it, leading to legitimate requests getting no response. In botnets, DoS attacks can be carried out simultaneously from a mass of devices, generating more traffic. These attacks, called DDoS attacks, were able to reach about 620 Gbps of traffic, as in the case of Mirai attacking the website KrebsOnSecurity.com [**?**].

2.2 Analyzing the Mirai Botnet

In this section, we explain how Mirai works and go into the specifics of the code.

Mirai consists of a *Bot* that is the part that runs on the device and the *C2 Server* sending commands to the bots, which can be seen in Figure 2.2. The difference is the wormlike behavior of every bot scanning the network for vulnerable devices. Mirai targets devices hosting a publicly available Telnet daemon that allow connections with default user credentials. The *Loader* component is then used to place the Mirai binary on a found device.

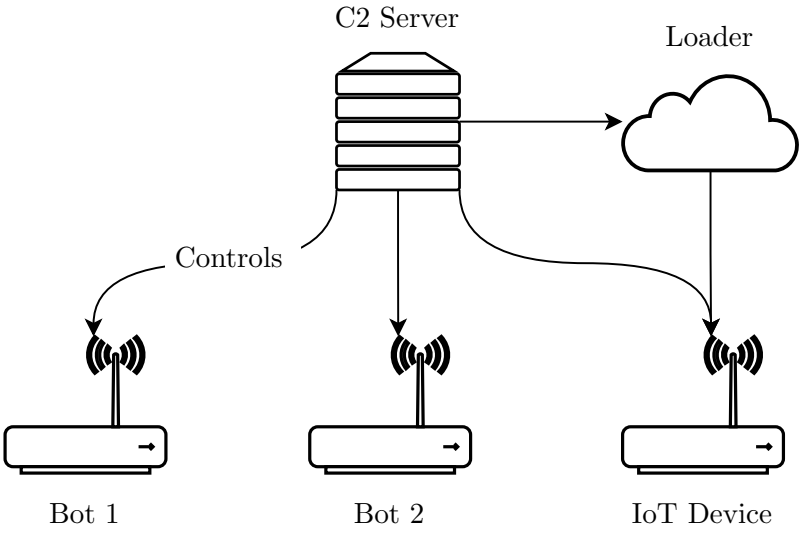

Figure 2.2: Mirai topology diagram

Reconnaissance phase. Every device infected with Mirai scans the network for other devices that use default credentials, that are hard coded in Mirai's code. In Figure 2.2 the devices labeled "Bot 1" and "Bot 2" run Mirai and scan vulnerable devices, such as the "IoT device".

Infection phase. When a vulnerable device is found, the *C2 Server* is notifying the *Loader* that then connects via SSH or Telnet to the victim with the probed credentials and downloads the binary.

Registration phase. After Mirai is run for the first time on the infected device, it connects to the *C2 server* and registers itself. It is now a fully functional bot and also scans the network for other vulnerable devices.

2.2.1 Bot Component

The bot is the part of Mirai that runs on a machine once it is infected. It connects to the C2 server and waits for commands. These include information for the bot to conduct DDoS attacks.

When Mirai is started, it first sets up itself by initializing variables and deleting its own executable to make detection harder, as no traces remain in the mass storage of the device. The drawback of this approach is that after the Random-access memory (RAM) is cleared, e.g. after a reboot, Mirai cannot start itself again, as the binary is not on the device anymore. Therefore, the system watchdog, that is responsible for restarting the device if it is not responding, is disabled to lower the likelihood of a system reboot, therefore keeping Mirai running as long as possible. The developers also implemented techniques with the goal of making reverse engineering harder, by tampering the `SIGTRAP` Portable Operating System Interface (POSIX) signal[1] to change behavior if the bot is run with GNU Debugger (GDB) attached. Notably, these persistation methods are not executed when the bot is compiled in debug mode.

Afterward, a function called `killer_init()` is called, that starts Mirai's killer module. Its task is to kill processes that bind to certain ports on the system to reduce the surface area that other rivalizing botnets can exploit.

Finally, the connection to the C2 server is established. For this, the domain name of the server is retrieved from an encrypted table and resolved with the hardcoded DNS server

[1] An overview of Linux signals can be found under `https://man7.org/linux/man-pages/man7/signal.7.html`

`8.8.8.8` that belongs to Google's public DNS service [20]. Communication is done over simple sockets that the bot uses to send a magic value to the C2 server to indicate its intent to register as a new bot. In line 1 of Listing 1, we can see the value `\x00\x00\x00\x01` that tells the C2 server that the device trying to connect is a new bot, where `\x01` indicates the version number of the bot. Additionally, the bot prepares a variable named `id_buf`, which is set to the first argument passed to the Mirai executable when started (`args[1]`[2]). Then the length of this string is evaluated and sent as a second message. If the string is not empty, it is sent as well, as seen in line 5 of Listing 1. Figure 2.3 shows the message containing the magic value in Wireshark.

Listing 1 Code for sending registration messages

```
1    send(fd_serv, "\x00\x00\x00\x01", 4, MSG_NOSIGNAL);
2    send(fd_serv, &id_len, sizeof (id_len), MSG_NOSIGNAL);
3    if (id_len > 0)
4    {
5        send(fd_serv, id_buf, id_len, MSG_NOSIGNAL);
6    }
```

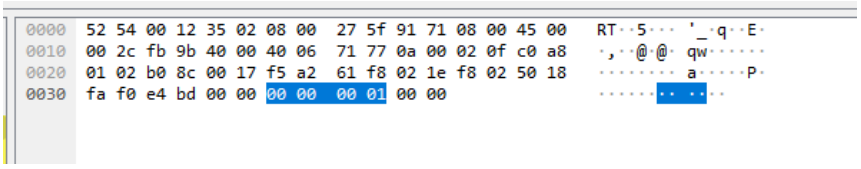

Figure 2.3: Magic value observed in Wireshark

Mirai also has a scanner component that regularly scans the network for devices to which it can connect over Telnet. IP addresses are generated randomly by the function `get_random_ip()`, which generates a 32-bit integer and splits it into four 8-bit values that form an IP address. If parts of this address are found on a hardcoded blacklist, the process is repeated and a new IP address is generated. Blocked IP ranges include invalid and internal addresses, but also ranges belonging to *Hewlett-Packard*, *US Postal Service*, and the *Department of Defense*, among others. This is likely to be done to avoid attention by infecting devices of these institutions and to reduce the number of unsuccessful connections. Once a successful connection is established to a device, Mirai tries to brute force the login credentials with a dictionary attack. A hard-coded list of common username and password combinations is used to repeatedly attempt logging into the device. Listing 2 shows an excerpt from this list. These values are not stored in plain text, but in encrypted form. The algorithm used to encrypt these values is a simple cipher with the key `0xDEADBEEF`. As the name of the decryption function `deobf()` suggests, this is done to obsfuscate these strings, making it harder during static analysis to detect these login credentials in plain text.

2.2.2 Command and Control (C2) Server

The C2 server is the component that keeps track of all bots and can instruct them to perform attacks. It provides a Telnet interface over which a user can interact with the botnet and issue attacks to the bots.

When the C2 is started, it listens on **port 101** for connections to the internal API and on **port 23** for Telnet connections. Port 23 is used for the registration of new bots and also

[2]Most C implementations prepend the name of the program to the `args` array, so `args[1]` equals to the first actual argument

Listing 2 Excerpt of the hardcoded credentials in the `scanner.c` file

```
// root xc3511
add_auth_entry("\x50\x4D\x4D\x56", "\x5A\x41\x11\x17\x13\x13", 10);
// root vizxv
add_auth_entry("\x50\x4D\x4D\x56", "\x54\x4B\x58\x5A\x54", 9);
// root admin
add_auth_entry("\x50\x4D\x4D\x56", "\x43\x46\x4F\x4B\x4C", 8);
// admin admin
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x43\x46\x4F\x4B\x4C", 7);
// root 888888
add_auth_entry("\x50\x4D\x4D\x56", "\x1A\x1A\x1A\x1A\x1A\x1A", 6);
...
```

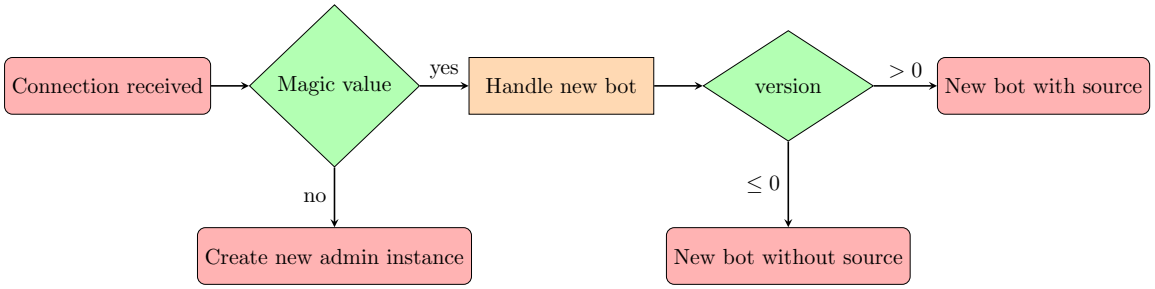

Figure 2.4: Telnet connection flow

for providing the control interface. Figure 2.4 shows how a connection is handled when received. The C2 first validates if the magic value is present. If it is present, the version number of the bot is evaluated and a new bot instance is created. The bot has no source value if the version number is zero or smaller. If it is one or larger, a source string is read from the connection and stored by the C2. Connections without the magic value are considered admin connections and handled differently.

The C2 server needs a MySQL database for storing various information. Most importantly, there is the `user` table that contains the list of login credentials, that can log into the admin interface.

- `users`: This table includes the list of users that can access the admin interface.
- `whitelist`: This table lists IP addresses that can not be targeted by DDoS attacks, like the *Department of Defense* or the *US Postal Service*. Despite its name, it works rather like a blacklist.
- `history`: This table contains a history of all previously started attacks.

The admin interface can be opened by connecting to port 23 of the C2 server via Telnet. After the C2 server internally created the admin instance, a login prompt in Russian is shown, as can be seen in Figure 2.5. It consists of a header, that is read from a file called `prompt.txt`, that by default contains the sentence ”я люблю куриные наггетсы” that translates to ”I love chicken nuggets”. We do not think this serves any particular reason, rather than being a placeholder. Then there are prompts for a username (пользователь) and password (пароль). These values are then queried against the `user` database table and the user is logged in if the credentials are correct. Before commands can be executed, various log entries are printed that serve no purpose other than for visuals.

When a user is logged into the admin interface, some commands can be executed.

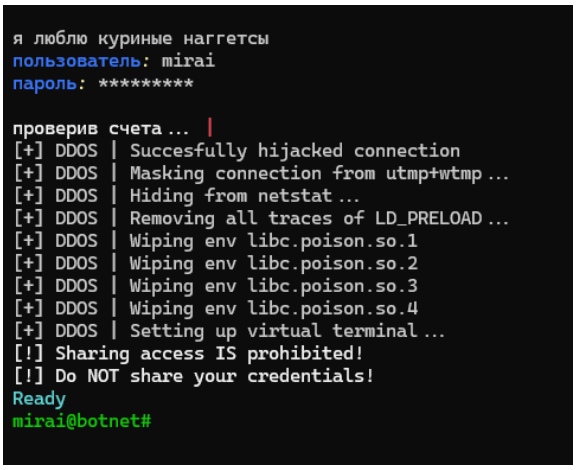

Figure 2.5: Telnet connection to C2 admin interface

- `botcount`: This command will print the number of bots currently connected to the C2 server.
- `adduser`: An interactive prompt will be shown for adding new users with access to the admin interface and the ability to launch attacks.
- `?`: This will print a list of all available attack types. Figure 2.6 shows the output of this command.

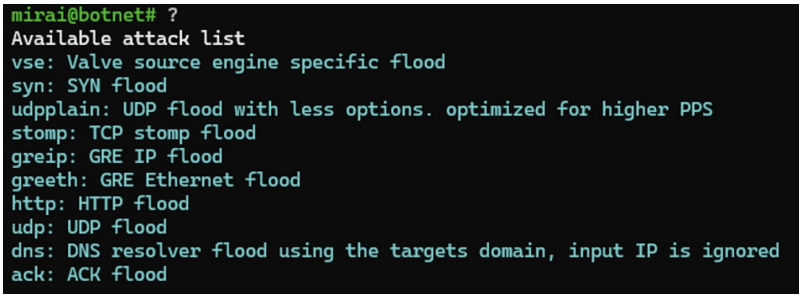

Figure 2.6: Help command listing available attack types

When the user wants to start a DDoS attack against a target, a supported attack type has to be specified, along with the target IP address and the desired duration of the attack. The C2 then proceeds with checking if the target IP is contained in the `whitelist` table, if so, preventing the attack. Each user also has a cooldown, which is a time that must be waited before another attack command can be issued. If these checks are valid, a command is sent to the maximum allowed number of bots defined for the user. Figure 2.7 shows a captured package sent to a bot that contains the details of an attack.

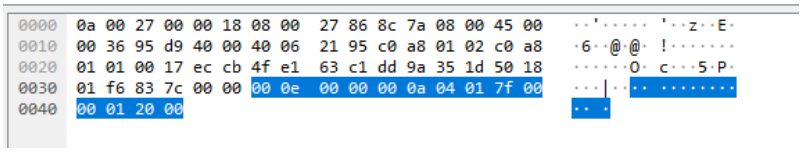

Figure 2.7: Attack command package sent to a bot

Upon receiving the command, the bot will parse its contents and initialize the attack. As every attack is assigned an integer ID, the bot has to map the IDs to the corresponding

functions, which is done in the `attack_init` method in the `attack.c` file. There, each ID is mapped to a pointer that references the function that should be called for a specified attack.

2.2.3 ScanListen and Loader Components

The ScanListen component, which is implemented in a single Go file called `scanListen.go`, binds to port `48101` and listens to reports of vulnerable devices sent by Mirai bots. Every time such a report is received, it formats the data in the format `ip:port username:password` and prints it to Standard output (stdout).

The loader part of Mirai is responsible for logging into devices over a Telnet connection and placing the bot. It is also written in C and can be run on the same server as the C2 server or on a completely different server. Once started, it consumes the Standard input (stdin) for the credentials and IP addresses of vulnerable devices, in the same format as the ScanListen returns. This makes it obvious that the ScanListen is designed to be piped into the loader, for continuous operation. If known, the architecture of the device can also be appended, such as *x86*, *arm*, *mips*, etc. The loader then loops over the entries and tries to establish a Telnet connection and log in with the information provided. If the login is successful, it is tested if *busybox* is available and copies the `echo` binary into the current working directory. If the architecture is not provided initially, it tries to determine it at this point. Then, one of the following methods is used to upload the bot binary, depending on what tools are available on the device:

- If `wget` is available, the bot is downloaded from an HTTP server that has to host the binary.
- Otherwise, `tftp` can be used to get the binary from a tftp server.
- If neither tool is found on the system, a small downloader binary is echoloaded on the device. This works by putting the binary content of the downloader directly into a shell command and redirecting it to a file on the device with the `echo` command. This downloader then retrieves the file over HTTP from the same server as the `wget` method.

Finally, the correct permissions for the bot executable are set with `chmod 777`, and the bot is executed, with the command shown in Listing 3. `FN_BINARY` defines the file name to be executed (`mirai.x86` in case of an x86-based device) followed by the

Listing 3 Shell command that gets build and send over Telnet to the target device

```
util_sockprintf(conn->fd, "./" FN_BINARY " %s.%s; " EXEC_QUERY "\r\n",
↪  id_tag, conn->info.arch);
```

2.3 Analyzing Mirai's Source Code

Since the Mirai source code was leaked in 2016 [1] many copies appeared on GitHub, as a search revealed. The most common one was created by *Jerry Gamblin* available on GitHub [21], shown in Figure 2.8.

This repository contains the leaked source code of the Mirai bot and C2 server (located in the `mirai` folder), the loader and a download binary for systems without `wget` or `tftp`, as seen in 2.9. Additionally, the original forum post of the leak is included, that provides an explanation and steps to get Mirai running.

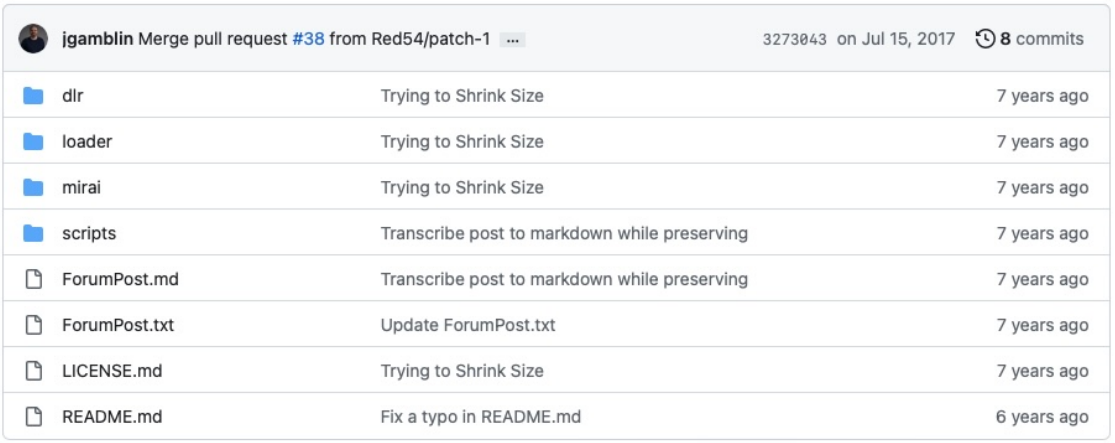

Figure 2.8: Mirai source code repository

```
├── dlr
├── loader
├── mirai
│   ├── bot
│   │   └── <C source files>
│   ├── cnc
│   │   └── <Go source files>
│   ├── tools
│   │   └── <various utilities in C and Go>
│   ├── build.sh
│   └── prompt.txt
└── scripts
```

Figure 2.9: Directory structure of the Mirai source code

When looking into the `mirai` folder, we are presented with the whole Mirai source code in C and Go. During the inspection of the source code, our attention was primarily directed towards the sections where configurations are implemented, enabling us to make modifications for specific setup.

Bot. The `mirai/bot` folder contains various C source files that make up the bot part of Mirai. This code is written in plain C, which can be compiled for different processor architectures like x86, ARM, MIPS, etc., allowing the bot to run on a large array of devices. Throughout the code, there is extensive use of *preprocessor directives* that control debug functionality (by setting the flag `DEBUG`) and also enable Telnet related parts of the code (with `MIRAI_TELNET`), such as the scanner module. There is also a flag `MIRAI_SSH` in the build script, which seems to be unused in the code. When building the code with the `DEBUG` flag enabled, many obfuscation and anti-debug techniques are disabled and the bot is printing log entries to the terminal. This behavior is helpful when one is trying to set up Mirai or modifying parts of it.

The main configuration of the bot is done in the file `table.c`, that contains a key-value table of strings. Listing 4 shows the domain and port entries that are essential to successfully establish a connection to the C2 server and to report vulnerable devices. Like the list of credentials in Listing 2, these values are encrypted with the same algorithm and key, to also prevent plain domains from appearing in the binary.

Listing 4 Bot configuration in table.c

```
1 add_entry(TABLE_CNC_DOMAIN, "\x41\x4C\x41\x0C\x41\x4A\x43\x4C\x45\x47\
  ↪ x4F\x47\x0C\x41\x4D\x4F\x22", 30);
  ↪ //cnc.changeme.com
2 add_entry(TABLE_CNC_PORT, "\x22\x35", 2); //23
3
4 add_entry(TABLE_SCAN_CB_DOMAIN, "\x50\x47\x52\x4D\x50\x56\x0C\x41\x4A\
  ↪ x43\x4C\x45\x47\x4F\x47\x0C\x41\x4D\x4F\x22", 29);
  ↪ //report.changeme.com
5 add_entry(TABLE_SCAN_CB_PORT, "\x99\xC7", 2); //48101
```

The `scanner.c` file contains a list of username and password combinations that the bot uses to log on to other devices, as shown in Listing 2. During testing, it might be useful to include only the working credentials of the devices that should be infected to reduce the number of false attempts.

Similarly, a list of IP address ranges is included in this file, which are excluded from the generation of IP addresses. For testing, instead of implementing a blacklist for IP ranges, an alternative approach would be to return a fixed IP address to also reduce the number of attempts.

C2 Server. The `mirai/cnc` folder contains Go source files that can be compiled to the C2 server. The only modifications that might be necessary here, are setting the variables in the `main.go` file as shown in Listing 5.

Listing 5 Configuration constants in main.go

```
const DatabaseAddr string   = "127.0.0.1"
const DatabaseUser string   = "root"
const DatabasePass string   = "password"
const DatabaseTable string  = "mirai"
```

Other tools and scripts. The `mirai/tools` folder contains various programs written in C and Go. Especially interesting is the file `enc.c`, which can be compiled into a Command-line interface (CLI) tool for encrypting strings. These strings can then be used in the table implementation of the bot part. This is useful for generating custom values for testing purposes. The file `db.sql` is a Structured Query Language (SQL) script, that creates the base database structure required for the C2 server to work.

To build the Mirai source code into usable binaries, a `build.sh` script is included in the `mirai` folder. First, it requires a parameter that defines the mode of compilation. When `release` is used, the script optimizes the resulting binaries by passing various flags to the C compiler. In the case of `debug`, these optimization flags are replaced by a debug flag. Second, the parameter `telnet` or `ssh` is required. Depending on the value, the preprocessor directive `MIRAI_TELNET` or `MIRAI_SSH` is passed to the compiler to toggle parts of the code as explained earlier. Since our focus lies on analyzing and extending Mirai, we are more interested in compiling code in debug mode.

Listing 6 shows the part of the script responsible for building all the necessary binaries to run a full Mirai setup. In lines 1-2, the bot and the string encryption tool are compiled using the `gcc` [22] compiler. The bot also gets cross-compiled for different processor architectures, but in our case we are only interested in the x86 one. Afterwards, the C2 server and ScanListen tool are compiled in lines 3-4.

Listing 6 Part of the build script, compiling Mirai code with debug flags

```
gcc -std=c99 bot/*.c -DDEBUG "$FLAGS" -static -g -o debug/mirai.dbg
gcc -std=c99 tools/enc.c -g -o debug/enc
go build -o debug/cnc cnc/*.go
go build -o debug/scanListen tools/scanListen.go
```

2.4 History and Explanation of the Internet of Things (IoT)

The Internet of Things (IoT) describes the interconnection of everyday objects over the Internet, allowing them to collect and exchange data [23]. These devices can range from household items like routers or thermostats to industrial applications such as sensors or machinery. To better understand the term IoT, we will give a short overview of the history of the Internet and the anatomy of IoT devices. After that, we will discuss some specifics about their security measures.

History. Initially, the Internet was shaped as a collection of HTML pages, all linked through hyperlinks, building the foundation for the World Wide Web [24]. This first iteration is also referred to as *Web 1.0* [25]. The next step in its evolution was enabling users to interact with websites. Consequently, many websites adopted this idea and a plethora of dynamic web content was published on the Internet, called *Web 2.0* [25]. This led to the Internet we know today, with social networks, online shops, business applications, and much more [26].

With many more devices connected to the Internet, like cameras and sensors, there is a need for machines to be able to exchange and understand data. One way to achieve this is by using standardized data formats, called Semantic Web or Web 3.0. This allows machines to automatically process data from websites or IoT [26].

Devices and technology. While in the past IoT networks mostly comprised Radio-frequency identification (RFID) based tags [23] that were passively included, with time and technological advance more devices were added, that are also actively connected. Industrial applications for IoT include sensors that are used to collect, process and send data to other devices or computers. This could be a smart meter, for example, that can remotely sense and report the energy consumption to the provider [27].

Not only the industrial sector makes use of this [28], nowadays more and more consumer appliances are connected to the Internet [27], like sensors, cameras, TVs, thus being IoT devices. While simpler devices, such as smart bulbs are mostly using weaker processors, such as an ESP8266[3], to connect to a Wi-Fi network [29], more complex hardware like routers, set-top boxes and IP cameras, often use embedded variants of the Linux Operating system (OS) like OpenWRT [30].

Security of IoT devices. Although IoT devices share similarities with desktop computers, their low-end hardware poses some new security challenges to show consideration for. As often purpose-built for a specific use case, they are typically lacking in computing power. The storage capacity is also limited, mostly providing only enough storage to contain the bare software needed to function [31]. While this makes it ideal for cost- and energy-aware applications, deploying traditional security measures like Intrusion Detection System (IDS) and AV would result in major difficulties [3]. Additionally, due to the low prices of many IoT devices, the manufacturers take relatively low effort into applying secure measures or providing software updates. Furthermore, even if updates are provided by the

[3] A cheap, thus widely used 32-bit microprocessor with onboard WiFi

manufacturer, they must be installed on the devices to take effect. As IoT networks are often very heterogenic in terms of hardware and software [3], it is highly likely for devices to remain unpatched, e.g., if they get overlooked or forgotten [32]. To prevent this from happening, an inventory system can help keeping track of all those devices [32]. Whilst there are many inventory systems on the market, it is the most beneficial if the system is able to check for updates automatically, taking humans off this task.

2.5 Defensive Strategies

There are multiple approaches to secure IoT devices. Firstly, there can be protective measures right on the device. Secondly, the device's firmware can be analyzed to find vulnerabilities. Lastly, there are ways to verify the integrity of the device or detected malicious traffic externally.

On-device security. A rather traditional approach for detecting malware is using AV or IDS software directly on the device. AVs try to detect malicious programs by using signatures with patterns of known harmful files, but many also include features of Intrusion Prevention System (IPS) systems [33]. Their goal is to prevent the successful execution of malware by analyzing the behavior of programs and terminating them if the actions are considered malicious. Those systems often uses behavioral based approaches, that monitor the actions a program is performing on the system [34]. While signature based systems are easy to bypass with obfuscation techniques, behavioral detection is harder to evade [35]. These solutions are used on almost every personal computer and in business environments, but they are resource intensive and thus not suited for IoT [3].

Static analysis. Another approach for securing IoT devices is analyzing and detecting vulnerabilities in the firmware itself, before shipping the product or a potential software update. One way is static analysis, where the source code or the firmware binaries is analyzed without executing, obviating the need for physical access to devices [36]. This is a particularly simple method to automate, as no complex setups are necessary. The static analyzing tool chain can make use of multiple tools to cover different areas of the firmware to generate a report with possible vulnerabilities [37]. This method proved to be capable of finding private RSA and SSH keys, hard-coded passwords, outdated packages and unsafe configurations [36]. Moreover, with the use of binary static analysis, common attack vectors like buffer overflows could be also discovered [37].

Dynamic analysis. Contrary to static analysis, dynamic analysis does not work by analyzing a firmware image, but rather by performing tests against the running firmware. This means that it is irrelevant which technology stacks the firmware is using and if there are any static-analysis tools available. Instead, known attacks are run against, for example, the web-interface to test if a vulnerability exists [38]. Despite the benefits, getting the environment as real as possible is difficult, as IoT devices often use different hardware and CPU configurations [37]. To overcome this difficulty, researchers proposed a possibility to automatically emulate firmware images for analysis, removing the need for real devices [37].

Remote attestation. While the previously mentioned strategies work by detecting malicious payloads on the device or inspecting the firmware for potential vulnerabilities, another approach is to verify if the device is in a trusted state, for example, by validating a checksum of the firmware to detect modifications by malware or other means [39]. Remote attestation (RA) allows a *verifier* to check the integrity of the *prover* [40]. The *prover* therefore makes a claim about its internal state based on evidence supporting that claim. The *verifier* then decides whether to trust this claim and thus attest the device as

trustworthy [41]. This in turn only works in scenarios where there is an active instance that is interested in the integrity of the devices.

Traffic analysis. Since botnets rely on network connections to exchange data with C2 servers, one can try to detect this traffic. Many botnets use Internet Relay Chat (IRC) or HTTP as means of communication that feature characteristic properties which can be detected [42][43]. These approaches can reach from using machine-learning or analyzing the spatial-temporal correlations of network packages due to typical behavior like registering, scanning the network, etc.

2.6 Tools

In this section, we explain the specific tools that are used to implement Kako.

2.6.1 VirtualBox

VirtualBox [44] is an open-source virtualization software developed by Oracle that allows the creation of virtual machines. These can be used to run different operating systems and software on a single computer. Each VM is isolated from the host machine, which makes them perfect for running and analyzing malware, while preventing it from spreading to real machines [45].

2.6.2 Vagrant

Vagrant [11] is a tool developed by *HashiCorp* to automatically create virtual machines, based on a configuration file. It acts as a wrapper for different hypervisors like *VirtualBox* [44], *VMware* [46], *Hyper-V* [47] and more.

A configuration file, called `Vagrantfile`, is used to describe the desired VM setup that is created when the command `vagrant up` is called. This configuration can contain parameters such as:

- VM instances (including images)
- Networking
- Provisioners (Shell or Ansible script to install software on the VM)
- Specific options for the provider (the hypervisor used to spin up the VMs)

2.6.3 JetBrains IDEs

Integrated development environments are code editors with additional tools to aid in the software development process. These tools range from autocomplete and error analysis to more advanced features [48] such as an integrated debugger, which allows one to better understand the code and its behavior.

JetBrains [12] is a company that specializes in the development of IDEs for multiple languages. We use *GoLand* for code written in Go and *Clion* for C code, for our main development environment. As we execute the Mirai code within the VMs, we also utilize *JetBrains Gateway* [49] to directly run these IDEs within those VMs. This enables us to directly use tools such as the debugger without further configuration.

3. Related Work

This section describes recent and related work on the topics of this thesis. First, it surveys papers analyzing Mirai and its history and damage it did. Then, the current state-of-the-art work in defending and cleaning IoT devices is reviewed.

3.1 Botnet Research

When Mirai emerged in 2016, it was by far not the first botnet. *Eggdrop* [50] is considered the first IRC bot, despite being harmless and developed to moderate and protect IRC channels in 1993 [51][52]. In 1998, a malicious bot based on the *mIRC.exe* client was found called *GT-Bot* [53][54]. *Agobot*, a botnet first seen in 2002, is considered one of the most capable and widespread bots of the time and "marks a turning point in which botnets have become a more significant threat" according to Grizzard et al. [17]. It not only provides the capabilities to conduct DDoS attacks, but also allows sniffing packets, monitoring keystrokes, and installing rootkit components [51].

As part of the workshop *Steps to Reducing Unwanted Traffic on the Internet Workshop* (SRUTI) in 2005, Cooke et al.[55] presented an early overview over botnets. As little was known about the effects of botnets, the researchers first interviewed network backbone operators for their experience. They reported, that botnets pose indeed a real problem. To further validate these statements, a *honeypot* was set up to observe traffic. During the experiment, the system was repeatedly compromised and became part of one or multiple botnets at once. There are also papers that measured the activity of botnets on a larger basis. Abu Rajab et al. [56] suggested that 27% of the malicious connections attempts can be attributed to botnets. Another work by Dagon et al. [57] observed botnets over a period of six months and concluded that time zones have an important impact on botnet propagation, as users tend to turn off their computer at night. As the amount of botnets grew, they gained more interest in the research community resulting in further papers giving an overview [58, 59, 60, 61, 62, 63, 64]. Bailey et al. [65] surveyed current botnet technologies in different categories. These include propagation methods, the type of communication between bot and C2 and the different attacks that botnets are used to carry out.

There is also more specific work, focusing on detection techniques [66, 67, 68, 69, 70]. A commonly followed approach is *traffic analysis*, which was used by Binkley et al. [71] from the *Portland State University* in 2006. They presented an algorithm to detect botnets

based on anomalies in IRC and TCP traffic. Their approach was used in the university network and was effective in reducing the number of bots. Gu et al. [42] presented *BotMiner*, a framework capable of identifying traffic originating from botnets. By clustering similar network traffic, *BotMiner* can identify malicious traffic and thus detect bots without the need for *a priori* knowledge. A prototype demonstrated effectiveness when tested with real-world data. Another popular method is using Machine Learning (ML) for this traffic analysis. Miller et al. [72] provide a review on different ML-based detection systems and examine the role of each method. Newer publications exist that are primarily concerned with IoT-based botnets [73, 74, 75]. Ali et al. [76] present a systematic literature review examining 34 studies on IoT-based botnet attacks. Their analysis shows that most of the surveyed papers (65%) focus on botnet detection in contrast to avoiding infections in the first place (35%).

When researchers aim to measure the effects of botnets, the collection of bot samples becomes imperative. A commonly used method are *honeypots*. An early definition of the term can be found in the book *Honeypots: Tracking Hackers* [77], where the author defines honeypots as a "security resource whose value lies in being probed, attacked or compromised". They are computers intended to get infected to then gather information on these attacks and malware. Honeypots are a widely researched topic [78, 79, 80, 81, 82, 83].

3.2 Analysis and History of Mirai

Despite the fact, that the Mirai botnet posed an acute threat to the Internet at the end of 2016, not much was known about the attacks and the malware itself except from articles reporting the events [**?**][84]. But since its first appearance, it was analyzed by many researchers [6][85]. In 2017, Antonakakis et al. [1] presented their findings about the botnet. At first, the work presents all important events on a timeline in addition to a basic outline of how Mirai works. After that, the methodology is described with the different datasets and sources. Using this data, the researches then tracked the spread of Mirai and broke down different clusters with the goal to trace the different operators of the botnet. Finally, after the authors used DNS data to analyze its DDoS attacks, a discussion outlined the most important countermeasures.

While the previously mentioned work mostly focused on the attacks of already established Mirai botnets, Sinanovic et al. [10] took a deep look into the technical details. They statically analyzed the leaked source code, giving a good overview over the different bot components and how they work. Additionally, they ran the code in a virtual environment and captured the network traffic, that allowed the creation of IDS rules. A similar overview was given by Kolias et al. [5].

Mirai is not the only botnet used to carry out DDoS attacks. Van der Elzen et al. [85] also took a look at *BASHLITE*, another strain of botnet malware that was seen in 2014. Since Mirai's source code was leaked, it served as a basis for other malware. Botnet that share parts with the Mirai source code include *Persirai* [86] that exploits a known zero-day flaw and *Hajime* [9][87] featuring a decentralized C2 protocol. There are also reports of a bot called *BrickerBot* [88][89], that compromises devices with Mirai-like techniques and inflicts permanent damage by changing network settings and trying to wipe the file system.

3.3 Immunization and Disinfection with White-hat Malware

Cao et al. [4] proposed a solution to remotely implant a "white" Mirai into devices. They assume that many IoT devices do not support easy firmware updates, so that patching vulnerable devices often means much work for the manufacturer and customer by recalling

those devices. The authors propose that a restart of a possibly infected device should be done in a given time slot, then a manufacturer-operated "white" Mirai would be infecting the device. Once installed, it closes the most vulnerabilities used by the real Mirai malware, like open ports, and tries to kill any real Mirai processes that might have infected the device at the same time. As the benign bot would also be unloaded once the device restarts, a heartbeat is sent to the manufacturer's server, that will reinfect the device automatically. This work also serves as a baseline for this thesis to further research usability limitations imposed by this approach.

Even though these white-hat approaches seem to be able to prevent Mirai infections, there are multiple ways how these can be deployed. Kageyama et al. [90] evaluated the necessary number of white-hat deployments to effectively contain Mirai outbreaks based on its infection rate. They showed, that different deployment tactics can reduce the number of white-hats needed to prevent Mirai from spreading, instead of deploying it randomly.

3.4 Detection and Remote Attestation

Whitelisting is an established method for proactively preventing non-authorized (non-whitelisted) programs from running that is widely described [91][92][93][94][95][96]. Such an approach tailored to IoT devices is proposed by Gopal et al. [97] that was successfully demonstrated in a private network. Their solution comprises a profiling module that captures the hash of all executables on a clean device and stores them in a database. Afterward, the enforcement module watches the system for application launches and calculates its hash on-the-fly. If the hash is found in the previously created database, the execution is allowed. Otherwise, the application is flagged as malicious and is not permitted to run. Additionally, an online service can also be incorporated to verify hashes that are not yet in the local database. This approach showed to be effective for preventing Mirai from running on devices.

Whereas this whitelisting is aiming to prevent malware from running in the first place, one can also try to detect infected devices in hindsight by analyzing network communication for malicious traffic [42]. The authors in [43] use a machine-learning based approach to detect traffic from IRC based botnets. First, IRC traffic is distinguished from non-IRC traffic and afterwards, it is decided whether it is IRC communication originating from botnets. Since IRC is not the only way bots communicate with C2 servers, BotSniffer [42] extends this idea to both IRC and HTTP communication. In their evaluation, the researchers found that BotSniffer was able to successfully detect all botnets tested with few false-positives.

Another approach is Remote Attestation, that can be used to verify the integrity of devices. In [41] the authors define the term *Remote Attestation* and give an overview over the general architecture. A similar overview is given in a paper by Banks et al. [39] with the addition of different types of implementation. According to the authors, Remote Attestation can be based on hardware features, that need to be present in the device. These can include vendor-specific solutions like *ARM TrustZone* or *Trusted Platform Modules* (TPMs). Software-based attestation omits the need for special hardware features, making it cheaper to implement and more portable across different devices. As this method is time-dependent, it is not viable if there is no direct communication channel between the *prover* and the *verifier*. Finally, a hybrid approach can be used, combining software-based attestation with the trust provided by the hardware.

Since *Remote Attestation* needs a *verifier* that is responsible to attest the integrity of every device, this can result in scalability issues if there are many devices involved. In their paper, Petzi et al. [98] elaborate on the problems that can occur when trying to solve the

scalability problems with distributed *verifiers*. IoT devices not only have limited memory or computation power, moreover they might also suffer from unreliable connections, can be in a sleep mode or being disconnected. This makes it hard for traditional approaches that require synchronous communication between the *verifier* and the *prover*. The authors thus propose a scheme called *Scalable Collective Remote Attestation for Pub-Sub* (SCRAPS) that uses smart contracts. This omits the need for synchronous communication, reduces the computational overhead and leads to publically verifiable attestations due to the use of a blockchain.

4. Approach

Our approach Kako will be based on the idea of a white-hat Mirai proposed by Cao et al. [4]. A white-hat Mirai will infect the device and secure it, instead of using it for malicious actions such as DDoS attacks. This existing white-hat is loaded onto the device and closes the SSH and Telnet ports. Although it effectively prevents a real Mirai from infecting the device, closing ports is an intrusive way of securing a device, as it may break the possibility of a user connecting to the device. With Kako we want to propose a solution to this problem, preserving access to the device for the user. By not closing ports, but rather changing the vulnerable password, the user can still connect to the device, instead of being locked out. Additionally, we want to simplify the process for users to get an overview over the vulnerable devices found, by listing them on a web-based dashboard. There, the collected information, such as the new password, the Linux kernel version, and the MAC address can be accessed by the user. We also make recognizing devices easier since the device's manufacturer is automatically determined.

To develop and evaluate Kako, we set up a VM environment for Mirai with *Vagrant* [11], with the integration of IDEs from *JetBrains* [12]. This allows us to streamline the development process by using the code editor and integrated compiler workflow, whereas the debugger makes analyzing Mirai's code and issues faster.

From a high-level perspective, Kako will work in the following steps, as depicted in Figure 4.1:

1. The loader is instructed to load Kako on a given list of devices. An organization may run regular scans of its network and identify devices with vulnerable Telnet credentials. In addition to the binary that the original loader drops, we create a file on the target device containing the credentials used to log in.
2. Kako will read the file to get the current combination of username and password. The password is then changed to a random one.
3. Information is collected that will be presented to the user later. This includes the current credentials, the randomly generated password, and additional information such as the Linux kernel version and the MAC address.
4. These data are then packed into a single string, through a technique called *serialization*. Kako then registers on the C2, as a normal Mirai bot would do, but also includes this additional information.

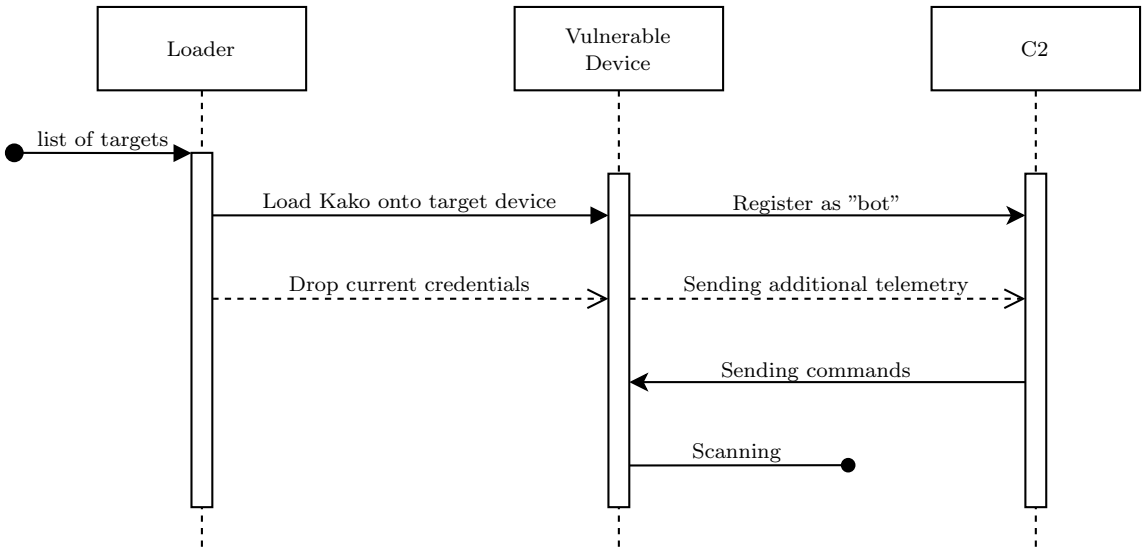

Figure 4.1: Sequence diagram showing Kako's functioning

5. The C2 server is extended with a dashboard that can be viewed in a user's web browser. It contains a list of all devices currently connected, showing their manufacturer, MAC address and whether Kako was able to automatically change the password.
6. A detail page can be opened for each device showing additional information. This includes the Linux kernel version, and most importantly the automatically set random password. This allows the user to continue to connect to the device.
7. If desired and the automatic password change was successful, the user can set a custom password. The custom password will be sent over Mirai's C2 protocol to the device, and the change will be triggered.
8. Now Mirai's scanner can scan the network for other vulnerable devices, and report those to the ScanListen component. The loader will then deploy Kako to the machine, securing it.

To achieve our goal, we have divided our approach into various components of Mirai and analyzed which parts require modifications and in which manner.

Testbed and development environment. Our first task is to set up a working Mirai instance in a VM environment. We will use a tool called *Vagrant* [11] to orchestrate the different VMs to build our environment. Afterwards, these VMs must be configured with additional software and packages to make them suitable for Mirai to work. When Mirai is running successfully, we can start integrating the IDEs. We choose *CLion* for C development and *GoLand* for the Go parts from the company *JetBrains* [12]. Since we want to make use of advanced features such as compiler integration and debugger, we use *JetBrains Gateway* to directly run the IDEs inside the VMs. This saves time and effort otherwise spent on the configuration of the IDE running outside the VM.

Bot. To implement the bot part of Kako, we will use the leaked source code of Mirai. We will first implement the logic required to retrieve the current password. With the current password for authentication, we can call the `passwd` command built into Linux to change the password of the current user. Afterwards, we collect all the required information and store pack them into into a single string, using a method called *MessagePack* [99]. Finally, we modify Mirai's registration method, to sent this payload to the C2.

C2 server and dashboard. The Mirai C2 server already implements the necessary bot management functionality, but we have to extend its data structures and modify code to support the reception and unpacking of our custom payload. Since the C2 is written in the Go programming language [100], we can use a web-framework called *Gin* [13] to serve a dashboard with this information to the user. The dashboard is implemented as an HTML page with placeholders, that get populated and rendered by the template engine of *Gin*.

When the user clicks on a device on the dashboard, a detail page will open. This is created similarly using a different HTML template that in addition to the information shown on the dashboard also displays the Linux kernel version and the random password set by Kako. It also includes a form, where users can set their own password. Upon submitting the new password, a route will be called that sends a command to the device, using the same protocol that Mirai uses to otherwise instruct DDoS attacks.

Loader. The loader requires only a simple modification. Once the loader successfully connects to a device, the bot executable is dropped. We extend this process by also creating a file called `password.kako`, that contains the username and password that were used to log on to the device.

Scanner. While Mirai already features a scanning component, we were unable to get it to work in our test environment. Since we still want to evaluate the scanner in conjunction with Kako, we decided to reimplement a simple scanner that works similarly to the existing one, but is based around the tools *netcat* [101] and *expect* [102]. For this, we loop over a list of IP addresses that are not random but belong to a predefined range of addresses. This ensures that no devices outside our possession are accessed. If an open Telnet server on port 23 is found, our scanner attempts to log in by using a stripped-down version of the credential list in Mirai's scanner to reduce failed attempts. When a working Telnet connection is established, the information is reported to the ScanListen component, in the same way that Mirai does.

5. Implementation

This chapter will outline the methodology used to construct our proposed white-hat Kako. At first, we create a virtual machine environment for running and testing Mirai, with focus on tools for the further development of Kako. Finally, we will provide a detailed description of our implementation of Kako.

5.1 Creating Virtual Machine Environment with Vagrant

There are several pre-configured Vagrant environments for Mirai available on GitHub, including the one developed by Boris Kirikov that we took as a starting point [103]. During the implementation of Kako, we heavily modified this setup to fit our needs and also replaced a Vagrant image that is no longer available for download. For our lab setup, we need two types of VMs:

- **C2 server**: A Debian 12 VM (`bento/debian-12`) with the static IP `192.168.1.2` that runs the C2 binary and a *MySQL* instance, providing the database that the C2 server requires. We also use this VM to run the ScanListen component and our modified loader.
- **Bot**: n instances of Ubuntu 16.04 VMs (`bento/ubuntu-16.04`) with static IP addresses `192.168.69.{100+i}`. Each VM is provisioned with the necessary tools and a new user is created with the name `admin` and password `admin` to make them exploitable by Mirai.

First, we modified the `Vagrantfile`, as it is the main entry point for the lab environment. It is responsible for spinning up VMs and configuring them to our needs. The VM for the C2 server is created as shown in Listing 7.

- A new VM called `c2` is created in line 1-2, using the base image `bento/debian-12`.
- The VM is customized by placing it within a private network and assigning it the static IP address `192.168.1.2` and hostname `c2` in lines 3-4. This network is shared with all other Vagrant VMs that also define `private_network` in their configuration.
- Then in line 5, a shell script is defined for provisioning. This script is executed once the machine boots for the first time.

- In addition to other configurations, we added logging of network traffic. With lines 7-9 we tell *VirtualBox* specifically to store the traffic of the second network adapter to a file called `dump_c2.pcap`. This file can then be opened with a tool like Wireshark to analyze it.

Listing 7 Vagrant definition for creating the C2 server VM

```
1  config.vm.define "c2" do |c2|
2      c2.vm.box = "bento/debian-12"
3      c2.vm.network "private_network", ip: "192.168.1.2"
4      c2.vm.hostname = "c2"
5      c2.vm.provision "shell", path: "provision/provision_dashboard.sh"
6
7      c2.vm.provider :virtualbox do |v|
8        v.customize ["modifyvm", :id, "--nictrace2", "on"]
9        v.customize ["modifyvm", :id, "--nictracefile2", "dump_c2.pcap"]
10     end
11 end
```

Kirikov's original provisioning script [103] was also responsible for compiling Mirai executables. Since we are using the IDE to build the code, we completely removed the installation of compilers for different architectures and all build steps. Our modified script does the following:

1. We install all necessary packages, most notably `golang` [100] for compiling Go code, `dnsmasq` [104] for providing a DNS and Trivial File Transfer Protocol (TFTP) server, and `mariadb-server, mariadb-client` [105] to provide the required database.
2. The value `net.ipv4.ip_unprivileged_port_start=0` is added to the file `/etc/sysctl.d/50-unprivileged-ports.conf`. This allows programs running without root access to be bound to all ports, since our IDE will run with user permissions. The configuration is then loaded with `sysctl --system`.
3. We disable Go module support with the command `go env -w GO111MODULE=off`, to simplify the installation of Go packages. Afterward, we install the packages required by Mirai in the C2's `main.go` file.
4. A SQL file is executed to create and populate the required database scheme. The SQL file is taken from Kirikov's setup and was modified to work in our environment. This is required to let Mirai run properly and to get access to the C2 interface by creating a admin user with the credentials `mirai:password`.
5. Finally, we configure `dnsmasq` as a local DNS and TFTP server. The domain `cnc.local` is resolved to the IP address `192.168.1.2` of our C2 server, and the TFTP server will serve files from the `tftp` folder. This is later used by the loader to download the bot executable to the devices.

The VMs for running the bot are created with a similar configuration, as shown in Listing 8. In line 2, there is again the definition for a new VM, but it is wrapped in a loop that thus can create an arbitrary number of bots. In our case, two of the same machines are created, but with different names and hostnames as indicated in lines 2 and 5.

The provisioning script is also simpler, since only required packages are installed and a vulnerable user with credentials `admin:admin` is created. In addition to `gcc` [22] for compiling C code, `busybox` [106], a collection of standard UNIX tools, is installed as this

Listing 8 Vagrant definition for creating the bot VM

```
(1..2).each do |i|
    config.vm.define "bot#{i}" do |bot|
      bot.vm.box = "bento/ubuntu-16.04"
      bot.vm.network "private_network", ip: "192.168.69.#{100+i}"
      bot.vm.hostname = "bot-#{i}"
      bot.vm.provision "shell", path: "provision/provision_kako.sh"
    end
end
```

is required by the Mirai bot to function. Since we implemented our own scanner module, we also have to install `expect` [102] for it to work, which is not required by Mirai by default.

The environment can then be started with the command `vagrant up`. Vagrant will download the necessary images and create VirtualBox VMs. If they are successfully booted, the provisioning scripts are executed, which finishes the setup. Now that the setup is complete, an SSH connection can be established with `vagrant ssh c2`, `vagrant ssh bot1`, `vagrant ssh c2` and so on. As Vagrant automatically mounts the directory containing the Vagrantfile to `/vagrant` in every VM, the code can be easily accessed.

To let Mirai run in our setup, some configurations have to be made within the code. As *dnsmasq* resolves `cnc.local` to the IP address of our C2 VM, we have to reflect this change in the bot code. We therefore change the value for `TABLE_CNC_DOMAIN` shown earlier in Listing 4 to the string `cnc.local` converted with the provided encryption tool. Since our C2 VM also runs the ScanListen module, we set the same value for `TABLE_SCAN_CB_DOMAIN`, which is the domain that receives the scan reports. Furthermore, the bot contains features that should make detecting Mirai more difficult. As these would interfere with our debugger and hinder analysis, we set `#define DEBUG` at the beginning of the `main.c` file. This already disables many of the anti-detection features, such as deleting the own executable and tampering with POSIX signals to prevent debugging, as explained in Section 2.2.1.

5.2 Setting Up Development Environment

Mirai by default uses a script that compiles all the necessary code files to produce the binaries for a successful Mirai deployment. This approach is enough for producing test build or even ”production” binaries, but when actively developing, this manual building procedure can be quite time-consuming. Therefore, we replace the build scripts with the integrated build functionality of the *JetBrains* IDEs.

We chose *JetBrains GoLand* and *CLion* as our IDEs, as they provide a powerful code editor, an integrated debugger for different languages, and are available free of charge for educational users. In their default configuration, the *run target* (the machine where the code is run and debugged) is the host machine, as can be seen in Figure 5.1. As this might be sufficient for most development tasks, we want our code to run isolated inside a VM, for security and networking purposes.

There are also other methods, such as using a remote machine (VM or physical computer) via SSH or using a Docker container. We tried the latter approach, but Mirai's networking did not seem to play well when running in Docker containers. To simplify the process and omit the need for debugging of networking in containers, we used *JetBrains Gateway* [49] to directly spin up the IDE backend inside the bot or C2 virtual machine, respectively. In

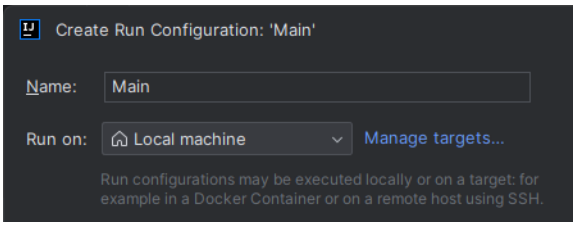

Figure 5.1: IntelliJ run configuration

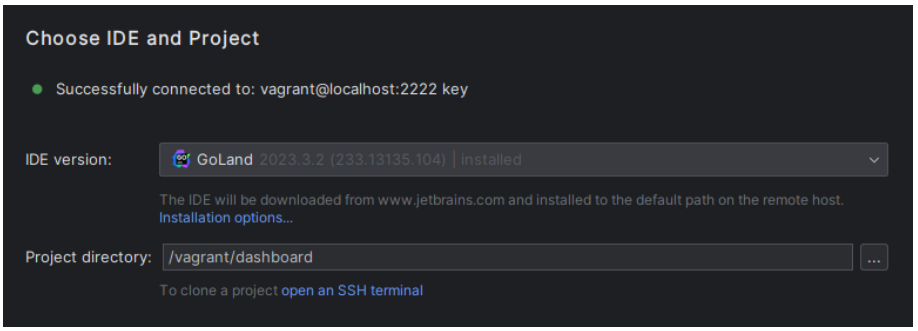

Figure 5.2: Wizard in JetBrains Gateway for creating a new IDE instance on the C2 VM

this way, the default *run configuration* can be used without modifications to directly run and debug code inside the Vagrant environment.

In *JetBrains Gateway* new SSH configurations can be added to connect to remote machines. When the machine is added and the connection is successful, a new IDE instance can be created with the wizard shown in Figure 5.2. In this example, we configure the C2 VM to run the *GoLand* IDE with the project stored in `/vagrant/dashboard`. When the wizard is completed, *Gateway* will download the IDE to the VM and connect to it.

In total, we create three instances, organized as follows:

- *GoLand* running on the C2 VM with project folder `/vagrant/dashboard` containing the dashboard
- *CLion* running on the C2 VM with project folder `/vagrant/source/loader/src` containing the loader
- *CLion* running on the bot1 VM with project folder `/vagrant/kako-bot` containing the Kako bot

We can retrieve the SSH configurations with the command `vagrant ssh-config` that returns the SSH configuration for each VM. When all instances are created, the start screen of *Gateway* will look like in Figure 5.3. The desired IDE can then be started with a click on the project folder.

The project should now be automatically detected by the JetBrains IDE, resulting in the creation of a run configuration. This allows the code to be executed or debugged with the integrated debugger by clicking *Run* or *Debug* in the IDE as shown in Figure 5.4. By starting the C2 server first and then the bot, we can successfully create a Mirai testbed within IDEs.

5.3 Implemening Kako

Now that we have a running Mirai instance, we can proceed to modify the source code. Initially, we make adjustments to the bot component to enable the gathering of information

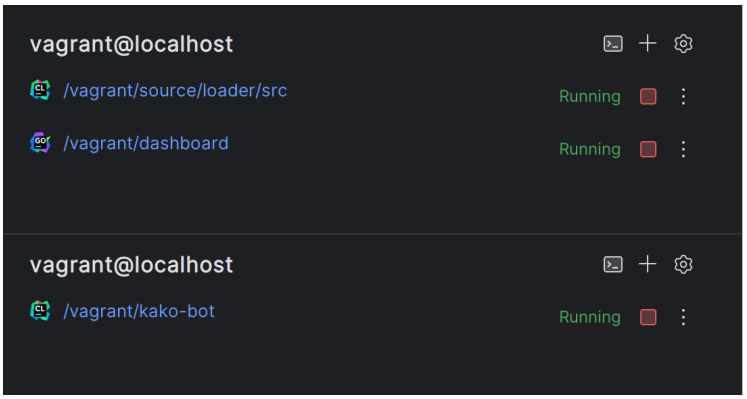

Figure 5.3: List of projects in *JetBrains Gateway*, showing all three instances

Figure 5.4: *Run* and *Debug* buttons in GoLand (left) and CLion (right)

and modify the C2 protocol to transmit this data. Then the C2 server is extended with the dashboard and the necessary modifications to receive data from the bot.

5.3.1 Implementing the Bot

The changes we have implemented in the bot are divided into two main areas. First, we need to implement the automatic change of the device's password to protect it from real Mirai login attempts. Second, we need to gather data, including the new credentials, to send them to the dashboard. We implemented most of the new functions in a new file called `kako.c`, but also modified existing code in other files.

Changing the user's password in `kako.c` The most important step is to change the vulnerable password that was used to infect the device to protect it from other Mirai variants. This is done directly after the bot has started. In Linux, there are multiple ways in which a password can be changed, but we make use of the `passwd` command, which is available on Linux and UNIX systems [107]. The user calling this command must have root permissions (being `root` user or using `sudo`) or provide the old password for authentication. Since the bot cannot retrieve the password in plain text from the system, we modified the loader to create a file `password.kako` containing the password used to log in and start the bot. This file is placed in the same directory as the bot and is read by Kako. The code responsible for changing the password is shown in Listing 9. In line 2 we create a string buffer that is used to store the shell command that is assembled in line 3. Since `passwd` prompts the user for input, we pipe a string into it that resembles the user typing the old password, followed by the new password twice, pressing enter after each line. Then we run the command on line 6 with the `system` command and store the exit code in `status`. Line 8 is where the exit code is checked, and then the function is returned with a Boolean value that indicates the result. The new password is generated randomly by Kako. Mirai already provides a pseudo-random number generator in a file called `rand.c`, that is based on simple calculations, including the current time and Process identifier (PID). Although its randomness may not meet the requirements for cryptographic use, it is sufficient to prevent basic dictionary attacks. We use the function `rand_alphastr(buf, len)` that creates a pseudo-random string of length `len` containing only alpha characters, and stores it in buffer `buf`.

Listing 9 Changing a Linux users password

```
1  bool changePassword(char* current_password, char* new_password) {
2      char command[100];
3      snprintf(command, sizeof(command), "echo '%s\\n%s\\n%s' | passwd",
       ↪  current_password, new_password, new_password);
4
5      printf("[kako] Changing password command: %s\n", command);
6      int status = system(command);
7
8      if (status == 0) {
9          printf("[kako] Password changed successfully.\n");
10         return true;
11     }
12
13     printf("[kako] Failed to change password with status %d\n", status);
14     return false;
15 }
```

Obtaining telemetry data. We also want to collect additional data that are useful to display on the dashboard. The Linux kernel version can be obtained using the `uname` method, which can be found on POSIX systems. Furthermore, we collect the MAC address of the device as an identifier. This can be problematic, since devices, especially routers, can have multiple network adapters and thus multiple MAC addresses. For simplicity, we use the address of `eth0`, which is the first network adapter on most systems.

Serializing the data. As we now collect a batch of data, we can use *serialization* to convert these data into a single string that is easier to transmit via Mirai's C2 protocol. The C2 can then covert this string back into its original structure. There are countless serialization formats like *Protocol Buffers* [108], but we opt for *MessagePack* [99], because it is easy to implement, offers libraries for many languages and is more space efficient than *JSON*. Of the several implementations available in C, we chose *MPack* [109], as it can be easily integrated into our project by only adding a source and a header file. We created a `prepare_payload` method that initializes an *MPack* instance and writes data to it in the form of a key-value map. Table 5.1 shows all the data that is now included in the structure to be serialized. Then *MPack* is instructed to complete the process and the resulting *MessagePack* encoded string is returned by the function.

Field name	Description
Uname	Linux Kernel version
Mac	MAC address of the `eth0` network adapter
User	Name of the current user
OldPw	Password used to infect the system
NewPw	Randomly generated password by Kako
PwSuccess	Boolean, if password was changed successfully

Table 5.1: Data that is sent during the registration process

Afterwards, we let Mirai continue sending its registration messages, as seen in Listing 10, but with the difference that instead of sending the `source` string in line 5, we modified it to send Kako's payload instead.

When we capture again the network traffic between the bot and C2 like in Section 2.2.1,

Listing 10 Sending the registration messages with the Kako payload

```
send(fd_serv, "\x00\x00\x00\x01", 4, MSG_NOSIGNAL);
send(fd_serv, &payload_len, sizeof (payload_len), MSG_NOSIGNAL);
if (payload_len > 0)
{
    send(fd_serv, payload_buf, payload_len, MSG_NOSIGNAL);
}
```

after the packet containing the magic value we can observe the former `source` packet now including our Kako payload. Figure 5.5 shows this packet captured in Wireguard, with our modified payload selected.

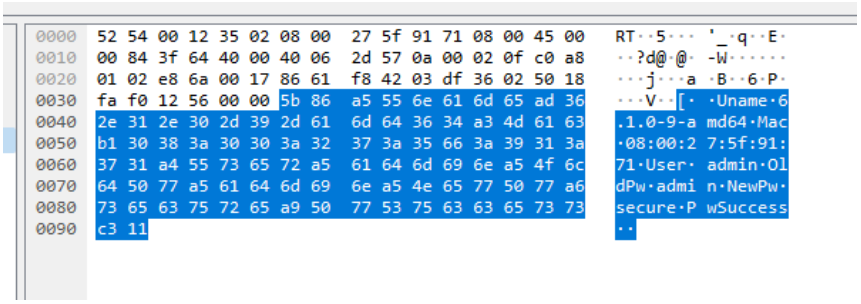

Figure 5.5: Kako payload message in Wireshark

Receiving custom commands. As mentioned in Section 2.2.2, Mirai supports receiving commands from the C2 server to trigger attacks. Since Kako is a benign white-hat approach, the attack functionality is not required. Therefore, we removed all files that contain code related to DoS attacks, to prevent them from accidentally running. Instead, we repurpose sections of the original code to create our own *command* infrastructure. This allows us to execute our own methods when instructed by the dashboard.

The commands we use are basically identical to those employed by Mirai to carry out attacks, but we modify how they are handled upon reception. When a command is received, it is parsed by Mirai into three main types:

- **type**, which indicates the type of attack
- **opts_len**, the number of options
- ***opts**, which is essentially an array of strings

We reuse this exact scheme, but instead of `attack_start()` we call our own `action_start()` method. This function, which can be seen in Listing 11, now contains the code we want to run instead. To let the user manually change the password of the device, we implemented a command with `type` 1, that will call the method `changePassword()` from Listing 9 which is also used when the bot is initially run.

Scanning module. The Mirai bot includes a scanner module that tries to connect to random IP addresses and probe ports 23 and 2323 for a Telnet server. When found, a dictionary attack on the device login is attempted. In our testing, we were unable to get the scanner running due to issues with the original code. Additionally, the scanner requires root permissions since it binds to the raw sockets of the system.

We ran Mirai on one of our bots with root permissions to start the scanning process. With the help of additional log output and the debugger, we observed that the scanner was

Listing 11 Starting action based on command from the dashboard

```
1 void action_start(int type, uint8_t opts_len, struct attack_option *opts)
2 {
3     switch (type) {
4         // change password
5         case 1:
6             printf("[kako] Changing password to: %s\n", opts[1].val);
7             bool result = changePassword(opts[1].val, opts[0].val);
8     }
9 }
```

able to send *SYN* packages over the Transmission Control Protocol (TCP) to port 23 of the VM, but the scanner was stuck at this step. We then tried to remove as much code as possible that is responsible for verifying the responses, which caused the scanner to detect a target VM as seen in Figure 5.6. At this point, the connection to the VM was lost, so no login attempts could be made. We further tried to solve the issue, but since Mirai's scanner is implemented in C by using sockets, this proved to be unfeasible within the context of the thesis.

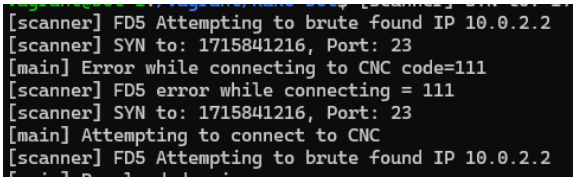

Figure 5.6: Excerpt of the log produced by Mirai's scanner module. Several attempts to brute force another VM can be observed.

Since we want to evaluate the scanning capability of Mirai with a white-hat approach like Kako, we decided to reimplement a simple scanner from scratch. Unlike the original scanner, we take the opportunity to only scan a specified IP address range, instead of randomly generated IP addresses. This not only results in fewer attempts to find devices in a known network but also mitigates the risk of legal problems caused by connecting to other devices without permission. Our implementation currently takes a start IP address, where the last octet can be incremented to a desired limit, e.g. `192.168.69.100 - 192.168.69.110`. Each address is tested for the existence of a Telnet server, by probing port 23. Due to the problems encountered with the original implementation of the scanner using sockets, we made the decision to replace it with the Linux utility *netcat* [101]. The command `nc -z <ip> 23` is used to probe port 23 of the given IP address. When the port is considered open, we use the standard *telnet* client to try different logins.

Mirai's included list of logins contains 62 different username and password combinations. To reduce the number of attempts required, we only include `admin:admin` and `admin:password`, with the possibility to extend later. Since Telnet requires that the login credentials are interactively entered (by manually responding to the login prompt), we have to automate this task. This could be implemented directly in C, but for simplicity we use another tool called *expect* [102], which can be used to automate interactive commands. Listing 12 shows the most important part of this script. Line 1 executes `telnet` with the host passed to the script. As this should return the device login promt, we tell *expect* in line 2 to wait for the string `"login:"` to appear. In line 3 we then send the user name and repeat the same process for the password in lines 4-5. Then we get two possible outcomes. If the device's response contains the word `"Welcome"` as in line 7, we exit with the return

code 0, indicating that the login credentials are correct. Otherwise, we return code 0 if the string `"Login incorrect"` is encountered.

Listing 12 Script to be executed by `expect` automating the interactive Telnet login process

```
1  spawn telnet $host
2  expect "login:"
3  send "$user\n"
4  expect "Password:"
5  send "$password\n"
6  expect {
7    -re "Welcome" {
8      exit 0
9    }
10   -re "Login incorrect" {
11     exit 1
12   }
13 }
```

This script is also dropped by our loader in the same directory as Kako, so it can be called within the code using the `system()` function, passing the found IP address and the first username and password pair from the credentials as seen in Listing 13. This is repeated until the credentials are found or the list is exhausted. In case of success, Mirai's original report functionality is used, only with a few fixes applied. We hardcoded the IP address of the C2 VM `192.168.1.2` as report server, where the unmodified ScanListen tool listens for those reports on the default port 48101.

Listing 13 Building the command to execute the *expect* script with the IP address and credentials

```
1  sprintf(command, "./telnet.expect %s %s %s", host, credentials[i].user,
   ↪  credentials[i].password);
2  int result = system(command);
```

5.3.2 Implementing the Dashboard

The dashboard is responsible for showing the information collected to the user and also for triggering custom actions on the bots. Therefore, we first need to implement the custom C2 protocol and deserialize the data and add them to the existing data structures. We can then start working on the dashboard to list bots and show detailed information.

Deserializing the Kako payload. As explained in Section 5.3.1 we serialize a key-value map of data to a string using *MessagePack*. This string can now be deserialized using a library, just like in the bot part. There are also multiple implementations, but we chose *msgpack* by developer *shamaton* [110]. It is added by installing and importing `github.com/shamaton/msgpack/v2` in the `main.go` file and is then ready to use. In order to represent the structure of our payload, we must create a Go *struct* to store the data, which can then be added to the existing Mirai bot struct. When a new bot registers at the C2, Mirai calls the `initialHandler` method to handle the registration, that is partially shown in Listing 14. Lines 1-9 show the original Mirai code that reads the *source* message, that now contains our Kako payload and stores it in an identically named string. In line 12 we create an empty instance of the `KakoInfo` struct that is then filled with the deserialized data in line 13. Finally, in line 18 the bot instance is created as usual, but

with the difference that the original *source* is set to an empty string and the payload is passed as an additional parameter.

Listing 14 Reading the source string and deserializing the Kako payload

```
var source string
if string_len[0] > 0 {
    source_buf := make([]byte, string_len[0])
    l, err := conn.Read(source_buf)
    if err != nil || l <= 0 {
        return
    }
    source = string(source_buf)
}

// Deserialize payload
payload := KakoInfo{}
msgpack_err := msgpack.Unmarshal([]byte(source), &payload)
if msgpack_err != nil {
    fmt.Println("MsgPack error:", msgpack_err)
}

bot := NewBot(conn, buf[3], "", &payload)
bot.Handle()
```

Determiting device vendor. To make it easier for the user to identify devices with Kako installed, we want to show the device vendor on the dashboard. The vendor can be determined with the help of the MAC address, as all addresses begin with six digits assigned to a manufacturer by the IEEE, called Organizationally Unique Identifier (OUI) [111][112]. The IEEE provides the list as a Comma-separated values (CSV) file, which can be obtained openly. However, for our needs, we only require a simplified map-like file. Therefore, we will use the version provided by Alasdair Allan [113], which includes only the OUI and the manufacturer's name. Upon starting the C2, the file is read and a map with the vendor and MAC addresses is created.

When the C2 server creates a new bot instance, the function `NewBot` is called, which can be seen in Listing 15. As the MAC address is transmitted in the format `xx:xx:xx:xx:xx:xx`, on line 2 we extended the function to take the first eight characters and remove the colons to get the first six digits. We can then search for them in the OUI list on line 3. Finally, we write the name to the *vendor* field of the bot. This value can then later be used in parts of the dashboard to allow users to recognize a device more easily.

Listing 15 Determiting the vendor during the creation of a new bot instance

```
1 func NewBot(conn net.Conn, version byte, source string, info *KakoInfo)
  ↪ *Bot {
2     vendorPart := strings.ReplaceAll(info.Mac[0:8], ":", "")
3     vendor := macVendorMap[vendorPart]
4
5     return &Bot{-1, conn, version, source, info, vendor}
6 }
```

Creating the dashboard. As the dashboard should be displayed in the browser, a web server is required to serve the site to the user. Since Mirai's C2 server is written in Go, we can make use of a web framework called *Gin* [13]. *Gin* provides a simple way to set up an HTTP API and serve static files. In addition, there is a basic templating system that allows HTML pages to be rendered with custom data.

It is added to the project by installing and importing the `github.com/gin-gonic/gin` package in the `main.go` file and then initialized in the `main()` function by creating a new *Gin* instance. In the default configuration, it binds to port `8080`. This way, existing C2 server data structures are available directly for use in the dashboard. In Listing 16, the first line initializes a new *Gin* instance and the second tells it to load all the templates in the `template` folder.

Listing 16 Initialization of the *Gin* framework

```
1 r := gin.Default()
2 r.LoadHTMLGlob("templates/*")
```

We implemented three main routes, which are `GET` requests to display pages to the user or `POST` requests to trigger actions.

`GET /` provides the entry point for showing the dashboard. The data displayed on the dashboard page are a list of connected clients. For this, we use the existing instance of the `ClientList` class, which is created and maintained by Mirai. In Listing 17, a new route is registered that is triggered when a `GET` request is received without a path. In our case, this is done by opening `http://192.168.1.2:8080` in a browser. When the route is hit, *Gin* is instructed in line 4 to respond with an HTTP *StatusOK* (status code 200) and return the template `index.tmpl`, which is a normal HTML file with placeholders. *Gin* then replaces these placeholders with the variables passed in lines 5-7. Figure 5.7 shows a screenshot of the dashboard, with two devices connected. On the first device, Kako was successfully able to change the password, as indicated by the *"Password changed"* badge. The second device shows another badge, indicating that further attention is required, as the automatic password change was not successful. Additionally, for the second device we manually changed the MAC address of its network adapter to imitate an IoT device by the manufacturer *Hikvision* [114], to demonstrate the vendor detection functionality.

Listing 17 Serving and populating the dashboard page

```
1 r.GET("/", func(c *gin.Context) {
2     var clients []*client
3     // ...
4     c.HTML(http.StatusOK, "index.tmpl", gin.H{
5         "clients":     clients,
6         "clientCount": len(clients),
7         "textPlural":  len(clients) != 1,
8     })
9 })
```

`GET /device/:uid` displays a detail page for a specific device. Similarly to the dashboard, a new route is created with a matching template. `:uid` in the route tells *Gin* to handle this part of the URL as a parameter that can be used in the route definition. The parameter is received with `c.Param("uid")` and used to retrieve the device from the list of clients. Figure 5.8 shows the detail page of an example device. In addition to basic information,

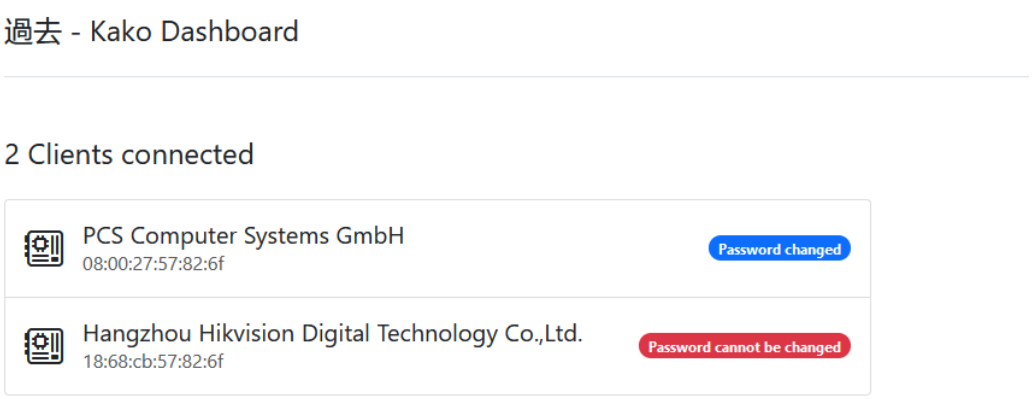

Figure 5.7: Screenshot of the Kako dashboard, showing two connected devices

such as the vendor and MAC address, different modules are displayed. The first module *"Default password"* tells the user the vulnerable credentials that were used to load Kako on the device. Additionally, if the automatic changing of the password was successful, the new randomly generated one is shown on the right side. This is important, as it allows the user to still connect via Telnet or other remote protocols that use these credentials. When desired, it is also possible to set another password with the form provided. In this case, the route `/device/:uid/vaccinate` is triggered.

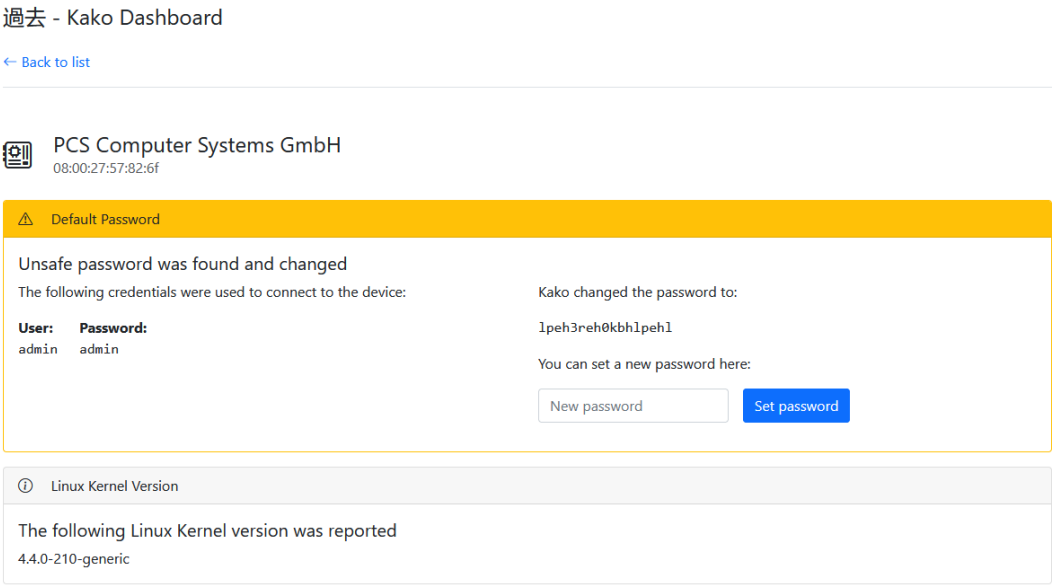

Figure 5.8: Detail page of a device, indicating the successful change of the password

If Kako was unable to automatically change the password, the module indicates this by displaying a warning message, as seen in Figure 5.9. Since it was initially not possible to modify the password, the option of manually changing it is removed. In this case, the user has to take other actions to secure the device.

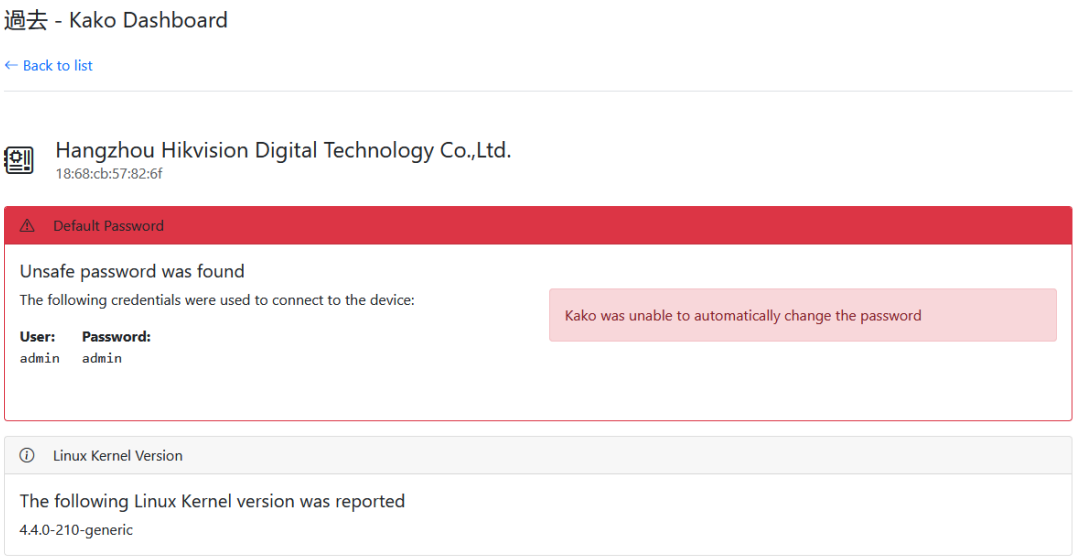

Figure 5.9: Detail page of a device, indicating the password change failed

The second module displays the version of the Linux kernel collected by Kako on the device. Although the version alone does not allow any statements to be made about the security of the device's configuration, it provides further information for professionals to make assumptions on the state and age of the firmware.

`POST /device/:uid/vaccinate` is triggered, when the user want to set an own password on the device. Beside the normal initialization of a new route, the parameter `:uid` is retrieved from the URL and converted to an Integer in line 2 of Listing 18. Line 3 furthermore stores the user-defined password from the body of the `POST` request. Similarly to the `/device/:uid` route, the ID of the client is also retrieved. With this information, a *Vaccine* instance can be created. As mentioned before during the implementation of the bot in Section 5.3.1, this struct is idenical to the original Mirai attack struct. We have decided on `1` as the type for changing the password on the client. As additional data in lines 8-9, we set `vac.Flags[0]` to the new user-defined password, and `vac.Flags[1]` to the generated password by Kako, as this is now the "old" password required for the bot to authenticate. Line 12 then sends this command over the existing C2 protocol to the bot. We now set the new password to the user-define one in line 14, and reload the page to reflect the changes. The URL for the detail page of the current device is regenerated in line 16-18. The additional URL parameter `pwChanged` is included in order to provide functionality in the template, such as displaying a success message. Finally, in line 20 a redirect to the generated URL is triggered, effectively refreshing the current page with the new user-defined password.

Listing 18 Route for changing the device password to a user-defined string

```
r.POST("/device/:uid/vaccinate", func(c *gin.Context) {
    clientId, _ := strconv.Atoi(c.Param("uid"))
    password := c.PostForm("password")
    bot := clientList.clients[clientId]

    vac := &Vaccine{0, make(map[uint8]string)}
    vac.Type = 1
    vac.Flags[0] = password
    vac.Flags[1] = bot.info.NewPw
    buf, _ := vac.Build()

    bot.conn.Write(buf)

    bot.info.NewPw = password

    q := url.Values{}
    q.Set("pwChanged", "true")
    location := url.URL{Path: fmt.Sprintf("/device/%d", clientId),
    ↪  RawQuery: q.Encode()}

    c.Redirect(http.StatusFound, location.RequestURI())
})
```

6. Evaluation

In this chapter, we will evaluate the effectiveness of Kako by demonstrating it in a simulated environment within the virtual machine setup created in Section 5.1. Figure 6.1 gives an overview of the VMs in our test setup and the services they are running. The C2 VM is running our Kako C2 server, listening for new connections and *dnsmasq* for resolving the C2 domain. It is also used to run the modified loader that receives its targets from the original ScanListen module. We start it with the command `./scanListen | ./loader` to pipe the stdout stream of the ScanListen to stdin of the loader. Both Bot VMs are setup equally with a Telnet server listening on default port 23. They only differ in their vulnerable credentials and IP address.

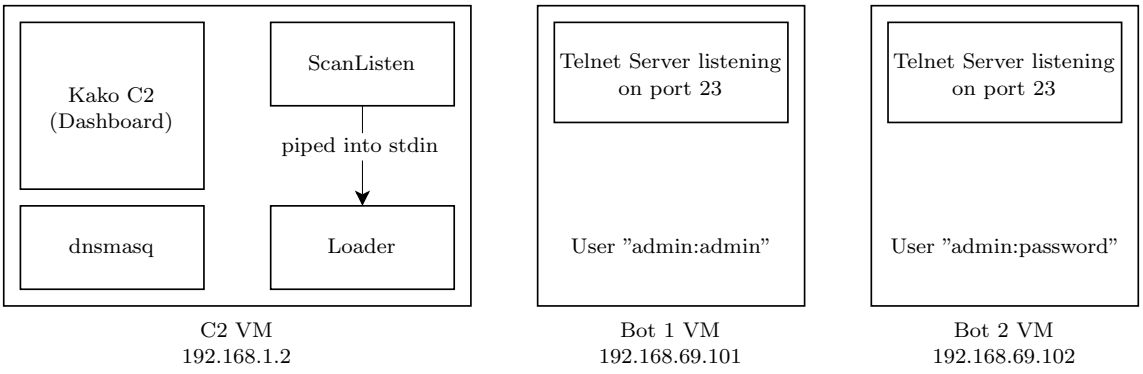

Figure 6.1: Lab setup for evaluation, showing all VMs with their services

We first tried to connect to our Bot 1 VM manually via Telnet. In Figure 6.2 we are able to successfully connect to the bot1 VM with the default credentials. This means that the device will be vulnerable to Mirai.

Now we want to immunize the machine by loading Kako onto it. Since we need at least one device with Kako running, to scan for other vulnerable devices, we manually generate a file containing the necessary target information for the Bot 1 VM as seen in Listing 19. In practice, this step can be realized by using external network scanning tools, or by modifing the existing scanner to work outside the bot.

Then we can pipe the contents of this file to our modified loader with the command `cat targets.txt | ./loader`. As we compiled the loader with all the original debug

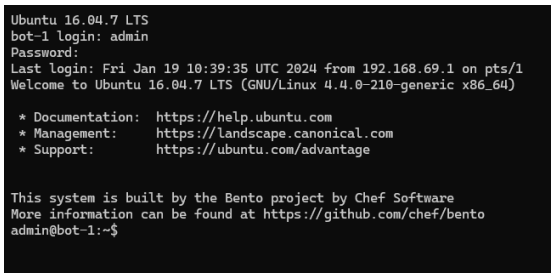

Figure 6.2: Successful Telnet conntection to the bot1 VM

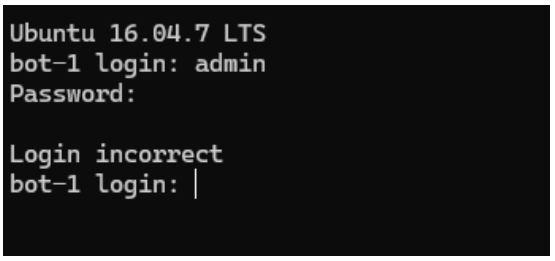

Figure 6.3: Telnet connection to bot1 fails, due to an incorrect login

Listing 19 `targets.txt` file containing the required information for the loader to infect the Bot 1 VM

```
192.168.69.102:23 admin:admin x86
```

functionality enabled, we get a lot of verbose logging. Listing 20 shows an extract of this log, where in line 1, we can observe that our modified loader creates the `password.kako` file with the credentials used to log in and runs the downloaded Kako executable. In line 3, we can observe that Kako runs the `passwd` command to change the password. Line 5 is a string Mirai prints by default to signal the loader that the loading process was successful.

Listing 20 Part of Kako's log on Bot 1, scanning and finding the vulnerable Bot 2

```
1 echo 'admin:admin' > password.kako; ./dvrHelper telnet.x86;
  ↪ /bin/busybox IHCCE..DEBUG MODE YO..
2 [main] We are the only process on this system!..Current working dir:
  ↪ /home/admin..
3 [kako] Changing password command: echo
  ↪ 'admin\nav3gqa3grg7ghv3ga\nav3gqasgrg7ghv3ga' | passwd.. Changing
  ↪ password for admin...(current) UNIX password: Enter new UNIX
  ↪ password: Retype new UNIX password: passwd: password updated
  ↪ successfully..
4 [kako] Password changed successfully ...
5 Listening tun 0...
```

When we now try to connect to the VM with the same credentials as in Figure 6.2, we get an error that our password is incorrect as seen in Figure 6.3. This confirms that now a real Mirai is no longer able to infect the device, since the weak default password for user `admin` was automatically changed. In contrast, this means that a legitimate user is also no longer able to use these credentials to log into the device. However, since Kako sends the new password to the C2 server, the user can view it on the dashboard.

We first open the dashboard, where our Bot 1 VM appeared depicted in Figure 6.4. The user can select the device to open the detail page, shown in Figure 6.5. There, the insecure credentials used to connect to the device and also the new password generated by Kako can be viewed. Furthermore, we can confirm that the MAC address and vendor[1] is correct by obtaining the value in the VM as shown in Figure 6.6. The kernel version also matches that obtained with the command `uname -r`.

We can validate that the displayed password is correct by trying to log in to the VM using it. In fact, we were able to connect successfully using the new Kako-generated password,

[1] *PCS Computer Systems GmbH* is the default vendor used for all VirtualBox VMs.

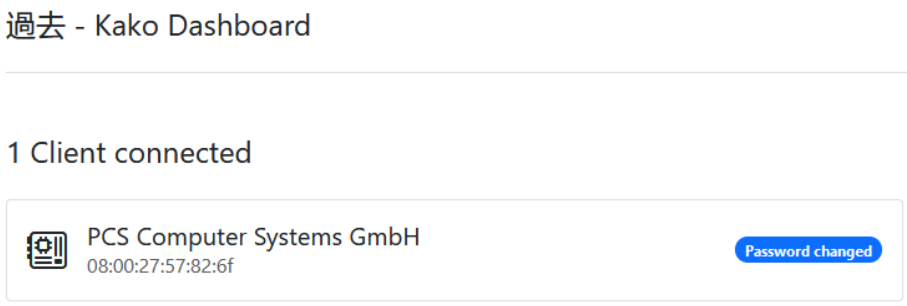

Figure 6.4: Overview of the dashboard showing the Bot 1 VM with a changed password

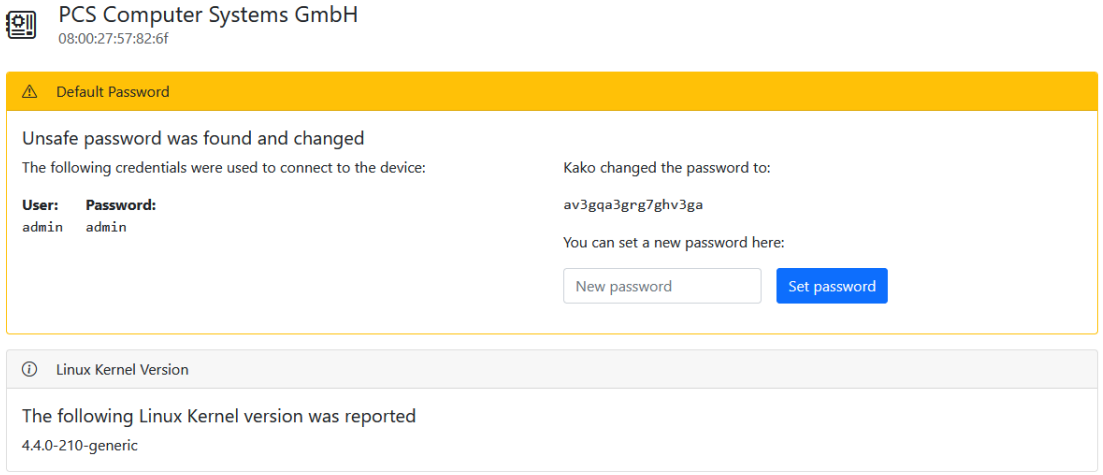

Figure 6.5: Detail page of Bot 1, showing the information collected

since we got the same successful connection as seen in 6.2. Additionally, with the `ls` command we can list the files in the user's home directory and confirm that both the bot executable `dvrHelper` and the `password.kako` file were successfully created.

The next scenario that we want to evaluate is to manually set a new password. Therefore, on the detail screen of Figure 6.5 we enter a new password and click the *Set password* button. Finally, we try again to log onto the machine over Telnet with the randomly generated password by Kako, confirming that the credentials are now wrong.

Kako will now use its scanner module to scan the network for other devices that accept Telnet connections and will try a list of login credentials. Mirai by default uses randomly generated IP addresses with only a few exceptions. Since connecting to other devices without permission might cause legal problems, we specifically set an IP range to scan. Our test VMs are in the IP range `192.168.69.10x`, so we restricted the scan area for the last octet to 100-105. Furthermore, we limited the list of credentials to `admin:admin` and `admin:password` to reduce the number of failed attempts. In Figure 6.7 we can see the part of Kako's log that contains the entries produced by the scanner. Our scanner correctly identifies that only the IPs ending *101* and *102* respond to Telnet port 23, which are both of our test VMs and the only devices in this range. Whereas all credentials are unsuccessfully probed against Bot 1, demonstrating that Kako, in fact, immunized it, it was able to connect to Bot 2 with one of the credentials.

```
admin@bot-1:~$ cat /sys/class/net/eth0/address
08:00:27:57:82:6f
admin@bot-1:~$ uname -r
4.4.0-210-generic
admin@bot-1:~$
```

Figure 6.6: Obtaining the MAC address and kernel version on the Bot 1 VM

```
[main] Resolved domain
[main] Connected to CNC. Local address = 251789322
[kako_scan] 192.168.69.100: no Telnet
[kako_scan] 192.168.69.101: found Telnet
[kako_scan] 192.168.69.101: trying 'admin' 'password'
[kako_scan] 192.168.69.101: trying 'admin' 'admin'
[kako_scan] 192.168.69.102: found Telnet
[kako_scan] 192.168.69.102: trying 'admin' 'password'
[kako_scan] 192.168.69.102: Found working credentials 'admin' 'password'
[kako_report] Send scan result to loader
[kako_scan] 192.168.69.103: no Telnet
[kako_scan] 192.168.69.104: no Telnet
```

Figure 6.7: Part of Kako's log on Bot 1, scanning and finding the vulnerable Bot 2

We can now repeat the steps of Bot 1 and view the detail page of Bot 2 as depicted in Figure 6.8. Again, we can verify that the same set of credentials is shown, as seen in the log in Figure 6.7. We can also confirm that the password is different from the one generated for Bot 1.

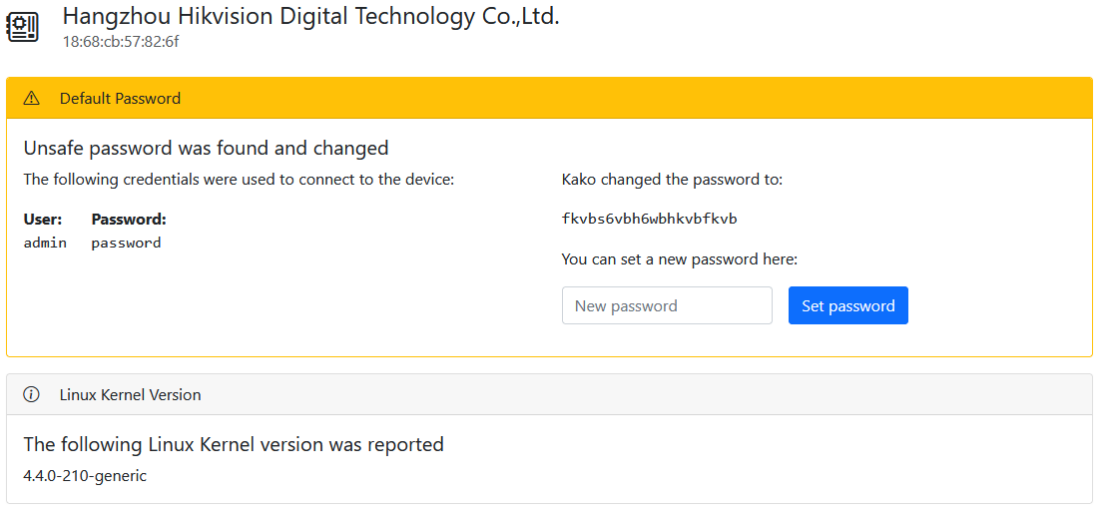

Figure 6.8: Detail page of Bot 2

As a last test, we configured our test setup to spin up four instead of two VMs and directly ran Kako on Bot 1. As we manually execute Kako, we can modify the contents of the `password.kako` file with a wrong password, so that an automatic password change will fail. In Figure 6.9 we can see the dashboard that lists the four different VMs. Bots 2-4 have their password successfully changed, while the dashboard correctly indicates the failed password change attempt for Bot 1.

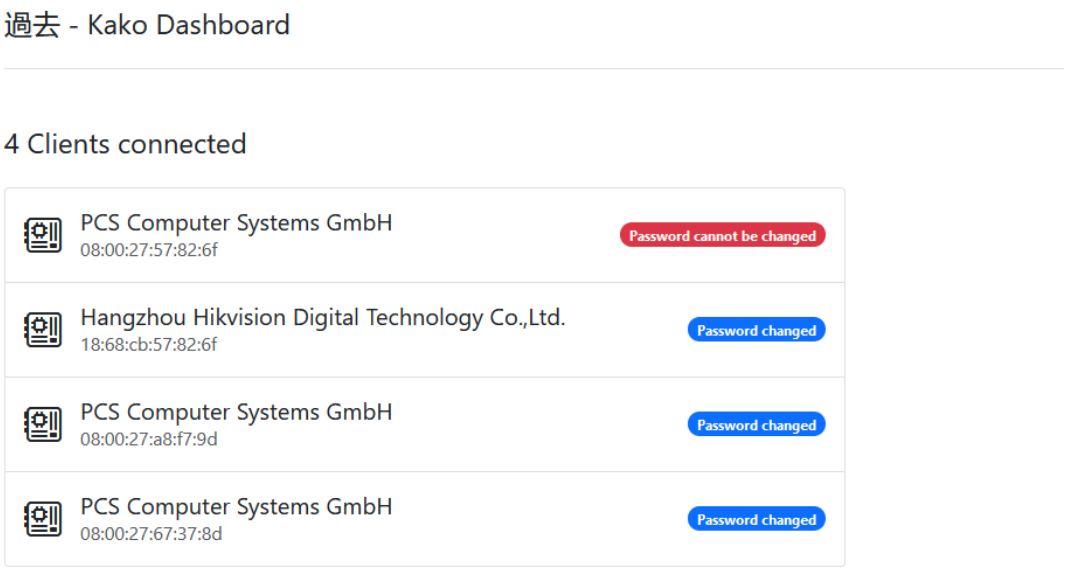

Figure 6.9: Dashboard showing four devices running Kako, with Bot 1 failed to change the password

7. Conclusion and Future Work

Mirai and other botnets continue to be a threat to the growing number of IoT devices. While traditional security approaches often struggle with IoT devices, new techniques have been developed. In this thesis, we explored how botnets work and what their goal is. We explained the fundamentals of IoT and what the security implications are. We then reviewed the different approaches that are currently used for securing IoT devices, like on-device security, firmware analysis, and traffic analysis. Mirai and its differences from classic botnets are explained, as well as how it exploits the most prevalent security flaws in IoT devices. Subsequently, we explored the current state of research concerning the defense against Mirai, which uses Mirai's own propagation techniques to create a kind of vaccine for all vulnerable devices. Cao et al. [4] demonstrated that their white-hat approach is feasible for securing devices from Mirai, as the Telnet and SSH ports are closed. Although this effectively prevents Mirai from infecting the device, usability is reduced since a legitimate user is locked out, if Telnet access to the device is still required. Consequently, this thesis proposes *Kako*, our approach for a white-hat Mirai solution, which provides a user-friendly dashboard showing all infected devices and detected vulnerabilities. During the infection of a device, *Kako* will automatically change the password to a new one, instead of completely closing the ports. As a result, a user who requires an SSH or Telnet connection can obtain the updated password from the dashboard to gain access to the device.

Our work began with setting up a Virtual Machine (VM) setup that allows us to run our own Mirai instance in an isolated environment, tailored to our needs. In this context, we also configured a development environment based around the *JetBrains* IDEs, that allowed us to implement new features for Mirai faster, with the help of the integrated debugger and compiler support.

We then modified the bot to automatically change the insecure password of the user that the loader utilized to connect to the device. As this also prevents legitimate connections from being made to the device, Kako sends this new password together with other information to the C2 server. For this to work, we had to modify the C2 protocol to incorporate these additional data into the bot registration message.

The C2 side of Mirai was adapted to build our proposed dashboard, that shows all the devices with the gathered information to the user. The original C2 server was extended with a Web server that can render pages with direct access to the original data of the Mirai C2. For a specific device, the dashboard shows the vulnerable credentials, including the updated password. The user can also set a custom password directly from the dashboard.

Future work. Our approach proved to be successful in preventing the devices from being infected by Mirai, without locking the user out of the device. Since *Kako* is based on the original Mirai source code from 2016, we only tested its effectiveness against the vulnerability of using weak default credentials. As Mirai continues to evolve through the development of modified versions and the discovery of botnets derived from it, it is possible that new vulnerabilities in addition to the default password are used to target devices. Further research has to be done to identify the most prevanlent new botnets and attack vectors they are using. Then existing approaches like *Kako* have to be tested against these variants to evaluate their effectiveness.

During the time working with Mirai, we discovered that specific assumptions are made for the victim device. First, Mirai in its original form only uses Telnet and requires *busybox* to be installed. While this might not be a problem with the goal of infecting a large number of random machines, the goal of a white-hat should be to protect as many devices as possible. Thus, future work can be done to ensure that a white-hat is able to be installed on a broad spectrum of various devices, vulnerable to different exploits. A specific step that can be done is to add SSH to the set of protocols, alongside Telnet.

List of Figures

List of Tables

List of Listings

Acronyms

DoS Denial of Service

DDoS Distributed Denial of Service

IoT Internet of Things

C2 Command and Control

RFID Radio-frequency identification

OS Operating system

DNS Domain Name System

IDS Intrusion Detection System

IPS Intrusion Prevention System

AV Anti-virus software

IDE Integrated Development Environment

RA Remote attestation

IRC Internet Relay Chat

ML Machine Learning

POSIX Portable Operating System Interface

GDB GNU Debugger

MAC Medium Access Control

OUI Organizationally Unique Identifier

CSV Comma-separated values

RAM Random-access memory

VM Virtual Machine

SQL Structured Query Language

TFTP Trivial File Transfer Protocol

PID Process identifier

stdin Standard input

stdout Standard output

CLI Command-line interface

TCP Transmission Control Protocol

Bibliography

[1] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou, "Understanding the Mirai Botnet," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 1093–1110, USENIX Association, 2017.

[2] B. Krebs, "Who Makes the IoT Things Under Attack? – Krebs on Security." `https://krebsonsecurity.com/2016/10/who-makes-the-iot-things-under-attack/`. Accessed: 22.01.2024.

[3] Z.-K. Zhang, M. C. Y. Cho, C.-W. Wang, C.-W. Hsu, C.-K. Chen, and S. Shieh, "IoT Security: Ongoing Challenges and Research Opportunities," in *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pp. 230–234, IEEE, 2014.

[4] C. Cao, Le Guan, P. Liu, N. Gao, J. Lin, and J. Xiang, "Hey, you, keep away from my device: remotely implanting a virus expeller to defeat Mirai on IoT devices."

[5] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and Other Botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[6] J. Margolis, T. T. Oh, S. Jadhav, Y. H. Kim, and J. N. Kim, "An In-Depth Analysis of the Mirai Botnet," in *2017 International Conference on Software Security and Assurance (ICSSA)*, pp. 6–12, IEEE, 2017.

[7] A. Affinito, S. Zinno, G. Stanco, A. Botta, and G. Ventre, "The evolution of Mirai botnet scans over a six-year period," *Journal of Information Security and Applications*, vol. 79, p. 103629, 2023.

[8] abuse.ch, "MalwareBazaar | Malware sample exchange." `https://bazaar.abuse.ch/`. Accessed: 10.01.2024.

[9] S. Herwig, K. Harvey, G. Hughey, R. Roberts, and D. Levin, "Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet," in *Proceedings 2019 Network and Distributed System Security Symposium* (A. Oprea and D. Xu, eds.), (Reston, VA), Internet Society, 2019.

[10] H. Sinanovic and S. Mrdovic, "Analysis of Mirai malicious software," in *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 1–5, IEEE, 2017.

[11] HashiCorp, "Vagrant." `https://www.vagrantup.com/`. Accessed: 10.01.2024.

[12] JetBrains, "JetBrains: Essential tools for software developers and teams." `https://www.jetbrains.com/`. Accessed: 14.01.2024.

[13] gin gonic, "Gin Web Framework." `https://github.com/gin-gonic/gin`. Accessed: 10.01.2024.

[14] H. R. Zeidanloo and A. A. Manaf, "Botnet Command and Control Mechanisms," in *2009 Second International Conference on Computer and Electrical Engineering*, pp. 564–568, IEEE, 2009.

[15] E. Bertino and N. Islam, "Botnets and Internet of Things Security," *Computer*, vol. 50, no. 2, pp. 76–79, 2017.

[16] J. Gardiner, M. Cova, and S. Nagaraja, "Command & Control: Understanding, Denying and Detecting - A review of malware C2 techniques, detection and defences."

[17] J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon, "Peer-to-Peer Botnets: Overview and Case Study," in *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets*, HotBots'07, (USA), p. 1, USENIX Association, 2007.

[18] Y. Chen, P. Kintis, M. Antonakakis, Y. Nadji, D. Dagon, and M. Farrell, "Measuring lower bounds of the financial abuse to online advertisers: A four year case study of the TDSS/TDL4 Botnet," *Computers & Security*, vol. 67, pp. 164–180, 2017.

[19] T. Blizard and N. Livic, "Click-fraud monetizing malware: A survey and case study," in *2012 7th International Conference on Malicious and Unwanted Software*, pp. 67–72, IEEE, 2012.

[20] Google, "Google Public DNS,"

[21] J. Gamblin, "jgamblin/Mirai-Source-Code: Leaked Mirai Source Code for Research/IoC Development Purposes." `https://github.com/jgamblin/Mirai-Source-Code`. Accessed: 10.01.2024.

[22] GCC Team, "GCC, the GNU Compiler Collection - GNU Project." `https://gcc.gnu.org/`. Accessed: 18.01.2024.

[23] S. Li, L. Da Xu, and S. Zhao, "The internet of things: a survey," *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[24] R. Cohen-Almagor, "Internet History," *International Journal of Technoethics*, vol. 2, no. 2, pp. 45–64, 2011.

[25] T. Kollmann, ed., *Handbuch digitale Wirtschaft*. Springer eBook Collection, Wiesbaden: Springer Gabler, 2020.

[26] A. Whitmore, A. Agarwal, and L. Da Xu, "The Internet of Things—A survey of topics and trends," *Information Systems Frontiers*, vol. 17, no. 2, pp. 261–274, 2015.

[27] J. Wurm, K. Hoang, O. Arias, A.-R. Sadeghi, and Y. Jin, "Security analysis on consumer and industrial IoT devices," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 519–524, IEEE, 2016.

[28] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund, "Industrial Internet of Things: Challenges, Opportunities, and Directions," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 11, pp. 4724–4734, 2018.

[29] M. Steigerwald, "Smart Home - Smart Hack," 2018.

[30] R. Brown, "Welcome to the OpenWrt Project." `https://openwrt.org/`. Accessed: 18.01.2024.

[31] A. Musaddiq, Y. B. Zikria, O. Hahm, H. Yu, A. K. Bashir, and S. W. Kim, "A Survey on Resource Management in IoT Operating Systems," *IEEE Access*, vol. 6, pp. 8459–8482, 2018.

[32] T. Uhlemann, "Sicherheitsfallen: Alte und vergessene IoT-Geräte im industriellen Einsatz." `https://www.welivesecurity.com/deutsch/2021/06/08/sicherheitsfallen-alte-und-vergessene-iot-geraete-im-industriellen-einsatz/`. Accessed: 23.12.2021.

[33] J. Koret and E. Bachaalany, *The Antivirus Hacker's Handbook*. New York, NY: John Wiley & Sons, 1. auflage ed., 2015.

[34] G. Wagener, R. State, and A. Dulaunoy, "Malware behaviour analysis," *Journal in Computer Virology*, vol. 4, no. 4, pp. 279–287, 2008.

[35] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1–12, 2017.

[36] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares," in *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA XVII)*, (Berkeley, Calif.), pp. 95–110, USENIX Association, 2003.

[37] A. Costin, A. Zarras, and A. Francillon, "Automated Dynamic Firmware Analysis at Scale," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (X. Chen, X. Wang, and X. Huang, eds.), (New York, NY, USA), pp. 437–448, ACM, 2016.

[38] D. Wang, X. Zhang, T. Chen, and J. Li, "Discovering Vulnerabilities in COTS IoT Devices through Blackbox Fuzzing Web Management Interface," *Security and Communication Networks*, vol. 2019, pp. 1–19, 2019.

[39] A. S. Banks, M. Kisiel, and P. Korsholm, "Remote Attestation: A Literature Review."

[40] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to Remote Attestation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, (New Jersey), pp. 1–6, IEEE Conference Publications, 2014.

[41] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011.

[42] Guofei Gu, Junjie Zhang, and Wenke Lee, "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic," *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008.

[43] Carl Livadas, Bob Walsh, David Lapsley, Tim Strayer, "Using Machine Learning Techniques to Identify Botnet Traffic," *2006 31st IEEE conference on local computer networks*, 2006.

[44] Oracle, "Oracle VM VirtualBox." `https://www.virtualbox.org/`. Accessed: 14.01.2024.

[45] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic Malware Analysis in the Modern Era—A State of the Art Survey," *ACM Computing Surveys*, vol. 52, no. 5, pp. 1–48, 2020.

[46] VMware, "Windows-VM | Workstation Pro | VMware." https://www.vmware.com/de/products/workstation-pro.html. Accessed: 18.01.2024.

[47] Microsoft, "Hyper-V in Windows 10." https://learn.microsoft.com/de-de/virtualization/hyper-v-on-windows/. Accessed: 18.01.2024.

[48] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis of integrated development environment recommendations," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 669–682, 2012.

[49] JetBrains, "Remote Development by JetBrains." https://www.jetbrains.com/remote-development/. Accessed: 10.01.2024.

[50] Eggdrop Development, "Eggheads – Eggdrop Development." https://www.eggheads.org/. Accessed: 28.08.2023.

[51] P. Wainwright and H. Kettani, "An Analysis of Botnet Models," in *Proceedings of the 2019 3rd International Conference on Compute and Data Analysis*, (New York, NY, USA), pp. 116–121, ACM, 2019.

[52] S. S. Silva, R. M. Silva, R. C. Pinto, and R. M. Salles, "Botnets: A survey," *Computer Networks*, vol. 57, no. 2, pp. 378–403, 2013.

[53] C. Li, W. Jiang, and X. Zou, "Botnet: Survey and Case Study," in *2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*, pp. 1184–1187, IEEE, 2009.

[54] P. Barford and V. Yegneswaran, "An Inside Look at Botnets," in *Malware Detection* (M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, eds.), vol. 27 of *Advances in Information Security*, pp. 171–191, Boston, MA: Scholars Portal, 2007.

[55] Evan Cooke and Farnam Jahanian, "The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets," in *Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI 05)*, (Cambridge, MA), USENIX Association, 2005.

[56] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "A Multifaceted Approach to Understanding the Botnet Phenomenon," in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, (New York, NY, USA), pp. 41–52, Association for Computing Machinery, 2006.

[57] D. Dagon, C. C. Zou, and W. Lee, "Modeling Botnet Propagation Using Time Zones," in *NDSS*, vol. 6, pp. 2–13, 2006.

[58] P. Amini, M. A. Araghizadeh, and R. Azmi, "A survey on Botnet: Classification, detection and defense," in *2015 International Electronics Symposium (IES)*, pp. 233–238, 2015.

[59] M. Eslahi, R. Salleh, and N. B. Anuar, "Bots and botnets: An overview of characteristics, detection and challenges," in *2012 IEEE International Conference on Control System, Computing and Engineering*, pp. 349–354, 2012.

[60] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita, "Botnet in DDoS attacks: trends and challenges," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2242–2270, 2015.

[61] S. Khattak, N. R. Ramay, K. R. Khan, A. A. Syed, and S. A. Khayam, "A taxonomy of botnet behavior, detection, and defense," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 898–924, 2013.

[62] J. Liu, Y. Xiao, K. Ghaboosi, H. Deng, and J. Zhang, "Botnet: classification, attacks, detection, tracing, and preventive measures," *EURASIP journal on wireless communications and networking*, vol. 2009, pp. 1–11, 2009.

[63] P. Rodr\'\ia-Teodoro, "Survey and taxonomy of botnet research through life-cycle," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, pp. 1–33, 2013.

[64] A. K. Tyagi and G. Aghila, "A wide scale survey on botnet," *International Journal of Computer Applications*, vol. 34, no. 9, pp. 9–22, 2011.

[65] M. Bailey, E. Cooke, F. Jahanian, Y. Xu, and M. Karir, "A survey of botnet technology and defenses," in *2009 Cybersecurity Applications & Technology Conference for Homeland Security*, pp. 299–304, 2009.

[66] K. Alieyan, A. ALmomani, A. Manasrah, and M. M. Kadhum, "A survey of botnet detection based on DNS," *Neural Computing and Applications*, vol. 28, pp. 1541–1558, 2017.

[67] A. Karim, R. B. Salleh, M. Shiraz, S. A. A. Shah, I. Awan, and N. B. Anuar, "Botnet detection techniques: review, future trends, and issues," *Journal of Zhejiang University SCIENCE C*, vol. 15, pp. 943–983, 2014.

[68] G. Khehra and S. Sofat, "Botnet detection techniques: A review," in *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*, pp. 1319–1326, 2018.

[69] M. Singh, M. Singh, and S. Kaur, "Issues and challenges in DNS based botnet detection: A survey," *Computers & Security*, vol. 86, pp. 28–52, 2019.

[70] Y. Xing, H. Shu, H. Zhao, D. Li, and L. Guo, "Survey on botnet detection techniques: Classification, methods, and evaluation," *Mathematical Problems in Engineering*, vol. 2021, pp. 1–24, 2021.

[71] James R. Binkley and Suresh Singh, "An Algorithm for Anomaly-based Botnet Detection," in *2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI 06)*, (San Jose, CA), USENIX Association, 2006.

[72] S. Miller and C. Busby-Earle, "The role of machine learning in botnet detection," in *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pp. 359–364, 2016.

[73] H. Hamid, R. M. Noor, S. N. Omar, I. Ahmedy, S. S. Anjum, S. A. A. Shah, S. Kaur, F. Othman, and E. M. Tamil, "IoT-based botnet attacks systematic mapping study of literature," *Scientometrics*, vol. 126, pp. 2759–2800, 2021.

[74] M. Wazzan, D. Algazzawi, O. Bamasaq, A. Albeshri, and L. Cheng, "Internet of Things botnet detection approaches: Analysis and recommendations for future research," *Applied Sciences*, vol. 11, no. 12, p. 5713, 2021.

[75] C. Wei, G. Xie, and Z. Diao, "A lightweight deep learning framework for botnet detecting at the IoT edge," *Computers & Security*, p. 103195, 2023.

[76] I. Ali, A. I. A. Ahmed, A. Almogren, M. A. Raza, S. A. Shah, A. Khan, and A. Gani, "Systematic literature review on IoT-based botnet attack," *IEEE Access*, vol. 8, pp. 212220–212232, 2020.

[77] L. Spitzner, "Honeypots: catching the insider threat," in *19th Annual Computer Security Applications Conference, 2003. Proceedings*, IEEE, 2003.

[78] I. Mokube and M. Adams, "Honeypots: Concepts, Approaches, and Challenges," in *Proceedings of the 45th annual southeast regional conference*, (New York, NY, USA), ACM, 2007.

[79] N. Kambow and L. K. Passi, "Honeypots: The need of network security," *International Journal of Computer Science and Information Technologies*, vol. 5, no. 5, pp. 6098–6101, 2014.

[80] L. Spitzner, *Honeypots: Tracking hackers*. Boston: Addison-Wesley, 2003.

[81] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen, "Honeystat: Local worm detection using honeypots," in *Recent Advances in Intrusion Detection: 7th International Symposium, RAID 2004, Sophia Antipolis, France, September 15-17, 2004. Proceedings 7*, pp. 39–58, 2004.

[82] C. Kreibich and J. Crowcroft, "Honeycomb – Creating Intrusion Detection Signatures Using Honeypots," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, pp. 51–56, 2004.

[83] C. Seifert, I. Welch, and P. Komisarczuk, "Taxonomy of honeypots," *Victoria University of Wellington, Wellington*, 2006.

[84] S. Mansfield-Devine, "DDoS goes mainstream: how headline-grabbing attacks could make this threat an organisation's biggest nightmare," *Network Security*, vol. 2016, no. 11, pp. 7–13, 2016.

[85] Van der Elzen, Ivo and van Heugten, Jeroen, *Techniques for detecting compromised IoT devices*. PhD thesis, University of Amsterdam, Amsterdam, 2017.

[86] T. Yeh, D. Chiu, and K. Lu, "Persirai: New IoT Botnet Targets IP Cameras." `https://www.trendmicro.com/en_us/research/17/e/persirai-new-internet-things-iot-botnet-targets-ip-cameras.html`. Accessed: 27.08.2023.

[87] S. Edwards and Ioannis Profetis, "Hajime: Analysis of a decentralized internet worm for IoT devices."

[88] Radware, "BrickerBot: Back With A Vengeance." `https://www.radware.com/security/ddos-threats-attacks/brickerbot-pdos-back-with-vengeance/`. Accessed: 27.08.2023.

[89] Radware, ""BrickerBot" Results In Permanent Denial-of-Service." `https://www.radware.com/security/ddos-threats-attacks/brickerbot-pdos-permanent-denial-of-service/`. Accessed: 27.08.2023.

[90] T. Kageyama and S. Yamaguchi, "On Tactics to Deploy White-Hat Worms in Botnet Defense System," in *2021 IEEE 10th Global Conference on Consumer Electronics (GCCE)*, pp. 294–297, IEEE, 2021.

[91] G. R. Galloway, *Application Whitelisting as a Malicious Code Protection Control*. PhD thesis, Capitol Technology University, South Laurel, 2020.

[92] C. Gates, N. Li, J. Chen, and R. Proctor, "CodeShield," in *Proceedings of the 28th Annual Computer Security Applications Conference*, (New York, NY, USA), ACM, 2012.

[93] J. Hizver and T.-c. Chiueh, "Cloud-Based Application Whitelisting," in *2013 IEEE Sixth International Conference on Cloud Computing*, IEEE, 2013.

[94] S. Obermeier, R. Schierholz, and A. Hristova, "Securing industrial automation and control systems using application whitelisting," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, IEEE, 2014.

[95] H. Pareek, "Application Whitelisting: Approaches and Challenges," *International Journal of Computer Science, Engineering and Information Technology*, vol. 2, no. 5, pp. 13–18, 2012.

[96] Sandeep Romana, Amit Jha, Janardhan Reddy, Himanshu Pareek, and P R Lakshmi Eswari, *Practical Application Whitelisting*, vol. Vol. 10. 2015.

[97] Tatikayala Sai Gopal, Mallesh Meerolla, G Jyostna, P Reddy Lakshmi Eswari, and E Magesh, "Mitigating Mirai Malware Spreading in IoT Environment: 19-22 Sept. 2018," 2018.

[98] Lukas Petzi and Ala Eddine Ben Yahya and Alexandra Dmitrienko and Gene Tsudik and Thomas Prantl and Samuel Kounev, "SCRAPS: Scalable Collective Remote Attestation for Pub-Sub IoT Networks with Untrusted Proxy Verifier," in *31st USENIX Security Symposium (USENIX Security 22)*, (Boston, MA), pp. 3485–3501, USENIX Association, 2022.

[99] S. Furuhashi, "MessagePack: It's like JSON. but fast and small." `https://msgpack.org/`. Accessed: 10.01.2024.

[100] "The Go Programming Language." `https://go.dev/`. Accessed: 18.01.2024.

[101] D. Libes, "nc(1) — netcat-openbsd — Debian testing — Debian Manpages." `https://manpages.debian.org/testing/netcat-openbsd/nc.1.en.html`. Accessed: 22.01.2024.

[102] "expect(1) - Linux man page." `https://linux.die.net/man/1/expect`. Accessed: 22.01.2024.

[103] B. Kirikov, "kribesk/security-project-mirai: Running mirai botnet in lab environment." `https://github.com/kribesk/security-project-mirai`. Accessed: 10.01.2024.

[104] "Dnsmasq - network services for small networks." `https://dnsmasq.org/`. Accessed: 18.01.2024.

[105] MariaDB Foundation, "MariaDB.org." `https://mariadb.org/`. Accessed: 18.01.2024.

[106] Erik Andersen, "BusyBox." `https://www.busybox.net/`. Accessed: 18.01.2024.

[107] V. Gite, "How To Set or Change Linux User Password." `https://www.cyberciti.biz/faq/linux-set-change-password-how-to/`. Accessed: 15.01.2024.

[108] Google, "Protocol Buffers." `https://protobuf.dev/`. Accessed: 10.01.2024.

[109] N. Fraser, "ludocode/mpack: MPack - A C encoder/decoder for the MessagePack serialization format / msgpack.org[C]." `https://github.com/ludocode/mpack`. Accessed: 10.01.2024.

[110] shamaton, "shamaton/msgpack: easier, faster, but extendable MessagePack Serializer for Golang. / msgpack.org[Go]." `https://github.com/shamaton/msgpack/`. Accessed: 16.01.2024.

[111] D. Groth, *Network+ Study Guide: Exam N10-003*. Hoboken: John Wiley & Sons Inc, 4th ed. ed., 2006.

[112] IEEE Standards Association, "IEEE Registration Authority." https://standards.ieee.org/faqs/regauth/. Accessed: 11.01.2024.

[113] A. Allan, "List of MAC addresses with vendors identities." https://gist.github.com/aallan/b4bb86db86079509e6159810ae9bd3e4. Accessed: 11.01.2024.

[114] Hikvision, "Hikvision Global English Site." https://www.hikvision.com/en/. Accessed: 20.01.2024.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Place, 23. January 2024

.....
(Felix Hack)