Honeytrack

Persistent honeypot for the Internet of Things

by

S. Kamoen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday July 24, 2018 at 13:00 PM.

Student number: 1534866

Thesis committee:	Dr. ir. J. C. A. van der Lubbe,	TU Delft, committee chair
	Dr. C. Doerr,	TU Delft, supervisor
	Dr. T. Abeel,	TU Delft, committee member

Contents

1 Introduction

In recent years, there has been an enormous increase in so called “smart” devices. Starting with the rise of the smartphone such as the iPhone and Samsung Galaxy devices, more and more consumer devices get the “smart” label. While there is no formal definition for what a makes a device “smart”, it usually involves data collection, an Internet connection, or both. These devices usually involve some logic based on this data, to automate certain tasks users had to do manually before.

This trend of giving devices access to the Internet continued to spread to devices outside of the wearable and mobile market. Next to “smart” devices, other types of devices with Internet access were introduced. The main purpose was to collect data rather than to perform automated tasks. The term “Internet of Things” was introduced to describe these types of devices. As this term is relatively broad and vague, again there is no formal specifications on what such a device must do, but effectively, they serve as Internet-connected sensors. To get an idea about what types of devices the “Internet of Things” contains, one can think of these examples:

1. Temperature sensors around your house, connected to Wi-Fi to control the heating in a house more efficiently.
2. To increase (the sense of) security, cameras are now connected to the Internet and made remotely accessible through an app so you can always monitor your home, from wherever you are.
3. Existing concepts are transformed into connected systems, such as lighting systems that can be fully automated and controlled through an app with Philips Hue or Osram Lightify.
4. Common household items such as a laundry machine or a fridge are being connected to the Internet to report on the status of your laundry or the remaining amount of eggs in your fridge.

As connected devices become the standard, more types of devices are connected to home networks and made accessible from the Internet for convenience. According to a report by Gartner, an estimated 11.2 billion Internet of Things devices will be in use in 2018, and will reach 20.4 billion by 2020 [25]. Next to home networks in the form of Wi-Fi, several specialized wireless protocols for home-automation networks emerged. These protocols such as Z-Wave[1] and Zigbee[2] allow for more efficient communication between automation products which can often be considered IoT devices as well. These networks can be exposed to the Internet through a bridge, making them accessible to other devices such as computers and smartphones which do not support these specialized protocols but do have Wi-Fi.

On the industrial side of the market, the IoT trend is starting to become clearly visible as well. According to the same report by Gartner mentioned earlier, it is estimated that businesses will spend over 1.1 trillion dollars on IoT devices [25]. Instead of expensive recurring maintenance, where specialized equipment is required for measurements, industrial machines can be equipped with cheap integrated sensors, which can broadcast their data directly over a network. When the sensor indicates something

[1] http://www.z-wave.com/
[2] http://www.zigbee.org/

is wrong, maintenance can be done based on necessity instead of periodically. Automation can go even further than just sensor data. For example, machines can be automatically and dynamically oiled from a reservoir based on sensor data, which can measure how well the machine is running and whether extra lubrication is required to prevent wear and tear. This can save a lot of expensive labour, as inspecting and maintaining these machines requires specialized personnel. IoT devices can generate a lot of data in real-time, allowing for new methods of analysis with recent techniques such as machine learning. Sensors measure the entire lifetime of a machine, including points of failure. With enough data, characteristics of a failure can be predicted by recognizing indicators leading to such failures. If the failure of an expensive industrial machine can be prevented by analyzing previous sensor data before such an event, companies can save a lot of money and productivity.

IoT adoption is also accelerated by the availability of widely deployed connectivity networks. Where traditional Wide-Area Networks (WAN) offer high speed connections through fiber, copper or 4G, such technology requires a lot of power to operate. As IoT devices only require a low bandwidth solution, Low-Power Wide-Area Networks (LPWAN) were introduced as a type of network suitable for these requirements. A number of different standards are available, of which LoRa is the most well known in Europe. This proprietary technology enables very-long-range transmissions of more than 10 km with low power consumption [22]. The LoRa alliance is a network of telecom providers across the world operating LoRa networks. In order to give these devices their required connectivity, special LoRa network modules are available, or built into the IoT devices. This way, real-time monitoring was enabled for a low price. A LoRa transmitter can operate for 15 years on two batteries [15], enabling many possibilities for widespread deployment without high maintenance cost for these sensors. In the Netherlands, KPN has a LoRa covering the entire country.

As development continues, newer products will be brought to market. For vendors, it is not economically attractive to keep supporting their entire product line-up until the last device is taken out of operation. Maintaining software is expensive, and getting updates to devices is a challenge in itself. Instead, if devices are supported beyond warranty at all, any software update often stops shortly after a new version of the product is introduced. The business model is not about a lengthy support term after purchase, but rather the purchase itself and fast replacement by a product which has more features, or is cheaper to produce.

However, products are being used much longer than the period covered under support. If an industrial sensor can operate independently for 15 years, it is unlikely it will be replaced after the warranty period ends. The same goes for consumer products. Even if replacement was an option, one can think of many scenarios where IoT devices are deeply integrated in structures or other places where a device is simply unreachable. Sensors deeply embedded in an industrial plant likely require a full plant shutdown to allow access to the sensor. Simply leaving them be is much more viable than spending millions on replacing or updating. This leads to products running on outdated software, while still connected to the Internet. Even when a device is brand new, the software controlling it might not be. Vendors often rely on hardware manufacturers themselves for software support, for example in terms of drivers for their chips. As stated by Schneier [23], the Linux version running on IoT devices was at least 4 years older than the device itself. Updates for certain vulnerabilities might not even be possible, as source code for example drivers is often not available and patching binary blobs is not feasible.

IoT devices are often directly connected to the Internet for their connectivity needs. With such a direct connection, these devices can be vulnerable when vulnerabilities are found in their software. New vulnerabilities are found very regularly and can allow unrestricted access to a device. And it is not just the devices that can contain security flaws. As new protocols are rushed into a production environment, they are not properly evaluated and can also contain more fundamental flaws, as has been shown by Yang [27] These vulnerabilities are caused by various flaws such as outdated software with known exploits, mistakes in software implementation, or the use of default credentials. Such vulnerabilities allow a complete compromise of the device with ease. Compared to compromise of a company-server containing sensitive information, the impact can be considered limited as typical IoT devices have limited capabilities. However, due to the large scale these devices are deployed and connected to the Internet, the number of compromises can become significant.

1.1. IoT malware

As IoT devices are widely deployed in mass numbers, they can be easily exploited once a vulnerability has been published. As we have established earlier, many devices will never be updated and remain vulnerable for their entire lifespan. For vulnerabilities found in desktop or server software, adversaries usually have to keep up and act fast. Serious threats are often patched relatively quickly, rendering exploits ineffective. This has lead to the rise of IoT malware, focussing specifically on low-powered devices connected to the Internet.

Once a device has been successfully breached, the adversary has full control and can do anything on the device. In cases where these devices are part of a corporate network without proper segmentation, this could even mean adversaries gain a permanent foothold inside the network from which they can continue laterally through the network[3]. By forming a so called botnet of compromised low-capability devices, a combined attack can be very effective. Vendors won't publish updates for their entire product lines for various because the devices are no longer supported, patches aren't available from subvendors, or because certain functionality essential to the device relies on the vulnerable piece of code. Next to that, the value of cheap and old devices can make the vendor decide it is no longer economically feasible to invest in updates, or they simply do not care anymore once the product has been sold. Even if a patch is available, getting the updates to these devices can be impossible, for example if it is integrated in industrial hardware. As software updates aren't being distributed to these devices, they remain vulnerable for the rest of their operating lifetime. This makes attacks against them very effective and dangerous.

An example of this effectiveness was demonstrated in various attacks of the Mirai malware. A well known and high impact attack was a major DDoS attack on DNS provider Dyn, which occurred on October 21st, 2016[4]. This attack, which caused widely used websites such as Twitter, Github, Reddit and Netflix to become unavailable. The attack was later attributed to the Mirai botnet, which turned out to consist of thousands of compromised Internet-connected security cameras with default username/-password combinations. With alleged peaks of 1.2Tbps coming from over 100,000 malicious devices, it was an attack twice as big as anything recorded before that time.

On September 30, 2016, source code of the Mirai malware was released by its author under the nickname "Anna-senpai"[5]. While this allowed for an in-depth analysis of the malware, it also allowed anyone to create their own variant. In the period after publishing the source code, many variants were discovered. For example, a variant called Satori added a zero-day exploit in the Huawei HG532 routers to infect additional hosts, next to the traditional default passwords[6].

Another notable IoT malware variant is "Hajime". This malware was first observed early as October 2016, but unlike traditional malware, Hajime does not have any capability for malicious activity at this point. Instead, it is considered a "vigilante IoT worm". Hajime attacks hosts using the same credentials Mirai uses, but the exploitation method is far more advanced than Mirai. A detailed analysis of Hajime is discussed in Chapter 3.4. It supports multiple CPU architectures such as ARM, AMD64 and MIPS. For all these architectures, the payloads contain handwritten assembly code in the binaries. Additionally, instead of a centralized C&C infrastructure, Hajime uses the decentralized BitTorrent DHT (Distributed Hash Table) protocol to discover other infected hosts and the uTorrent Transport Protocol for data exchange. Hajime does not have any malicious modules enabled in its current form, but it behaves and spreads like any other malware. Security researchers often assume a malicious module will eventually be added, but it is also possible the author of Hajime did not have any malicious intent and just wanted to prevent other malware from spreading.

Where previously discussed malware primarily focused on the Telnet protocol, that isn't the only protocol used by IoT malware. As is demonstrated by BrickerBot, a so-called "permanent denial-of-service botnet"[7]. BrickerBot uses vulnerabilities and default credentials of services listening on port 22, commonly used for SSH. Instead of using the compromised device for DDoS attacks, it disables the device

[3] https://www.zscaler.com/blogs/research/iot-devices-enterprise
[4] https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/
[5] https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/
[6] https://research.checkpoint.com/good-zero-day-skiddie/
[7] https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-back-with-vengeance/

by wiping drives and changing network parameters. This way, the device becomes inoperable and difficult to restore to a working state. While this also prevents the device from being infected by other malware, BrickerBot can definitely be considered malicious, unlike Hajime.

1.2. Adversary motivation

Sometimes, authors of malware make themselves known through a nickname on forums. They allow interviews or make statements by themselves, allowing some insight in the motivation and goals behind the malware.

When it comes to IoT malware, we see several distinct types of motivation. One category, for example Mirai's authors, aim to create a large and powerful botnet, which they can sell as a service to perform DDoS attacks. Also belonging to this category is a botnet author known as "Wicked", author of both the "SORA" and "OWARI" botnets. In an interview with Ankit Anubhav [3], "Wicked" explains his motivation behind his work. Next to making money by renting his botnet to so called "web stressers", "Wicked" mostly enjoys working on a project like this. These web stressers are a service where its clients can launch DDoS attacks on victims of their choice, by renting the capacity of the botnet.

Another category of authors is either looking for fame or trying to make name for themselves. While eventually they have a similar goal than the first group, they make themselves known online and aren't always as successful. An example of such an author is known as "Daddyl33t". This author was also tracked down by Ankit Anubhav, which was made easy as "Daddyl33t" used identifying information such as his Skype name on multiple places on the Internet, and even job interviews. From both the interview and malware samples observed in honeypots, it became clear the author fully relied on existing code and help from others on an Internet forum.

A third category aims to prevent other malware from spreading. In an article written by Cimpanu [7], a team of the website Bleepingcomputer.com managed to get an interview with the author of BrickerBot. The interview provides insight into the motivation behind this malware. The author, who goes by the nickname "janit0r", shares his dissatisfaction with the way the industry handles vulnerabilities in their devices. Rather than waiting for vendors to solve the problems, "janit0r" took manners into his own hands and allegedly disabled over 2 million devices which otherwise could have been compromised by botnets such as Mirai. While the means and execution wildly varies, both Hajime and BrickerBot fall under this category.

Even if authors make themselves known and express their motivation behind creating a bot, it is often after the damage has been done. To investigate the motivation of adversaries before a botnet is widespread can be crucial in devising security measures against them.

The various botnets also compete with each other. In a message addressed to owners of botnets, the author of BrickerBot states that he took over a significant number of devices that were previously part of competing botnets. With such competition, it is not unlikely attackers may return to their previously infected target. When an attacker returns, a number of scenarios can occur in such a setting. For instance, an attacker wants to reinforce the device's defenses against a competing botnet, or the Command and Control software requires an update that can't be done through the previously installed malware. If we want a full understanding of attacker motivations and strategies, we need a method to account for these scenarios. For this purpose, a form of persistence is needed between sessions of the same attacker. If an attacker infects a honeypot, but such a honeypot is reset after a period of time, a follow-up attack can not be observed.

1.3. IoT threat research

As the IoT market is unregulated, it is likely this trend of insecure devices hitting the market will continue for the forseeable future. In order to understand the vulnerabilities and devise measures to protect them, we need to know how these devices are being attacked.

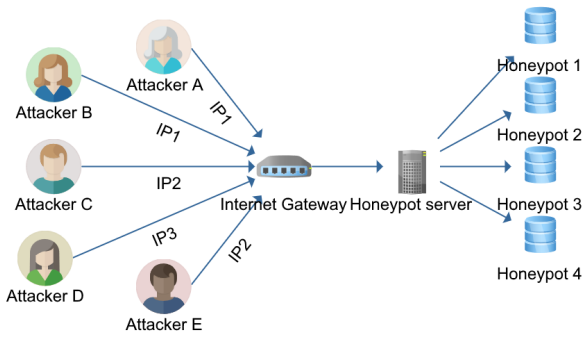
So far, research into IoT malware is mostly done by cyber security companies after a major exploit or other incident. Only after a significant attack, the cause is investigated and the responsible malware

analysed. This helps understanding the cause of the attack and possible measures to prevent further spreading. These samples can be retrieved using honeypots of various interaction levels. However, this method only allows research into the malware itself, and a very limited insight into the movement patterns. In order to discover new attacks sooner than an incident with major impact, research needs to be done in the behavioral part of adversaries. Instead of malware-based research, we have to look at IoT threats from a broader perspective. Only by answering questions covering the entire attack path from reconnaissance to post-infection behavior can we start looking at a more fundamental solution. For example, by analyzing attacks as part of the cyber killchain as introduced by Hutchins et al. [14], common patterns might be discovered that were previously missed. The killchain concept and how this can help with IoT threat research, is discussed in more detail in Chapter 2.2.

Using high-interaction honeypots, every aspect of an attack can be monitored and observed. However, once a (virtual) box is compromised, new attackers are either shut out, or encounter an already compromised machine. This can be compensated by regularly wiping the machine, allowing for new infections to take place. Recent work, such as research done by Minn et al. [17] proves this is an effective method of discovering attacks on IoT devices, but leaves room for improvement. A downside of this approach is the fact you lose any progress an adversary made up until that point. As shown in the analysis of recent botnets [4, 10], part of command sequence involves checking for previous infections. In case an adversary returns, he either finds a machine compromised by someone else, or a freshly wiped machine.

To overcome these limitations, we need a form of persistence in honeypots. By saving machine state, and binding this state to a certain attacker, we can serve attackers their “own” previously attacked honeypot, serving a large number of adversaries at the same time in parallel. The same attacker can connect to a honeypot from different IP addresses, so the IP address is not the only metric that should be used to recognize attackers. By combining different metrics, individual attackers or related groups can be connected to a single honeypot (Figure 1.1). Not only does this significantly increases the amount of potential infections, it also allows for more in-depth research towards follow-up adversary behavior.

Figure 1.1: A setup where multiple attackers can connect to different IP addresses, but are routed to the same central server. Based on different metrics, the attacker can be connected to one of the honeypots running in the backend



1.4. Research outline

This thesis introduces Honeytrack, a persistent scalable virtual high-interaction honeypot for the Internet of Things. Honeytrack aims to solve the limitations of the current available honeypots by providing the means to analyze adversaries in large networks. By using isolated containers for the high-interaction module, it allows for saving state for each adversary. In addition to that, the data collected by Honeytrack allows for an in-depth analysis of every phase of an attack, going beyond the traditional malware-sample based research.

With Honeytrack, we aim to answer the following research questions:

1) How do adversaries discover, scan and exploit potentially vulnerable devices?
    a) Can we scale honeypot deployment to gain insight in global attack behavior?
    b) What methods are used to discover devices?

   c) Can we identify groups or clusters of similar discovery methods?

   d) What are the used methods of exploitation and how do these methods relate to each other?

2) What do adversaries do after successful exploitation, what are the Tactics, Techniques and Procedures (TTP)?

   a) How are devices suitable for infection selected and identified?

   b) How is malware installed?

3) What do adversaries do with a compromised machine, what do they do to remote machines?

   a) What type of attacks are conducted with compromised machines?

The remainder of this thesis is organized as follows. Chapter 2 will introduce the background knowledge related to aspects regarding this research. We discuss the Telnet protocol, the Killchain model and finally an overview of honeypot concepts. In Chapter 3, an overview is presented of work done in the field over the past years. Next to a number of currently available honeypots, we also discuss work done in the field of honeypot evasion, adversary behavior analysis and IoT specific honeypots. From these related works, we derive a number of requirements for the honeypot in order to achieve the goals set out in this chapter. Chapter 4 introduces a number of concept for the engineering required for this thesis. The implementation developed for this thesis is described in Chapter 5. First, Honeytrap is discussed, which is the basis for Honeytrack, as described in Chapter 5.5. The evaluation of the results are discussed in Chapter 6, followed by the conclusion in Chapter 7.

2

Background

In the previous chapter, we discussed the rise of the Internet of Things and the threats that come with it. In this chapter, we will discuss a number of aspects which are important for the context of this research. First, the Telnet protocol itself is explained. Secondly, we discuss the Killchain model, which can be used to identify the different phases of a cyber attack and analyse them from a more campaign-oriented perspective. Finally, we go over the honeypot concepts. We discuss the different types, their advantages and disadvantages and how they perform in terms of killchain analysis.

2.1. The Telnet protocol

An important attack vector for IoT devices is an enabled Telnet service. Telnet is a very old inter-device communication protocol, allowing administrators to log on to a device remotely and execute commands. The Telnet protocol provides the means to establish communication with a remote server. How that communication is handled, and what the possibilities are once connected, depends on the implementation of the Telnet server.

Common Telnet servers provide a standard terminal, very similar to what one would get when connecting a monitor to a device without a graphical user interface. Other custom implementations have a limited set of commands that can be executed, specific to the device. When the Telnet protocol was designed, security was not as important as it is these days. Before encryption or security became an important aspect of the Internet, Telnet was a widely used protocol for remote interaction with systems. As proper security is essential for a remote interaction protocol, Telnet has been superseded by more modern alternatives such as SSH. However, despite the availability of protocols much better suited for this purpose, Telnet still has a significant presence on the Internet. While there are some Telnet servers that are kept online on purpose, most legitimate use of Telnet belongs to old systems that never got the support of upgrading to a modern communication protocol. Telnet is also popular on devices like modems and routers, as a way to configure these devices when they are being setup for production usage. An advantage of Telnet is the fact it is very lightweight as there are no cryptographic calculations to be done, which can be expensive on embedded chips.

The Telnet protocol is relatively straightforward and allows flexibility in its implementation. By itself, Telnet is a thin layer on top of a TCP socket. The protocol itself is described in RFC 854 [19] and RFC 855 [18]. The first standard concerns the Telnet protocol itself and the goal it aims to achieve. From RFC854, page 1:

> The TELNET Protocol is built upon three main ideas: first, the concept of a "Network Virtual Terminal"; second, the principle of negotiated options; and third, a symmetric view of terminals and processes.

This second notion plays an important role in determining device capabilities, and the level of flexibility an attacking client has. Telnet without a negotiation is very minimal, but it supports setting several

Table 2.1: The defined TELNET commands [19]

Name	Code	Description
SE	240	End of sub-negotiation parameters.
NOP	241	No operation.
Data Mark	242	The data stream portion of a Synch. This should always be accompanied by a TCP Urgent notification.
Break	243	NVT character BRK
Interrupt Process	244	The function IP
Abort output	245	The function AO
Are You There	246	The function AYT
Erase character	247	The function EC
Erase Line	248	The function EL
Go ahead	249	The GA signal
SB	250	Indicates that what follows is sub-negotiation of the indicated option
WILL (option code)	251	Indicates the desire to begin performing, or confirmation that you are now performing, the indicated option
WON'T (option code)	252	Indicates the refusal to perform, or continue performing, the indicated option
DO (option code)	253	Indicates the request that the other party perform, or confirmation that you are expecting the other party to perform, the indicated option.
DON'T (option code)	254	Indicates the demand that the other party stop performing, or confirmation that you are no longer expecting the other party to perform, the indicated option.
IAC	255	Data Byte 255

Table 2.2: An overview of structured Telnet negotiation options

Sender	Response	Description
IAC WILL [OPTION]	IAC DO [OPTION]	Sender wants to enable, receiver confirms
IAC WILL [OPTION]	IAC DONT [OPTION]	Sender wants to enable, receiver denies
IAC DO [OPTION]	IAC WILL [OPTION]	Sender wants receiver to enable, receiver confirms
IAC DO [OPTION]	IAC WON'T [OPTION]	Sender wants receiver to enable, receiver refuses
IAC WON'T [OPTION]	IAC DON'T [OPTION]	Sender wants to disable, receiver confirms
IAC DON'T [OPTION]	IAC WON'T [OPTION]	Sender wants receiver to disable, receiver confirms

conventions for a connection between two parties, in order to create a more elegant experience, such as changing the character set or echo mode. In the negotiation phase, the client and server set several parameters for the later communication. The negotiation revolves around a number of Telnet *options*, as described in RFC 855 [18], which can be set between server and client. Either party can propose setting a Telnet option using a predefined structure, to which the other party can respond with a confirmation or rejection. See Table 2.1 for an overview of all Telnet commands. Negotiation around an Option code parameter can work both ways, see Table 2.2 for an overview of typical structured negotiation scenarios. For Options where a binary `DO` or `DON'T` is not sufficient, it is possible to have sub-negotiations. An example option where this is uses, is the negotiation for the Terminal type, using the `TTYPE` option. For a sub-negotiation, a regular Option sequence is initiated as, `IAC WILL TERMINAL-TYPE`, just like the scenarios in Table 2.2. However, instead of a regular confirmation by the server, it requests additional information with `IAC SB TERMINAL-TYPE SEND IAC SE`. The client is asked to send the parameters for this option, and can do so like `IAC SB TERMINAL-TYPE IS "ANSI" IAC SE`, where `ANSI` is the Terminal Type that the client wants to use for this session.

Several Telnet implementations exist, but no reference or default implementation ever became dominant. For that reason, not all clients using Telnet fully implement the standard, meaning not all options are equally supported. This presents an opportunity to fingerprint certain devices, by sending clients negotiation requests with uncommon Options to find out which options were supported.

Support for the Telnet protocol is present in a large number of Linux-based operating systems, and by extension, in many IoT devices. By enabling Telnet capabilities by default, often using standard weak credentials, these devices became the target of recent hack attacks. Functionality that was meant to stay local is sometimes exposed to the Internet without a firewall, making them accessible to the world. These attacks are well known as they were extensively covered in the media, as discussed in Chapter 1.1. Even today, recent scans by Shodan[1] indicate there are over 6.5 million publicly available Telnet servers connected to the Internet.

An encrypted alternative was introduced in the form of TelnetS, but that never became as popular as Telnet. Instead, SSH is by far the most popular protocol with similar features like Telnet, but with much better security features and encryption.

[1] `https://www.shodan.io/`

2.2. The Cyber Killchain

An attack on a device often consists of various stages. One of the methods to divide these stages was introduced by Hutchins et al. [14], named the Cyber Killchain. The killchain is a method to map the different attack stages to a single model. By linking these separate stages together, it is possible to analyse an attack as a whole or even as part of a campaign. If two stages are often used together, the killchain model can help uncover the initial reconnaissance an attacker did after a certain type of malware was discovered on a system. It also makes the reverse possible, allowing defenders to take specific action as soon as an attack is detected in the early stages. The different stages in the killchain are:

1. Reconnaissance
2. Weaponization
3. Delivery
4. Exploitation
5. Installation
6. Command & Control
7. Actions on Objectives

In the first phase, reconnaissance, the goal is to gain as much information on the target as possible. As the killchain model covers all attacks related to cyber security, this can be through any means, both on- and offline. Depending on the target, one can think of different methods to do reconnaissance. If the target is a server connected to the machine, attackers could employ vulnerability scanners to find weaknesses in the servers' connected services. However, a target can also include the system administrator of a company, in which case one could think of observation and social media usage as methods of reconnaissance. When we consider attacks on IoT devices, this phase could, amongst other techniques, involve scanning the Internet for devices with open ports or grabbing banners from open Telnet servers.

When an attacker gathered enough intelligence to proceed with an actual attack, a remote access trojan is combined with an exploit to create a deliverable payload. The exploit is the part that takes advantage of a vulnerability in a program or other security measure, allowing the execution of code on the target system. The code that is being executed installs the remote access trojan, which is a program that allows full control over the system it is being executed on. Multiple strategies exist for both the exploit and the remote access trojan installation, all with different characteristics and suitable usages. Selecting the right combination, and putting them together is what happens in the *weaponization* phase. Mapping this phase to an IoT device, the exploit could be as easy as obtaining a set of default credentials, known to work on the device since they are hardcoded or not changed since installation. As for the payload, a variant of Mirai can be used based on its source code, but other malware compatible with IoT devices can be selected as well.

In the Delivery phase, the attacker aims to deliver the weaponized package from the previous phase to the target. There are various ways this can be achieved, for example through (spear) phishing, luring the target to a malicious website, dropping an USB-stick nearby, or connecting directly to the server. IoT devices are often directly connected to the Internet, which enables a trivial delivery phase. Simply connecting to the device allows for the delivery of the exploit in the form of default credentials, after which code can be executed through the shell access granted.

Once the package has been successfully been delivered to the target, a vulnerability is used by the exploit to execute the attackers' code on the system. This can be interpreted in different ways, and does not always rely on a technical vulnerability in a program. If a user is lured into running a malicious program, code is executed without abusing a technical vulnerability, but rather the user's trust is abused. An example on the other side of the spectrum is for example a bug in the web server software, allowing code execution on a machine without any user involved at all.

Running the attacker's code on a system results in the installation of the remote access trojan part of the weaponized package. Just like the exploit, there are many different types of trojans, all with their own benefits and features. Depending on the attacker's goal, this trojan is selected and installed. The goal of the installation phase is to maintain persistence on the system, instead of relying on exploits to run code on the system. This means the attacker has an independent foothold on the system that can't just be solved by fixing the original vulnerabilities on the system. Where attacks on IoT devices are instantiated through a poorly secured Telnet connection, malware like Mirai does not rely on Telnet to control the system. Instead, a binary is downloaded and executed, enabling control over the device independent from the Telnet connection.

The remote access trojan uses it is own methods of communication with the attacker. This communication is called Command and Control traffic, also known as C&C or C2 traffic. As it is an application communicating from inside the network to the outside, it is often not blocked by the firewall, opposed to incoming connections. The trojan connects to a server outside of the network, where it can find instructions on how to proceed with the attack.

The possibilities are endless but dependent on the features the trojan has. Common features can include opening a shell on the system, extracting files, making screenshots or spreading itself throughout the internal network. This can either be fully automated, but can also be done manually in an attempt to evade possible detection measures in place. This final phase of the attack is the last phase of the cyber killchain model, called "Actions on Objectives". Objectives vary widely per attacker. For IoT botnets, "renting" the devices for extensive DDoS attacks is a common objective, but more advanced attackers might use IoT devices to gain initial foothold inside a company network and proceed from there. Previous studies investigating IoT compromises [17] observed DDoS attacks in the form of UDP floods and many different types of TCP floods. Furthermore a DNS water torture attack[2] and SSL attacks were observed.

For accurate analysis of an attacker's methods and motivation, it is important that each stage of the killchain can independently be investigated. By creating a fingerprint for each stage, patterns can be derived to identify adversaries or groups of adversaries. More importantly, by linking the different stages together as part of a single attack or campaign, attacks can be detected considerably earlier in the killchain, reducing the amount of damage an attacker can do.

2.3. Honeypot research

In recent literature, the term honeypot is commonly used to define a concept in network security research. The first mention in literature about a network security honeypot was by Spitzner in his book "Honeypots: Tracking Hackers" [24]. In this book, a number of definitions and terms are formalized, which are later often cited in scientific literature. His definition of a honeypot is as follows:

> It is a security resource whose value lies in being probed, attacked, or compromised.

Additionally, he first describes the different types of honeypots, "physical" and "virtual":

> A physical honeypot is a real machine on the network with its own IP address. A virtual honeypot is simulated by another machine that responds to network traffic sent to the virtual honeypot.

Deploying honeypots to detect possible attack vectors isn't a novel approach. It is unclear when the honeypot concept was first introduced, but one of the earliest works which defines a honeypot which is still used to this day is HoneyD, introduced by Provos [20] in 2003. Provos proposes a framework for virtual honeypot, but also refines the definition for a honeypot as initially set by Spitzner [24].

> A honeypot is a closely monitored computing resource that we intend to be probed, attacked, or compromised. The value of a honeypot is determined by the information that we can obtain from it. Monitoring the data that enters and leaves a honeypot lets us gather information that is not available to NIDS.

[2] https://secure64.com/water-torture-slow-drip-dns-ddos-attack/

Despite its age, it is still a commonly used implementation and is used as a basis or inspiration for later honeypots. Provos introduces a lot of good concepts and requirements which overlap with the goal of this project, but the framework itself only provides a low-interaction network stack emulation.

In later work, various approaches to honeypot implementations have been introduced which can be divided in two main groups; low-interaction and high-interaction honeypots. Many implementations currently used in practice are created without published work backing it up with scientific methodology and evaluation. Past work shows a couple of different approaches. We compare these approaches using the killchain, in terms of observability and detection functionality. As these implementations still include various innovations and push the state-of-the art in terms of functionality, they are important to include in scientific research. Therefore, these implementations will be discussed later in Chapter 3.

2.3.1. Low-interaction honeypot

Low-interaction honeypots are simple applications which emulate a vulnerable system in a static and predefined manner. When an attacker connects to a low-interaction honeypot, the input is parsed and commands are extracted. When a command is supported by the honeypot, a response matching that command is returned. This means all interaction is static, a command is mapped to a static response. Some variation is possible based on some parameters from the user input, but commands are never actually executed on a system. With a low-interaction honeypot, you can research known threats by tailoring the honeypot to a specific malware variant. As commands are not really executed, deploying a low-interaction honeypot poses no significant risk to a network as there is no possibility for abuse and code execution.

Another scenario where a low-interaction honeypot is useful is for initial exploration. When emulating certain devices, the initial entry point is often shared between attack paths. Bruteforcing credentials for example is a common attack vector. While this collects adequate information about possible attackers, it lacks the functionality to analyze attacker behavior. A usecase for a low-interaction honeypot could be the collection of known malicious IP addresses. These IP addresses could then be blocked in the firewall to prevent access to any other accessible systems.

A downside to low-interaction honeypots is the limited flexibility. When an adversary connects, it is very easy to detect that the device is not real. As pre-configured commands always result in the same response, small variations in those commands can reveal if the command is actually executed or being processed by something else. Similarly, when new malware is released that does not follow the attack paths of previously observed malware, it is unlikely that low-interaction honeypots will observe a successful infection or capture any payload samples. As we will see in Chapter 6.6.5, malware relies on certain triggers to determine of a system is suitable for infection. If a low-interaction honeypot is not configured to provide a suitable response to the exploratory commands, an attacker may abort its attack.

When we map low-interaction honeypot capabilities to the killchain, we can state that each step up the killchain requires a more specific implementation. The maximum phase possible for low-interaction honeypots is an attempt at installation. For the first phase, Reconnaissance, a generic low-interaction honeypot is enough to observe an adversary scanning or probing a system.

When we move up the killchain, one can argue that delivery and exploitation are possible on a low-interaction honeypot, though with an important limitation that each step of the way requires predefined responses so that an adversary “thinks” an attack is successful. Actual *exploitation* or *installation* is not possible due to the nature of the low-interaction approach. However, these attempts can be captured and malware samples can be collected for analysis, outside the regular operation.

A low-interaction honeypot can be extended with a number of more advanced features. When it can do more than just parsing commands and returning a response, it is called a medium-interaction honeypot. Such features can include the possibility to download files when such a command is recognized, or executing a parsed command on a real system using the SSH Execute function. The major difference between low-, medium- and high-interaction honeypot remains that an attacker will not have full control over a system with the first two systems, while it does for high-interaction.

2.3.2. High-interaction honeypot

On the other end of the spectrum, you have the high-interaction honeypots. High-interaction honeypots can be further divided in physical and virtual honeypots. A physical honeypot is an actual system, assigned to a single IP address, being directly attacked by adversaries. When scaling this up to larger networks, having a physical system becomes unfeasible as it does not scale very well. That is where virtual honeypots come in. Virtual honeypots are virtual machines, which allow for multiple of them running on a single physical machine. Both types are tightly controlled by their owners as it is expected or even desired that these devices get compromised.

Opposed to lower-interaction machines where each command needs a predefined response, high-interaction machines directly execute commands issued by adversaries. This makes it very hard for an attacker to distinguish the honeypot from a real system. By letting attackers fully compromise a machine, new threats and techniques can be detected without specific knowledge of these threats.

A downside of high-interaction honeypots is the lack of control you have over what attackers are doing on these systems. If the goal is to provide a realistic experience, adversaries should be able to do anything they can do on a real system. This often includes malicious activity such as DDoS attacks or spreading itself like a worm. A trade-off needs to be made between the level of freedom and the ethical aspect of letting the device partake in malicious activity. Even if you closely monitor the incoming connections through a honeypot, it is possible an attacker installs its own software for communication such as a custom SSH server. This way, you quickly lose control and turn a honeypot into a serious security risk.

Mapping high-interaction to the killchain immediately shows the major advantage this type of honeypot has. Every phase of the killchain can be achieved, but more importantly, monitored using this approach. An adversary has a high level of freedom, but that does not mean it can't be closely monitored and collect data on what is being done on the system. This data can be analysed to get a better understanding of a malware variant or attacker motivation.

3

Requirements & Related work

While the Internet of Things may be a concept introduced in recent years, methods to research threats from online adversaries exist much longer. This chapter will provide an overview of related work in the field of honeypot research, detection and adversary analysis. Additionally, several honeypots applicable to the Internet of Things will be discussed. In Chapter 1, a number of aspects were discussed which are needed to perform research towards the current IoT threat landscape. Works discussed in this chapter are evaluated against these aspects and any shortcomings are identified.

3.1. Honeypot research

Botnets have been around longer than the recent IoT trend. This section discusses some of the work that has been done in the field of honeypots.

3.1.1. Honeypot evasion and detection

When an attacker compromises a honeypot, it allows the operator of that honeypot to fully analyze the attack path. This can lead to unwanted attention, or even unmask the attacker itself when specific pointers are found in for example the binary or the Command and Control infrastructure. For this reason, malware commonly has functionality that detects a honeypot environment, at which point the malware might abort its attack. As we're interested in analyzing attacker behavior and tracking adversaries over time, it is important to mask honeypot indicators from attackers.

A famous example of such functionality was present in the Wanacry ransomware. The security researcher known as MalwareTech found a domain name when analyzing a sample of the malware which was not registered yet[1]. Honeypot-like environments often resolve all DNS queries in an attempt to trick the malware in sending traffic to a system controlled by the researcher. In this case, Wanacry assumed that if that domain resolved, it must be running in a controlled environment, aborting further infection. By registering this domain, the malware was able to resolve the domain globally which it did not expect in a real environment. As such, it wrongly assumed all systems were a honeypot-like environment effectively stopping it from spreading any further.

While this was a poorly implemented method of honeypot detection, the principle remains an important aspect to consider. This section discusses a number of published works about honeypot detection.

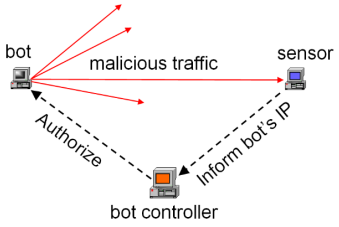
Zou and Cunningham [28] introduce several techniques and tools to detect suspicious environments. An example of such a technique, is sending a "fake" malicious instruction to the honeypot, telling it

[1] https://www.theguardian.com/technology/2017/may/13/accidental-hero-finds-kill-switch-to-stop-spread-of-ransomware-cyber-attack

to attack a target under the attackers' control. As it is impossible to determine if the target is a legitimate target or not, it is possible to verify if a honeypot will actually participate in such an attack (see Figure 3.1). This technique assumes honeypot administrators will not allow their honeypots to attack innocent computers.

Holz and Raynal [13] present a more technical approach for this problem. Based on several techniques honeypots use to secure their system from compromises, they describe how these techniques can be identified. For example, User-mode Linux (UML) is a way to embed a Linux kernel within another kernel. This way, escalated privileges on one kernel does not necessarily mean root privileges on the other. This method leaves some easily detectable traces, such as fake mounted devices and CPU information indicating such techniques are used.

Figure 3.1: Detecting a honeypot by testing the willingness to participate in an actual attack using a sensor under control of the attacker [28].



Additionally, they present techniques for detecting virtualization software VMWare, by looking at the graphics card, network card and again the names of the hard drives. As all hardware in a virtual machine (VM) is virtualized, just the names of these devices will indicate the system is actually a Virtual Machine. The detection of these aspects depend on the default settings of VMWare. It is possible to change the names of these values, but that requires changes to the actual VMware binary. Next to that, VMWare contains an I/O backdoor that allows configuration of the machine during runtime. As it turns out, it is possible to detect the presence of this backdoor, indicating the system is running in VMWare.

Finally, a method for detecting debuggers is presented. A debugger provides the operating system the means to supervise the way a process is running. Debuggers can be used to gain insight how a process is running, for example learning about the inner workings of a piece of malware. The technique described is quite complex, but it comes down to the fact a process that is being debugged, can not start a debugger itself. If a process being debugged tries to call a debugger itself, this will fail, which could indicate the presence of a debugger.

3.1.2. Adversary behavior

Once an attacker gained access to the system, there are various ways to create a profile of the attacker. In work done by Ramsbrock et al. [21], the authors ran an SSH honeypot with easy-to-guess passwords. To create a profile of attacker behavior, they looked for specific actions taken by the attacker and the order in which they occurred. This profile contained actions such as password changes, file downloads and system configuration changes. From these profiles, a state machine was built showing the number of times attackers changed from one state to another. From this data, a statistical analysis lead to a number of recommendations to increase security of systems with an exposed SSH service.

A more advanced technique of analyzing attacker behavior, is keystroke analysis. Dowland et al. [8] attempt to use keystroke analysis for user authentication. They propose to use this technique to verify periodically whether the current keystroke profile matches regular behavior. By splitting the profiling up into static profiling and dynamic profiling, access is not only granted on login, but also during system usage. While this usecase is not directly applicable to honeypot systems, having profiles for each attacker which can be compared in real-time to a current session can be useful for adversary identification. This method however, requires a lot more data that simply the network delays between characters. Their models include the exact times a key is pressed, staying pressed and then released. Such information is not available for a honeypot implementation. Ahmed et al. [1] attempts to minimize the data required to construct a reliable fingerprint based on keystrokes. Their models require less data than Dowland et al., but experiments are based on client-side software, which make it not directly applicable to a honeypot situation. Their detector includes a false acceptance rate (FAR) of 0.0152% and a false rejection rate (FRR) of 4.82%. Telnet allows per-character transmission of input using the `LINEMODE` option (as described in Chapter 2.1). By measuring the timing between input, it could be possible to identify returning attackers based on their keystroke patterns.

3.2. IoT honeypots

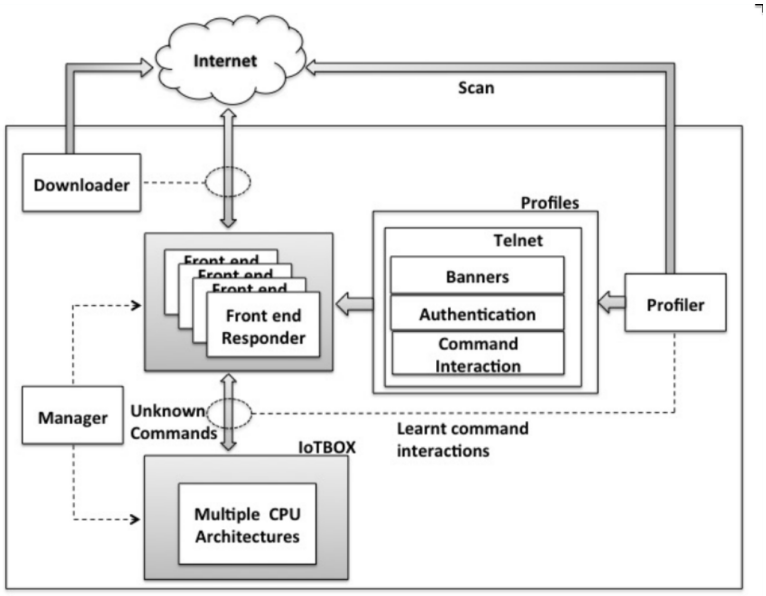
The rising threat against IoT devices did not go unnoticed and the need for honeypots was recognized by several developers. Honeypots discussed in this section all aim to gather intelligence on malware targeting the Internet of Things.

3.2.1. IoTPot

An initiative from Japan by Minn et al. [17] called IoTPot shows a lot of similarities with our research goals. They propose a honeypot that aims to retrieve IoT malware binaries by emulating multiple CPU architectures. By supporting multiple architectures, a wide array of malware can be captured and analysed. This way, they've been able to identify several different groups of malware, attackers and analyse their behavior.

The design is specifically tailored to emulate various Telnet devices, based on the Telnet protocol (Chapter 2.1) The banners are collected by scanning active devices on port 23, accessible from the Internet. For authentication, IoTPot supports various strategies for accepting credentials. For example, it can accept certain specific credentials, or accept any credentials after a number of attempts. As for the interaction phase, IoTPot supports running commands on different CPU architectures, as not all IoT devices run on the same hardware platform.

Figure 3.2: An overview of IoTPot [17]



A design overview of IoTPot can be found in Figure 3.2. The implementation for IoTPot consists of the following modules:

Frontend Responder The Frontend responder handles the incoming connections from attackers. The frontend responder has a set of "device profiles", which consists of details on the different phases of a Telnet session such as Telnet negotiation options, banner, accepted credentials and how to respond to interaction commands. The Manager component selects which device profile is linked to which IP, so that multiple device configuration can be tested on the same instance. When an attacker enters a command now previously observed, the Frontend Responder sets up a Telnet connection to IoTBox, which is the execution sandbox discussed later.

Profiler When the Frontend Responder sends an unknown command to IoTBox, the Profiler is used to extract the response. This response is saved in the device profiles for later use. This way, when this command is seen in a later session, the Frontend Responder knows how to respond.

Downloader When an attacker executes commands, it is possible they attempt to download malicious files. The Downloader intercepts such commands and downloads these files in a safe environment for later analysis.

IoTBox When an unknown command is received by the Frontend Responder, it is sent to IoTBox. IoTBox is a set of sandbox environments that run a Linux distribution for embedded devices with multiple different CPU architectures.

IoTPot was deployed for 39 days across 165 IP addresses. The device profiles consisted of 29 banner profiles, and a single command interaction profile based on a commonly exploited DVR. The IoTBox backend supported 8 different CPU architectures, all running the OpenWRT Operating system, commonly found in routers. During the observation period, 49.141 out of 76.605 attackers successfully logged in, and 76.605 download attempts were intercepted.

From their observations, they divide an attack path into three different stages: intrusion, infection and monetization. These stages are described as a simplified version of the Killchain, as discussed in Chapter 2.2. Findings include the discovery of 4 malware families with unique characteristics for each of the described stages. Additionally, they found that all families but one share a common goal, namely DoS attacks and Telnet scans. Finally, they conclude that the different malware families have a different level of aggression. For example, a family named ZORRO updated its command sequences twice during the observation period.

The work done by Minn et al. was a major source of inspiration for this project. Their design approach ensures a wide variety of devices emulated for the first phases of a Telnet connection before the interaction. However, a downside of their approach is that interaction is saved for later use, where the sandbox environment (called IoTBox), is reset after a while. This way, follow-up commands to an already infected machines can not be reliably reproduced, and commands are sent to a clean machine instead. As observed during their experiments, adversaries sometimes return to the same device with a different command sequence. As the device was supposed to be compromised before, this should be accounted for when commands are executed. Furthermore, the research seems to be focussed on collecting malware samples, rather than adversary behavior. By splitting up the attacks in only 3 phases, valuable intermediate steps are combined instead of analysed separately.

From IoTPot, we can derive a number of requirements, to both replicate and extend their work.

1. Support for configurable Telnet protocol responses
2. Support for configurable Banner
3. Able to run commands on “real” systems
4. Account for attackers returning to systems previously compromised

3.2.2. IoTCandyJar

While acknowledging the limitations of low-interaction honeypots, Luo et al. [16] propose a machine-learning based approach, named “IoTCandyJar”. They state that building a comprehensive low-interaction honeypot is very time-consuming and difficult due to the large variety in IoT devices. High-interaction honeypots also pose a challenge, as emulators aren’t available and deploying physical machines would pose increased risk and is not very cost-effective. Instead, Luo et al. propose a new interaction level: “intelligent-interaction”. The goal is to extract malware samples from the honeypot, and for that all they need is to come up with a response such that an adversary executes the commands leading to a payload. From the requests received, a suitable response has to be derived. Using a module “IoT-Scanner”, they scan the Internet, searching for IoT devices and repeating the previously received requests to the devices. This way, they get a response a real IoT device would return, adding to the knowledge required for a convincing reply.

This concept has some similarities with IoTPot, though given enough time, “IoTCandyJar” could learn to deal with a wider array of responses than the limited number of devices IoTPot has at its disposal. Their approach is novel, but the goals differ from the goals set for this research. There is no persistence, or even a environment that allows adversaries to return to, as the honeypot still effectively behaves as a low-interaction honeypot. An approach like this however, does help solve the problem of honeypot evasion techniques.

3.3. Available implementations

Most honeypots used in practice are developed without a scientific paper to back them up. There are many implementations available, both freely available or as a commercial product. There is a lot of overlap in functionality, and each honeypot has their own advantages and disadvantages, as each honeypot is designed and implemented with a specific goal in mind. We will evaluate a number of potentially suitable options in this chapter.

Many implementations are available to mimic Databases, web applications and mailservers. For our goal, the Internet of Things, a number of potential candidates are available. The following honeypots were selected based on their popularity on Github and features relevant for our research. Especially Telnet support was an important consideration.

3.3.1. Kippo, Cowrie and Kojoney2

Kippo[2] is a popular SSH honeypot written in Python. It is currently no longer active development, but has been forked by Michel Oosterhof, who named it Cowrie[3]. Cowrie is a medium interaction SSH and Telnet honeypot designed to log brute force attacks and the shell interaction performed by the attacker. Next to the typical low-interaction features such as credential collection and fake shells, it is possible to forward unknown commands to an SSH instance using the “Execute” functionality. This is not considered a true high-interaction honeypot based on the definition from Chapter 2.3.2, but because of the dynamic nature it is often referred to as medium interaction.

There is a lot of functionality in Cowrie that makes it behave like a high-interaction honeypot. For example, there is support for a fake filesystem, so if an attacker tries to access files through the emulated shell, a number of files are actually available. This enables users to configure a honeypot with a high level of flexibility, but with a much lower risk than a typical high-interaction honeypot.

Another popular medium-interaction honeypot is Kojoney2[4]. It offers similar functionality than Cowrie, but is focused on SSH instead of both SSH and Telnet. With Kojoney2, download requests made by attackers are actually performed in the background so they can be analyzed later. These files are not usable in the emulated environment so there is no risk of infection through binary execution.

3.3.2. Malware-specific

Additionally, various Mirai specific honeypots are available as well. Based on analysis of the Mirai malware, these honeypots are designed in such a way that responses to the known commands are within the Mirai selection criteria. This makes it very useful to collect Mirai malware samples and to research attackers with a very similar attack path as Mirai itself. However, for more generic research discovery of previously unknown malware, these honeypots aren’t advanced enough. These low-interaction implementations can be useful for initial data gathering, but for the more advanced requirements, a high-interaction honeypot is needed.

MTPot is a project by Cymmetria[5], designed to capture Mirai samples. It is designed to listen for Telnet connections, and behave exactly as Mirai expects it, triggering a sample download. It is a pure low-interaction honeypot, with predefined expected commands and related responses. The application is useful to investigate the circumstances in which Mirai will attempt to infect a machine. However, as a whole, it is far too simple and is missing a lot of configuration options to be suitable for our goals.

3.4. IoT botnet analysis

In Chapter 1.1, we discussed a number of current botnets active on IoT devices. A number of these botnets were extensively analysed, providing insight into the way adversaries attack and infect de-

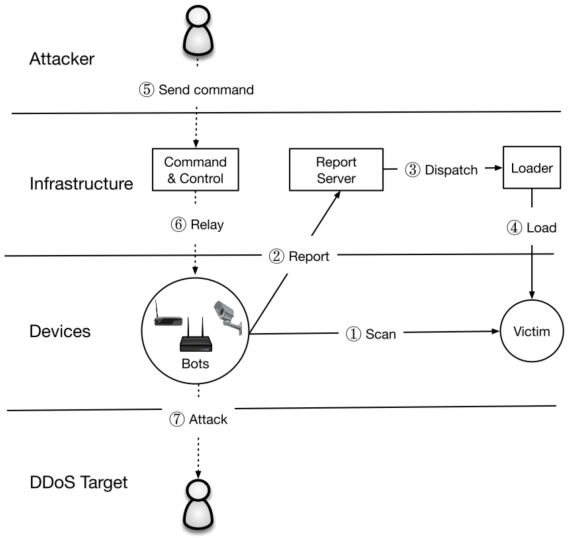
[2] https://github.com/desaster/kippo
[3] https://github.com/micheloosterhof/cowrie
[4] https://github.com/madirish/kojoney2
[5] https://github.com/Cymmetria/MTPot

vices.

3.4.1. Mirai

Mirai was analysed by Antonakakis et al. [4]. While Mirai only gained widespread attention by a number of massive DDoS attacks on “Krebs on Security”, OVH and Dyn, reports of Mirai were seen several months earlier on August 31 2016. Based on the source code which was released, the authors developed a measurement methodology. A copy of the source code itself can be found on Github[6]. Figure 3.3 depicts the operation method of Mirai.

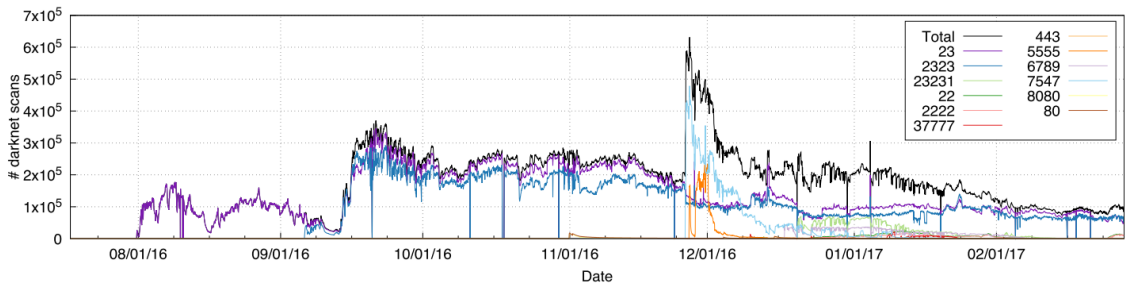
Figure 3.3: Operation of the Mirai botnet [4].



Mirai finds its victims by scanning the Internet in a pseudo-random order using TCP SYN probes. These scans are done on port 23 and port 2323, and certain IP ranges are blacklisted, likely to prevent drawing attention from high-profile companies or governments. After a potential victim is found using the scans, a Telnet connection is established and 10 random entries are selected from a pre-configured list of 62 credentials. On successful login, the victim IP and correct credentials are sent to a reporting server. Separate from the reporting server, a loader program logs into the victim machine using the reported credentials. This loader program determines the underlying architecture, after which it loads and executes the specific malware. After a successful infection, Mirai attempts to hide itself by deleting the executables and using pseudorandom names for its process. As a result, Mirai will not persist across reboots, making it harder to investigate. In order to prevent other malware from infecting the device, processes bound to port 22 and 23 are killed, as well as known processes from rivaling malware such as `.anime` and Qbot. Finally, the infected device will listen for Command and Control traffic while scanning for other potential victims using the method described above.

The number of infected devices changed a lot over time. As can be seen in Figure 3.4, Mirai had between 200,000 and 300,000 infections in September 2016. In November of that year, a peak of 600,000 infections was observed, with a decrease to 100,000 devices around the end of February 2017. This big variance in infection numbers was due to the evolution of Mirai during the observed time period.

Figure 3.4: Number of Mirai infections over time [4].



The authors note that even before the source code was released on September 30 2016, 24 unique Mirai samples were uploaded to VirusTotal. These samples were used to analyse Mirai’s initial evolution.

[6] `https://github.com/jgamblin/Mirai-Source-Code`

Where the C&C infrastructure was initially IP address based, this was later upgraded to a domain-based variant in mid-September. Additionally, around the same time the malware began to delete its own binaries and obfuscate its process ID. In one of the last samples collected before the source code release, features such as the closing of infection ports and aggressive killing of competitive infections were observed. After the release, the malware evolved rapidly, with different actors adding various features. These features include new sets of credentials, changes to the IP blacklist and scanning on ports related to the CWMP protocol.

Based on a DNS dataset in combination with the obtained malware samples, Antonakakis et al. were able to cluster Command and Control Server IP addresses and shared network infrastructure. These clusters varied greatly in size, the smallest being a single host and the largest containing 11 domains with 92 IP addresses. As these clusters do not share any infrastructure, it is possible that multiple actors were actively operating botnets using Mirai. These clusters showed different levels of evolution, but a large number of features did not equal successful infection volumes. Using data from a large ISP, the authors were able to determine DNS lookup volumes for domain names previous obtained through reverse engineering. The cluster with the highest number of features and added credentials was only 19th out of 33 clusters in terms of DNS lookup volume. Oh the other hand, a cluster which showed no evolution at all achieved the highest lookup volume of all clusters.

During the monitoring period, thousands of DDoS attacks were observed conducted by Mirai. Next to the high-profile attacks on “Krebs on Security”, Dyn and Liberia’s Lonestar, Mirai showed the same characteristics as a so called stresser- or booter service. These services allow customers to pay for DDoS attacks against targets of their choice. Some Mirai operators were targeting gaming platforms such as Steam, Minecraft and Runescape. These attacks employed a wide range of strategies, such as volumetric attacks, TCP state exhaustion and application-layer attacks. An interesting observation was made regarding one of the attack targets. The 7th most popular target was an IP address associated with one of the Command and Control servers. Out of the 484 observed Mirai Command and Control IPs, 47 were the target of a DDoS attack themselves. When clustered by attack commands, 93 unique clusters were identified, of which 26 clusters were targeted at least once. This supports the hypothesis that Mirai actually consists of multiple rivaling botnet operators.

Where common DDoS attack relies on amplification strategies, Mirai primarily used direct non-reflective attacks. Even without relying on amplification strategies, Mirai was still able to inflict serious damage as shown by the high-profile attacks mentioned earlier.

The Mirai botnet shows that even a relatively simple dictionary attacks can still lead to the compromise of hundreds of thousands of IoT devices. In the future, it is likely that IoT devices will become the target of software vulnerabilities as well, which is why it is imperative vendors need to adapt a much stricter hardening strategy for their devices.

3.4.2. Hajime

Another notable botnet consisting of IoT devices is Hajime. Hajime was extensively analysed by Edwards and Profetis [10] in October 2016. Three stages are identified, namely:

Reconnaissance and infection phase The reconnaissance stage of Hajime runs over a Telnet session, very similar to other IoT malware discussed before. It scans the Internet using a randomization algorithm which generates a random IP address to which a connection is made. When a connection can be established on port 23, it goes through a list of credentials sequentially. This is a significant difference compared to Mirai and its variants, which attempts credentials in a random order. When a username/password combination is rejected, Hajime disconnects and then reconnects to attempt a different set of credentials. This way, only one combination is attempted per session while still going through a dictionary.

Once a set of credentials is accepted, a number of commands are executed to determine the suitability of the compromised device. A suitable location on the filesystem is selected and the system architecture is determined by inspecting the “echo” application. Once the target is deemed suitable for infection, a binary is uploaded by entering encoded bytes through the terminal session.

Downloader stub The binary downloaded at the end of the first stage is a simple program, aiming to connect to a remote server and store all received bytes in a file. The authors noticed that each architecture receives a different binary, handwritten specifically for that architecture. They arrive at this conclusion based on two indicators, which are characteristics not fitting to a compiler which can create executables for different architectures from the same source code.

DHT downloader The binary downloaded in the previous phase is the final stage of the Hajime worm. This binary is responsible for retrieving and executing additional payloads using a Peer-to-Peer (P2P) network, consisting of infected hosts. It uses several protocols used in BitTorrent. For example DHT protocol is used to discover other infected hosts, and the uTorrent Transport Protocol is used to exchange data over the network.

From the inner workings of the binary from the final stage, it appears that Hajime supports a number of different modules. Up until the point of publication, only the exploitation “exp” module was observed on the Internet. No other modules are currently active, though Hajime supports loading new and possibly malicious modules over its P2P network, still posing a serious threat. In Chapter 6, we will compare the findings from Edwards and Profetis [10] with observations done of our own.

From this chapter, we can conclude no implementation currently exists that meets all requirements. However, many of the concepts discussed in this chapter partially cover our requirements. From these concepts, combined with missing requirements, we designed Honeytrack, which will be discussed in Chapter 5.

4

Design Considerations

To design and implement an effective IoT honeypot requires predefined design parameters. This chapter describes which steps to take in order determine the optimal design parameters and deployment strategies for our honeypot. It involves a number of different phases, ranging from dataset exploration, network traffic analysis and passive data gathering. Before we can effectively deploy a honeypot as envisioned, there are a number of steps we can take to get an overview of the current exposure of Telnet.

4.1. Censys data analysis

As IoT botnets can cause a lot of damage, a lot of vulnerable devices must exist. Censys is a search engine that collects data on hosts and websites through daily scans of the Internet. Their dataset includes a scan on port 23, on which Telnet servers listen by default. Results include all publicly reachable devices on port 23 and lists the IP address, banner and negotiation components. This banner is the initial (textual) data a Telnet server returns, and is often used to identify the device, as well as present a login prompt for authentication.

From this dataset, we can establish an understanding of how many Telnet devices are actually exposed to the Internet. Additionally, we can extract common characteristics and identify the nature of such devices. As a honeypot aims to mimic a vulnerable device, this provides essential information for the implementation.

By analyzing this data, we can answer the questions “What sort of IoT devices are connected to the Internet” and “What are characteristics of IoT devices connected to the Internet”

4.2. TU Delft Telescope

After it is established what sort of IoT devices are connected to the Internet, we can investigate who is connecting to them. While the usage of a Telnet service is justified in a few cases, the devices which were used in the attacks mentioned in Chapter 1 likely had no critical dependency on Telnet. From various research done on one of the most comprehensive Telnet-based worms in recent history, Mirai, we know that malware spreads by scanning the entire Internet for potential targets [4]. If we want to capture infection attempts on a honeypot, we need to investigate where and how often attackers attempt to connect to these devices.

These scans can follow a number of different scenarios, each requiring a different strategy to achieve sufficient honeypot coverage.

Random When an attacker scans the internet in a random fashion, there is no pattern to be detected. When this scenario is the case, the distribution over a number of observed IP addresses will be a normal distribution for the number of hits. In this case, it does not matter in which part of a

network you set up your honeypots as they'll be hit with the same probability as other parts of the network. Therefore the number of hits is not dependent on the location of your honeypots.

Pseudo-random Another scenario is a the pseudo-random approach. Such an approach consists of a round-robin algorithm which initially looks random, but ensures every IP address is only hit once before hitting an IP address again. An example of a program using such an approach is zMap [9].

Targeted scan The final scenario we will consider is a targeted scan. In this case, an attacker is specifically targeting a range of IPs while ignoring the others. For this scenario, you would expect certain areas to be hit a lot more then others.

To draw conclusions on scanning traffic, we will analyze a large number of consecutive IP addresses. We have access to a large dataset consisting of all incoming network traffic of 3 partial /16 networks. From these networks, all traffic to unroutable IP addresses is forwarded to a network telescope. Over the course of 8 days, this came down to 1,704,422,219 incoming connections. Due to the large number of IP addresses present in the dataset, it provides a good indication of what we can expect in terms of incoming connections.

The three networks each have their own characteristics. The first two network, `130.161.0.0/16` and `131.180.0.0/16` are employee networks, consisting of workstations, servers and other relatively static devices. The third range at our disposal is `145.94.0.0/16`, which is the Eduroam network. Eduroam is a global university network, providing Internet access to students all around the world. This network consists of devices owned by students and employees, such as laptops and smartphones.

As we will see in Chapter 6.1, the Eduroam network is targeted 8% less than the other two networks available. The number of connections is normally distributed over all networks, indicating a random scanning scenario. For the design of our honeypot, this means that no special attention is required for the location of the honeypots in the networks.

4.3. Telnet metrics

When an attacker connects to a device, there are various methods of identification possible. Even before giving access to a shell, a lot can be derived from the connection setup and activity before any commands are executed.

The goal of a honeypot is to mimic a vulnerable Telnet device. This means we have all aspects of Telnet available as a metric to identify attackers. As discussed in Chapter 2.1, when a device establishes a Telnet connection, a Telnet-specific negotiation protocol is the first type of communication. This negotiation is part of the Telnet protocol specification and is used to set the parameters for the upcoming session. By controlling this negotiation, we can both set the terms for the Telnet session, and use the negotiation response as a metric for fingerprinting. Next to the negotiation, we can use other aspects of the connections at this point as well. Such aspects include IP address and Geo-location, but also timing data about the received bytes. A Telnet negotiation is done under the hood by Telnet clients, invisible to the user. However, when checked manually, a non-standard Telnet negotiation might the raise suspicion of an attacker. It should therefore be investigated if there is a difference in behavior when presented a non-standard Telnet negotiation.

Telnet negotiation allows for several possibilities for data gathering. If a connecting device is fully Telnet compliant, we can ask it to send data per character, the so-called `LINEMODE` option. This is meant for devices with very small input buffers, or devices that can't handle lines. While our honeypot is technically capable of accepting full lines, it is very useful to receive input per character, as this reveals a number of characteristics of the remote device. When an attack is automated, this will likely result in very low input lag. However, in the case of manual (human) input, the higher input delays will immediately stand out. Additionally, as `LINEMODE` is not a common negotiation parameter, we can obtain an indication on the Telnet-compliancy of the connecting device.

The next feature of Telnet negotiation which could be applied to our honeypot is `ECHO` Telnet option. The `ECHO` option determines whether or not the server is responsible for what is being displayed on

the screen, or if the connecting client keeps track of the terminal screen itself. Server-controlled echo is mostly used to hide the password input, emulating default SSH behavior. It is not as useful as `LINEMODE`, but the preference for server- or client side echo can still be used as a fingerprint property.

As Telnet is only a protocol specification, many different implementations exist on the different devices. Responses to Telnet negotiation options can therefore be answered in different ways, or not responded to at all. To make our honeypot as convincing as possible, the Telnet negotiation should consider and act on a number of common responses.

Considering the above, this gives us the following possibilities for attacker fingerprinting, just based on the initial connection establishment:

1. IP address
2. Geo-IP location
3. Time to first received byte
4. Input delay
5. Telnet Negotiation response

4.4. Interaction data

Depending on the level of clustering, the parameters described in the previous sections may not be sufficient to identify attackers. When a Telnet session is successfully established, the attacker is presented with a banner of a device we're emulating. The Censys dataset forms the source of the banners. The attacker is then presented with a login prompt, for the attacker to submit credentials.

As we're in full control of the Telnet session, this is not an actual login prompt. Instead, we gather all used credentials, and present the attacker with an error message stating the credentials are wrong. This way, we aim to collect the usernames and passwords used by attackers. We can use this data to form dictionaries for better classification of different types of attackers.

As a final step, the first set of commands executed can be used to determine the target's intention. As automated scripts often have easily identifiable characteristics, this should generate enough features to distinguish different malware variants.

4.5. Sandboxing

Honeypots are meant to be infected with malware. Executing these malicious files is often needed to investigate, it also poses great risk to the network. Malware can take control over an entire system, and even spread over the network if the infected host is not isolated properly. In order to analyze malware in a controlled environment, reducing the risk of unwanted spreading, there are several approaches possible.

First, we have virtual machines, for example implemented by VirtualBox of VMWare. Virtual machines are fully virtualized, including the hardware they run on. This allows for good isolation, as very little system resources are shared between the actual system and the virtual machine. Because all the hardware has to be virtualized and the machine runs completely separate from the host, this approach is resource heavy. Several virtual machines can run simultaneously on a single system, but resource usage and boot-times are comparable to actual systems. As adversaries expect to be attacking functioning running systems, requiring the allocation of both resources and time to boot virtual machines does not scale well. This can be partially overcome by having a cache of running systems that can be linked with incoming sessions, but having persistence throughout sessions becomes increasingly complex using this approach.

There are different methods of isolating processes from a host system. Containers are an approach that isolates processes from the host system, but without the full virtualization of all hardware. Instead,

containers share the host kernel and optionally have access to other parts of the system through the use of namespaces. This way, the applications running in a container have their own view of the hosts' resources and hardware. This greatly reduces both resource usage and boot-time for container-based systems. Well-known implementations of containers are Docker and Linux Containers (LXC).

An additional security feature of LXC is very useful when allowing unknown malicious activity on the container. LXC containers can be mapped to a permissionless user, meaning the processes running as root in the container, actually run as a non-root user on the host system. This means that when an LXC container is exploited and an adversary manages to escape the container environment, permissions on the host are limited to this user. This does not guarantee unwanted or critical damage to the system as various privilege-escalating attacks could still be possible, but it greatly reduces the risk of a compromised host.

4.6. Scaling high-interaction honeypots

Running a high-interaction honeypot is relatively much more expensive than a low- or medium interaction honeypot. The fact a real system needs to be behind one significantly increases the computing power to run one. When scaling up a honeypot network (sometimes called a honeynet), this become an increasingly large problem. Various research has been done in order to scale up high-interaction honeypots, with different approaches.

Bailey et al. [5] state that high-interaction honeypots are not feasible when deployed on a large scale. It is important to note that for this conclusion, it is assumed that high-interaction honeypots are implemented as fully virtualized machines. On the other hand, low-interaction honeypots can serve large IP ranges at once with relative ease. All IPv4 addresses are allocated, so large ranges of IP addresses on which to deploy honeypots on are not always easily available. Bailey et al. propose a framework that aims to solve both these issues. The scaling problem is addressed by combining low- and high-interaction honeypots and utilizing the best of both. Many lightweight low-interaction honeypots are deployed over a number of different ranges. These low-interaction honeypots serve as filters for a high-interaction setup. This high-interaction setup is a collection of centralized VMware hosts, running a wide variety of operating systems and applications.

The role of the low-interaction machines is to filter most of the traffic that is uninteresting or seen before. As they have the capability to respond to these requests themselves, there is no interaction with the high-interaction backend, saving a lot of resources. Based on several metrics such as connection state and payload hash, a low-interaction honeypot can hand over the connection to the high-interaction backend. This process is dynamic, so this decision is not only based on static rules, but accounts for different types of investigation goals, which we will get back to later.

The high-interaction machines use VMWare to provide several hosts per physical machine. All outgoing network traffic is prevented from directly going out, but instead is routed through the heavyweight honeypots. This traffic can be monitored to provide indicators of successful infections. The setup enables worms to propagate within the backend, as long as machines are available. This provides insights into the methods of spreading once a host has been infected. As soon as such an infection is detected, snapshots are created automatically, enabling analysis at a later point in time.

Both low- and high-interaction machines are monitored and controlled by a central control component. This component is responsible for data aggregation and analysis for all machines, so it is able to combine data from both the low- and high-interaction machines. This way, it has the possibility to present an attacker a previously infected machine to monitor possible secondary attacks.

Details on how this is achieved, or under which circumstances the control component decides to check for follow-up infections remain unclear. The experiments and results discussed in the paper mostly go into the filtering aspect and possible applications of the presented frameworks. Results show that low-interaction filtering is highly effective in terms of load reduction, while still achieving enough coverage to investigate attacker behavior. Their work serves as a good starting point to incorporate both scaling and persistence into a honeypot system, but lacks specifics on technical implementations and use cases for low-powered devices such as the IoT.

In work done by Guarnizo et al. [12], physical IoT devices are exposed to so-called wormholes. These wormholes are deployed across different IP ranges worldwide, similar to the lightweight honeypots as described by Bailey et al. [5]. Using a “forwarder” module, traffic is re-written in real time to make it look like the physical device is actually at the location of the wormhole. Multiple wormholes are connected to different ports on the IoT device, allowing multiple attackers to interact with the same physical device. As the devices are very likely to be compromised, they propose to periodically reset the devices and block any potential outgoing attacks to prevent collateral damage.

The authors estimate that by combining IoT devices with different wormholes, it is possible to expose $n * m$ services to the Internet, where n is the amount of physical IoT devices and m is the number of wormholes. While this does scale significantly better than binding a single device to an IP address, it still requires dozes of IoT devices to be set up. The proposed solution however, is suitable for IoT devices and allows for a more realistic setup than for example IoTPot [17].

So far, scaling up honeypot deployment was achieved by reducing the level of interaction of honeypots in order to reduce system load. Virtual machines allow for limited scaling per physical machines, but this does not scale beyond dozens of machines. Containers are suitable to isolate the different attackers from the system and each other, but so far using containers to run a large number of high-interaction honeypots has not been proposed.

4.7. Persistence

Maintaining the state of a honeypot after an attacker is disconnected could be very useful in tracking adversary activity over time. However, this functionality is absent in most currently available implementations.

Work done by Fairbanks et al. [11] proposes a method to store metadata from honeypots, for later analysis. While this can’t be considered persistence in the sense that attackers can return to their previously compromised machine, it does work towards the solution for the problem that a honeypot needs to be reset periodically. A method is proposed to extract and analyze a richer set of forensic information from the file system journal of honeypots in spite of anti-forensic tool use. This forensic information can then be used for system recovery, research on attack techniques, insight into attacker motives, and for criminal investigations.

The research of Bailey et al. [5] describes a form of persistence. When an intrusion is detected based on a number of metrics, the state of the virtual machine is saved with the checkpoint functionality. These checkpoints are for analysis purposes only, in order to investigate the exact changes made to the system.

While some work was done to enable persistence in honeypots, a scalable solution to store *and restore* machine state has not been proposed in past works. An alternative method previously unexplored, is the use of LXC containers to store machine state. LXC containers can be cloned in an efficient way. Where a clone virtual machine requires a full copy of the filesystem, a container clone uses the same root filesystem as the template. When changes are made, only these changes are saved to disk. This way, many containers can be created on a single machine without requiring a lot of space on the filesystem.

5

Honeytrack: Persistent IoT honeypot

For the implementation, we use the open source Honeytrap "Advanced Honeypot framework" [26]. Honeytrap is a modular framework for running, monitoring and managing honeypots. It aims to simplify the creation of custom honeypots, consisting of (possible) combinations of simple services, isolated containers or even real hosts. Using Honeytrap you can use sensors, high interaction and low interaction honeypots togethers, while still using the same event mechanisms for data collection. Honeytrap is a project under heavy development, but contained many of the features necessary for meeting the requirement set in earlier chapters. An overview of the requirements set in Chapter 3 can be found in Table 5.1.

Table 5.1: An overview of requirements for a persistent IoT honeypot.

Requirements
Scalable
Allow for extensive data collection
Configurable for A/B testing
Support for persistence

Honeytrap is both scalable in terms of architecture and implementation, but was lacking support for the Telnet protocol, commonly used for the IoT adversaries. Honeytrap is made up of several components, each responsible for a different phase of the handling of a connection. The connection enters the application through the *listener*. This *listener* accepts the incoming connection and attempts to match it to the next phase of Honeytrack, the *service*. A service is a module where the connections are handled based on a specific protocol or applications. Behavior such as reading and writing to the connection is done manually and hardcoded in the Honeytrap module. When a higher level of interaction is required, Honeytrap allows the connection to be passed on to an external application, such as a virtual machine running on the system. The module that takes care of this connection is called a *director*. Both *services* and *directors* can notify Honeytrap of important events. These events are all handled in a unified way, and can be passed on to various data collectors, through modules called *pushers*. A general overview of Honeytrap's design can be found in Figure 5.1. This chapter will discuss Honeytrap's design and architecture more in depth, as well as modifications necessary to meet the requirements.

5.1. Listener

At the entry point of Honeytrap is the *listener*. The *listener* accepts connections and forwards those connections to the appropriate *services* further down the pipeline. While Honeytrap can only run with one *listener*, there are several options possible. A schematic overview of the different types of *listeners* can be found in Figure 5.2.

Figure 5.1: Honeytrap Design overview

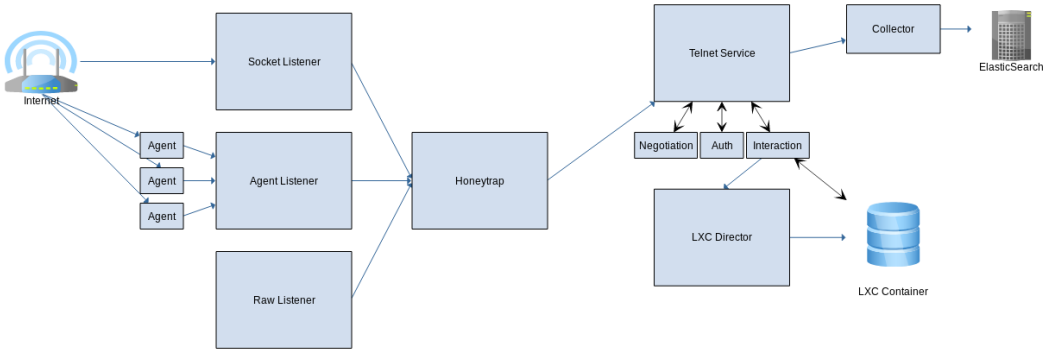
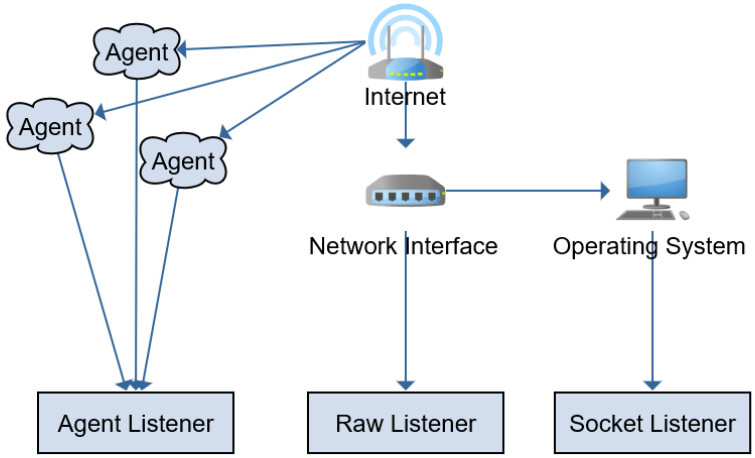


Figure 5.2: Honeytrap Listeners



Socket Listener The first listener is the socket listener. The socket listener accepts connections on all configured ports, but relies on the OS to create sockets to accept. This can be compared with a webserver listening on the default port 80. When a remote machine makes a connection to this server on port 80, this connection is picked up by the listening application which handles the request. The dependency on the operating for example means that it must recognize the incoming packets as its own. Simply sending packets to the server's network interface is not enough, as the destination IP address has to match the IP configured for the network interface. The Socket Listener is the default listener for Honeytrap, and enables support for all application layer communication between Honeytrap and remote hosts.

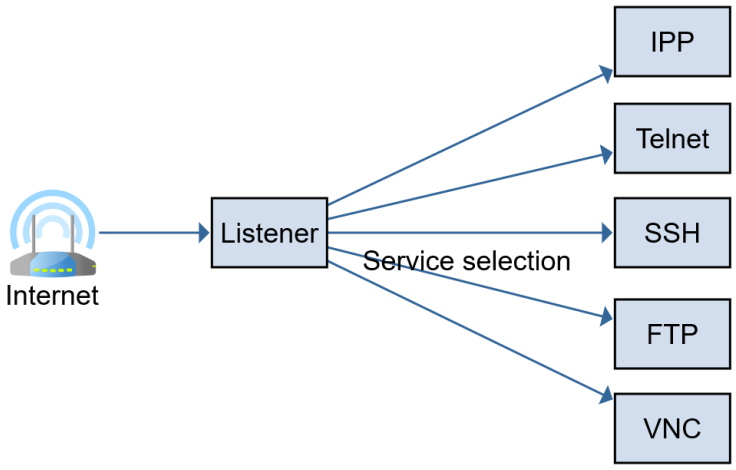
Agent Listener Instead of accepting connections on the machine running Honeytrap, the agent listener accepts connections which in turn are accepted by listeners on remote agents. Agents are devices running the Honeytrap Agent program, which forwards connection to a Honeytrap server. After an agent connects to the server, the configuration is retrieved and local (socket) listeners are started according to this configuration. This way, agent behavior can be controlled through a central server, turning Honeytrap into a distributed honeypot network. Connections accepted on

an agent listener are encrypted and sent to the server for processing, where they will go through the regular service matching flow as described above. This means that from the service component on, there is no difference between an agent connection and a socket connection. Data is also sent back over this connection, turning the agent into a man-in-the-middle probe. The device which is running the agent is not at risk of being compromised by itself. All login attempts and commands are simply forwarded to the central Honeytrap server, so the agent does not do any of the processing itself.

Raw Listener When the Raw listener is enabled, Honeytrap binds directly to the network interface. Where the socket listener relies on the operating system to turn incoming connections into sockets, the Raw Listener responds to all packets arriving at the network interface. This way, it can capture the first few bytes of data transmitted by a remote host. When a SYN packet is received, the Raw Listener is able to respond to it like a regular application, and follow-up packets are all confirmed with ACK packets. This way, the remote host gets confirmation packets are successfully received, tricking the remote host into thinking an actual application is listening on the other side. However, as no socket is created, application layer traffic is not possible so interaction is only possible on the Link Layer. This mode is very suitable to monitor all traffic sent to a server, but when a higher level of interaction is not required.

5.2. Services

Figure 5.3: Honeytrap Services



Every type of honeypot has it is own service, for example SSH, Telnet or a webserver. All those types require different responses to different requests, which is why each service is a separate module, capable of handling connections as if they were a SSH, Telnet or webserver. A service can be considered as the low-interaction component, where the connection is manually read, processed and responded to. Connections for each service come in through the central listener, so after a connection is accepted, Honeytrap has to determine for which service the connection was intended. A schematic overview of the relation between the listener and services can be found in Figure 5.3. The mapping of connections to a service can be done in multiple ways. The most common approach is to define a port on which the service listens. In the case of a Telnet service, made to handle Telnet traffic, this means all traffic on port 23 should be directed to this service. Services can also indicate if they are capable of handling a certain connection by looking at the first few bytes read from a connection. This could be used for

webservers, which can run on a variety of ports apart from the default port 80. Webservers typically receive HTTP requests, which have the format like `GET / HTTP/1.0`, where the first few bytes consist of `GET`. A service for a webserver can indicate that connections starting with `GET` should be handled by it, regardless on which port that connection was established.

Services allow full control over a connection, but all downsides of low-interaction honeypots discussed previously apply here. However, as the transition from a service to a director based connection can be done from the service, the best of both worlds can be combined. Currently, a wide variety of services is implemented such as FTP (File Transfer Protocol), IPP (Internet Printing Protocol), SSH, VNC (Remote Desktop) and ElasticSearch (Database). Honeytrap only supported very limited Telnet capabilities. The Telnet service developed for Honeytrack will be discussed later in Section 5.5.

5.3. Directors

When a higher level of interaction is required for a connection, a service can pass the connection on to an external application such as a Virtual Machine. A service can call a *director*, which is responsible for passing on this connection. A *director* calls an external program and establishes a network connection to be used by the Honeytrap service. For example, a connection can be established to an external remote device, to which the service can forward all received traffic, effectively becoming a man-in-the-middle. The same functionality currently exists for a Qemu virtual machine and an LXC container. For our implementation, a custom director is implemented with more refined control over the container selection process. This allows for better matching and grouping of attackers to the same containers. Details of this director will be discussed later in Section 5.5.

5.4. Data collection

Next to the *listeners*, *services* and *directors*, there are several utilities available for easy data collection. The event system is a centralized module where all Honeytrap components can send events to for central collection. Based on the configuration, these events are forwarded to various data collection modules. These modules, called *pushers* translate and send the event in a suitable format to their respective processing component. These components currently include basic output such as console and logfile, but also more advanced options such as Elasticsearch, Kafka, Splunk and Slack are available. Events can be filtered and directed to specific pushers, based on any information contained in the event. For instance the originating Honeytrap service, but also connection-specific details like destination IP address or origin Agent IP address.

Finally, it is possible to capture all network traffic on a given network interface. The sniffer can be started from any Honeytrap component, which allows for direct control over what is captured and when in the session flow. This interface can be the interface where sockets are created on the OS, but can also include the virtual interfaces made for each LXC container. This way, a full network capture can be generated for each session and processed directly, or it can be saved in a Packet Capture file (PCAP).

Due to the modularity of the system, all these components can be integrated or chained together, allowing for easy extension and detailed data connection. As Honeytrap is written in the Go language, it enables the use of powerful Go-routines[1]. These Go-routines are comparable to native OS threads, but much less resource intensive allowing many concurrent processes without losing performance. With each connection running in its own Go-routine, the software scales very well and can handle thousands of connections per hour without the need for extreme system specifications. This makes Honeytrap very suitable to achieve the requirements set in earlier chapters.

[1] `https://golang.org/ref/spec#Go_statements`

5.5. Honeytrack Additions

Where Honeytrap has most of the architectural requirements needed for the intended research goal, it needed some specific additions. For example the Telnet service was very basic, static and most importantly, only low-interaction. This section describes the modifications and additions made to Honeytrap to create Honeytrack.

5.5.1. Telnet Service

The Telnet service is the primary addition to Honeytrap. The existing service is very limited in terms of flexibility, and was low-interaction only.

As described in Chapter 2.1, a Telnet session typically consists of 3 different phases.

1. Negotiation
2. Authentication
3. Interaction

The Telnet service implemented for Honeytrack consists of different modules to support those phases. Each module allows full control over each phase, and can be independently configured for various experiments. After a Telnet phase is completed, the results are made available to the consecutive modules to account for variables set during those phases.

5.5.2. Negotiation Module

The first phase of a Telnet session is the Telnet negotiation. The technical details of the negotiation phase of Telnet are discussed in Chapter 2.1. In the negotiation module, attackers are presented with a custom negotiation challenge to test whether they connect using a proper Telnet client. The negotiation consists of two Telnet Option challenges, to which Honeytrack expects a valid response. When the response is valid and contains expected values, these values are stored for use later in the session. As Telnet implementations can vary, the response to the challenges posed by Honeytrack do not have to be valid for the session to continue. For example, an automated attack can be configured to have a fixed response to a negotiation, regardless of the challenges posed by a Telnet server. When a response does not meet the expected values, the session is marked as `raw`, so later analysis can be done to see what type of traffic was received anyway. A connection is only terminated in the negotiation phase when an attacker does not respond to the challenge within 5 seconds. As the negotiation is always automated, a failure to respond within 5 seconds is highly likely not to receive a response at all, wasting resources by waiting. From observation, we see that a response typically occurs within a couple hundred milliseconds. No sessions were observed where a response was recorded later than a response time of two seconds. After the negotiation phase, a banner is sent to the attacker. This banner is configurable and depends on the destination IP address to allow for experiments with different banners. Results from the negotiation module are saved and passed on to the next module, which is the authentication module.

5.5.3. Authentication Module

The authentication module is responsible for collecting credentials entered by attackers. It is a low-interaction login prompt service that consists of two states. In the first state, it expects a username. This isn't directly made clear with a `Username:` prompt, as this is the responsibility of the banner previously sent. As the module has access to the negotiation results, it can adhere to the established parameters, such as terminal echo. The module stores all input and looks for a newline character sequence. Once it observes this, all input so far is considered a username entry and moves to the second state, password collection. The first step is explicitly asking for a password with a prompt. The rest of the password collection state is identical to the username collecting, with the only difference being that characters are not echoed, no matter the negotiation value. When a newline character sequence is observed in the password collection phase, the entry is matched against a list of accepted credentials, which is set

through a configuration file. The configuration allows for both user- and root credentials, intended to be passed on to a director for correct handling. If a given entry does not match any of the pre-configured credentials, it notifies the user of wrong credentials, and goes back to the username collection state and start over. All entries, regardless of validity or correctness, are stored for later analysis. When the given entry matches one of the configured allowed credentials, the authentication is marked as successful and the phase ends. The authentication module is terminated when nothing is received after 30 seconds. Where automated scripts are expected to instantly start their brute-force attempts, humans might need a bit longer to enter credentials.

5.5.4. Interaction Module

When an attacker enters accepted credentials, the connection is passed on to the interaction module. Both low-interaction and high-interaction sessions are supported, depending on the availability of LXC containers and configuration. The difference between low- and high-interaction honeypots was discussed in Chapter 2. In low-interaction, Honeytrack presents the attacker with a fake prompt (`~#`), pretending a successful login. In a very similar fashion as the authentication phase, Honeytrack stores all input and looks for a newline character sequence. When a newline character sequence is detected, all input so far is processed by a string-matching algorithm to detect any known commands. These known commands are hardcoded and provide a fixed response when a command is matched. For example, when a command contains `/bin/busybox`, Honeytrack returns the first argument after `busybox` followed by `applet not found`. Several other strings were added in order to investigate common commands and the difference in behavior upon response.

If high-interaction is selected, the LXC Director is called to provide a connection to an LXC container. The implementation of this director is discussed later. The director returns a connection to an authenticated Telnet session on an LXC container. The interaction module reads from the connection with the attacker, stores all input, and passes the input directly to the container. This includes non-ascii characters such as the arrow-key bytes and other input supported by an actual Telnet session, increasing realism and predictability. A similar approach is used for the traffic coming from the container. Before the output of the container is passed to the attacker, it is first processed by a filter method. As a real LXC based system is hard to hide as being such, this filter function matches for strings known to expose the system as a honeypot. Such strings for example include lines containing `LXC` and `virtual`. Lines containing matching strings are stripped from the response, and not passed on to the attacker. While it is acknowledged this can lead to unexpected results on the adversaries side, it is an efficient way of covering some of the potential exposing characteristics of an LXC container. Output from the container is unknown and hard to predict. In order to prevent cluttering up the data collection with large and unnecessary output, traffic coming from the container is not stored using the event system. However, this data is still worth collecting. In order to log this traffic in a better way, we capture all traffic going over the LXC bridge adapter. The interaction module terminates the connection when the remote adversary closes the connection, or if no activity was observed for 60 seconds.

5.5.5. LXC Director

The LXC director is the component of Honeytrack that links the honeypot logic to LXC container management. It handles everything related to the LXC containers. Services can call the director, and it is expected to deliver a working connection to a container corresponding to the remote attacker. Binding with LXC is achieved through go-lxc bindings, enabling the director to perform any operation on the containers that is also possible through the native command line utility. As there is no direct control over these bindings, the director is designed to minimize interaction with the LXC bindings. Honeytrack is highly concurrent with every incoming connection being handled in (various) separate threads, the number of calls to the LXC system should be minimized to prevent concurrency issues and excessive load on the LXC control application. In order to reduce this load as much as possible, most LXC information is wrapped in a custom data structure, allowing the director to read and store information about a container without accessing the LXC subsystem. This information mostly regards name, IP address, idle time and other information that is unlikely to change from the container itself. The name of the container serves as a unique identifier and is made up of a hash from the various parameters

that make it unique. For example, if a container is requested for a network connection, both the source and destination IP address can be used as arguments for the hash function, resulting in a name unique for that combination. When a request for a container connection is received, the Director derives the name for the corresponding container first, and then checks if that container is already active. If this is the case, a new network connection is established with the already-running container without any additional changes. In most cases, a container is not active yet.

When the a container is created, a “housekeeper” routine is also started to monitor the LXC activity. This housekeeper checks for network activity coming from and to the container. If a container is idle for two minutes, meaning no network traffic whatsoever was observed, the container is shut down and stored for possible later use. When an attacker returns after having previously been connected to a container, the container is started, resulting in a connection to the container previously visited.

If a container does not exist yet, a new container is cloned from the configured template. The fresh container is then started and assigned an IP address by the DHCP server running on the host. This information is used to set up a new network connection to the container, and this connection is returned to the service for further handling.

5.5.6. Objectives

When this design is evaluated against the requirements discussed in Chapter 3, we can conclude Honeytrack overcomes the limitations existing honeypot implementations have.

1. Honeytrack has a hybrid architecture, leveraging the best of both worlds from low- and high-interaction honeypots. Only when an attacker successfully manages to log in to a honeypot, a high-interaction system is started.
2. Using LXC containers, it is possible to provide many attackers dedicated and separate high-interaction environments. These containers can be dynamically enabled and disabled, while saving state between different sessions enabling persistence across distinct sessions.
3. Honeytrack allows for an extensive distributed sensor network using the Agent listener, deployable on cheap hardware.
4. Honeytrack is able to analyse adversary behavior in detail, by directly controlling the Telnet negotiation and authentication phase, and by acting as man-in-the-middle for the high-interaction module.
5. Honeytrack can disguise itself from being a honeypot by filtering exposing lines coming from the high-interaction environment.
6. Honeytrack allows for extensive adversary analysis with several configurable variables, enabling A/B testing in both accepted credentials as high-interaction system images.

6

Evaluation

After several small experiments being conducted over the course of 6 months, the final experiments were conducted over the course of 2 months. During the first phase of the experiments, system performance was tweaked and a number of configurable variables were tested for viability and impact. With these experiments, we wanted to answer the following questions:

1) How do adversaries discover, scan and exploit potentially vulnerable devices?
    a) Can we scale honeypot deployment to gain insight in global attack behavior?
    b) What methods are used to discover devices?
    c) Can we identify groups or clusters of similar discovery methods?
    d) What are the used methods of exploitation and how do these methods relate to each other?
2) What do adversaries after successful exploitation, what are the Tactics, Techniques and Procedures (TTP)?
    a) How are devices suitable for infection selected and identified?
    b) How is malware installed?

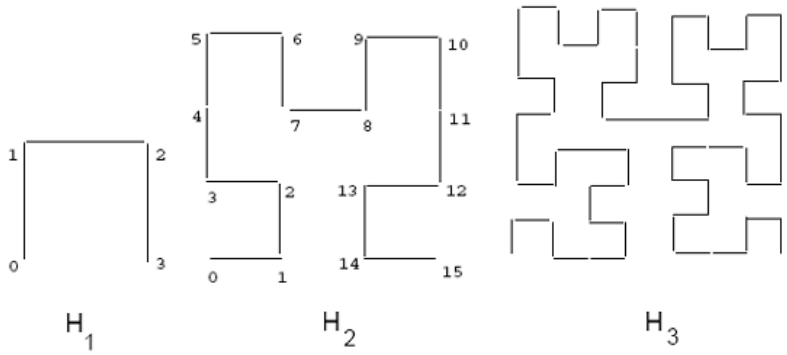
These questions are directly linked to the different phases of the killchain, which was discussed in Chapter 2.2. Each section addresses an aspect of the Telnet protocol, and links this to the killchain model. Section 6.1 and can be linked to the reconnaissance phase of the killchain model. There, scanning behavior and connection statistics provide insight into the methods adversaries use to discover potentially vulnerable devices. The exploitation phase can be found in Section 6.4. In that section, login attempts are discussed which are the primary method of exploitation observed on our honeypots. Going back two phases in the killchain, we find weaponization in Sections 6.6, where observations regarding device selection are discussed. Not all malware is suitable for each device. As part of the weaponization phase, a malware package is prepared for a certain type of device, which is selected by extracting a number of characteristics of the device. The final phases discussed in this chapter are the delivery and installation phases, which can be linked to Section 6.5. The last phases of the killchain, “Command & Control” and “Actions on Objectives” were out of scope for this thesis. However, with our honeypot we collected relevant data for analysis of these phases. Chapter 7 will contain a discussion of possible future work based on this collected data.

Throughout this chapter, the following terminology is used: When talking about *groups*, a cluster of similarly behaving malware is used. When talking about *variants*, a piece of malware is meant that is based on another piece of malware, or shares attack methods. In the broadest sense, all malware discussed in this chapter operates over Telnet, which classifies them all as variants of malware using this attack path. When talking about *actors*, the human actors behind the creation, spreading and/or exploitation of the malware is discussed.

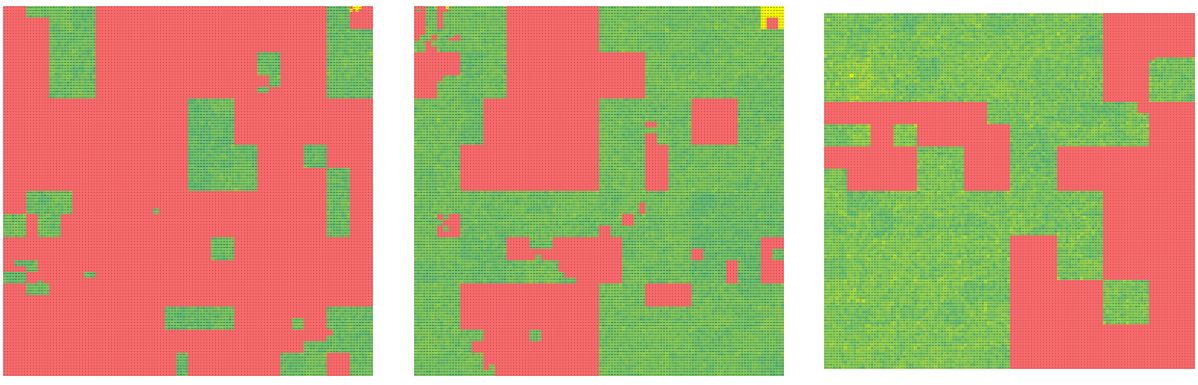
6.1. Honeypot Discovery

To discover how devices are being scanned on the Internet, we do not need data from the entire Internet to derive conclusions on scanning behavior. When data is collected from a large enough subnet, conclusions derived from it are statistically significant. As shown by Blenn et al. [6], data from a /17 network (consisting of 32,768 IP addresses) is already enough to get error margins down to below 2% when dealing with 20 Mbps attacks. For this section, we have data available from three distinct /16 ranges. The nature of these ranges are described in Chapter 4.2. To summarize, we use three /16 partially used networks where all traffic directed to unroutable IP addresses is collected and stored. To gain insight into scanning behavior on the IP ranges available to us, we are going to visualize activity in them. A suitable visualization for this purpose is the Hilbert curve, a space filling curve which fills up a square with 1D data. When applied on IP addresses, subnets form coherent blocks, as consecutive IPs will translate a compact region of the map. Figure 6.1 shows the first three iterations of such a mapping.

Figure 6.1: The first three orders of a Hilbert curve.



For our visualization, we use a Hilbert curve that creates a coordinate system with 128^2 possible coordinates, on which we map the 2^{16} possible addresses. Each of these coordinates represents 4 distinct consecutive IP addresses in their respective range. By counting each incoming connection on these addresses, we get a visual representation of network activity and the spread across the networks.



(a) 130.161.0.0/16 (b) 131.180.0.0/16 (c) 145.94.0.0/16, Eduroam

Figure 6.2: The Hilbert visualization of the three available ranges. The number of connections are mapped to a color gradient from red to green. Red indicates no connections, green indicate the highest number of connections.

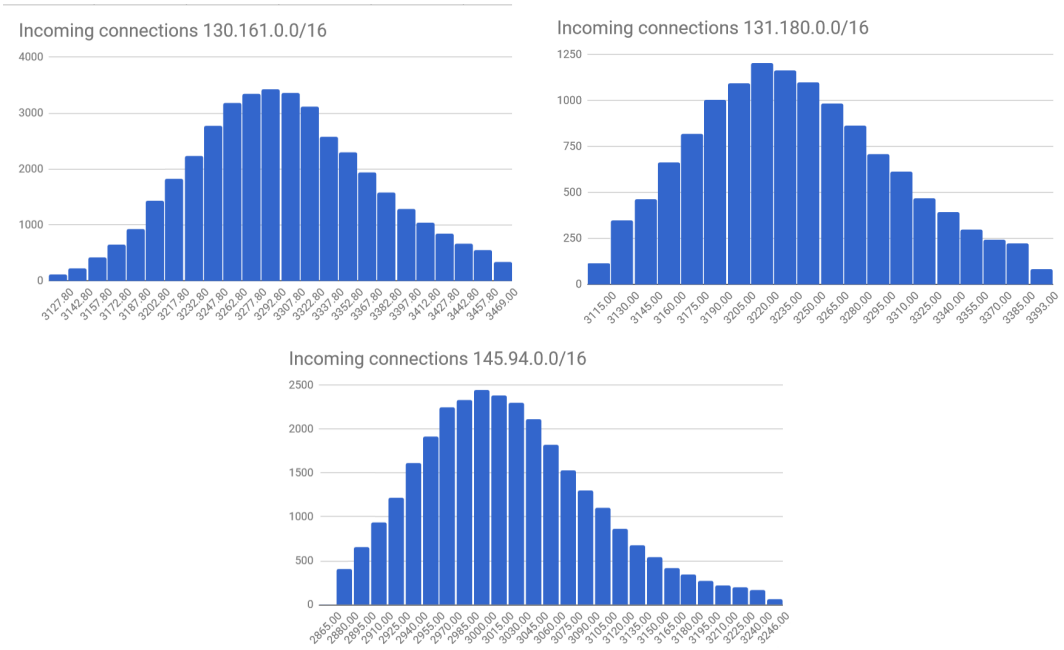
The resulting Hilbert curve for each of the 3 IP ranges can be found in Figure 6.2. The number of connections counted on each square of 4 consecutive IP addresses is mapped to a red-to-green color gradient. From these visualizations, we can conclude that the spread of the scanning activity is evenly distributed. The size of the green areas is not of importance in this case, as this only indicates the availability of the IP addresses. While there is some variation between the blocks, there are no areas

Table 6.1: The median and average values for each of the separate ranges

Range	130.161.0.0/16	131.180.0.0/16	145.94.0.0/16
Median	3291	3230	3007
Average	3203.03	3243.34	3082.87

that show anomalies in the amount of data received. The blocks with used IP addresses (and therefore not present in this data set) can be very clearly distinguished by the red areas.

Figure 6.3: The number of incoming connections show a normal distribution for all measured ranges. See Table 6.1 for the median and average values for these ranges.



From the same data, we also make histograms of the number of connections observed on each IP address. By looking at the distribution of the number of incoming SYN packets per IP address, we can see a normal distribution. As discussed in Chapter 4.2, this corresponds to the random scanning scenario where scanning behavior is random and not targeted at any specific IP (range). However, at the time of collection, there was no response to incoming SYN packets for these ranges. When we compare this distribution with incoming connections on the honeypot, we can determine if there is a difference between an active and inactive IP address.

Our honeypots were deployed across these three distinct ranges with different characteristics. For one range, there is a notable difference from the rest of the ranges. This shows from the average and median values (Table 6.1), both of which are 8% lower for the 145.94.0.0/16 range than the other ranges.

6.1.1. Honeytrack deployment & strategies

The 4 machines used for Honeytrack were each assigned between 1000 and 2600 addresses from IP ranges described above. Next to the central gathering in the TU Delft IP ranges, various agents were deployed at Dutch consumer IP addresses, as well as at Google and Microsoft Cloud instances.

One of the aspects to investigate is the public IP address on which the honeypots listen. For the setup, honeypots are deployed using 3 strategies:

1. Small non-consecutive subnets

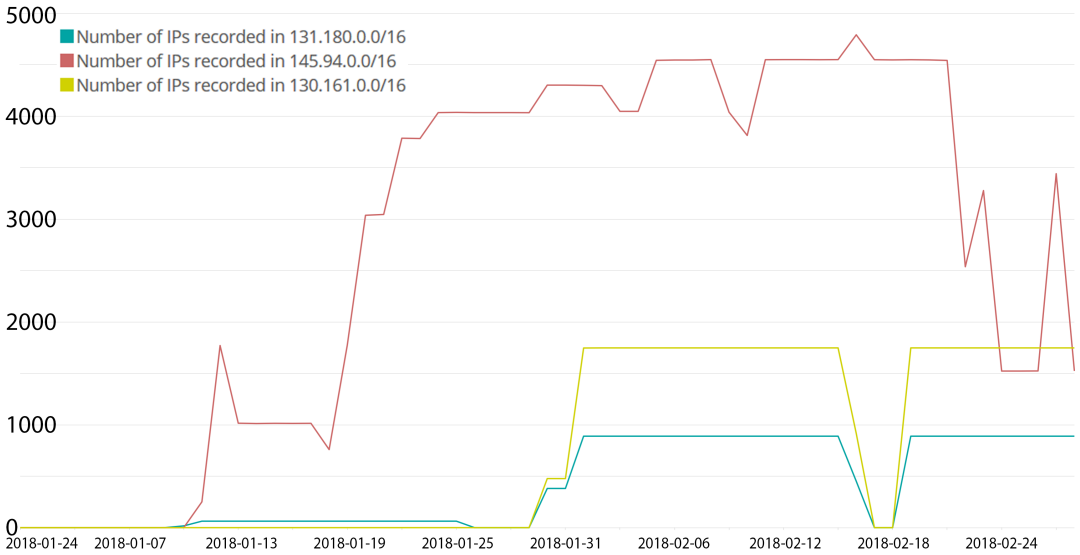
2. Large consecutive subnet
3. Distributed, non-consecutive addresses using agents

The ranges used for the first two are discussed in Chapter 4.2. All servers were assigned between 2600 and 1000 IP addresses, following different strategies in deployment. Two servers were deployed using the first strategy, one with the second, and the fourth server was set up as a server for the agents following the third strategy. As we had no control over the routing of these IP addresses, minor changes in the number of active IP addresses were possible due to IP addresses being activated and deactivated, removing them from the scope. Table 6.2 shows an overview of IP distribution at the peak operating times.

Table 6.2: The number of IP addresses active at peak operation

Server	Listening IPs	Strategy
Mimas	2,636	Consecutive
Ganymede	2,283	Consecutive
Io	1,769	Consecutive
Europa	1,015	Low-interaction
Callisto	77	Agents

Figure 6.4: The number of IP addresses monitored, split by range. The sudden drops were the result of technical difficulties.



IP addresses were added over time, as can be seen from Figure 6.4. During peak operations, slightly more than 7800 IP addresses had an active listener on port 23 combined over all servers. The agents were deployed on various ranges, reaching from Cloud providers such as Google, Microsoft and Amazon, to Dutch consumer IP addresses of the major Internet Service Providers in the Netherlands. As part of a Masters' course, students deployed agents on their home IP addresses for at least 3 days.

6.1.2. Connection statistics

Before analyzing the Telnet sessions, we start by comparing the differences between the three ranges in terms of connections.

When we look at hits over time, a few observations can be made as seen in Figure 6.5. The 145.94.0.0/16 range is mostly uncorrelated with the other two ranges, except for a peak around January 13th. Both other ranges show close correlation, both in terms of changes in activity as well as the number of sessions per IP address. As can be seen from the Figure, IP addresses in the 145.94.0.0/16 receive a

significantly larger amount of sessions compared to the other ranges. This does not correspond to the statistics obtained in Section 6.1. This could be explained by changes in network usage, which caused it to become more active between the time of the exploration data and the moment the honeypots were deployed. Additionally, the 145.94.0.0/16 range is more dynamic in nature than the other ranges. IP addresses are removed and added back to the monitored pool as demand requires it.

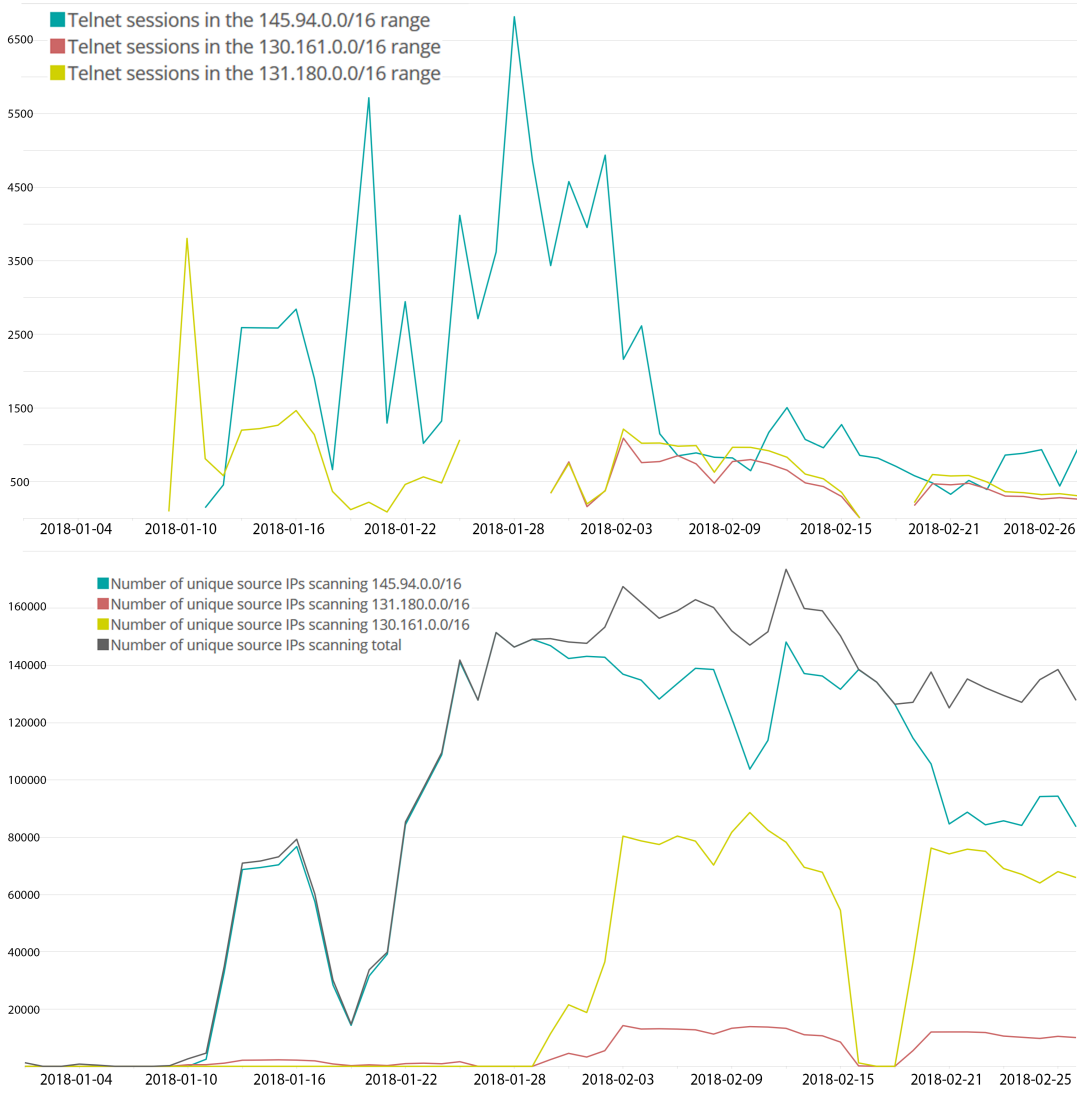


Figure 6.5: Number of hits per destination IP address, split over the three distinct ranges.

6.2. Telnet Negotiation

Scanning is done in most cases by sending TCP SYN packets. Such a packet indicates to the receiving client that the remote server would like to set up a connection. When a response to this SYN packet is received by the client, it indicates something is listening on the server side. Upon a successful TCP handshake, a session is established. The first thing Honeytrack sends to the remote host, are the Telnet negotiation bytes. This negotiation phase of the Telnet protocol is discussed in Chapter 2.1.

All honeypots were configured to check for two *option* codes. The first being the well known and often used `ECHO` *option* code, indicating whether or not the Telnet server should mirror any input received from the client. The second *option* code is the `LINEMODE` *option* code. This *option* code is less common, and provides the means to fingerprint based on Telnet compliance level, as well as if the `LINEMODE`

feature is actually correctly implemented.

From this negotiation phase, we can already make a distinction between attackers, based on the response to this initial request. If a remote host supports some form of negotiation, we expect a valid response to the Telnet options, starting with the `IAC` byte. If the first response to Honeytrack's request does not correspond to a valid negotiation response, we can conclude the remote host does not support the Telnet protocol. This comes down to if the first byte being read from the connection isn't the `IAC` byte, which is the only way a Telnet response (or request) can start. Honeytrack proceeds to send the banner after an invalid negotiation is detected, in order to capture any payload that the remote host might send, regardless of the negotiation. This is possible because the Telnet protocol is a thin layer on top of the TCP protocol. If the Telnet negotiation phase is not supported, follow-up communication is handled through the standard TCP protocol. From this first byte, the inter-arrival time between bytes is measured. As a TCP packet can contain more than one byte, bytes with an inter-arrival time of 0 are sent using the same packet, or without any delay at all.

6.2.1. Invalid Telnet responses

From all Telnet sessions recorded, 11.3% had an invalid Telnet negotiation response. If we look at TCP sessions with an invalid response to Honeytrack's negotiation request, we can make some interesting observations. Often, the remote host does not wait for an initial request from Honeytrack, but starts sending it is payload right after a TCP session has been established. If we interpret the incoming bytes as ASCII characters, the first category of payloads can be identified as HTTP requests. Even though HTTP servers commonly run on port 80 and port 23 is the default for Telnet servers, we still see a large number of HTTP requests with various complexity. Out of 208 million sessions, 117,518 sessions have an invalid Telnet negotiation and contain "HTTP" as part of their payload. These sessions are distributed across 1103 unique IP addresses. Many of these IP addresses belong to the same actors, as we observe clusters of consecutive IP addresses with a similar amount of requests, using a very similar payload. As IPs being scanned more than once by a combined range are not observed, it is likely they are operating as a distributed Internet scanner. Some requests contain the bare minimal (`GET / HTTP/1.1`), while others contain various headers, such as User-agents, Cookies, Host or specific paths. An overview of the most common payloads containing `HTTP` can be found in Table 6.3.

Table 6.3: HTTP Requests on port 23

Payload	Percentage
GET / HTTP/1.1	88%
GET / HTTP/1.0	5%
GET / HTTP/1.0: Host: [IP]:23	3%

Based on the information visible in the User-agent header, this behavior matches well known scanners like zGrab.

Another notable presence is a request for a SIP service. SIP, which stands for Session Initiation Protocol, is commonly used for media transfer and Internet telephony. Various SIP exploits have been published in the past, so the SIP related requests are likely part of an effort to discover potentially vulnerable devices. Unlike the HTTP requests, *all* SIP requests are the same, and match the signature for an NMap SIP version scan.

When we look at remote IP addresses who send these requests, we see that these hosts never have sessions with both valid and invalid negotiations. A single IP address does not necessarily scan the Internet with a single request. Different requests can be observed coming from the same remote IP. The same local IP address is hit with various requests from the same remote IP address as well, but not in consistent patterns. Likely, the used software has some sort of randomization algorithm to prevent easy detection or network congestion.

Next to fully invalid Telnet negotiation responses, 1,915,778 (0.9%) sessions were recorded with partial Telnet negotiations. These negotiations did start with the 255 `IAC` byte as specified in the Telnet protocol, but did not include a valid response to both the `ECHO` and `LINEMODE` requests. Due to the way

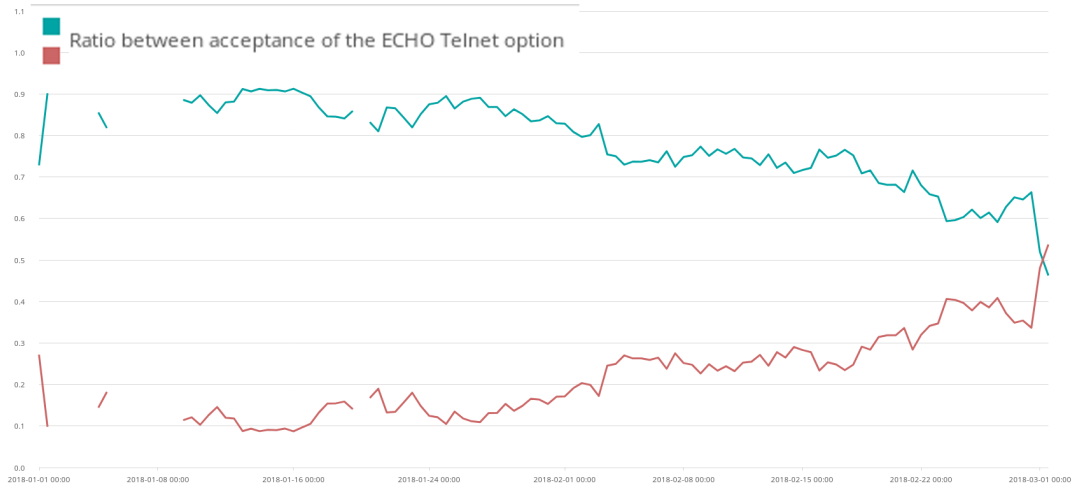
Honeytrack determines an invalid negotiation, these sessions were timed out, as a valid response was expected after a partially correct response. As a result it is not possible to determine the differences between partially valid and fully valid responses.

When looking into the source IPs behind these partially invalid responses, we find some inconsistent behavior. For the top 10 remote IP addresses showing this behavior, other sessions recorded coming from these IPs show different types of negotiation responses.

6.2.2. Valid Telnet responses

The majority (with 88.7%) of Telnet sessions recorded have a valid response to the Telnet negotiation. We consider a negotiation valid when a reply is received for both the `ECHO` and `LINEMODE` options, regardless of what this reply entails. Honeytrack checks for two Telnet *options* during this negotiation. The first option is the `ECHO` option, a very common option, which is widely supported in many Telnet implementations. The second option is `LINEMODE`, which, when enabled, tells the client to send each character separately instead of per line.

Figure 6.6: Ratio between Telnet option ECHO acceptance. Over time, the number of sessions with a positive response to the ECHO option converges to the number of sessions with a negative response.



When looking at the `ECHO` option, we see that the response to this option changes over time as can be seen in Figure 6.6. As we will see later in this chapter, malware variants often respond the same to the Telnet negotiation, so this diverging trend could indicate a dominance in variants employed. From the sessions where `ECHO` is set to false, the overwhelming majority comes from a single variant using a single credential entry per session. This strategy will be discussed later in Section 6.4. During the observation period, only a couple of dozen sessions were recorded with `ECHO` set to false, and not using the single-password strategy.

By itself, the support of the `ECHO` option does not say much about the capabilities of an attacker. However, as there are differences in response, this is still useful to divide attackers up into clusters, finding connections that would have otherwise been overlooked. In the following sections, the effect of an enabled or disabled `ECHO` option will be discussed compared to other aspects of the Telnet session.

From all recorded sessions, only 61 sessions supported the `LINEMODE` option. The `LINEMODE` option proves to be a very uncommon option, as a very small percentage of incoming clients reply positively to this negotiation request. Additionally, sessions with enabled `LINEMODE` respond with a more extensive negotiation reply than just the bare minimum, as observed with the majority of the request. The reply contains a number of sub-negotiations, which correspond to a negotiation. A noteworthy observation is that debug sessions, done with both Windows, Linux and Mac Telnet clients on default sessions supported `LINEMODE` out of the box. Based on these observations, we can hypothesis that a Telnet negotiation with `LINEMODE` enabled is a good classifier for human interaction. A nice side-effect of a client supporting `LINEMODE`, is that each character is transmitted to Honeytrack as it is typed. More on

Table 6.4: Banners used for Honeytrack honeypots

	Banner
1	RICOH Maintenance Shell. User access verification. login:
2	MikroTik v6.35.4 (stable) Login:
3	Welcome Visiting Huawei Home Gateway Copyright by Huawei Technologies Co., Ltd. Login:
4	Remote Management Console login:
5	Welcome to Microsoft Telnet Service login:
6	********** Welcome to ZXR10 Carrier-class High-end Routing Switch of ZTE Corporation **********
7	User Access Verification Username:
8	This is a honeypot, please don't log in

Table 6.5: Differences in credentials entered per banner

	With credentials	Without credentials	Percentage with credentials
Control Banner	772,238	18,336,482	4%
Other Banner	145,161,384	22,339,230	86%

how this feature can be used to classify human behavior is discussed in Chapter 6.4.

6.3. Telnet Banners

One of the A/B tests performed on the honeypot system is a number of different banners presented upon successful Telnet negotiation. As there is a large variety of devices with exposed Telnet servers, we would like to observe how attackers behave on these devices, and if different devices are attacked in a different manner. From the Censys Dataset, 8 different banners were selected that cover a wide array of different types of devices. An overview of these banners can be found in Table 6.4. Next to the 8 common banners, a control banner was added. This control banner stated "This is a honeypot, please don't log in", and was added to investigate whether attackers pay attention to non-standard banners.

From the Censys data set, we can already make a number of observations regarding the type of devices connected to the Internet. The banners often clearly indicate what device type it is, and even what version of the software is currently running. Given this information, one could expect that attackers craft their attacks based on specific vulnerabilities found in these devices.

The different banners were equally distributed amongst the IP ranges assigned to Honeytrack, with a maximum deviation of 2.7% in number of sessions recorded. Even though the banners indicated a wide range of different devices, all attackers show identical login behavior for all banners, except the control banner. For the control banner, which clearly states the device is a honeypot, we see a major difference in the number of sessions where credentials are entered. For honeypots where a banner was transmitted, 145,161,384 had entered credentials, and 23,339,230 did not. On average, when the session had the control banner displayed, 18,336,482 sessions did not have any entered credentials, and only 772,238 with credentials (see Table 6.5). Additionally, we observe a sharp decrease in the amount of sessions that enter credentials after February 12th. This is an interesting observation, as no changes were made to the setup of Honeytrack at this time, and the control banner is the only one affected. Overnight, the number of sessions with entered credentials drop from 10,000's per day, to about 200 to 100 attempts per day. When we filter only for valid Telnet negotiations, only 6 sessions were recorded since February 13th, *in total*. In that same period, over 4.1 million connections were established with a fully valid Telnet negotiation to IPs with another banner than the control banner.

As can be seen from Table 6.4, the banners are formatted in various ways and end in different ways. For example, both "Username" and "Login" were used as prompt, both with and without a newline. The fact that our control banner is rejected by almost all adversaries indicates a form of blacklisting on the transmitted banner. To fully determine whether black- or whitelist behavior is the used method, further research is required. As the control banner was the only banner that was not based on an existing banner, we can not derive a definitive conclusion about this.

6.4. Credentials

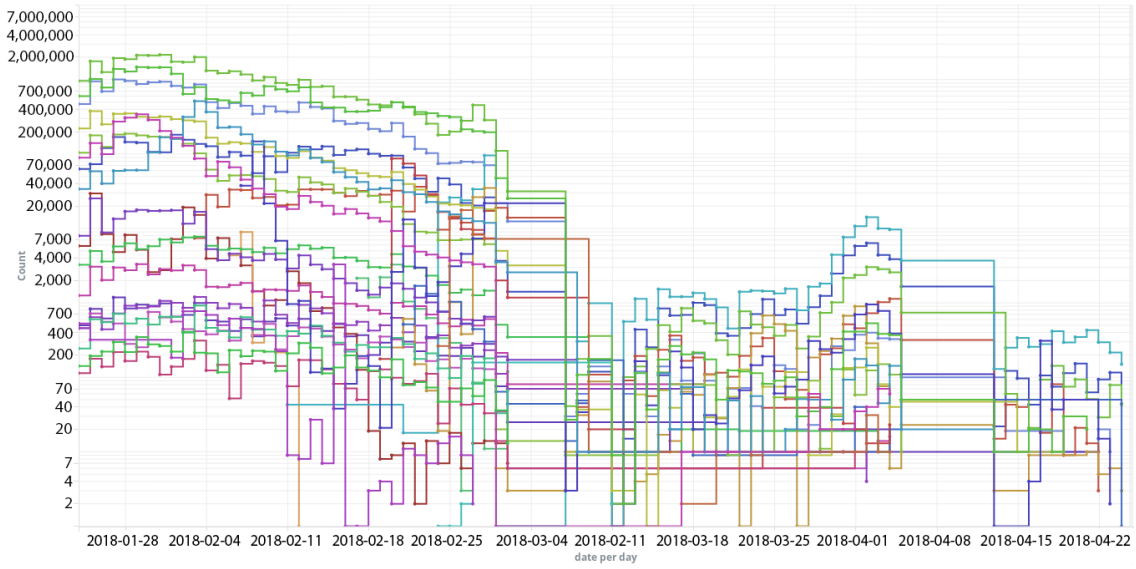
After the banner is presented to the connecting host, a response is expected in the form of login credentials. These credentials are the first non-automated payload coming from adversaries, after the Telnet negotiation phase. From the credentials, credential lists can be derived of commonly used credentials and patterns between different groups of malware can be made.

For the authentication phase, Honeytrack allows for 2 parameters to be configured. The first parameter is a list of accepted credentials, indicating whether or not an adversary is presented with an interactive shell. The list of accepted credentials was different for each of the three servers, but with some overlap for commonly used credentials. From various known botnets, malware analysis and vulnerability reports, a list of 6 passwords were chosen which the honeypot would accept. When selecting the credentials, the goal was to get a wide range of possible adversary access. Malware like Mirai has a known dictionary, but we're interested in more than just Mirai.

Before looking at the actual used credentials, we can already make an observation regarding the way credentials are entered. A large malware family opens 5 to 6 parallel sessions at once, and in each session a single password/username combination is entered. Instead of waiting for feedback whether the credentials are accepted, it immediately enters a series of bash commands. This family is, or closely related to, the Mirai botnet. This can be derived from the bash command `/bin/busybox MIRAI`, which will be discussed later in this chapter. From this command, we can extract an identifier, `MIRAI` in this case, which we will call the Busybox identifier. This string varies per malware group and can be used for clustering sessions. Mirai isn't the only variant using this strategy. Many variants can be derived from these bash commands, with `MIORI` and `daddyl33t` standing out as very active groups, with `XWIFZ` and `MASUTA` notable, though less active.

In total, we identified 33 unique Busybox identifiers. Not all single-password strategy variant were constantly active at any given time. Where the active variants like `MIRAI` and `daddyl33t` have a constant presence, the smaller variants show more distinct peak activity. A timeline of the different variants is shown in Figure 6.7. From this timeline, we can see that there's a large variance in attack volume for each variant.

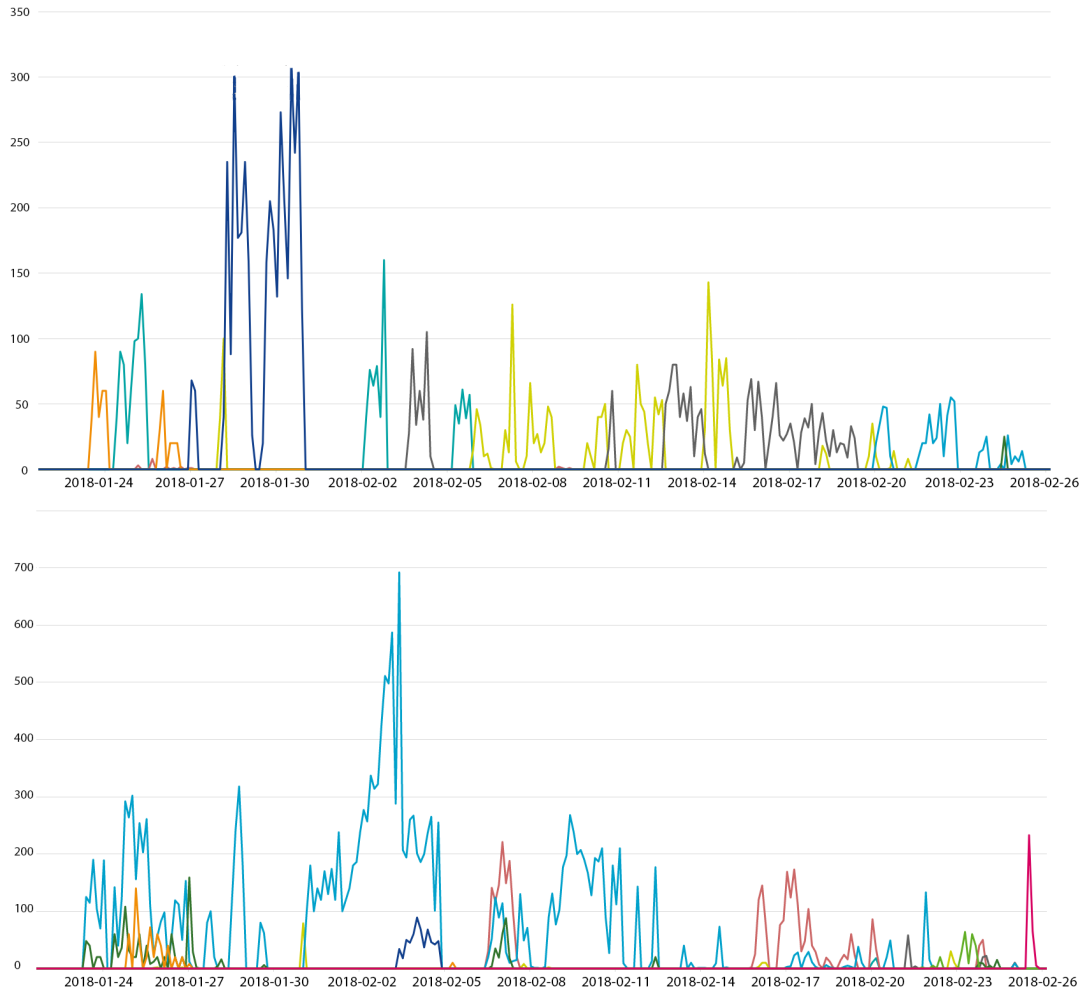
Figure 6.7: Activity of the different variants over time. The logarithmic Y-axis shows the number of sessions recorded containing a certain variant, represented by the individual colors.



A single source IP is not bound to a single variant. Of the 2,675,389 unique source IP addresses observed during the experiment, 186,439 used the single-password strategy. Of those IPs, 6,632 showed more than one Busybox identifier over time, where the others have only been observed with one identifier. The maximum number of variants for a single IP address is no less than 10 different variants over time. When we plot variant activity of the most active IP addresses, we see two distinct

scenarios. For the most time, different variants are active on the same IP at the same time. However, at a different point in time, we see the variants clearly alternating. This alternating behavior is also observed on more IPs with more then 4 different variants during the experiment.

Figure 6.8: Activity of the different variants from two distinct source IPs. Each color indicates activity of a different variant. In the two scenarios below, the first shows only consecutive activity while the second shows an overlap in activity from different variants.



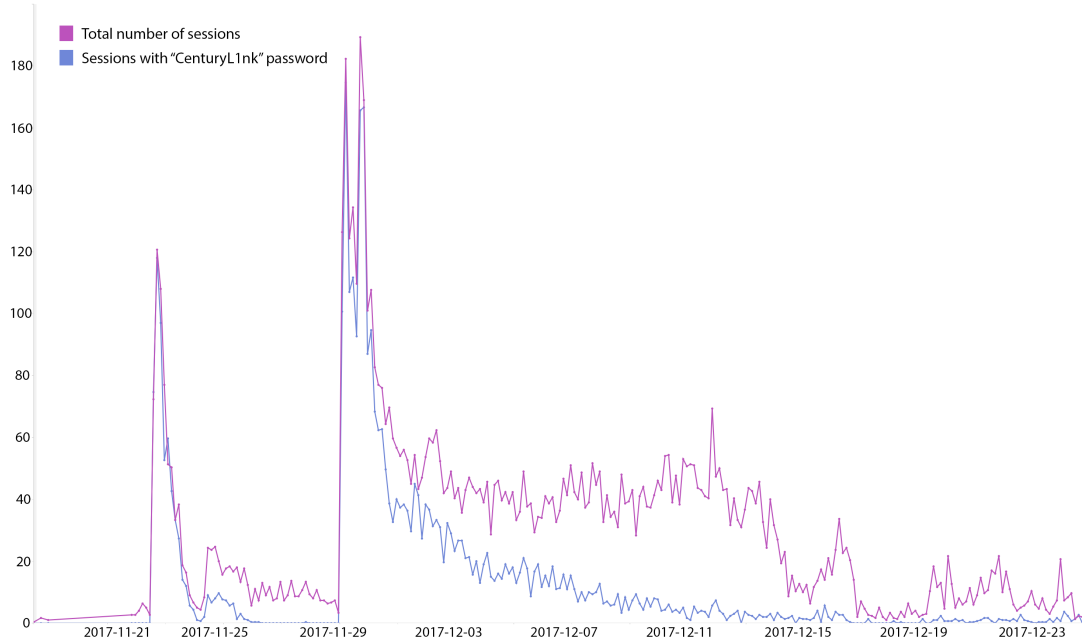
In Figure 6.8, we plot the activity of the different variants for two IPs showing the two scenarios discussed above. The first scenario can be explained by the fact that a public IP address may have different devices behind a firewall or proxy server. The outgoing route of those devices may be shared, much like Honeytrack's setup. Another possibility is that a device is infected by multiple malware variants at once. Previous work showed that some malware removes rival malware from a device before installing, but it is possible not all variants do this.

After the source code of Mirai was published online, anyone with sufficient programming knowledge could create their own variant, adding credentials to the dictionary at will. The other major groups, `MIORI` and `daddyl33t`, have significantly smaller dictionaries than Mirai, but the overlap is not large enough to classify them as a single actor. Between these groups or even amongst devices infected by the same actor, there does not seem to be any communication about correct credentials on a certain IP address. Over time, Honeytrack accepted different credentials. However, at times where these credentials were introduced, there wasn't a significant increase in attempts using those passwords. What is however notable, is that actors are updating their dictionary when a new vulnerability is disclosed. For example, when the existence of a backdoor account in certain ZyXEL modems[1] was published,

[1] https://www.exploit-db.com/exploits/43105/

this gave rise to a new Mirai variant using credentials specific to this exploit[2]. Shortly after publishing, an enormous increase of roughly 800% in sessions per hour was recorded in Honeytrack, using this username/password combination.

Figure 6.9: Usage of the CenturyL1nk password greatly increased after an exploit was published. The top line shows the total number of sessions over time, with the bottom line showing the number of sessions where the CenturyL1nk password was used.



Not all sessions with a single password contain a busybox identifier. 7.1% of the recorded sessions contain a single password, followed by the typical `enable` command, but do not proceed with a busybox identifier, as seen by for example Mirai (which uses `/bin/busybox MIRAI` after a single password attempt).

6.4.1. Multi-password sessions

A second group has a more sophisticated method of entering passwords. Instead of blindly entering credentials and an immediate follow-up bash command, this group supports entering multiple credentials during the same session. By looking at sessions with more than one attempt, effectively ignoring the group mentioned earlier, we still observe a large overlap in dictionaries. As these variants can not be identified using a Busybox identifier, and a large overlap exists between dictionaries, we need to look at the interaction phase of the session in order to identify variants of this type. These will be discussed later in Section 6.5.

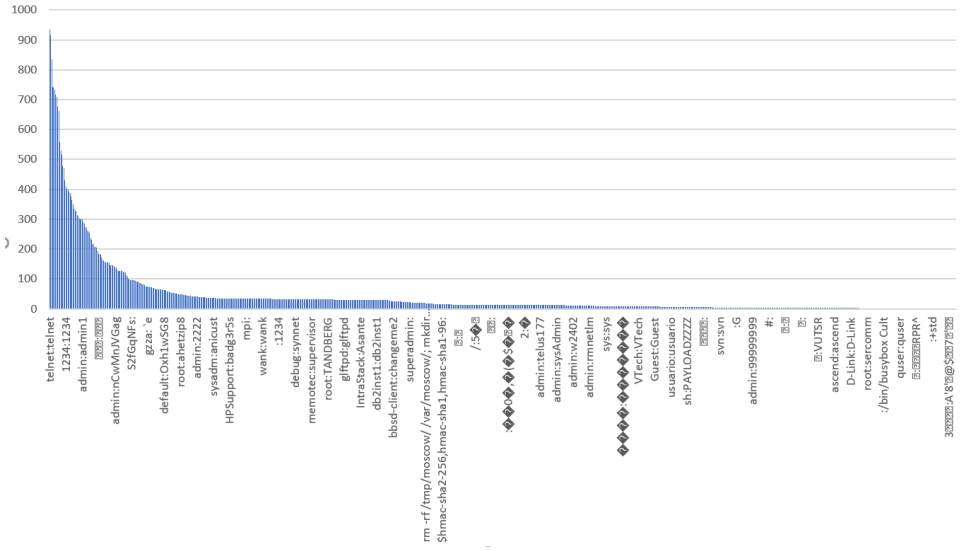
Looking at the other side of the spectrum, we can observe credentials used only a handful of times, recorded over millions of sessions. Many of these entries are part of a dictionary of a single piece of malware, identified by `/bin/busybox OBJPRN` as they're using the single-credential-per-session strategy.

6.4.2. Dictionaries

From previous work, we know malware has fixed dictionaries of credentials from which it attempts either a fixed number, or attempts them all. From the groups of malware identified in previous sections, it can be observed that many credentials are shared between them. When looking at an overview of most used credentials, we can identify a number of credentials are much more often used than others.

[2]https://www.trendmicro.com/vinfo/us/security/news/Internet-of-things/new-Mirai-variant-found-spreading-like-wildfire

Figure 6.10: Distribution of the used credentials. From this plot we can derive there are no clearly distinguishable dictionaries belonging to just one variant.



From Figure 6.10 we can see the popularity does not show any clustering by itself. Another method of clustering is looking at combinations of credentials used per session. As discussed in the previous section, different password entry strategies exist. These strategies indicate a different group of malware anyway, so we analyze single-password sessions separately.

When we combine all passwords used by the single-password strategy and group them by Busybox identifier, we can determine several dictionaries used by these variants. From the 33 busybox identifiers discussed previously, the side of the dictionary varies greatly. The smallest dictionary contained a single entry, while the largest contained 446 unique username/password combinations. This largest dictionary belongs to the `MIRAI| Busybox` identifier. Previous analysis of Mirai concluded that the original Mirai malware only has a dictionary of 62 credentials. As this does not match with our findings, it is likely this dictionary actually consists of multiple variants which all use the MIRAI identifier. From the credential entries alone however, it is difficult to derive sub-dictionaries. When looking at the distinct credentials used for these groups, it is easy to observe there is a large overlap between dictionaries.

In Table B.1 and B.2 in the Appendix, we compare the similarity between all variants. Each column header shows the overlap with the variant named in its respective row header. Rows containing a large number of high-percentage values indicate a relatively small dictionary, with credentials often found in other variants. On the other hand, columns containing many low-percentage values indicate a variant with a largely unique dictionary. Examples of the latter are variants with Busybox identifier `ORION`, `Word`, `ZEAZO` and `ECCHI`. Some of these variants simply have a very small dictionary consisting of 5 or less unique credentials which results in the high percentages. Next to relative overlap, an interesting observation can also be made regarding cross-variant overlap. When looking at the number of unique credentials per variant, we see a different picture than the individual overlap between variants. If the uniqueness is considered for all variants combined, the number of unique entries per variant drops drastically. It seems dictionaries consist of subsets of the “global” dictionary with a few optional additions, but there does not seem to be a relation between these subsets. It is possible that the choice of credentials is related to the targeted devices, but our experimental setup did not contain enough device specific configuration to make this type of distinction.

From the interview with the author of both the `OWARI` and `SORA` variants [3], we know the two are from the same actor. However, when comparing the dictionary of both variants, the relation between the two is not obvious. Both variants have their own unique entries the other does not have and the number of entries is not the same either. `OWARI` has been observed using 30 unique username/password combination, where `SORA` has a dictionary of 44 credentials.

6.4.3. Input delays

Honeytrack records the delay between each incoming character during the authentication phase. This feature relies mostly on the correct implementation of the `LINEMODE` option on the remote host. When `LINEMODE` is not enabled or incorrectly implemented, characters are only transmitted by line. In this case, the delay between characters will be 0, even if the characters are typed manually. However, when `LINEMODE` is enabled, and correctly implemented, the remote host will send each character separately. By recording the time between these characters, we can measure the typing speed between sessions. A clear distinction can be made between two groups with valid enabled `LINEMODE` options. The first group uses an automated script to enter commands. Delay between characters are likely caused by TCP delays.

The second group shows a significantly bigger delay for each character, as well as a bigger variance between each measurement. Where automated scripts have a small variance within 10 milliseconds, the other groups shows a variance an order of magnitude bigger. What also stands out for this second group, is the delay before the first character is received. Automated scripts have a very small or no delay at all, while the second group often takes one to four seconds before input is transmitted. When comparing these values to manual debug sessions, we see similar values for both initial and between-character delays.

So far we covered the first 4 phases of the killchain model. Adversaries scan the entire Internet looking for vulnerable devices. The presented banner is of hardly any influence, except for an apparent blacklist for certain keywords like “honeypot”. By using automated scripts, which attempt a number of known default credentials, they’re able to weaponize these security weaknesses. Delivery is trivial, as these devices are directly accessible on the Internet. By simply connecting to the device, the attack can be carried out directly and immediately. Once the correct credentials have been found, an attacker has direct access to the device and execute arbitrary code.

6.4.4. Misconfigured hosts

As the credentials phase records all input coming from an attacker, several hosts can be observed where existing software was used in a misconfigured way. A notable mistake by attackers running the a variant of Mirai, is that the full output of the executable is sent to the victim (our honeypot). In the Mirai source code, various debug statements are present to indicate the status of certain operations such as scanning, credential input and other phases of the attack path. These statements can be identified by the component responsible for the action, which is printed between brackets. For example, debug statements regarding operations of the scanner component are shown as `[scanner]` in Mirai’s execution window. However, due to a misconfiguration, these debug statements are also sent to the victim, resulting in a full debug log entered during the credentials phase.

This variant has the busybox identifier SORA and 16 sessions with this behavior were observed coming from different source IPs. However, next to these misconfigured sessions, regular single-password sessions coming from the same IPs are also observed during the same time frame. The debug log session can be directly linked to the other sessions running in parallel, with the logs showing the debug output for entries found in another session. This could indicate that the SORA variant supports a form of remote debug logging to a system under the attacker’s control. Additionally, it is shown that connections nearly always time out, instead of SORA recognizing a failed attempt instead.

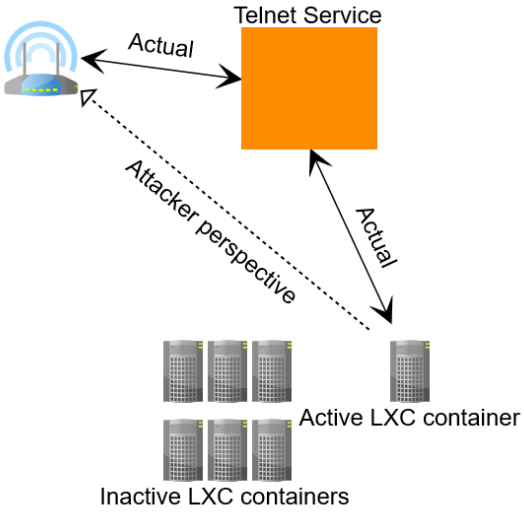
Honeytrack is designed to verify the entered credentials after a newline character is observed and a password is expected. Due to the way the single-password strategy works, the last entry (often containing the Busybox identifier) is considered a username. Where SORA expects a response to its input, Honeytrack expects a password, resulting in a timed-out connection. This time-out behavior SORA displays is also observed with other variants using the single-password strategy. Therefore it is likely other variants with this strategy are affected by the same situation.

6.5. Installation

The next step in the kill chain is “Installation”. The device needs to be prepared to serve the adversary’s goals. This often happens by installing customized malware on the device, making it part of a larger network containing more infected devices. In this section, the next phases of the killchain are discussed, namely “Installation” and “Command & Control”.

When an adversary enters accepted credentials, an LXC container is started and the connection is patched through to the container. Network traffic is directly forwarded by Honeytrack, effectively becoming a Man-in-the-Middle (MitM), see Figure 6.11 for a schematic overview of this setup. This way, the connection has all the features one can expect from a Telnet terminal, but is still observable by Honeytrack. In the case of an Agent listener (Chapter 5.2), this also means that any malicious activity only occurs on the Honeytrack server. An advantage of that approach is that users can run this safely from their home network without risking any breaches as a result from running the agent.

Figure 6.11: Overview of Honeytrack acting as a Man-in-the-Middle between the LXC container and an attacker. The dotted line is the perceived situation from the attacker’s perspective.



Based on the commands entered in an LXC session, it becomes easier to distinguish groups of adversaries. Most attack paths were shared up until this point in order to exploit the device. From the list of selected credentials accepted by Honeytrack, it is possible to select the level of access the adversary gains on the container. For each server, a different set of credentials was selected that would give root access to the interaction phase. Combined with the other configurable options, this provided the means to investigate differences in behavior if an attacker had root access from the start.

For the experiments described in this chapter, LXC version 2.1.1 was used, in combination with LXCFS 2.0.8 and libseccomp 2.3.1. The template for the LXC containers was based on the Ubuntu Xenial image. Two users were made available, one with user privileges and one with root privileges. Next do the default programs available in the LXC Ubuntu image, the following programs were installed:

Busybox Busybox is a program that provides a number of common Unix tools, from a single executable. It is designed to operate in embedded environments where processing power is scarce, to give basic functionality to a device without providing a full Unix environment. As Telnet devices usually fall in that category, Busybox is an easy and reliable way to get access to useful Unix tools.

Wget & cURL To download files from the Internet, Wget and cURL are two common Unix tools often used for this task. Both tools are often used for the same task, but cURL supports a huge amount of file transfer protocols, where Wget only supports HTTP, HTTPS, FTP and FTPS.

TFTP The Trivial File Transfer Protocol, better known as TFTP, is an application used for file transfers. It uses very few system resources, making it suitable for embedded systems with low amounts of memory. TFTP does not support authentication or encryption, but is very easy to implement, which makes it a popular tool.

TelnetD TelnetD is the program that enables Telnet access to a machine. For TelnetD to work as desired for Honeytrack, settings were adjusted to allow direct root login, something which is disabled by default as a security precaution. In order to present an adversary with a device as neutral as possible, Message-of-the-Day (MotD) display on successful login was disabled. The MotD is shown upon login and often shows some information about the current state of the system, such as resource usage, system temperatures and available updates. In addition to that, the $PS1 environment variable was adjusted to not display the hostname and username. This was done to initially hide the fact that the username of the current user may not correspond with the credentials offered.

LXC containers are running in unprivileged mode, meaning that the root user in the container, was mapped to an unprivileged user on the host system. In case an adversary discovers the environment is actually a container, it might be possible to break out of this secure environment due to security flaws in LXC. However, should this occur, the attacker only has the rights of an unprivileged user, instead of the administrator privileges LXC is running on. As it is possible our honeypots will be infected with real malware, the possibility of actively participating in attacks is not excluded. In order to prevent attackers from taking full advantage of the full TU Delft infrastructure, containers had a combined restricted bandwidth of 20480 bits down and 8096 bits up. While this does not eliminate it, this restriction greatly reduces the possible impact an attacker has, while still maintaining a level of realism. As described in Chapter 3.1.1, a common approach to detect honeypots is to verify it is able to conduct real attacks.

6.6. Device selection

Once an attacker has successfully logged in, it does not instantly start to download malware. As there are many devices vulnerable to its attacks, it needs to determine what sort of device its dealing with in order to determine further steps. Additionally, its possible that an attacker is accounting for possible honeypots, so steps need to be taken in order to verify its a legitimate target. This section describes a number of strategies observed that deal with initial exploration and device detection. The download and execution of malware will be discussed in Section 6.7.

6.6.1. Single-password strategy

The first installation strategy to stand out is actually one we've seen before during the authentication phase. This strategy consisted of concurrently entering credentials in different sessions, followed by a bash command directly. When the entered credentials weren't accepted, the attacker would get a "Wrong password, please try again" message after every second newline. However, when the single username/password combination tried in that session was one Honeytrack accepted, these commands were are actually executed in an LXC environment.

For the most common example, Mirai, the command calls on the Busybox program, which is described above. When a command is not recognized, for example a tool is called that does not exist, Busybox returns `applet not found`. In the case of the bash command Mirai executes, `/bin/busybox Mirai` will result in `Mirai: applet not found`. A large group using a similar strategy never went further than this first command. Identifying strings include `MIORI` and `daddyl33t`. This could indicate that the response wasn't what Mirai expected or that there were other factors leading to an abort of the attack.

From one of the interviews discussed in Chapter 1.2, we learned that malware authors test their own

malware with honeypots themselves. One of the honeypots used is “telnet-iot-honeypot”[3]. The source code contains a number of analysis from common attacks as well, providing insight into behavior of a number of variants in other honeypots. From these analyses, as well as observations made by Minn et al. [17], we learn that Mirai consists of two separate components responsible for infection. From the single-password strategy sessions, we observe behavior that closely resembles both. However, due to the large scale of deployment and limited number of functional credentials, it proves very difficult to directly match the reconnaissance phase to the infection phase by a second host.

6.6.2. Braces strategy

Another strategy often observed in Honeytrack is far more sophisticated than plain Busybox calls. At the start of this strategy lies a command in the form of `cat /proc/mounts; (/bin/busybox OSEWX || :)|`. While this may seem like a single command, it actually consists of a number of distinct sub-commands and seems to be designed around fingerprinting the system it is run on. In order to understand the sophistication behind this command better, we break it down in separate pieces.

The first part, `cat /proc/mounts`, is a command that shows the content of the “mounts” file. “cat” is a command available in any Unix environment, and also one of the commands present in the Busybox program which was discussed earlier. While originally intended to concatenate files, it is often used for it is ability to simply print the contents of a file. In this case, it is used to print the contents of the “mounts file”. This file contains information about the various file systems mounted on the device. Next to physical hard drives, this also includes temporary file systems and information regarding Linux cgroups. As this information is quite detailed and highly dependent on the device, it provides a good insight in the type of device it is run on.

Right after the first part, we find a `;`. This semicolon indicates to execute whatever comes after it, right after the part that comes before it. This way, you can combine multiple commands with a single line of input.

The part of the command behind the semicolon, `(/bin/busybox OSEWX || :)|`, is actually a statement evaluation rather than a command. This command can again be split up in two parts. The first part `/bin/busybox OSEWX` is equivalent to the command we’ve seen before. Instead of an identifying string observed before such as `MIRAI` or `MIORI`, this strategy uses random 5-character strings. This makes it much more difficult to identify specific groups, and adds an extra challenge for possible low-interaction systems in recognizing this command. The command is followed by ||, which is the logic operator for XOR. If the command before it fails, it executes the command that comes after it. For the example described above, this means that if the Busybox command fails with a non-zero exit code, : is executed instead. The colon character is specified by POSIX as a shorthand for “true”. This completes the statement between the brackets and turns it into a complete logic check. In practice, it is a complicated way to check if Busybox is present on the system.

Due to the way it is designed, it is likely the command was intended to fool honeypot or other automated command parsing systems. A common method of executing commands in a safe environment employed by honeypots and sandboxes, is by the use of a “Execute” function. Many implementations, such as PHP, LXC and Docker take a string array as input, representing the command and its arguments. This works quite well for most commands, but without correct parsing, does not work for logic comparisons as described above. The implementation used by the LXC Execute function does not accept the command unless fully parsed, making it difficult to realistically execute this function. This could be solved by implementing an advanced command parser, but that would defeat the purpose of running the commands in an container environment.

Adversaries using this strategy are very common. In total, 4.1 million sessions were recorded following this strategy, out of 6.1 million sessions where commands were executed. Adversaries using this strategy proved to be tough to convince that Honeytrack was a legitimate target when running in an LXC environment. Initially, the attack stopped after executing this command in the container environment. Based on the output of `/proc/mounts`, it is easy to discover that the environment is LXC-based and being used as a honeypot. Based on this behavior, it was hypothesized that the contents of the

[3] `https://github.com/Phype/telnet-iot-honeypot`

mounts file was likely the cause of these aborted attacks. To test this hypothesis, 3 experiments were conducted to investigate this behavior. For the first experiment, Honeytrack was configured to operate in low-interaction mode for a range of IP addresses in the 145.94.0.0/16 range.

A modified version of the Honeytrack low-interaction module was configured to recognize the command explained above. When this command was detected, a string was returned, emulating the actual execution of the cat command. This string was based on the mounts file of a freshly installed Raspberry Pi running Raspbian, which can be seen in Listing 6.1.

Listing 6.1: The mounts file of a freshly installed Raspberry Pi

```
sysfs /sys sysfs rw,nosuid,nodev,noexec,relatime 0 0
proc /proc proc rw,nosuid,nodev,noexec,relatime 0 0
udev /dev devtmpfs rw,nosuid,relatime,size=243480k,nr_inodes=60870,mode=755 0 0
devpts /dev/pts devpts rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000 0 0
tmpfs /run tmpfs rw,nosuid,noexec,relatime,size=50952k,mode=755 0 0
/dev/mmcblk0p2 / ext4 rw,relatime,data=ordered 0 0
tmpfs /dev/shm tmpfs rw,nosuid,nodev 0 0
tmpfs /run/lock tmpfs rw,nosuid,nodev,noexec,relatime,size=5120k 0 0
tmpfs /sys/fs/cgroup tmpfs ro,nosuid,nodev,noexec,mode=755 0 0
cgroup /sys/fs/cgroup/systemd cgroup rw,nosuid,nodev,noexec,relatime,xattr,release_agent=/lib
    ↪ /systemd/systemd-cgroups-agent,name=systemd 0 0
cgroup /sys/fs/cgroup/net_cls cgroup rw,nosuid,nodev,noexec,relatime,net_cls 0 0
mqueue /dev/mqueue mqueue rw,relatime 0 0
debugfs /sys/kernel/debug debugfs rw,relatime 0 0
configfs /sys/kernel/config configfs rw,relatime 0 0
fusectl /sys/fs/fuse/connections fusectl rw,relatime 0 0
/dev/mmcblk0p1 /boot vfat rw,relatime,fmask=0022,dmask=0022,codepage=437,iocharset=ascii,
    ↪ shortname=mixed,utf8,errors=remount-ro 0 0
tmpfs /run/user/1000 tmpfs rw,nosuid,nodev,relatime,size=50952k,mode=700,uid=1000,gid=1000 0
    ↪ 0
tmpfs /run/user/0 tmpfs rw,nosuid,nodev,relatime,size=50952k,mode=700 0 0
```

This mounts file does not contain any references to LXC or honeypot activity as it is an actual system suitable for infection, if an attacker would connect to such a device. When an attacker enters accepted credentials, they're routed to an emulated environment with very basic support for a few commands, instead of a full LXC environment. This approach proved to be more successful in convincing the attacker to continue, as 100% of the sessions recorded using this strategy, proceeded with the second batch of commands. However, due to the very limited amount of commands supported by the low-interaction module, the attack still did not result in an attempted malware download. Still, a significant number of commands were executed before the attack was aborted, giving us valuable insight in the follow-up strategy of this attacker. The next sequence of commands is as follows:

```
cd /dev/shm; cat .s || cp /bin/echo .s; (/bin/busybox UBWTS || :)
tftp; wget; (/bin/busybox UBWTS || :)
dd bs=52 count=1 if=.s || cat .s || while read i; do echo \$i; done < .s
(/bin/busybox UBWTS || :)
tmpfs /run/user/0 tmpfs rw,nosuid,nodev,relatime,size=50952k,mode=700 0 0
```

These follow-up commands use similar techniques as used before. On the first line, the attacker changes the working directory to `/dev/shm`, which is the filesystem that uses virtual memory. All files created in this directory exists while the device is running, but will be removed when it is rebooted. It will not leave any trace, as these files completely reside in memory. In this directory, a copy of the `echo` program is copied to a new file `.s`. This very simple program simply prints the given input to the standard output. For example `echo "Hello World"` will print "Hello World" to the standard output. The line is followed by another check if Busybox exists, equal to the command used before.

In the second line, the presence of two applications is tested. tftp and wget are both common applications to download files from a remote ftp host or webserver. Simply executing these commands in this form does not actually download anything, but the output can be easily parsed for relevant information regarding the existence of these programs.

The third line contains three commands with very similar output. The dd command prints the first 52 bytes of the ".s" file. If that command succeeds, the whole line terminates there. However, when that command exits with a non-zero exit code, the next command is executed. "cat .s" will print the contents

of the ".s" file in total. Finally, if the cat command fails, the third option achieves a similar goal, resulting in an equal output as the cat command.

The point behind this command sequence seems to be testing the suitability of the infected device. If all commands execute as expected, this indicates the current user has rights to both read and write to the ramdisk, making it suitable for stealthy infection. With either tftp or wget available, the device enables the download of the malware required for the actual infection. Due to the way these commands are formatted and executed, it filters a significant amount of honeypot systems, as it is a challenge to execute these commands in a generic way.

Now that it is confirmed that a different output for the mounts file results in different behavior for this attack, we can conclude this strategy uses either a whitelist or a blacklist on the content. To test this hypothesis, the LXC output filter was configured to prevent any revealing lines from the mounts file from getting to the adversary. As Honeytrack acts like a man-in-the-middle, traffic from the container can easily be manipulated before forwarding the traffic to the remote host. Lines from the mounts file have a very characteristic pattern and can be easily recognized by the LXC filter. All lines referencing LXC, virtualized filesystems or non-default security measures were intercepted and filtered from the forwarding one by one. Instead of filtering all lines at once, this was done one at a time in an attempt to discover what lines of the mounts file were responsible for the abort. While this strategy was aimed at the braces strategy, it also affected all other variants which depended on the mounts file. The impact on other strategies will be discussed in Section 6.6.5. As this strategy included changing the working directory to the ramdisk, all entries regarding this, as well as other writable filesystems, remained untouched. Even with a minimal mounts file, not containing any reference to LXC whatsoever, the attack aborted after the first command. This supports the hypothesis that this strategy does not work by blacklist, but rather uses a whitelist to scan for suitable filesystems.

For the third experiment, the aim was to replicate the out from the low-interaction module on the high-interaction module completely; in the LXC container environment. As mentioned before, the mounts file of the LXC container environment contains a lot of information, giving away that the user is in such an environment. For this experiment, all output from the mounts file was replaced by the same lines as used in the low-interaction module, as described in the first experiment using the low-interaction module. The result of applying the filter was that the `cat /proc/mounts` command printed the contents of the Raspberry Pi mounts file, instead of the LXC container file. Despite this equal output, the attack does not continue on after the first command when executed in the high-interaction module. At this point it remains unclear what triggers the abort.

6.6.3. Hajime family

Next to Mirai, there is another well known IoT worm currently active on the Internet. This worm was briefly discussed in Chapter 1.1. Hajime shares several characteristics with strategies discussed in this chapter, but has different triggers to abort an attack. Hajime is very active, as Honeytrack recorded 580,000 sessions using the characteristic `ECCHI` string as Busybox parameter, checking its availability. The follow-up commands are listed in Listing 6.2. Other groups have different triggers, or rely less on the ouput of the initial command. This strategy included an initial command with an identifying string, but followed up immediately with commands such as the following:

Listing 6.2: The command sequence of the Hajime family

```
1  enable
2  shell
3  sh
4  /bin/busybox ECCHI
5  /bin/busybox ps; /bin/busybox ECCHI
6  /bin/busybox kill -9 2713
7  /bin/busybox kill -9 2714
8  /bin/busybox kill -9 2715
9  /bin/busybox kill -9 2717
10 /bin/busybox kill -9 2718
11 /bin/busybox kill -9 2719
12 /bin/busybox cat /proc/mounts; /bin/busybox ECCHI
```

The nature of these commands are similar to the ones used in the braces strategy. Besides the commands discussed before such as `cat /proc/mounts`, Hajime also calls the `ps` command. `ps` is another tool supported by Busybox, and lists all processes running on a system. After running the ps commands, Hajime attempts to kill a number processes using the `pkill` command. The `pkill` command uses ProcessIDs, or PIDs as input, which uniquely identify a running proces on a system. The PIDs used by Hajime vary widely per session with over 8000 different PIDs used. Howver, these are not based on the previous commands as the plan `ps` command does not contain any information about the PIDs of processes running on the system. A relation between used PIDs and output from previous commands could not be found. In addition to the PID, the number of kill commands per session also varies from a single command to six per session. These PIDs are incremental per pair, for example:

```
1 /bin/busybox kill -9 591, /bin/busybox kill -9 592
2 /bin/busybox kill -9 673, /bin/busybox kill -9 674
```

However, these pairs do not seem related to other pairs. For some sessions, the attack ends after the kill command, where in other sessions a Busybox identifier command is executed afterwards and a final scenario consists of multiple file checks as described below. These differences did not correlate with differences in Honeytrack responses, used credentials or other measured metrics.

Similar to the braces strategy, Hajime checks certain filesystems for accessibility. A major difference here is that each series of command is applied to every line of the mounts file. Additionally, not every command sequence is equal for every session.

Hajime has already been extensively analyzed by Edwards and Profetis [10]. When we compare their findings with observations made with Honeytrack, we can conclude that the original Hajime as analyzed by Edwards and Profetis is still present. However, there are a number of variants which initially look like Hajime, but either use a different Busybox identifier or use the `ECCHI` identifier but show different behavior.

6.6.4. Other strategies

Aside from the strategies discussed above, Honeytrack observed dozens of sessions using slightly different variations.

`/bin/busybox echo -e '\x6b\x61\x6d\x69/dev' > /dev/.nippon;` followed by `/bin/busybox cat /dev/.nippon; /bin/busybox rm /dev/.nippon` occurs in several variations, based on the mounts file which was discussed previously. This command attempts to create a file containing “kami/dev” at a number of locations. The escaped bytes `\x6b \x61 \x6d \x69` are part of a device compatibility check, as not all shells correctly process escaped byte input. Next to the escaped bytes, the file also contains the path it is currently being written to. If the file is being written to “/run/lock/”, it would attempt to write “kami/run/lock”. In this case, the file is written to the “/dev” directory, but is always part of a series of commands iterating over the entries of a mounts file. This behaviour is observed in multiple variants, including ones identied by their busybox identifiers `daddyl33t` and `ECCHI`. Additionally, variants using the single-password strategy, but do not include a Busybox identifier are also observed using similar commands. The iteration over the different mounts file entries is the same, as well as the structure of the commands. However, the escaped bytes are different for these variants. From observations found with existing honeypots, this behaviour is originally attributed to the Hajime worm[4], but the daddyl33t Busybox identifier indicates this strategy was adopted by multiple variants.

`>/tmp/.ptmx && cd /tmp/` is a command observed in about 110,000 sessions. This command is combined with very similar combinations of the same command, in different locations and is also based on the mounts file. Attackers using this strategy use a limited dictionary, and a noticable detail is that all source IP addresses only use correct credentials. It is likely this is part of the loader strategy as used by Mirai [4, 17], where the password probing and actual exploitation are done by

[4] `https://github.com/Phype/telnet-iot-honeypot`

two different machines. The used credentials are `admin:admin1234`, `admin:CenturyL1nk`, `admin:atlantis`, `admin:ho4uku6at` and `admin:zyad5001` but it is possible the discovery phase employs a wider dictionary. Due to the fact that these passwords are also used for different strategies, we could not distuingish the discovery machines directly related to this strategy.

6.6.5. Decision patterns

As discussed earlier in this section, many different commands are used to gather information on the system. Where different groups of malware act different on the same commands, we can identify a number commands which are responsible for these actions and decisions. These decisions can help us understand the type of device a piece of malware is targetting, and what it does to evade honeypots.

In general, the observed malware use one or more of the following commands to determine if an attacked device is suitable for further infection:

```
1 cat /proc/mounts
2 tftp, wget, curl
3 /bin/busybox IDENTIFIER
4 ps
5 echo
6 kill
```

Based on the output of these commands, attacks either continue or are aborted instantly. These commands play an important role in catching as many different attackers as possible in a honeypot. Where the used LXC container has the same response for all attackers to these commands, a number of variants likely specifically targetting certain devices abort their attack early in the killchain. Especially the content of the mounts file seems to be important, as the attackers use the contents of this file to both check suitability, as well as a source for potentially writable destinations required for follow-up steps. Evading potential honeypot detection methods by replacing the output of the `cat /proc/mounts` turns out to be counter-productive, as subsequent steps would fail because the fake file locations were not actually writeable.

6.6.6. Honeypot detection

There is a difference between how different groups of malware behave based on their initial commands. These commands can partially be attributed to architecture detection or vulnerability detection. However, based on the complexity of some of the commands, it is possible some groups are actively looking for ways to detect honeypots. It is difficult to determine if an attacker aborts a command sequence because a honeypot environment has been detected.

A way to check the universal knowledge of honeypots, Shodan has a metric called “honeyscore”[5]. Based on its scans, Shodan attempts to identify honeypot environments and assigns a score between 0 and 1, indicating the probability the given IP address is a honeypot. For all IP addresses used in the experiments, the honeyscore was requested. None of the IPs had a score other than 0.0, indicating the IP address did not show any sign of honeypot activity. While this can be interpreted as a good indicator, it requires an active scan conducted in the period the honeypot was active. Despite the regular scanning activity by Shodan, none of the IPs in the TU Delft range used in the experiments were scanned during that time. One of the agents which has been running for a longer period of time was scanned, but still indicated a honeyscore of 0.0.

6.7. Malware installation

Once an attacker has determined a device is suitable for infection using for example one of the strategies described above, the malware is downloaded. To achieve this, various methods can be used to get malware on the system. A common approach is the use of download tools available in the Busybox

[5] `https://honeyscore.shodan.io/`

toolset, such as wget and tftp. As observed earlier, it is common for attackers to check the availability of these tools on the system. Using this method, it is easy to get a malicious binary on the system, but requires a server hosting the files.

6.7.1. Base64 encoded payload

Using a Busybox download tool is not the only method to get a malicious binary on the system. The variants discussed earlier, which use the single-password strategy but do not include a Busybox identifier, have been observed writing a malicious binary directly through the Telnet session. This technique is used as both a fallback or as the primary method for downloading a malicious binary on the system. In the snippet below, the fallback method is shown.

```
1 /bin/busybox tftp -g -l /dev/.none -r yCz31g9 37.79.60.38 ; cat /dev/.none ; rm /dev/.
      ↪ none ; /bin/busybox SSfAh2cK
2 /bin/busybox wget http://37.79.60.38:15798/lvn3/eU -O /dev/.none ; cat /dev/.none ; rm /
      ↪ dev/.none ; /bin/busybox SSfAh2cK
3 /bin/busybox echo bFRP | base64 -d; /bin/busybox echo Nkty | openssl base64 -d; /bin/
      ↪ busybox SSfAh2cK
4 base64 -d > /dev/con << .
```

The first line shows an attempt to download a file using the TFTP protocol (explained in Section 6.5). A notable difference from other variants attempting such a download, is the fact that in the same command, the downloaded file is examined using `cat` but then immediately removed again. Usually, when a binary is successfully downloaded, that binary is used for further communication with the Command and Control server. After the TFTP command, a similar command sequence is executed, only then using the Wget application. Again, after the file is downloaded, it is examined with `cat` and directly removed without actually executing the binary. The final exploratory command checks the presence the base64 application. As this can be found both as a standalone program or part of the openssl toolkit, both are checked using a method comparable to application presence checks seen previously. This time, the output of an `echo` indicates which version of the base64 application is available. Both versions of the base64 applications are effectively the same, and allow the encoding and decoding of input to and from base64. In the last line, input is read from the terminal, decoded from base64, and written to `/dev/con`. What follows is a full base64 encoded binary executable file, which when decoded, is about 75 kB.

We observed 4,653 sessions where the presence of base64 was checked. However, not all attackers proceeded with the input of the binary file. To separate the sessions where a base64 encoded payload was completely transferred, we can look at the first part of the payload. As these first few bytes consist of the file header (`ELF`) required to make the file executable, they are always present. Filtering for this value (“f0VMRg” in its base64 encoded form) results in 989 observed sessions using this strategy. Those 989 sessions came from 714 unique source IPs and 683 unique destination IPs were hit. What stands out is that there’s very little duplicate infection attempts. At most, a single destination IP was infected 5 times by 3 different IP addresses.

When this attack path is traced back to earlier phases of the killchain, we observe this variant uses the single-password strategy without exposing a Busybox identifier. Just like other variants using the single-password strategy, the control banner is completely ignored by this variant.

7

Conclusions and Future Work

These days, malware targeting Internet of Things devices pose a serious threat to consumers, companies and the Internet as a whole. There are many actors active in the field and attack methods are becoming more sophisticated with each large scale attack. While the security of the Internet of Things is slowly gaining attention in both the media and vendor priorities, devices that are already deployed will make an impact for the forseeable future. Until good security becomes the standard, continuous research towards new threats is required to minimize the possible impact. As the expected number of active IoT devices will grow towards billions of devices in the coming years, research needs to scale up with this growth in order to keep up with the rising threat.

With Honeytrack, we were able to collect data on Telnet attacks on over 8000 observed IP addresses. To the best of our knowledge, this is the biggest high-interaction honeypot experiment compared to other published work.

The attacks were analyzed and mapped to a phase of the killchain model, allowing us to identify a number of strategies for each phase of a Telnet session.

For the negotiation phase, the chosen option codes proved insufficient for effective clustering. As nearly all attacks were automated and from previously infected machines, the hypothesis that the `LINEMODE` option would reveal human attackers turned out difficult to confirm. Only a handful of sessions were recorded with this option enabled, and those sessions could not prove the presence of a human attacker. The other option, `ECHO`, showed a clear separation of variants, with each client belonging to a specific variant showing identical behavior.

The credentials attempted after a successful negotiation prove to be dynamic in nature. As new hardcoded credentials are discovered in a device, malware authors quickly incorporate them into their dictionaries. Between all variants, only a few have truly unique dictionaries. Most variants use their own subset of all available credentials, where a full overlap between variants is rare.

When an attacker compromises a device by guessing the correct credentials, a number of strategies can be identified. Most attacks involve a number of commands to determine if the device is suitable for infection and to identify the underlying system architecture. After this is done, a suitable location for a payload is determined based on the mounts file. This file serves an important purpose, as the contents of this file are reason for number of variants to abort their attack. The retrieval of the payload is done in a number of different ways. These methods vary from the utilization of already present utilities such as wget or TFTP, to the manual input of a base64 encoded binary through the Telnet terminal.

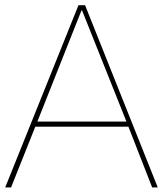
For the final two phases of the killchain, Command & Control and Actions on Objectives, further study is required as these were out of scope for this thesis. With Honeytrack, we gathered a large number of malware samples downloaded through various methods such as wget and TFTP. A small number of samples were checked against the VirusTotal database. Several of these samples were already known malicious files marked as Mirai variants. A number of files described in Chapter 6.7 were manually reconstructed and hashed. None of the resulting hashes were found on VirusTotal, indicating that the

malware is either unique in nature or only introduced very recently. Further analysis is required, but could lead to better insights into this malware retrieval strategy. Additionally, using full packet capture all traffic to and from the container was captured and stored. While not analyzed in detail, these packet captures likely contains communication with Command and Control servers such as instructions to attack certain targets.

When analyzing the data gathered with Honeytrack during the experimentation phase, we discovered more advanced methods are required to fully utilize the data gathered. As a large number of metrics are stored for each phase of a Telnet session, it is likely more subtle relationships between the sessions and related malware variants can be discovered using more advanced clustering algorithms.

Next to the original research questions, much more data was collected than we were able to analyze within the scope of this thesis. In the process of answering our research questions based on the data collected, we also discovered that more data is required to be able to cluster related sessions together. Due to the large overlap in used credentials, similar discovery procedures and data structure used to collect the data, attributing different IP addresses to the same attacker proved difficult. Finally, a more dynamic method of A/B testing is necessary in order to maximize the number of malware variants fully completing their attack. With the current setup, only a subset of variants completed all stages of the killchain model.

For future work, Honeytrack can be extended to include on-the-fly analysis of a Telnet session, allowing for a better way of matching a current attack to an already existing container. This way, the number of required containers can be reduced and an attack coming from different machines and/or attackers can be executed on a single container, increasing simulation accuracy.



LXC mounts file

```
/dev/sda1 / ext4 rw,relatime,errors=remount-ro,data=ordered 0 0
none /dev tmpfs rw,nodev,relatime,size=492k,mode=755,uid=1000000,gid=1000000 0 0
proc /proc proc rw,nosuid,nodev,noexec,relatime 0 0
proc /proc/sys/net proc rw,nosuid,nodev,noexec,relatime 0 0
proc /proc/sys proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/sysrq-trigger proc ro,nosuid,nodev,noexec,relatime 0 0
sysfs /sys sysfs rw,nosuid,nodev,noexec,relatime 0 0
sysfs /sys sysfs ro,nosuid,nodev,noexec,relatime 0 0
sysfs /sys/devices/virtual/net sysfs rw,nodev,relatime 0 0
sysfs /sys/devices/virtual/net sysfs rw,nosuid,nodev,noexec,relatime 0 0
fusectl /sys/fs/fuse/connections fusectl rw,relatime 0 0
debugfs /sys/kernel/debug debugfs rw,relatime 0 0
securityfs /sys/kernel/security securityfs rw,nosuid,nodev,noexec,relatime 0 0
pstore /sys/fs/pstore pstore rw,nosuid,nodev,noexec,relatime 0 0
mqueue /dev/mqueue mqueue rw,nodev,relatime 0 0
binfmt_misc /proc/sys/fs/binfmt_misc binfmt_misc rw,relatime 0 0
lxcfs /proc/cpuinfo fuse.lxcfs rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other 0 0
lxcfs /proc/diskstats fuse.lxcfs rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other 0
    ↪ 0
lxcfs /proc/meminfo fuse.lxcfs rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other 0 0
lxcfs /proc/stat fuse.lxcfs rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other 0 0
lxcfs /proc/swaps fuse.lxcfs rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other 0 0
lxcfs /proc/uptime fuse.lxcfs rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other 0 0
udev /dev/null devtmpfs rw,nosuid,relatime,size=24705676k,nr_inodes=6176419,mode=755 0 0
udev /dev/zero devtmpfs rw,nosuid,relatime,size=24705676k,nr_inodes=6176419,mode=755 0 0
udev /dev/full devtmpfs rw,nosuid,relatime,size=24705676k,nr_inodes=6176419,mode=755 0 0
udev /dev/urandom devtmpfs rw,nosuid,relatime,size=24705676k,nr_inodes=6176419,mode=755 0 0
udev /dev/random devtmpfs rw,nosuid,relatime,size=24705676k,nr_inodes=6176419,mode=755 0 0
udev /dev/tty devtmpfs rw,nosuid,relatime,size=24705676k,nr_inodes=6176419,mode=755 0 0
devpts /dev/console devpts rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000 0 0
devpts /dev/pts devpts rw,relatime,gid=1000005,mode=620,ptmxmode=666,max=1024 0 0
devpts /dev/ptmx devpts rw,relatime,gid=1000005,mode=620,ptmxmode=666,max=1024 0 0
devpts /dev/tty1 devpts rw,relatime,gid=1000005,mode=620,ptmxmode=666,max=1024 0 0
devpts /dev/tty2 devpts rw,relatime,gid=1000005,mode=620,ptmxmode=666,max=1024 0 0
devpts /dev/tty3 devpts rw,relatime,gid=1000005,mode=620,ptmxmode=666,max=1024 0 0
devpts /dev/tty4 devpts rw,relatime,gid=1000005,mode=620,ptmxmode=666,max=1024 0 0
tmpfs /dev/shm tmpfs rw,nosuid,nodev,uid=1000000,gid=1000000 0 0
tmpfs /run tmpfs rw,nosuid,nodev,mode=755,uid=1000000,gid=1000000 0 0
tmpfs /run/lock tmpfs rw,nosuid,nodev,noexec,relatime,size=5120k,uid=1000000,gid=1000000 0 0
tmpfs /sys/fs/cgroup tmpfs ro,nosuid,nodev,noexec,mode=755,uid=1000000,gid=1000000 0 0
cgroup /sys/fs/cgroup/systemd cgroup rw,nosuid,nodev,noexec,relatime,xattr,release_agent=/lib
    ↪ /systemd/systemd-cgroups-agent,name=systemd 0 0
cgroup /sys/fs/cgroup/memory cgroup rw,nosuid,nodev,noexec,relatime,memory 0 0
cgroup /sys/fs/cgroup/cpu,cpuacct cgroup rw,nosuid,nodev,noexec,relatime,cpu,cpuacct 0 0
cgroup /sys/fs/cgroup/perf_event cgroup rw,nosuid,nodev,noexec,relatime,perf_event 0 0
cgroup /sys/fs/cgroup/cpuset cgroup rw,nosuid,nodev,noexec,relatime,cpuset,clone_children 0 0
cgroup /sys/fs/cgroup/freezer cgroup rw,nosuid,nodev,noexec,relatime,freezer 0 0
cgroup /sys/fs/cgroup/devices cgroup rw,nosuid,nodev,noexec,relatime,devices 0 0
cgroup /sys/fs/cgroup/net_cls,net_prio cgroup rw,nosuid,nodev,noexec,relatime,net_cls,
```

```
    ↪ net_prio 0 0
48 cgroup /sys/fs/cgroup/hugetlb cgroup rw,nosuid,nodev,noexec,relatime,hugetlb 0 0
49 cgroup /sys/fs/cgroup/blkio cgroup rw,nosuid,nodev,noexec,relatime,blkio 0 0
50 cgroup /sys/fs/cgroup/pids cgroup rw,nosuid,nodev,noexec,relatime,pids 0 0
```

B

Variant Dictionary overlap

Table B.1: The overlap between dictionaries of all single-password strategy variants. Read from right to up; for example variant “s” shares 83.72% of its credentials with variant “sunless”, and “sunless”dictionary consists 45.78% of passwords also in “daddyl33t”dictionary

	s	sunless	daddyl33t	OBJPRN	MM	dwickedgod	MATOS	ORION	Cult	NGRLS	MIRAI	satori	PUTIN	QBOTV1	SORA
s	100.00%	83.72%	45.35%	29.07%	22.09%	20.93%	18.60%	1.16%	18.60%	16.28%	47.67%	24.42%	30.23%	30.23%	26.74%
sunless	86.75%	100.00%	45.78%	33.73%	24.10%	20.48%	19.28%	1.20%	16.87%	16.87%	50.60%	30.12%	34.94%	34.94%	30.12%
daddyl33t	11.61%	11.31%	100.00%	19.94%	10.42%	39.58%	5.36%	0.30%	4.46%	7.74%	56.55%	13.39%	15.48%	16.37%	10.12%
OBJPRN	7.60%	8.51%	20.36%	100.00%	11.25%	5.17%	2.13%	0.00%	3.04%	3.34%	27.05%	15.81%	15.50%	16.72%	6.38%
MM	47.50%	50.00%	87.50%	92.50%	100.00%	37.50%	12.50%	2.50%	22.50%	27.50%	100.00%	95.00%	95.00%	100.00%	40.00%
dwickedgod	13.53%	12.78%	100.00%	12.78%	11.28%	100.00%	5.26%	0.75%	8.27%	11.28%	25.56%	13.53%	15.04%	15.79%	14.29%
MATOS	69.57%	69.57%	78.26%	30.43%	21.74%	30.43%	100.00%	4.35%	34.78%	26.09%	73.91%	26.09%	43.48%	39.13%	65.22%
ORION	100.00%	100.00%	100.00%	0.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Cult	8.08%	7.07%	7.58%	5.05%	4.55%	5.56%	4.04%	0.51%	100.00%	5.05%	7.07%	4.55%	6.57%	6.57%	6.06%
NGRLS	46.67%	46.67%	86.67%	36.67%	36.67%	50.00%	20.00%	3.33%	33.33%	100.00%	90.00%	40.00%	53.33%	56.67%	53.33%
MIRAI	9.34%	9.57%	43.28%	20.27%	9.11%	7.74%	3.87%	0.23%	3.19%	6.15%	100.00%	12.76%	13.44%	15.26%	8.43%
satori	36.84%	43.86%	78.95%	91.23%	66.67%	31.58%	10.53%	1.75%	15.79%	21.05%	98.25%	100.00%	87.72%	96.49%	35.09%
PUTIN	41.27%	46.03%	82.54%	80.95%	60.32%	31.75%	15.87%	1.59%	20.63%	25.40%	93.65%	79.37%	100.00%	92.06%	39.68%
QBOTV1	37.14%	41.43%	78.57%	78.57%	57.14%	30.00%	12.86%	1.43%	18.57%	24.29%	95.71%	78.57%	82.86%	100.00%	37.14%
SORA	52.27%	56.82%	77.27%	47.73%	36.36%	43.18%	34.09%	2.27%	27.27%	36.36%	84.09%	45.45%	56.82%	59.09%	100.00%
Word	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
EXTENDO	8.94%	7.82%	8.38%	5.59%	5.03%	6.15%	4.47%	0.56%	100.00%	5.59%	7.82%	5.03%	7.26%	7.26%	6.70%
ZEAZO	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
AKUMA	17.02%	16.60%	45.11%	30.64%	16.60%	14.47%	7.23%	0.43%	5.96%	11.49%	59.57%	21.28%	22.98%	25.11%	16.17%
JOSHO	23.55%	21.80%	17.73%	9.01%	5.81%	6.98%	5.81%	0.29%	56.69%	5.23%	24.13%	7.27%	8.43%	9.01%	9.30%
ASUNA	90.91%	90.91%	100.00%	72.73%	63.64%	72.73%	36.36%	9.09%	90.91%	63.64%	100.00%	72.73%	81.82%	81.82%	81.82%
HHHH	35.71%	44.64%	78.57%	91.07%	66.07%	32.14%	10.71%	1.79%	16.07%	21.43%	98.21%	100.00%	89.29%	96.43%	35.71%
OOMGA	6.25%	5.21%	15.62%	15.62%	8.33%	7.29%	2.08%	1.04%	4.17%	7.29%	29.17%	12.50%	11.46%	12.50%	9.38%
XWIFZ	27.59%	27.59%	50.00%	18.97%	15.52%	32.76%	13.79%	1.72%	18.97%	39.66%	84.48%	18.97%	25.86%	31.03%	32.76%
OWARI	70.00%	76.67%	93.33%	43.33%	36.67%	43.33%	60.00%	3.33%	33.33%	43.33%	90.00%	40.00%	56.67%	53.33%	76.67%
Zeus	14.48%	15.17%	28.28%	13.79%	10.34%	13.10%	9.66%	0.69%	6.21%	9.66%	40.69%	11.72%	12.41%	12.41%	15.86%
MASUTA	18.49%	21.92%	49.32%	39.04%	27.40%	21.92%	6.16%	0.68%	8.90%	16.44%	72.60%	37.67%	38.36%	42.47%	18.49%
MIORI	46.34%	39.02%	53.66%	26.83%	21.95%	36.59%	17.07%	2.44%	39.02%	31.71%	56.10%	24.39%	36.59%	36.59%	36.59%
ECCHI	8.33%	8.33%	33.33%	8.33%	8.33%	0.00%	0.00%	0.00%	0.00%	0.00%	16.67%	8.33%	8.33%	8.33%	0.00%
RBGLZ	49.06%	43.40%	81.13%	50.94%	37.74%	41.51%	24.53%	1.89%	24.53%	49.06%	94.34%	45.28%	54.72%	58.49%	45.28%
FREEPEIN	36.07%	42.62%	81.97%	90.16%	65.57%	31.15%	9.84%	1.64%	16.39%	21.31%	100.00%	90.16%	90.16%	98.36%	32.79%

Table B.2: The overlap between dictionaries of all single-password strategy variants. Read from right to up

	Word	EXTENDO	ZEAZO	AKUMA	JOSHO	ASUNA	HHHH	OOMGA	XWIFZ	OWARI	Zeus	MASUTA	MIORI	ECCHI	RBGLZ	FREEPEIN
s	0.00%	18.60%	0.00%	46.51%	94.19%	11.63%	23.26%	6.98%	18.60%	24.42%	24.42%	31.40%	22.09%	1.16%	30.23%	25.58%
sunless	0.00%	16.87%	0.00%	46.99%	90.36%	12.05%	30.12%	6.02%	19.28%	27.71%	26.51%	38.55%	19.28%	1.20%	27.71%	31.33%
daddyl33t	0.00%	4.46%	0.00%	31.55%	18.15%	3.27%	13.10%	4.46%	8.63%	8.33%	12.20%	21.43%	6.55%	1.19%	12.80%	14.88%
OBJPRN	0.00%	3.04%	0.00%	21.88%	9.42%	2.43%	15.50%	4.56%	3.34%	3.95%	6.08%	17.33%	3.34%	0.30%	8.21%	16.72%
MM	0.00%	22.50%	0.00%	97.50%	50.00%	17.50%	92.50%	20.00%	22.50%	27.50%	37.50%	100.00%	22.50%	2.50%	50.00%	100.00%
dwickedgod	0.00%	8.27%	0.00%	25.56%	18.05%	6.02%	13.53%	5.26%	14.29%	9.77%	14.29%	24.06%	11.28%	0.00%	16.54%	14.29%
MATOS	0.00%	34.78%	0.00%	73.91%	86.96%	17.39%	26.09%	8.70%	34.78%	78.26%	60.87%	39.13%	30.43%	0.00%	56.52%	26.09%
ORION	0.00%	100.00%	0.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	0.00%	100.00%	100.00%
Cult	0.00%	90.40%	0.00%	7.07%	98.48%	5.05%	4.55%	2.02%	5.56%	5.05%	4.55%	6.57%	8.08%	0.00%	6.57%	5.05%
NGRLS	0.00%	33.33%	0.00%	90.00%	60.00%	23.33%	40.00%	23.33%	76.67%	43.33%	46.67%	80.00%	43.33%	0.00%	86.67%	43.33%
MIRAI	0.00%	3.19%	0.00%	31.89%	18.91%	2.51%	12.53%	6.38%	11.16%	6.15%	13.44%	24.15%	5.24%	0.46%	11.39%	13.90%
satori	0.00%	15.79%	0.00%	87.72%	43.86%	14.04%	98.25%	21.05%	19.30%	21.05%	29.82%	96.49%	17.54%	1.75%	42.11%	96.49%
PUTIN	0.00%	20.63%	0.00%	85.71%	46.03%	14.29%	79.37%	17.46%	23.81%	26.98%	28.57%	88.89%	23.81%	1.59%	46.03%	87.30%
QBOTV1	0.00%	18.57%	0.00%	84.29%	44.29%	12.86%	77.14%	17.14%	25.71%	22.86%	25.71%	88.57%	21.43%	1.43%	44.29%	85.71%
SORA	0.00%	27.27%	0.00%	86.36%	72.73%	20.45%	45.45%	20.45%	43.18%	52.27%	52.27%	61.36%	34.09%	0.00%	54.55%	45.45%
Word	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
EXTENDO	0.00%	100.00%	0.00%	7.82%	98.32%	5.59%	5.03%	2.23%	6.15%	5.59%	5.03%	7.26%	8.94%	0.00%	7.26%	5.59%
ZEAZO	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
AKUMA	0.00%	5.96%	0.00%	100.00%	25.53%	4.68%	20.85%	8.09%	15.32%	11.06%	16.60%	38.30%	9.79%	0.43%	20.85%	22.55%
JOSHO	0.00%	51.16%	0.00%	17.44%	100.00%	3.20%	6.98%	2.91%	6.40%	8.14%	18.31%	11.05%	6.40%	0.58%	8.14%	7.56%
ASUNA	0.00%	90.91%	0.00%	100.00%	100.00%	100.00%	72.73%	36.36%	72.73%	72.73%	72.73%	100.00%	90.91%	0.00%	90.91%	72.73%
HHHH	0.00%	16.07%	0.00%	87.50%	42.86%	14.29%	100.00%	21.43%	19.64%	21.43%	30.36%	96.43%	17.86%	1.79%	42.86%	96.43%
OOMGA	0.00%	4.17%	0.00%	19.79%	10.42%	4.17%	12.50%	100.00%	7.29%	5.21%	10.42%	16.67%	5.21%	0.00%	11.46%	12.50%
XWIFZ	0.00%	18.97%	0.00%	62.07%	37.93%	13.79%	18.97%	12.07%	100.00%	27.59%	29.31%	75.86%	27.59%	0.00%	53.45%	20.69%
OWARI	0.00%	33.33%	0.00%	86.67%	93.33%	26.67%	40.00%	16.67%	53.33%	100.00%	66.67%	60.00%	36.67%	6.67%	66.67%	40.00%
Zeus	0.00%	6.21%	0.00%	26.90%	43.45%	5.52%	11.72%	6.90%	11.72%	13.79%	100.00%	18.62%	8.28%	0.00%	15.86%	11.72%
MASUTA	0.00%	8.90%	0.00%	61.64%	26.03%	7.53%	36.99%	10.96%	30.14%	12.33%	18.49%	100.00%	13.01%	0.68%	27.40%	41.78%
MIORI	0.00%	39.02%	0.00%	56.10%	53.66%	24.39%	24.39%	12.20%	39.02%	26.83%	29.27%	46.34%	100.00%	0.00%	43.90%	26.83%
ECCHI	0.00%	0.00%	0.00%	8.33%	16.67%	0.00%	8.33%	0.00%	0.00%	16.67%	0.00%	8.33%	0.00%	100.00%	0.00%	8.33%
RBGLZ	0.00%	24.53%	0.00%	92.45%	52.83%	18.87%	45.28%	20.75%	58.49%	37.74%	43.40%	75.47%	33.96%	0.00%	100.00%	47.17%
FREEPEIN	0.00%	16.39%	0.00%	86.89%	42.62%	13.11%	88.52%	19.67%	19.67%	19.67%	27.87%	100.00%	18.03%	1.64%	40.98%	100.00%

Bibliography

[1] Ahmed Awad E. Ahmed, Issa Traoré, and Ahmad Almulhem. Digital Fingerprinting Based on Keystroke Dynamics. *Second International Symposium on Human Aspects of Information Security & Assurance (HAISA 2008) Digital*, (Haisa):94–104, 2008.

[2] Ankit Anubhav. IoT Hackers Shift to the Dark Side, 2017. URL https://blog.newskysecurity.com/iot-hackers-shift-to-the-dark-side-cd3d0005a5e0.

[3] Ankit Anubhav. Understanding the IoT Hacker — A Conversation With Owari/Sora IoT Botnet Author, 2018. URL https://blog.newskysecurity.com/understanding-the-iot-hacker-a-conversation-with-owari-sora-iot-botnet-author-117feff

[4] Manos Antonakakis, Tim April, Michael Bailey, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, Yi Zhou, Manos Antonakakis Tim April Michael Bailey Matthew Bernhard Elie Bursztein, Bullet J Jaime Cochran Zakir Durumeric Alex Halderman Luca Invernizzi, Bullet Michalis Kallitsis Deepak Kumar Chaz Lever Zane Ma, Joshua Mason Damian Menscher, Bullet Chad Seaman Nick Sullivan Kurt Thomas, and Bullet Yi Zhou. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium*, pages 1093–1110, 2017. ISBN 978-1-931971-40-9.

[5] Michael Bailey, Evan Cooke, and David Watson. A hybrid honeypot architecture for scalable network monitoring. *… -Tr-499-04, …*, 2004.

[6] Norbert Blenn, Vincent Ghiëtte, and Christian Doerr. Quantifying the Spectrum of Denial-of-Service Attacks through Internet Backscatter. In *Proceedings of the 12th International Conference on Availability, Reliability and Security - ARES '17*, pages 1–10, New York, New York, USA, 2017. ACM Press. ISBN 9781450352574. doi: 10.1145/3098954.3098985. URL http://dl.acm.org/citation.cfm?doid=3098954.3098985.

[7] Catalin Cimpanu. BrickerBot Author Claims He Bricked Two Million Devices, 2017. URL https://www.bleepingcomputer.com/news/security/brickerbot-author-claims-he-bricked-two-million-devices/.

[8] P. S. Dowland, S. M. Furnell, and M. Papadaki. Keystroke Analysis as a Method of Advanced User Authentication and Response. pages 215–226. Springer, Boston, MA, 2002. ISBN 1-4020-7030-6. doi: 10.1007/978-0-387-35586-3_17. URL http://link.springer.com/10.1007/978-0-387-35586-3{_}17.

[9] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Proceedings of the 22nd USENIX Security Symposium*, number August, pages 605–619, 2013. ISBN 9781931971034.

[10] Sam Edwards and Ioannis Profetis. Hajime: Analysis of a decentralized internet worm for IoT devices. 2016.

[11] Kevin D. Fairbanks, Christopher P. Lee, Ying H. Xia, and Henry L. Owen. Timekeeper: A metadata archiving method for honeypot forensics. In *Proceedings of the 2007 IEEE Workshop on Information Assurance, IAW*, pages 114–118. IEEE, jun 2007. ISBN 1424413044. doi: 10.1109/IAW.2007.381922.

[12] Juan Guarnizo, Amit Tambe, Suman Sankar Bhunia, Martín Ochoa, Nils Tippenhauer, Asaf Shabtai, and Yuval Elovici. SIPHON: Towards Scalable High-Interaction Physical Honeypots. jan 2017. URL http://arxiv.org/abs/1701.02446.

[13] Thorsten Holz and Frederic Raynal. Detecting honeypots and other suspicious environments. In *Proceedings from the 6th Annual IEEE System, Man and Cybernetics Information Assurance Workshop, SMC 2005*, volume 2005, pages 29–36. IEEE, 2005. ISBN 0780392906. doi: 10.1109/IAW.2005.1495930. URL http://ieeexplore.ieee.org/document/1495930/.

[14] Eric M Hutchins, Michael J Cloppert, and Rohan M Amin. Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains. *6th Annual International Conference on Information Warfare and Security*, (July 2005):1–14, 2011.

[15] KPN. LoRa connectivity | KPN IoT. URL https://www.kpn.com/zakelijk/grootzakelijk/internet-of-things/en/lora-connectivity.htm.

[16] Tongbo Luo, Zhaoyan Xu, Xing Jin, Yanhui Jia, and Xin Ouyang. IoTCandyJar: Towards an Intelligent-Interaction Honeypot for IoT Devices. *Blackhat*, 2017.

[17] Yin Minn, Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. IoTPOT: Analysing the Rise of IoT Compromises. URL https://www.usenix.org/system/files/conference/woot15/woot15-paper-pa.pdf.

[18] J. Postel and J.K. Reynolds. Telnet Option Specifications. Technical report, 1983. URL https://tools.ietf.org/html/rfc855https://www.rfc-editor.org/info/rfc0855.

[19] J. Postel and J.K. Reynolds. Telnet Protocol Specification. Technical report, 1983. URL https://tools.ietf.org/html/rfc854https://www.rfc-editor.org/info/rfc0854.

[20] Niels Provos. Honeyd: A Virtual Honeypot Daemon. *Proceedings of the 10th DFNCERT Workshop*, pages 1–7, 2003. doi: 10.1.1.63.8084. URL http://repository.mdp.ac.id/ebook/library-sw-hw/linux-1/HONEYPOTS/honeyd/docs/honeyd-eabstract.pdf.

[21] Daniel Ramsbrock, Robin Berthier, and Michel Cukier. Profiling attacker behavior following SSH compromises. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 119–124. IEEE, jun 2007. ISBN 0769528554. doi: 10.1109/DSN.2007.76. URL http://ieeexplore.ieee.org/document/4272962/.

[22] Ramon Sanchez-Iborra, Jesus Sanchez-Gomez, Juan Ballesta-Viñas, Maria Dolores Cano, and Antonio F. Skarmeta. Performance evaluation of lora considering scenario conditions. *Sensors (Switzerland)*, 18(3):772, mar 2018. ISSN 14248220. doi: 10.3390/s18030772. URL http://www.mdpi.com/1424-8220/18/3/772.

[23] Bruce Schneier. Essays: The Internet of Things Is Wildly Insecure—And Often Unpatchable, 2014. URL https://www.schneier.com/essays/archives/2014/01/the{_}internet{_}of{_}thin.html.

[24] Lance. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2002. ISBN 0321108957. doi: 10.1128/AAC.03728-14. URL https://dl.acm.org/citation.cfm?id=515237.

[25] Rob van der Meulen. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016, 2017. ISSN 1098-6596. URL https://www.gartner.com/newsroom/id/3598917https://www.gartner.com/newsroom/id/3598917{%}0Ahttps://www.gartner.com/newsroom/id/3165317{%}0Ahttp://www.gartner.com/newsroom/id/3598917.

[26] Remco Verhoef. What is Honeytrap | Honeytrap. URL http://docs.honeytrap.io/docs/concepts/overview/what-is-honeytrap/.

[27] Xueying Yang. LoRaWAN: Vulnerability Analysis and Practical Exploitation. 2017.

[28] Cliff C. Zou and Ryan Cunningham. Honeypot-aware advanced botnet construction and maintenance. In *Proceedings of the International Conference on Dependable Systems and Networks*, volume 2006, pages 199–208. IEEE, 2006. ISBN 0769526071. doi: 10.1109/DSN.2006.38. URL http://ieeexplore.ieee.org/document/1633509/.