CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Developing a modular, high-interaction honeypot using container virtualization

Bachelor's thesis in Computer science and engineering

Magnus Jonsson
Alexander Lysholm
Sophia Pham
Benjamin Sannholm
Oskar Svanström
Johan Valfridsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Bachelor's thesis 2021

Developing a modular, high-interaction honeypot using container virtualization

Magnus Jonsson
Alexander Lysholm
Sophia Pham
Benjamin Sannholm
Oskar Svanström
Johan Valfridsson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Developing a modular, high-interaction honeypot using container virtualization

Supervisors: Francisco Blas Izquierdo Riera and Magnus Almgren, Department of Computer Science and Engineering
Examiner: Arne Linde, Department of Computer Science and Engineering

Bachelor's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An illustration of an adversary being caught in a honeypot, made by Sophia Pham.

Typeset in LaTeX
Gothenburg, Sweden 2021

Abstract

A significant increase of attacks combined with companies neglecting cybersecurity poses a problem for the ever-growing number of Internet-connected devices. With honeypot technology being an invaluable tool for understanding adversaries' strategies, this project aims to develop a modular honeypot that is able to evolve. The report focuses on the honeypot's design and implementation. Results from deploying the honeypot are used to evaluate its properties and present a brief analysis of the information collected. To achieve a modular design, the honeypot is divided into three network-connected components. Container virtualization is utilized to allow easy deployment, imitating systems that adversaries interact with, and isolating performed actions. The result of the project is an easily deployable and distributable high-interaction research honeypot using container virtualization with support for the Secure Shell network protocol. The findings of the report indicate that the developed honeypot functions well and may support research within the field of computer security. However, there is room for deeper analysis of collected data to determine whether the honeypot's data collection capabilities are enough to draw valuable conclusions.

Sammandrag

En ökning av attacker i kombination med att företag försummar cybersäkerhet utgör ett problem för det ständigt växande antalet internetanslutna enheter. En honeypot är ett ovärderligt verktyg för att förstå angripares strategier, och därmed ämnar detta kandidatarbete att designa och utveckla en modulär honeypot som kan vidareutvecklas enligt framtida behov. Baserat på erfarenheter från driftsättning och insamlad information under drift presenteras en utvärdering av honeypotens egenskaper. För att åstadkomma en modulär design är honeypoten uppdelad i tre komponenter som samarbetar via nätverk och stödjer Secure Shell-protokollet. Container-virtualisering används för att underlätta distribution, möjliggöra imitation av system som angripare interagerar med och för att isolera angriparens handlingar. Projektet har resulterat i en forskningshoneypot med hög interaktionsnivå som är enkel att driftsätta och distribuera över flera system. Rapportens resultat tyder på att den utvecklade honeypoten fungerar väl, men det finns utrymme för djupare analys för att avgöra om honeypotens funktioner är tillräckliga för att dra meningsfulla slutsatser.

Keywords: honeypot, container-based virtualization, docker, security, data collection, modular software design, SSH, database design, network monitorization

Acknowledgements

We would like to thank our supervisor Francisco Blas Izquierdo Riera, who always had a word of encouragement and shared his vast expertise with us. We would also want to thank our second supervisor Magnus Almgren for the constructive criticism. We also want to mention and thank the Swedish Civil Contingencies Agency (MSB) for funding such great supervisors through the Resilient IoT project.

Magnus Jonsson, Alexander Lysholm, Sophia Pham, Benjamin Sannholm, Oskar Svanström, Johan Valfridsson, Gothenburg, June 2021

Contents

List of Figures

List of Tables

Glossary

backend The implementation of the target system provider, implemented by controlling Docker containers. 3, 21–25, 27–30, 34, 37, 38

botnet A network of hijacked IoT devices controlled by an adversary. 1

containerization A form of virtualization, where applications are run in an isolated container. 2, 4, 7, 10, 11

denial-of-service attack An attack in which the perpetrator tries to make a machine or network inoperable. 1, 37

DNS Domain Name System. A system for translating human readable addresses like "https://slashdot.org" into IP addresses, e.g., "216.105.38.1". 1

Docker An open-source tool for packaging applications and their dependencies using containerization. 2, 7, 13, 14, 21–25, 27, 28, 34, 35, 38

frontend The component of the honeypot that receives connections from adversaries, forwards them to a target system and logs the activity. 3, 9–13, 17, 18, 21–23, 25–29, 34, 36, 38

honeypot A security mechanism whose objective is to detect attacks or gather information about attacks. A honeypot can be of two types, production or research, and has a level of interaction between low, medium, and high. xi, 1–5, 8–10, 13, 14, 20, 22, 24, 28–32, 34–40

IANA The Internet Assigned Numbers Authority is an organization that oversees global IP address allocation. 5, 23

IoT Internet of Things. All Internet-connected devices together can be regarded as the Internet of Things. The devices can, for instance, be small household items with sensors, surveillance cameras or industrial robots. 1, 2

multiplexing A method of combining multiple signals or streams of information into one signal, over a shared medium. 5

RPC A remote procedure call is when control is synchronously transferred between programs running in different address spaces over a narrow communication channel, often across a computer network. 25, 27

SSH Secure Shell. A communication protocol enabling two computers to communicate in a secure way. xi, 2, 5, 6, 9, 10, 15, 17–21, 26, 28, 29, 31, 34, 36, 37, 40

target container An implementation of a target system in the form of a Docker container. 21–24, 28, 29, 34, 35, 37, 38

target system A controlled environment where adversaries may safely execute potentially malicious actions. 9–12, 18–24, 26, 27, 34, 36, 38, 39

target system provider The component of the honeypot responsible for preparing and managing target systems. 9–13, 18, 21, 22, 25, 27, 34

TCP The Transmission Control Protocol is a connection-oriented network protocol that provides a reliable, ordered, and error-checked transfer of data between two computers. 23

TLS Transport Layer Security is the improved and successor protocol to SSL which also uses encryption to protect the transfer of data and information. 25, 39

VM A virtual machine is an application environment that behaves like an actual computer. 28

1

Introduction

Companies producing devices for the "Internet of Things" (IoT) often aim to maximize their profits [1], resulting in a lower budget for security mechanisms and lack of long-term security patches in their devices. Weak security leads to unpatched vulnerabilities that attackers can exploit to form botnets. One such botnet was Mirai, which in 2016 caused the biggest Internet blackout ever by bringing down the DNS provider Dyn through a distributed denial-of-service attack [1], [2].

Further exacerbating the problem, the number of IoT devices seems to be growing every year. According to a report from Business Insider, there were around 8 billion Internet-connected devices in 2019 [3]. Together, these devices make up the IoT. The authors of the report estimate that the number of devices in the IoT will grow, resulting in 41 billion devices by 2027. The telecommunication company Ericsson estimates that there will be 24 billion Internet-connected devices by 2050 [4].

Using tools such as firewalls can help protect against many threats aimed toward IoT devices, but many tools are not proactive [5]. To understand what actions an attacker performs when reaching a device, there is a need for tools that can safely let an attacker into the device and monitor their behavior. This is where a honeypot has its place.

The number of attacks on IoT devices tripled in the first half of 2019, according to F-Secure [6]. Furthermore, Google's *Project Zero* [7] states that, on average, a new exploit is discovered every 17 days. These facts implicate a need of honeypots that can be adapted to discover new attacks.

1.1 Purpose and goals

The purpose of the project is to design and develop a honeypot that will be able to evolve with the ever-changing environment in computer security. The honeypot shall be easy to use, modular, and released as free software to aid future advancement in the domain of computer security. Information collected by the honeypot should be easily accessible in order to shed light on adversaries' strategies of attacking and compromising systems.

Developed from this purpose, the honeypot shall:

- be able to collect information that can be analyzed and used to draw conclusions regarding attackers and their objectives.

- support at least one network protocol for attackers to connect to, and allow implementing new protocols in the future.
- support imitating at least one type of system, and allow for imitating more systems in the future.
- remain operational regardless of alterations done to it by an attacker.
- limit the attacks an adversary can perform on other systems via the honeypot.
- be able to run on low-resource systems for extended amounts of time.

To evaluate the work, the honeypot's maintainability, modularity, ease of use, security, and detectability will be considered. The experience of deploying the honeypot publicly on the Internet and the information collected by it will be used to evaluate the mentioned properties.

1.2 Scope

To keep the project within a reasonable time frame, only the Secure Shell (SSH) protocol will be implemented. Existing libraries for SSH will be utilized. However, if no satisfactory options exist, the necessary functionality will be implemented. Furthermore, this report will not go into any deep analysis of the data collected during deployment as the focus lies on design and implementation. Additionally, the project will not actively attempt to stop any attacks.

1.3 Related work

Research of honeypots has been underway since at least the nineties [8] and as technology has advanced, some researchers have started to investigate the use of containerization technology for utilization in honeypots. Kyriakou and Skalavos conclude that pre-built low- and medium-interaction honeypots using Docker containerization can gather valuable information and that attackers seem to be shifting their focus toward IoT devices [9].

Eftimie and Racuciu also state that containerization of low-interaction honeypots can be used in the industry [10]. They conclude that using containers could remove some of the complexity in designing vulnerable images and enable creation of detecting intrusions early through a scalable Security-as-a-Service.

Research regarding the use of containers for high-interaction honeypots has also been conducted, with Valicek, Schramm, Pirker, and Schrittwieser [11]. They developed a honeypot that has an encrypted tunnel connection into a production network. This allows for a remote honeypot instead of having the physical host for the honeypot alongside the production servers.

1.4 Workflow

Planning the design and implementation of developing a honeypot held many uncertainties, which led to background research before continuing. The group functioned well from the beginning, which led to the design phase being a collaborative effort with all members involved. After an initial design had been constructed (described in chapter 3), the group split into three sub-groups where each group worked in parallel. Each group was assigned one of the design's three modules frontend, backend, and database. The sub-groups utilized pair programming as their internal way of working, with communication at the project level being handled through group meetings at least twice per week.

An approach similar to the agile framework Scrum was used, where the group planned iterative sprints with user stories that should be done before a deadline. In the group meetings, each sub-group reported their status and any troubles that had to be resolved before continuing. GitHub Issues and GitHub Project Boards were used to keep track of user stories, their status, and their assignees.

The version control system Git, with commits being uploaded to GitHub[1], was used to keep track of code changes and help document the changes. GitHub also allowed code review to be conducted, as new changes would be proposed in the form of pull requests. If the majority of the group accepted the changes, they were merged into the main branch. Continuous integration (CI) and automated testing were used to help with code reviews.

[1] https://github.com/Botnet-Honeypot/Honeypot

2

Technical background

This section aims to give a deeper understanding of the concepts that the project has utilized while developing the honeypot. Details about honeypots, the Secure Shell network protocol, virtualization and containerization are covered in this section.

2.1 Honeypots

A honeypot is a security mechanism that exists to detect and possibly lure an adversary away from another system or gain information about an adversary. Lance Spitzner, who formally introduced the notion of honeypots, defines a honeypot accordingly:

> "A honeypot is [sic] security resource whose value lies in being probed, attacked, or compromised" [8, p. 35].

In [12] Joshi and Sardana, the authors of *Honeypots: A New Paradigm to Information Security*, elaborate on what a honeypot is. They compare it to a surveillance camera, as the honeypot exists to observe every action against itself. These actions can vary from login attempts toward the honeypot to compromising and infecting the honeypot. The honeypot logs every action made toward it to gain information about the adversary. Alternatively, the information can be used to deflect either an ongoing attack or possible future attacks.

There are two kinds of honeypots: production honeypots and research honeypots [8], [13]. Production honeypots aim to detect intrusions and alert a system administrator, while research honeypots aim to reveal how attackers compromise systems so that security patches can be developed [8]. Both types of honeypots can have three levels of interaction, low, medium and high [8], [13]. The level of interaction of a honeypot determines how complex it is and its capabilities [8].

Low-interaction honeypots offer easy deployment and maintainability, with less risk to the host system but also limited collection of information [8]. Typically, a low-interaction honeypot can collect the IP address, port, and when an attack happened. Medium-interaction honeypots go one step further and send replies to the attacker [8], [13]. They let the attacker interact with an imitated service, for example sending web server responses to typical HTTP requests. This extends the collection capabilities to include payloads and some of attacker's exploit methods, but also increases the risks and effort needed to maintain the honeypot, since there are many different responses and services to imitate. High-interaction honeypots provide an, ideally, unmodified system for the adversary to

interact with [8]. Constructing, maintaining and supervising a high-interaction honeypot requires much effort, but provides the ability to reveal almost all methods adversaries utilize [8], including zero-day exploits [13].

2.2 Secure Shell

The Secure Shell (SSH) protocol is defined in the request for comments (RFC) document 4251:

> "The Secure Shell (SSH) Protocol is a protocol for secure remote login and other secure network services over an insecure network" [14].

SSH provides three services apart from secure remote login. The first one is the remote execution of a program. The program may be a shell, an application, a system command, or some built-in subsystem [15]. Secondly, SSH provides a means of forwarding X11 connections [15]. The X Window System, or X11, is a windowing system popular with UNIX-like operating systems [16]. With X11 forwarding, a remote application's user interface may be forwarded over the SSH connection. Lastly, SSH allows forwarding TCP/IP ports between the two hosts connected over SSH [15].

Today, SSH is used by millions of computers. Shodan, the world's first search engine for Internet-connected devices, currently lists (as of 2021-04-29) 18 million devices on the Internet running an SSH server [17].

2.2.1 Connection and authentication

SSH uses a client/server architecture, where clients initiate a connection to SSH servers [18]. Although not required, SSH typically operates over TCP/IP, for which port 22 has been assigned to SSH by the Internet Assigned Numbers Authority (IANA) [18].

One way for an SSH server to authenticate a client, which all implementations must support, is through password authentication [19]. Here, the client proves his identity by providing a username and a password [19]. The server can then verify that the password corresponds to the provided username and grants access if so is the case. Two additional authentication methods are described in Appendix A.1.

2.2.2 Channels and requests

SSH channels provide a means of multiplexing the different services offered by SSH, mentioned in section 2.2, inside a single connection [15]. One channel type exists for each service, and the channels have to be opened by either the server or the client before the service can be provided [15]. The channels that allow clients to execute commands on the remote server are of type "session" [15]. Described in the list below are a few examples of requests that can be sent over channels of type "session" to achieve different outcomes.

Shell request

To open a remote shell, the client may issue a "shell" request [15]. The server then starts up a shell which the client can interact with. At this point, the client may execute any

number of further commands directly to the shell supplied by the SSH server.

Exec request

If a client wishes to execute a single command, requesting a shell on the remote end might be excessive. Instead, the "exec" request is more suited for this purpose. It allows the client to send a single command over the channel, which is then executed on the remote host [15].

Pseudo-terminal request

A client can optionally send a "pty-req" request to allocate a pseudo-terminal with the same functionality as the client's terminal on the server [15]. The request contains information such as the client's terminal type, terminal modes, width, height, and font size.

Window dimension change request

If the client's terminal size changes during an SSH session, a "window-change" request may be sent by the client to inform the server of the new dimension [15]. The server can then use this information to ensure text is rendered in accordance with the client's terminal.

2.3 Virtualization and containerization

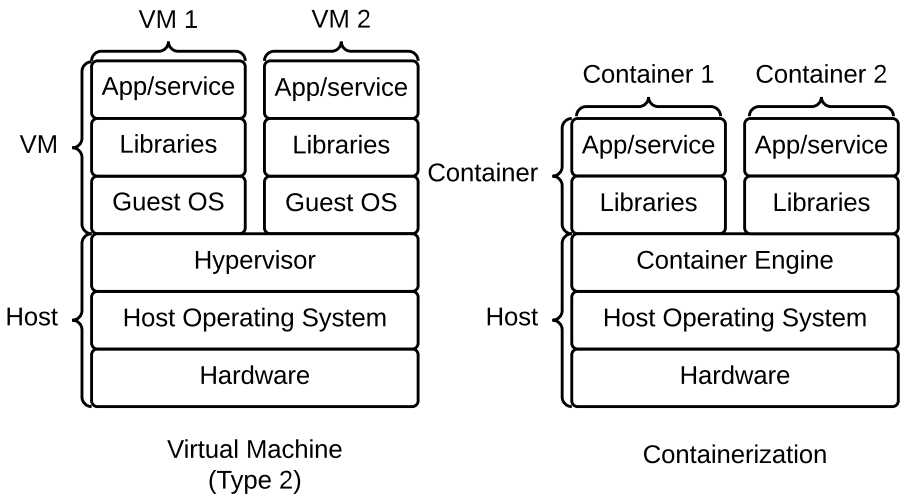

Figure 2.1: Topology of virtualization and containerization.[1]

Virtualization is a technology that can be used to utilize hardware more efficiently.[20] It is also a way of achieving security and, to an extent, reliability by using a technique called sandboxing. Sandboxing is built around giving each service or application a separate space to run in, and a way to access resources independently from other services or applications. Traditionally, this was achieved without virtualization by purchasing one computer for each service. However, it was seen as an inefficient use of the available budget, and thus other methods were sought after and researched [20].

[1] Adapted from "What is Kubernetes?" by "The Kubernetes Authors", licensed under CC BY 4.0

2.3.1 Virtual machines

Virtual machines (VMs) are used when a single physical computer must simulate multiple virtual computers such that each service can be given an entire operating system with custom configurations [21]. Each virtual machine is logically separated from the others, giving the illusion of an entirely separate physical machine.

There are two types of virtual machines, typically referred to as having type 1 or type 2 hypervisors [21]. The main difference between the two is the underlying operating system on the physical host. Type 1 hypervisors only add a minimal software layer between the virtual machine and the hardware, supplying only the necessities for operation instead of an entire operating system. Type 2 hypervisors run on top of an existing operating system and runs the virtual machines, utilizing the functionality provided by the operating system [20]. While the type 1 hypervisor is slightly more resource-efficient than the type 2, both still work by providing multiple operating systems on the same computer where each virtual machine consumes the resources needed by an entire operating system in addition to the service they run [21].

2.3.2 Containers

Containerization, or application virtualization, creates a container for each application [21]. These containers contain the environment in which to run the applications, complete with libraries and other dependencies. A container does not contain an operating system like the virtual machines, as can be seen in Figure 2.1. Instead it shares the kernel of the host operating system and thus the functionalities provided by it. Since a container only runs the application, each container consumes fewer resources than a virtual machine [21].

Although the separation between the host operating system and a container is not as strong security-wise, there is still some sandboxing in place as well as additional measures one can take to strengthen the separation [22]. Due to containers usually being very small in terms of disk space required, they can be easily deployed and moved, even across different hardware. Since containers do not have to set up their own operating system, there can be hundreds of lightweight container instances running on the same host. This also helps reduce the deployment time since there is no additional operating system to boot up [23].

2.3.3 Docker

Docker is an implementation of the containerization concept described in the previous section [24]. When using Docker, an interface called the Docker client communicates with the Docker daemon, which in turn performs the underlying work of managing containers [25]. To create a container, Docker uses a template called an image, where the application, file system, configuration, and dependencies are placed [25]. The image is created from a `Dockerfile`, in which a developer states how the image should be built. Additionally, containers can be started and further configured using a `docker-compose.yml` file. When starting a Docker container, it is possible to pass in environment variables and writable, persistent filesystems called volumes [26]. Docker containers aim to be distributable, scalable, and platform-independent regardless of the contents inside the container. The result is a software deployment unit that is portable, scalable, and offers isolation [27].

3

Design of honeypot system

In order to start developing the honeypot, a design was needed to identify the necessary components of the project. Starting off with the design, requirements were needed to know exactly what the honeypot should be capable of, as well as what features to include into the honeypot and the project as a whole.

3.1 Requirements

The process of determining requirements for the honeypot's design laid the foundation for the resulting capabilities of the honeypot. Through discussion between the project's members, several requirements were constructed:

- Since a honeypot fundamentally aims to collect information about an attacker and their actions, information storage is required. The information storage shall be structured for easy access and information retrieval.
- To make it possible to collect a wider variety of information, a high-interaction research honeypot shall be developed.
- To make it possible to collect a larger quantity of information, the honeypot shall be able to handle concurrent connections.
- To allow distinction between different adversaries' actions, each adversary session shall be kept separate.
- An adversary shall not be able to tell they are interacting with a honeypot, otherwise, they might alter their actions toward the honeypot.
- The honeypot shall be modular.
- The honeypot shall be easy to configure and deploy.
- The honeypot shall be able to visualize the collected information to the end-user.

3.2 Finished design

After determining product requirements, a finalized design was constructed through multiple iterations of discussions and suggestions. This section presents the design as a whole, its components, and motivation for design choices.

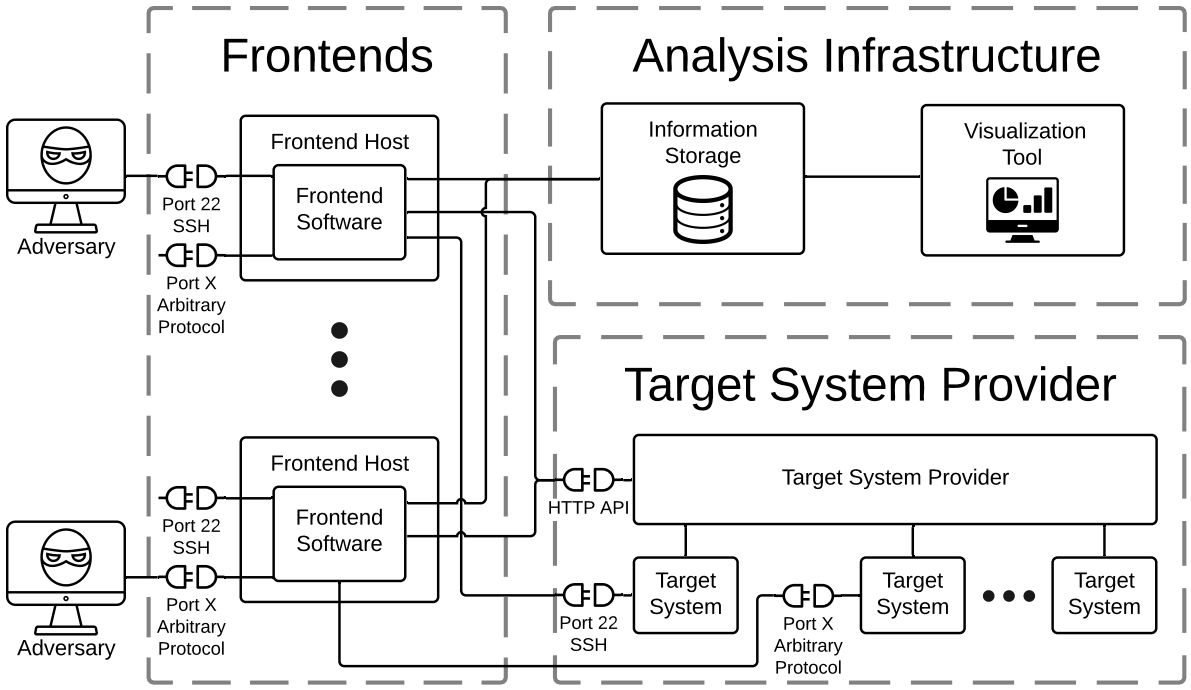

Figure 3.1: A schematic overview of the honeypot system's finished design.[1]Each frontend may concurrently connect to more than one target system.

As shown in Figure 3.1, the honeypot system consists of multiple distinct components: frontends, target systems, a target system provider, information storage, and a visualization tool. Each component serves a different purpose and aims to fulfill different parts of the established system requirements.

In this design, a target system is defined as an environment where adversaries will potentially perform malicious actions. This environment should, in some way, be regulated to minimize the impact of potentially malicious actions. It could, for example, be provided by a real system, a virtual machine, or a container. A target system will typically be running one or more exposed network-based services, for example, a web server or an SSH-server, as seen in Figure 3.1.

3.2.1 Information storage and visualization

Due to the fundamental concept that a honeypot should be able to collect information about adversaries, a place to store the information was essential (see Figure 3.1). As specified in the requirements, the information storage had to be structured for easy access to and retrieval of information. After discussing alternatives for information storage, a

[1]"Hacker screen" by Tom Fricker, "Online analytics" by ProSymbols, and "Linecons database" by Designmodo are licensed under CC BY 3.0 US, CC BY 3.0 US, and CC BY 3.0, respectively.

database management system (DBMS) was deemed fitting, as it allows for structuring the database itself [28]. Additionally, it also supports the requirement regarding storing a variety of information. An alternative would, for example, have been to store log files, but this would have made it more complicated to structure and retrieve the data, therefore it was dismissed.

Additionally, to comply with the requirement regarding visualization for the end-user, some form of visualization tool (see Figure 3.1) would have to be incorporated with the honeypot system. The end-user may not know how to query the DBMS for information. Therefore, a set of pre-defined queries would have to be packaged alongside the visualization tool.

3.2.2 Frontend

The frontend component of the honeypot system will be responsible for accepting and establishing connections with adversaries. As can be observed in Figure 3.1, the frontend is the only component where adversaries will establish a direct connection to, and thus it will also be a key component for collecting information. With regard to the scope of this project (see section 1.2), the frontend shall implement the SSH protocol and behave as a normal SSH server.

In accordance with the requirement that the honeypot should be modular, the decision of separating the frontend from other components of the honeypot system was made. This design allows for several frontends to run in parallel, which supports collecting a larger quantity of information.

To adhere to the requirement of creating a high-interaction honeypot commands sent by an adversary needed to be executed. Since these commands could potentially be dangerous, it was evident that they could not be executed on the same host machine the frontend was running on. Instead, these commands were to be relayed to a target system where they could be executed safely.

During a discussion around whether to have the frontend run virtualization or containerization software to provide target systems for attackers, it was decided to delegate this responsibility elsewhere. Requiring individual frontends to run virtualization or containerization software would significantly increase the hardware requirements and oppose the goal of running the honeypot on low-resource systems. Furthermore, by delegating the responsibility of acquiring and running target systems elsewhere, more flexibility is provided for the end-user to implement their own centralized system responsible for allotting target systems.

3.2.3 Target system provider

As was specified in the requirements of the design, the honeypot was to be implemented as a high-interaction research honeypot. Therefore, the honeypot would need target systems for adversaries to execute their commands in without rendering the host system inoperable or causing harm to other systems. Since it was decided that the frontend would delegate the responsibility of acquiring and running target systems elsewhere, the concept of a target system provider was introduced. Each time a target system is needed, the target

system provider would have to provide this target system and send back the connection details to the frontend. Derived from one of the project goals, this target system shall be able to imitate atleast one type of system.

Since an attacker was not supposed to be able to determine that the system is a honeypot, the target systems have to be provided clean and untouched. In addition, the attacker was not supposed to be able to render the host system inoperable, which led to the idea of using virtual machines. One problem identified with VMs was that they require at least a few seconds to start up, which could be solved by having a pool of VMs ready for each adversary. However, this approach would require the target systems to be cleaned from alterations after each use.

Instead, containerization offered a solution which requires less maintenance for the target systems. Containers could be started and stopped dynamically as their small resource footprint resulted in short start-up times. They could also be discarded after use, which made cleanup unnecessary.

3.2.4 Inter-component communication

Due to separating the system into a frontend and a target system provider, each potentially running on separate hosts, these two components needed a way to communicate. It was decided that the components should only require a network connection for communication to keep them as decoupled as possible. Therefore, a network protocol for interacting with a target system provider was deemed necessary. Since HTTP is a commonly used protocol for communication between services, it was decided that the protocol would be HTTP-based.

The following protocol was designed: As the frontend requires a target system to serve an incoming attacker connection, it sends a request to the target system provider. The target system provider prepares a target system and replies with connection details, allowing the frontend to proxy the attacker's connection to the target system. When the attacker decides to close the connection, the frontend sends a request to the target system provider to give up the target system previously acquired for the attacker's session. Information collected by the provider about the session in the target system is returned to the frontend for logging. This high-level overview of the protocol indicates it needs to be capable of two operations:

- One operation to allow asking to be provided a target system. Once the target system is ready, a unique identifier for the target system and the network address and port for connecting to the system should be returned.
- One operation to allow giving back a target system, identified by its ID, to the provider. Once the target system has been given back, potentially collected information about the target system and the actions performed by the attacker should be returned.

As can be seen in Figure 3.1, the frontend has one connection to the target system provider and one separate connection to each target system it has acquired. The connection between a frontend and a target system is used by the frontend to proxy attacker traffic to

a network service exposed by a target system. An alternative design could have been to proxy all attacker traffic coming from the frontend through the target system provider into each target system. However, this was decided against since the chosen design allows the HTTP target system provider protocol to be kept as simple as possible. Additionally, the chosen design makes it easier to support new protocols in the future since the target system provider does not need to concern itself with what protocol is being used between the frontend and a target system.

4

Development and implementation

This chapter presents the implementation of the honeypot design proposed in chapter 3. The chapter aims to clarify the decisions made while implementing the design and the underlying reasoning.

At the beginning of the development, Grafana [29] was planned to be used to fulfill the requirement of a visualization tool. However, due to time constraints this had a low priority and was not implemented.

4.1 Development tools and technologies

The following section introduces technologies used in the honeypot's implementation and tools used to assist the development.

4.1.1 Programming language

Python was the programming language chosen to implementation the frontend and the target system provider. The primary reason was that the group was interested in learning to program with Python. The secondary reason was the vast variety of already existing libraries that could be utilized to speed up the development process. The Python package index lists (as of 2021-05-05) over 300 thousand projects [30].

4.1.2 Container manager

Due to the modular nature of the honeypot architecture, three separate components (frontend, analysis infrastructure, and target system provider) would be required to be run for all the features of the honeypot to be fully operational. To account for this inconvenience, the group felt that a tool was required to assist with the deployment of the honeypot. The tool settled on was Docker due to two reasons: Firstly, each component of the honeypot could be packaged inside Docker images. This would provide an easy way for anyone to deploy any or all components of the honeypot by downloading and running pre-packaged Docker images. Secondly, the Docker Compose tool would allow deploying and managing several images, thus all components of the honeypot, by running a single command [31].

4.2 Information storage

Following the design decision to include a database management system (DBMS) for information storage, both the type of DBMS and a design for the structure of the stored information had to be decided. The following sections explain the factors that were taken into account during the decision-making process and the results.

4.2.1 Database management system

It was decided that the open-source object-relational DBMS PostgreSQL would be used for the system. The main reasons for this decision were mostly a matter of ubiquity, good compatibility, high familiarity and strict time constraints.

Firstly, PostgreSQL is a popular and widely used database management system [32] with support for ensuring data integrity in a relational database design [33]. For instance, PostgreSQL has a wide variety of data types [34] allowing table columns' values to be appropriately constrained to only well-formed values with regard to the problem domain.

Secondly, PostgreSQL was found to work well with other technologies that were planned to be used. Multiple PostgreSQL client-side libraries for Python exist, with the most popular and recommended library being psycopg2 [35]–[37]. Furthermore, there is an official Docker image for running PostgreSQL [38] and Grafana, the tool that was planned to be used for information visualization, has a built-in data source plugin for PostgreSQL [39].

Ultimately, due to the development phase of the project being time-constrained, the DBMS had to be set up and working, including a first design, within a couple of weeks after the overall system design was finished. Since the group members had previous experience with PostgreSQL and SQL, and it seemed to work well within the context the group intended to use it in, further alternatives such as MariaDB or MySQL were not considered.

4.2.2 Choice of information to collect

As a prerequisite to designing the database, the first step was to consider what information could be useful to collect. Due to these considerations being performed early during the development, it was not fully known what information the honeypot would be capable of collecting. The decisions made were therefore partly speculative.

When considering what information would be useful to collect, one of the main thoughts was what events occur during an attack. The items below are the core events followed by the reason why they were included in the system:

- **When an adversary connects:** The source IP address and port of the connecting adversary could be collected to provide information on where the attack originated from. Furthermore, collecting auxiliary data about the adversary, such as their geographic location, would provide information about what country the attack originated from. Due to the possibility of deriving information from an IP address at a later time, using third-party tools such as Shodan [17] and RDAP [40], it was

decided that auxiliary data would be added afterwards and only the IP address and port would be collected by the honeypot.

- **When an attacker disconnects**: Storing the time of the disconnection is useful to determine the duration of a session.
- **When authentication is attempted:** Which credentials are used by the adversary to attempt authentication would be collected. One reason to collect this information is to allow analysis of commonly used credentials, which furthermore allows drawing conclusions such as what credentials should not be used in any important system.
- **When a shell command is run:** What command was run would be collected to see what type of attack is performed while the adversary is connected to the system.
- **When a file is downloaded:** The contents of the file and where it was downloaded from, e.g., a website, would be collected. This is to allow further analysis of programs used in an attack.

All these occurrences are associated with a point in time and therefore it is important to store the timestamp of each one. A potentially interesting usage for the timestamp could be to allow filtering and aggregating. It could be to find what time of day most attacks occur. Additionally, modified files and raw network traffic were also considered to be stored in the database, but were ultimately not included due to time constraints.

4.2.3 Database design

As mentioned, the initial thought followed when designing the database was to consider what data would be useful to collect and, further, what constraints should be imposed on the included data to maintain a good structure and data integrity. Thereafter, the set of data chosen to be included was translated into entities and relationships that were made between the entities using entity-relationship modeling. Furthermore, to implement the database design in PostgreSQL, the created entity-relationship model was translated into a database schema consisting of SQL data definition language `CREATE TABLE` statements.

When structuring the data it was mainly translated into the entities `Session` and `Event`. These were found to be inherent structures in the information chosen to be collected. `Session` encapsulates a span of time where an attack occurs that is associated with a single attacker connection. `Event` is a distinct occurrence in time directly or indirectly caused by the adversary during a `Session`.

`Session` is protocol-agnostic to allow information about new protocols, beyond SSH, to be stored in the future. However, `Session` still stores which protocol was used to make sure SSH-specific events may only be associated with an SSH session. To store information about a session specific to SSH, `SSHSession`, which is a sub-entity to `Session` was additionally added.

Since all events belong to a session and a moment in time when they occurred, the `Event` entity is a super-entity of every type of event. This avoids introducing redundancy in the design, which would make the design harder to use in queries and harder to maintain.

As mentioned in section 4.2.2, it was desirable to store the event of a file being downloaded. For this purpose the `Download` entity was introduced. When introducing this entity there were concerns about the file size of the downloaded files. If the same type of attack occurs more than once, the same downloads would likely occur. Consequently, if the contents of a downloaded file were to be stored directly in `Download`, the same file contents would potentially be stored multiple times, which would be wasteful. To remedy the problem, the `File` entity was introduced. It stores the contents of a file and a hash of the file, which is referenced by the `Download` entity.

Additionally, there were concerns about the risk of storing illegal file contents. The `data` attribute of the entity `File` may be set to `NULL` if the contents of the file is decided to not be saved. Not saving the contents would disallow analysis of a download. However, since the hash is always stored to identify files, it could be used to look up third-party analysis of files [41].

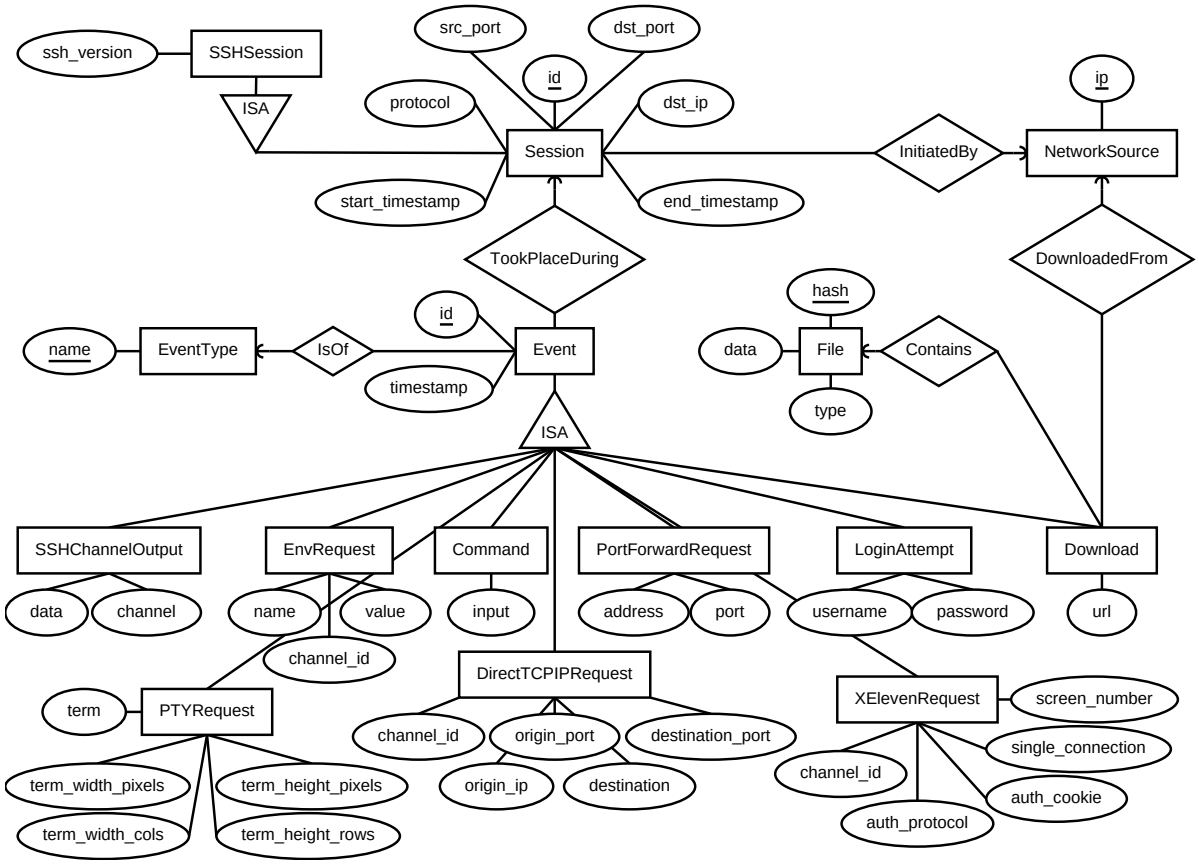

Figure 4.1: Entity-relationship model for the honeypot's database design.

At the end of the development the entity-relationship model seen in Figure 4.1 had been constructed. Each rectangular shape represents an entity and the oval shapes connected to an entity are its attributes [42]. An underlined attribute name indicates the attribute

used to identify an entity. The diamond shapes connecting two entities together represent a relationship between the entities. Finally, a triangle shape labeled “ISA” represents a sub-entity relationship where one entity inherits its attributes and identity from another and potentially extends the super-entity with its own attributes. For a more detailed view of the database design, including attribute data types and descriptions, see Appendix A.2.

4.3 Frontend

The frontend refers to the component that adversaries establish a direct connection to. It is intended to behave as a regular SSH server with the exception that events during SSH sessions will be logged and saved to the database.

Before development on the frontend would begin, it was critical to find an existing Python SSH library. Developing a new library would mean sacrificing valuable time which could be spent on the fundamental goals of the project described in section 1.1. When in search for a library it was deemed preferable that the library was mature and actively developed in order to ensure stability. With these goals in mind, the Paramiko SSH library was chosen [43]. Paramiko brands itself as being the leading native Python SSHv2 protocol library, the is claim supported by the almost 7 000 stars on GitHub [44]. Furthermore, the library has been developed since 2003 and has had commits on GitHub as recently as of February 2021 [44].

4.3.1 Managing SSH connections

In order to meet the requirement of serving several adversaries concurrently, a plan for managing potentially simultaneous SSH connections had to be put forward. The plan consisted of two components: A Connection Manager and an SSH Server. The Connection Manager listens for connections and establishes SSH sessions with adversaries connecting on port 22. The Connection Manager would then create and start a new SSH Server component to deal with further events over the SSH protocol. More specifically, the SSH Server component implements the `ServerInterface` class from Paramiko, where SSH-specific events such as login attempts and SSH requests can be handled.

During continued development and testing of the frontend software, it was realised that a third component would be required to better manage all concurrent SSH sessions. This new component was named Transport Manager and it keeps an up-to-date list of all active SSH sessions. With it, new functionality such as monitoring the life-cycle of SSH sessions or manually terminating individual SSH sessions became possible.

A visual representation of how an SSH connection is managed internally is pictured in Figure 4.2. Shown in the picture is the Transmission Control Protocol (TCP) and SSH handshake that occurs with the Connection Manager component. After the handshaking is completed, the SSH Server component takes over responsibility of serving the connection according to the SSH protocol, and the connection is relayed to the Transport Manager component.

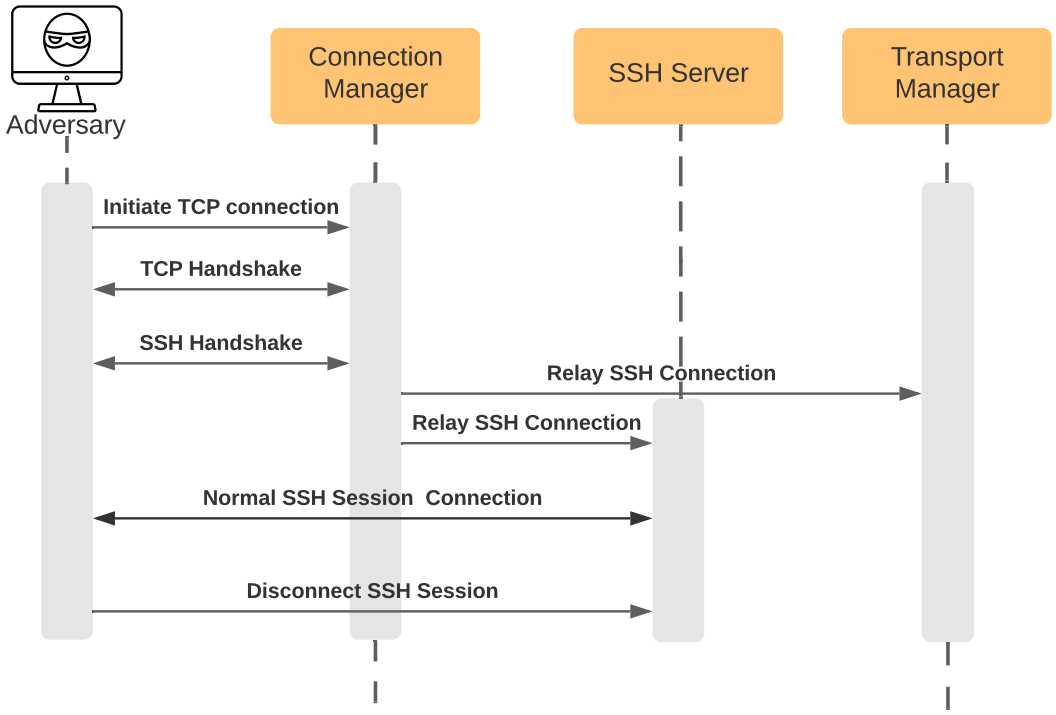

Figure 4.2: Sequence diagram illustrating SSH session connection establishment.[1]Time flows from top to bottom.

4.3.2 Proxying SSH

When implementing the SSH proxy, it was initially decided to only forward SSH requests to the target systems that could or would lead to the execution of a command as this would allow adversaries to interact with a target system. Two requests exist for this purpose: the exec and shell request, further explained in section 2.2.2. Also mentioned in section 2.2.2 is that these requests are sent over SSH channels of type "session". If the frontend receives a request to open an SSH channel of type session, an identical channel would have to be opened towards the target system. Thereafter, the two channels would have their data proxied between each other.

After further testing, it was also decided to forward the pseudo-terminal and window dimension change requests. The pseudo-terminal requests were forwarded to allow adversaries to allocate pseudo-terminals. The window dimension change requests were forwarded to ensure that text is rendered according to adversaries' preferences. Both were important in order to provide an experience similar to what they might be expecting.

To forward the requests received, a target system first had to be acquired. To acquire this target system, the frontend uses the API described in section 4.5 to send an `AcquistionRequest` to a target system provider. A target system would then be provided to which an SSH connection would be established using the Paramiko SSH client.

The proxying behavior is illustrated in Figure 4.3. In it, an adversary connected over SSH to the frontend sends a request to open a channel. The frontend then uses the target

[1]"Hacker screen" by Tom Fricker is licensed under CC BY 3.0 US

system provider protocol to acquire a target system, to which an identical channel open request is sent. Later, the adversary sends an exec request, which is also forwarded to the target system, and the result is returned to the adversary.

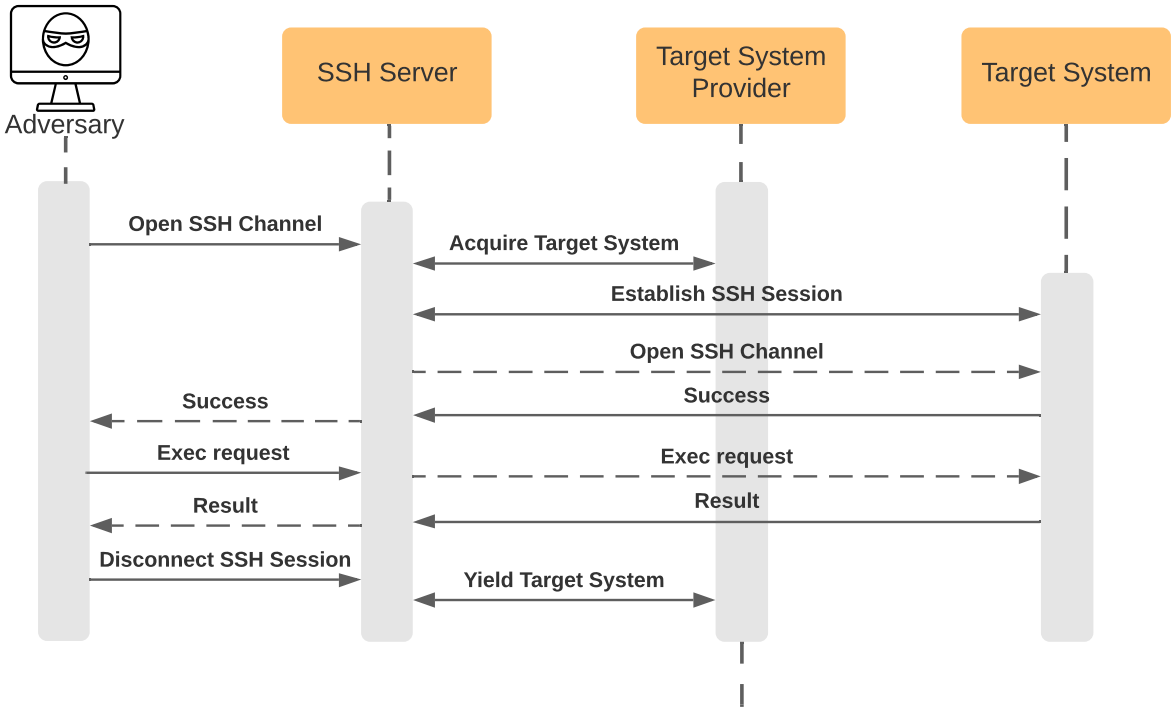

Figure 4.3: Sequence diagram illustrating target system acquisition and SSH session proxying.[2]Dashed arrows indicate proxied requests. Time flows from top to bottom.

4.3.3 Collecting SSH session information and events

The collection of SSH-related activity became a relatively easy task thanks to the Paramiko SSH library. The Paramiko `ServerInterface` class provides all data that SSH requests contain in method parameters. For example, if a login attempt were to occur over SSH, the `check_auth_password` method, shown in Listing 4.1, would be called in our SSH Server implementation. The same is true for other requests; for example, an exec request would have the method `check_channel_exec_request` called, also shown in Listing 4.1. It then became a matter of extracting the data associated with the request received and then relay it to the logging component described in section 4.3.5.

```
# This method is called when a password login attempt occurs
def check_auth_password(self, username, password):
    pass
# This method is called when an exec request is sent over a channel
def check_channel_exec_request(self, channel, command):
    pass
```

Listing 4.1: Examples of Paramiko `ServerInterface` methods.

[2]"Hacker screen" by Tom Fricker is licensed under CC BY 3.0 US

4.3.4 Collecting adversary commands and output

During SSH sessions, clients have two ways of executing commands on the remote host. The first one is by sending an exec request, described in section 2.2.2. In this case, the associated command to execute is sent along in the request, which makes it trivial to collect with Paramiko's `ServerInterface` mentioned in section 4.3.3. The other way would be for the client to first send a shell request and then execute the command in the remote shell supplied by the target system. In the latter case, extracting the command becomes more complex because the whole command is not included inside a single request. Instead, a stream of bytes is continuously sent over the SSH channel to the shell on the remote machine.

To collect commands executed inside a shell, a way of determining where a command begins and ends in the stream of bytes sent over a channel had to be discovered. Through trial and error, it was discovered that the stream of bytes sent over channels by adversaries was UTF-8 encoded data and could thus be converted to regular text. Furthermore, at the end of each command, the carriage return (CR) character would be sent.

With this knowledge, an implementation of extracting commands from shell sessions was put forward and used. To handle the continuous stream of bytes flowing through channels, a buffer for each channel would be allocated. When data arrives on a channel, it would be converted to a string and saved in the corresponding buffer. If however, the data contained a CR character, every character received prior to that CR character would be treated as a command and be sent to the logging component, described in section 4.3.5. The buffer would then be emptied in anticipation of future commands. Furthermore, it was also decided to save all data arriving through channels opened to target systems in the database as this could possibly be used to replay SSH sessions at a later date.

4.3.5 Logging information to database

To connect and log information to the PostgreSQL database server mentioned in section 4.2, a PostgreSQL Python library was needed. The library settled on was the psycopg2 library, briefly mentioned in section 4.2.1.

The following three-step process of logging information to the database was initially decided upon due to its simplicity:

1. When an event occurs, connect to the database.
2. Insert information in the relevant database table by executing an SQL insert statement.
3. Disconnect from the database.

This simple approach worked well while testing the whole honeypot system locally. However, it turned out to be problematic on other devices with a less stable Internet connection. Continuously connecting and disconnecting to the database each time an event occurred was time-consuming and held up the execution of the thread logging to the database. Additionally, since data was continuously inserted, incomplete SSH sessions would be stored in the database if a single insert were to fail during the course of an SSH

session.

Because of these problems, a new way of logging information into the database had to be put forward. Instead of continuously connecting and disconnecting to the database, a connection pool would be used. This connection pool would allow to recycle established database connections by putting them back into the pool instead of disconnecting them. Additionally, each SSH session to the frontend was to receive a unique connection to the database, which was to be returned to the connection pool when the session ended.

To eliminate the possibility of incomplete sessions inside the database, PostgreSQL transactions were used. Database transactions are a fundamental concept to database systems that allow to bundle several statements into a single all-or-nothing operation [45]. With transactions, insert statements relating to the same SSH session could thus be bundled together. If a single one of these statements were to fail, the database transaction would not be committed and no data relating to that SSH session would be saved. The following steps highlight the final approach for logging data to the database:

1. When SSH session begins, acquire connection to the database through a connection pool.
2. Begin database transaction.
3. For each event, insert information in the relevant database tables by executing SQL insert statements.
4. When SSH session ends, commit database transaction.
5. Return the database connection to the connection pool.

4.4 Backend

The backend is an implementation of the target system provider introduced in the design chapter. It provides target systems where an attacker can execute their actions. Since the target systems execute the adversaries' commands, this was where most of the concern regarding security was focused. Therefore, the implementation and security considerations for the backend are described in this section.

4.4.1 Target containers

The target systems mentioned in section 3.2 were decided to be implemented as Docker containers, and will further be referred to as target containers. Docker containers were chosen because the target systems needed to be dynamically started and stopped, which required the target systems to be lightweight to decrease start-up time. On the other hand, if a virtual machine was used, it would have to start up an entire operating system. Starting and stopping an operating system could be a demanding task [46], and could therefore not be utilized as it would cause a delay whenever the frontend requested a target system for an adversary.

The target containers were designed to provide the adversaries with an environment in

which they could execute their potentially dangerous actions. The containers had to support executing arbitrary commands and be customizable in terms of the kind of system they were designed to imitate. A Docker image that has a built-in OpenSSH-server and could easily be configured was chosen: *ghcr.io/linuxserver/openssh-server* [47]. The image also supported logging in as root and allowed password authentication, as was needed for the frontend to connect to target systems.

Two primary configuration changes were made to the Docker image to make the container environment appear legitimate to an adversary. Firstly, a user account and a home directory is created using the same credentials as the adversary provided to the frontend. Secondly, several decoy services are started to imitate an active system.

It is important to note is that there were no protections in the container for preventing the adversary from attacking other systems from within the target container. As anything that was placed inside the container could be modified or removed by an attacker, restrictions for preventing attacks on other systems had to be provided elsewhere (see section 5.1). In addition, restricting too much could give away that the system is a honeypot [8], so a balance has to be found.

4.4.2 Controlling target containers

As the backend needed a way to provide the frontend with target systems, a controller module was implemented, representing the target system provider from section 3.2.3. Since the honeypot was supposed to be easy to use, the controller module was packaged inside a Docker container. Considering the controller had to control the target containers, access to a Docker socket from where the Docker daemon could be controlled was required. There seemed to exist two ways to provide access to a Docker socket inside a container [48].

The first way considered involves a parent-child hierarchy where the controller container runs its target containers inside itself, also known as Docker in Docker. However, due to problems such as security concerns and container creation problems [48], the Docker in Docker method was ultimately not used. Therefore, the second method was used, which involved a sibling relation between containers. This sibling relation is also known as the socket solution [48].

The socket solution means that the controlling container is created with access to the Docker socket on the host system. The Docker socket gives it control over the Docker daemon, which is used to create, start, stop, and destroy other containers. The target containers are created on the same level as the controlling container, instead of being created as its children [48].

To assist with the management of containers, volumes, and images, the Docker SDK for Python [49] was used. Docker SDK for Python is maintained by Docker Inc. and provides a way of interacting with the Docker daemon through Python code [49].

Upon receiving a request to acquire a target system from the frontend, the controller container prepares a target container with the username and password provided by the frontend. Since the login credentials are provided by the frontend, the environment inside

the target system can be adapted to look more like what the adversary expects. The correct username is shown at the command prompt and the password used to log in will be accepted for running the `sudo` command. When prepared, the container is started and its port number, assigned by the Docker daemon from the IANA unregistered ports 49152–65535 [50], along with the container's ID and IP address of the backend host is sent back to the frontend.

When receiving a yield request for a target system from the frontend, the backend stops and destroys the corresponding target container, releasing its allocated resources. The environment variable `KEEP_TARGET_SYSTEM_VOLUMES` allows the user to control whether the container's volumes should be kept. Due to ethical concerns it was decided that the volumes are not kept by default.

4.4.3 Network traffic capture

Since the controller container has full control over and insight into the activities of the processes running in target containers through the Docker daemon, it was decided it would have the additional responsibility of capturing network traffic originating from a target container. The main goal of capturing network traffic is to gain insight into what files are downloaded by an attacker and other potentially interesting interactions with the outside world over the network. In the pursuit of achieving this goal, two primary problems were encountered:

- Network packets originating from all processes running in a specific target container need to be captured. To ensure an attacker cannot detect or bypass the network traffic being captured, the technique used for capturing must not be visible to any process running inside the target container.
- According to a 2020 report from the European Union Agency for Cybersecurity (ENISA), the overall use of encrypted internet protocols is steadily increasing [51]. Furthermore, ENISA report that more than 70% of malware will use encrypted protocols, like TLS, IPSec, or custom encryption schemes, in 2020. Therefore, to enable analysis of the captured traffic, some technique for decrypting captured network traffic is needed.

Capturing packets

The library libpcap, along with the command-line tool tcpdump, for capturing and filtering packets at the link-layer of the operating system's network stack [52] was chosen to be used. This choice was made due to multiple reasons. Libpcap is a widely used library by multiple well-known computer networking tools, such as Wireshark [53] and Nmap [54]. Furthermore, the pcap file format, originating from and used by libpcap to store network captures, is widely supported by many networking tools [55].

As an alternative approach, a proxy server, e.g., mitmproxy [56], PolarProxy [57] or SSLsplit [58] for capturing packets was also considered. However, this approach proved to be less user-friendly, more limited and potentially easier to detect and bypass than the libpcap solution.

Firstly, the mentioned proxies only support proxying either TCP or HTTP, compared

with libpcap, capable of capturing information from all layers of the network stack for any protocol. Secondly, the proxy server would run either directly on the host machine for the backend or, for ease of use, preferably inside a Docker container. For the proxy to capture any traffic, the traffic would have to be explicitly redirected to the proxy server [59] from all target system containers. According to the authors of the mentioned proxies, the recommended way for redirection on Linux is to use the Linux firewall [57], [60], [61]. Firewall configuration would have to be performed outside the container, directly on the backend host machine, to ensure the proxy cannot be detected by a process inside a target system container. No method was found for configuring the redirection on the host that would work reliably cross-platform without requiring manual setup by the user. Lastly, in order for the proxy to read encrypted traffic, a custom certificate authority (CA) would need to be installed in every target system image to ensure programs communicating through the proxy automatically trust the proxy's certificates [57], [62]. Unfortunately, installing a custom CA is performed differently for each operating system, programming language, and application, which complicates creating new target system images and supporting new systems that the honeypot could imitate. Further exacerbating the problem, an attacker could run a custom application that does not honor the custom CA, thereby preventing decryption of the application's network traffic.

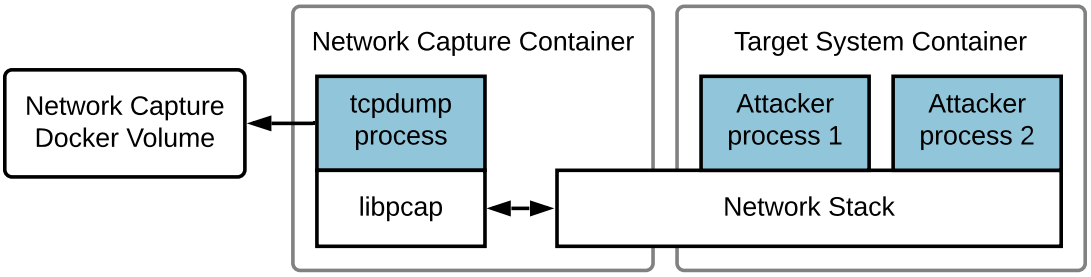

Figure 4.4: Network capture using libpcap and tcpdump in a separate Docker container. The network capture container attaches to the network stack of a target system container. A pcap-file is persisted in a Docker volume attached to the network capture container.

With the libpcap solution a dedicated network capture Docker container containing a single tcpdump process is used, as seen in Figure 4.4. There is a one-to-one correspondence between target system containers and network capture containers. For each target container created a corresponding capture container is created.

The Docker network of a newly created capture container is set to `container:<name>`, where `<name>` is the name of its corresponding target container. This causes the capture container to share the network stack used by its corresponding target container [63], [64]. A primary benefit of isolating the tcpdump process in a separate container is that while tcpdump can read all network traffic passing through the target container's network stack, the target container cannot detect the presence of the tcpdump process.

As a target container is instructed to start and stop, its corresponding capture container is instructed to start and stop alongside it. The tcpdump process is started with the `-w` flag, indicating that it should write a pcap-file containing the captured packets to disk [65]. The use of a one-to-one mapping combined with sharing the network stack means the output of tcpdump is naturally restricted to the network traffic of a single target system

container. Tcpdump writes the pcap-file to a Docker volume. This volume is persistent on the host system, allowing for future analysis of the captured network traffic.

Decrypting by extracting TLS session keys

To allow decryption, multiple techniques for extracting TLS keys from Docker containers were explored. However, none of the techniques were implemented due to time constraints. Furthermore, the techniques explored only cover the TLS protocol and are therefore not a complete solution. Possible future work regarding network traffic decryption is presented in section 6.7.

4.5 Frontend-backend communication

As discussed in section 3.2.4, a protocol for communicating with the target system provider had to be designed and implemented. This section discusses the choice of tools used to specify and implement the protocol, how it was designed using the chosen tools, and how it is utilized by the frontend and implemented in the backend.

4.5.1 API library and specification tools

When researching and choosing tools, the initial thought was that the protocol should be formally specified independently of the programming language used in a client or server implementation. This would in turn make it possible to use tools that automatically generate parts of the HTTP client and server implementation for any language supported by the tool. This automation was desired to speed up the development process and facilitate easier implementation of new target system providers or applications using the target system provider in the future.

GraphQL [66], OpenAPI [67] (with Swagger [68]) and gRPC [69], were the main options that were compared. All three options allow specifying the protocol in a language-agnostic way [67], [70], [71]. However, GraphQL was found to mainly be focused on providing a solution for allowing the client to query specific pieces of data [72], for example when retrieving account information for a user interface. Since this was not needed GraphQL was disregarded.

Both OpenAPI and gRPC provide similar functionality, including an interface definition language (IDL) for specifying services and remote procedures, allowing for remote procedure calls (RPCs) [73]. Essentially, RPCs allow calling a function that transparently sends a message and cause a function with the same signature to be run on a different networked system. For generating client and server stubs, gRPC uses Google's Protocol Buffers library [71] while Swagger Codegen can be used together with OpenAPI [74]. The differentiating factor, that lead to gRPC being chosen, was that gRPC abstracts away how the RPC-model is mapped to HTTP, while OpenAPI requires the protocol specification to contain details about HTTP [73].

4.5.2 Target system provider protocol

As mentioned in section 3.2.4, the target system provider protocol needs to support two types of operations:

- Acquiring — requesting to be provided a target system
- Yielding — giving back a target system to the provider

The protocol was implemented in the Protocol Buffers language [75] to be used with gRPC, as seen in Listing 4.2. The protocol specifies a gRPC service named `TargetSystemProvider`. The service has two remote procedure calls: `AcquireTargetSystem` and `YieldTargetSystem`.

```
service TargetSystemProvider {
    rpc AcquireTargetSystem (AcquisitionRequest) returns
    ↪  (AcquisitionResult) {}
    rpc YieldTargetSystem (YieldRequest) returns (YieldResult) {}
}
```

Listing 4.2: Target system provider protocol definition in Protocol Buffers language.

`AcquireTargetSystem` defines a request to be provided a target system. Since acquiring a target system could fail, e.g., due to resource exhaustion or configured limits, it was decided that the provider should return gRPC error code `UNAVAILABLE` [76] if no target system can currently be acquired. In this case, the client may retry calling `AcquireTargetSystem`.

The frontend may accept attacker connections with any SSH credentials. To make sure the provided target system behaves as expected by the attacker, a call to `AcquireTargetSystem` is given an `AcquisitionRequest`, as shown in Listing 4.3, containing the username and password to be used by the target system that will be acquired. Once a target system has been acquired, an `AcquisitionResult`, as shown in Listing 4.4, is returned containing a unique identifier for the system and the address and port for connecting to the system using SSH.

```
message AcquisitionRequest {
    string user = 1;
    string password = 2;
}
```

Listing 4.3: Data type sent when calling `AcquireTargetSystem`.

```
message AcquisitionResult {
    string id = 1;
    string address = 2;
    uint32 port = 3;
}
```

Listing 4.4: Data type returned when calling `AcquireTargetSystem`.

`YieldTargetSystem` defines a request to give back an acquired target system. The `YieldTargetSystem` call is given a `YieldRequest`, as shown in Listing 4.5, containing the identifier of the target system to be yielded. Upon yielding a target system a `YieldResult` is returned, as shown in Listing 4.6. `YieldResult` is intended to contain collected information about the target system and potential attacker actions. In the current protocol design no information is returned, however, the `YieldResult` allows this to be extended in the future.

```
message YieldRequest {
    string id = 1;
}
```

Listing 4.5: Data type sent when calling `YieldTargetSystem`.

```
message YieldResult {
    // Collected data about
    ↪ target system
}
```

Listing 4.6: Data type returned when calling `YieldTargetSystem`.

4.5.3 Frontend client and backend server

With the target system provider protocol specified using gRPC's interface definition language, the next step taken was to implement a client for the frontend and a server for the backend. The frontend would need a target system provider module, used by the frontend proxy, with an interface for provisioning target systems. It was desired that this interface would not expose the fact that the target systems are provisioned over a network. This would aid in not imposing a direct dependency for gRPC on other modules, and to facilitate other providers being implemented in the future. For the backend, a gRPC server running the `TargetSystemProvider` service for handling requests to provision target systems would need to be implemented.

To implement the frontend module and the backend service, gRPC client and server implementation stubs were automatically generated using the Protocol Buffers compiler. Consequently, the compiler created Python functions corresponding to the RPCs defined in the protocol specification, to be used by the client. For the server an abstract Python class containing methods, corresponding to the RPCs defined in the protocol specification, is generated by the compiler.

The implementation of the frontend target system provider module directly calls the generated Python functions. Meanwhile, the backend service is implemented by inheriting from the generated Python class. The implementation of the class directly forwards requests to acquire and yield target systems to the Docker container controller module (described in section 4.4.2). For an acquisition request, once the controller module has started a target system container, the container's assigned port is retrieved. Finally, the port, the target system's IP address and a randomly generated identifier for the target system is returned from the service.

5

Deployment and results

This chapter shows the deployment, properties of the honeypot implementation and information collected after deploying the honeypot between 2021-04-07 and 2021-04-28.

5.1 Deployment

During the deployment phase of the project, the cloud-based service Azure was used. Azure offers the ability to create and manage virtual machines [77], which was used to set up the honeypot software on different machines.

Each component (frontend, backend, and database) of the honeypot was placed on a separate VM. The reason for deploying the frontends separately was to facilitate collecting a greater quantity of information. Each VM received its own unique IP address and, therefore, had a greater chance of being found by an adversary.

For each new VM, Docker was installed together with the honeypot software. Additionally, on each frontend VM, the real SSH port was changed from 22 to 2121, so it is possible to administer the VM at a later time. Port 22 was assigned to forward an adversary to a target container. Furthermore, running a single Docker command in the directory of the desired component started it. There also exists a guide on GitHub[1]on how install Docker and run the components on a single machine or multiple machines.

Initially, three frontends were deployed on 2021-04-07 together with a backend and database. The VMs running the frontend were placed in Ireland, Sweden, and the US. Five additional frontends were deployed on 2021-04-17, located in Canada, India, Korea, the UK and the US. The first three frontends had a login success rate of 100%, while the latter five had a success rate of 20%. Figure 5.1 shows an overview of the components used during the deployment.

Since the target containers were unable to provide any protection or limitations to Internet access, a script was written for the host running the backend which modifies the firewall rules to provide some protection. The limitations included limiting the outgoing bandwidth and disallowing some protocols entirely, addressing the goal of disallowing our honeypot to be used to induce damage on other systems.

[1]https://github.com/Botnet-Honeypot/Honeypot

5.2 Properties of the honeypot

5.2.1 Modularity

The components frontend, backend, and database can be distributed over multiple host machines as only a network connection is required between them. During development the three components were all run on the same machine, which was observed to function well. Additionally, as Figure 5.1 shows, during deployment multiple frontends were run on separate machines and connected to one database and backend, which was also observed to function well. While it is possible to run multiple databases and backends simultaneously, each frontend can only be connected to one backend and one database at a time in the current implementation.

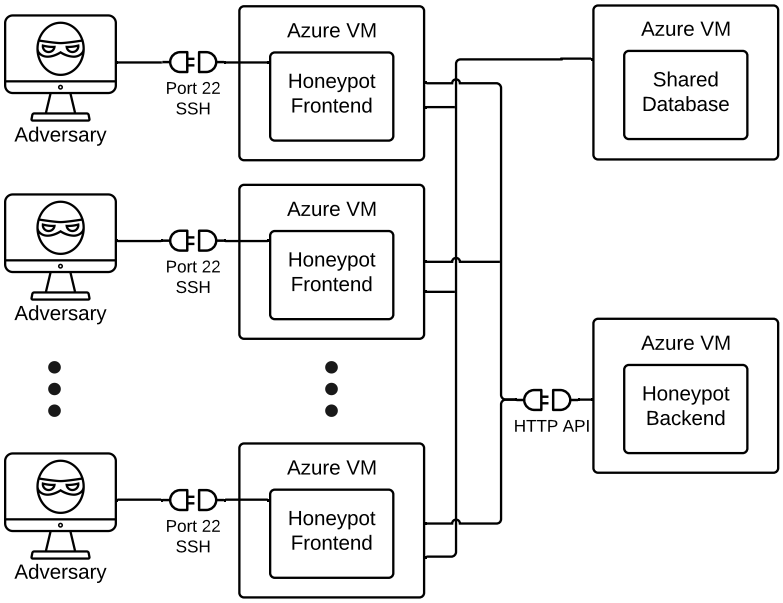

Figure 5.1: Example of how the honeypot's components can be distributed when deployed.[2]All frontends are connected to the same backend and database.

5.2.2 Detectability

Using automated testing utilities for detecting honeypots, such as Checkpot and Shodan [78], raises little concern for our systems being honeypots. Checkpot reported that the deployed systems have duplicate services running on different ports, namely SSH on ports 22 and 2121, which it marked as a red flag. Shodan did not raise any suspicion toward the deployed systems.

When the group tried to log in to the honeypot via SSH, it raised no concern as it behaved like any ordinary SSH session from the groups' experience. Since the target containers are full Linux systems, all commands attempted worked as expected. Additionally, as mentioned in section 4.4.1, the target containers are created with a user account with the same credentials as was used in the login request and thus provide a consistent experience.

[2]"Hacker screen" by Tom Fricker is licensed under CC BY 3.0 US

Command execution directly through “exec” requests or indirectly through a remote shell have been observed to work perfectly.

5.2.3 Performance and reliability

The frontends and database ran for almost a month on low-tier Azure instances with 1 GiB RAM and one processor core. The backend ran on a higher tier Azure instance with 16 GiB RAM and 4 processor cores, and throughout the deployment, the average processor usage was below 4%, with a peak of just under 8.5%. The deployed modules ran with no hiccups until they were stopped 2021-04-28 due to the time limitation of the project.

Artificial limits of 50 simultaneous connections from attackers to the frontends and 100 simultaneous connections for the database were set up. No performance issues were experienced using these configurations.

5.3 Information gathered

5.3.1 Login credentials

The honeypot collected 310 820 login attempts. Out of these login attempts, 5 059 were unique combinations. The most frequently attempted usernames are listed in Table 5.1a. As can be seen, the username “user” is heavily dominating. In addition, Table 5.1b shows the most frequently attempted passwords, where the password “user” is by far the most popular.

Table 5.1: The ten most frequently attempted usernames and passwords collected by the honeypot.

Username	Occurrences
user	132 028
admin	32 709
root	21 537
default	11 156
MikroTik	10 988
ubnt	10 769
profile1	10 712
user1	10 587
support	10 494
admin1	10 315

(a) Usernames used in login attempts collected by the honeypot.

Password	Occurrences
user	121 788
admin	29 093
password	16 135
1234	9 858
123456	8 923
1234567	8 240
	8 219
123	8 206
1	7 998
test	7 952

(b) Passwords used in login attempts collected by the honeypot.

5.3.2 Frequency of attacks

As can be seen in Figure 5.2, while the honeypot was deployed, it accumulated a total of 320 334 attacks. This does not correspond with the number of login attempts as

several login attempts can be made in one SSH session. As can be observed, the number of attacks during the first few days were low. As time passed, attacks increased more rapidly indicated by the small curve that occurs between days 6 and 10 of operation. The following subsequent days had roughly the same number of attacks, with a minimum and maximum difference of 477 and 7007, respectively.

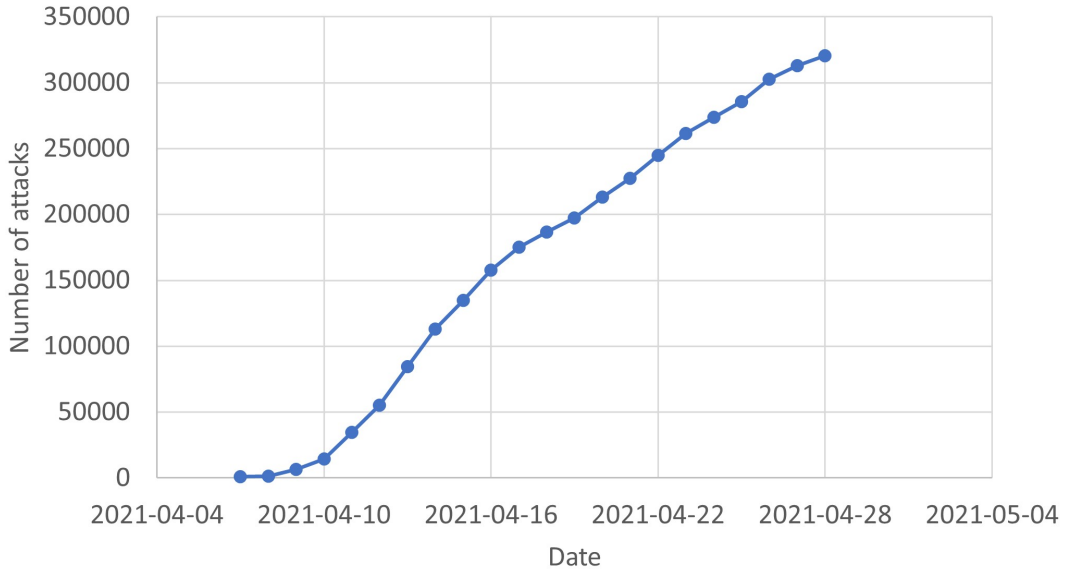

Figure 5.2: Number of attacks made toward the honeypot between 2021-04-07 and 2021-04-28. Each dot indicates a specific day in the given period.

Additionally, Table 5.2 shows the number of reoccurring attacks made from a single adversary. None of the adversaries in this table, executed any of the commands shown in Table 5.3. Instead, most of them executed `direct-tcpip` requests (see Table A.1).

Table 5.2: Number of reoccurring attacks grouped by adversary IP address. Information replaced with "x" has been redacted.

IP address	Attacks performed
x.x.255.205	6 610
x.x.86.169	5 571
x.x.86.178	5 362
x.x.86.212	5 353
x.x.86.216	5 325
x.x.86.206	5 297
x.x.87.53	5 221
x.x.86.221	5 210
x.x.86.168	5 165
x.x.87.51	4 960

5.3.3 Shell commands

The honeypot gathered 109 868 command executions. Table 5.3 shows commands that appeared over 1 000 times in the database, with some information replaced by uppercase words within <>. The redacted information could possibly be used to identify an attacker,

and was therefore excluded. As can be seen from the table, the distribution between commands was not very even. The remaining commands, which are not shown in the table, appeared less than 150 times each.

Table 5.3: Commands recorded at least 1000 times, along with the number of occurrences. Some information has been redacted, shown by uppercase words within <>.

	Command	Occurrences
1	`/system scheduler add name="U6" interval=10m on-event=` ↵ `"/toolfetch url=<URL> mode=http dst-path=` ↵ `file.rsc\r\n/importfile.rsc" policy=api,ftp,local,` ↵ `password,policy,read,reboot,sensitive,sniff,ssh,` ↵ `telnet,test,web,winbox,write`	88 015
2	`uname -a`	1 748
3	`uname -m`	1 323
4	`uname`	1 321
5	`ls -lh $(which ls)`	1 317
6	`top`	1 317
7	`free -m | grep Mem | awk '{print $2 ,$3, $4, $5, $6, $7}'`	1 317
8	`w`	1 317
9	`crontab -l`	1 317
10	`cat /proc/cpuinfo | grep name | head -n 1` ↵ `| awk '{print $4,$5,$6,$7,$8,$9;}'`	1 317
11	`cat /proc/cpuinfo | grep model | grep name | wc -l`	1 317
12	`lscpu | grep Model`	1 316
13	`cd ~ && rm -rf .ssh && mkdir .ssh` ↵ `&& echo "ssh-rsa <SSH_KEY>==mdrfckr"` ↵ `>>.ssh/authorized_keys && chmod -R go= ~/.ssh && cd ~`	1 303
14	`cat /proc/cpuinfo | grep name | wc -l`	1 270
15	`echo -e "<USERNAME>\n<PASSWORD>\n<PASSWORD>"|passwd|bash`	1 050
16	`echo -e "<USERNAME>\n<PASSWORD>\n<PASSWORD>"|passwd`	1 049

Most of the commands collected are generally found on Linux hosts, but some commands, like the one with most occurrences, are bound to specific manufacturers or operating systems. The `system scheduler` command is typically found in devices from the manufacturer MikroTik [79]. Although the command does nothing to the target system, some adversaries were observed to execute it several times. Commands with index 2, 3, 4, 6, 7, 10, 11, 12 and 14 return information about the system.

5.3.4 Common geographic locations

In total, the honeypot collected 5 899 unique IP addresses where attacks originated from. Figure 5.3 shows a heat map of the origin of attacks, based on IP address, from around the world. As can be seen, China holds first place and the United States holds second place. The accuracy of the map is limited to pointing out the country an attack originated from. However, more precise locations were also extracted from the data but are not presented and were not analyzed further due to ethical concerns and time constraints.

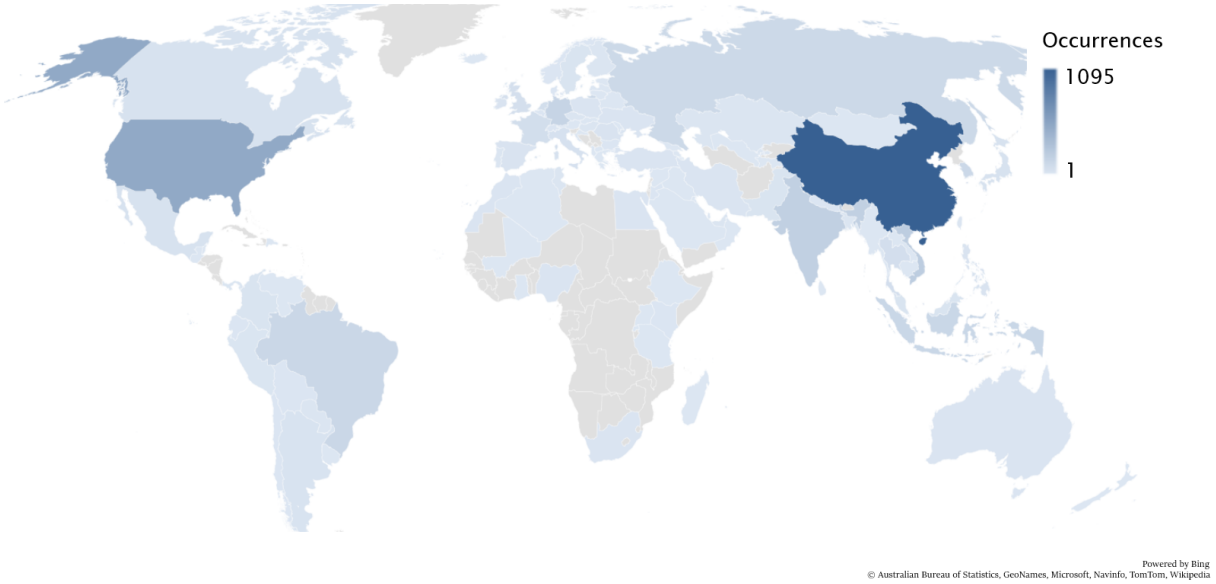

Figure 5.3: Heat map showing the number of unique IP addresses originating from each country, where a darker shade of blue means more addresses.

6

Discussion

This chapter discusses the resulting implementation of the honeypot system and evaluates its properties. Ethical concerns identified for the project and issues that could be further researched are also discussed.

6.1 Maintainability and modularity

As stated in section 1.1, the project's primary goal was to make a honeypot that is modular to enable it to evolve in the future. The system design presented in chapter 3 decouples the component responsible for target system acquisition and inspection from the component responsible for accepting adversary connections and therefore directly supports this goal of modularity. The only shared dependency between the frontend and the backend is the target system provider network protocol (see section 4.5). Therefore exchanging the frontend or backend would only require implementing this protocol. The choice to only require a network connection between the major components of the honeypot has been observed to provide flexibility in practice, as shown in section 5.2.1.

The honeypot design is intended to allow any system which supports SSH to be used as a target system. However, this is not possible in the current implementation, since the frontend is required to ask the backend for a target container. This problem is believed to be fairly straightforward to remedy due to the existence of the frontend target system provider interface, described in section 4.5.3. A simple implementation of this interface that returns the IP address of any SSH system could easily be implemented.

Another aspect of the modularity goal was to allow adding new systems to imitate and new protocols for an adversary to connect to. Since the target systems are implemented as Docker containers (see section 4.4.1), the ability to customize is ultimately limited by what can be run in a Docker container. Changing the image used by target containers is possible but requires modifying the backend code due to it not being configurable. Additionally, parameters injected into the container at startup, such as login credentials, need to be adjusted to fit other images.

Extending the system with another protocol adversaries can use also requires code changes, but this is to be expected. Since implementing any other protocol beyond SSH was out of scope for the project, no general interface was implemented in the frontend for supporting other protocols. This would be desired in the future to allow the proxying module to be reused and to allow a more plug-and-play approach to adding new protocols.

6.2 Ease of use

All of the honeypot's components run inside Docker, therefore, Docker is required to use the honeypot. This can complicate the process of a user wanting to use the honeypot. However, as mentioned in section 5.1, the honeypot's GitHub repository contains a guide on both how to install Docker and how to run the honeypot software. Since the guide exists, it can ease the process of getting the honeypot started for the user. Once Docker has been installed, all that is required from the user is to run a single Docker command, also included in the guide, to get the honeypot running.

An alternative solution for installing the honeypot could have been to distribute it with pip [30], making it easy to install for those familiar with pip. Nonetheless, Docker is still required as it is part of the honeypot, therefore, having Docker as an only requirement makes it easier on the user. Additionally, installation using pip would depend on the operating system and Python version the user is running.

As part of the project's purpose, the aim was to have the information easily accessible. The database that was implemented in section 4.2 supports this purpose since it is arguably well-structured as well as centralized in the honeypot system, which allows for easy extraction of data. However, to query the database, the user is required to know SQL. As the honeypot system currently has no support for users who do not know SQL, this could be improved in the future. The addition of a visualization tool along with pre-packaged queries (see section 3.2.1), would have made it easier for the user to digest the collected information. However, as mentioned at the start of chapter 4, time constraints prevented the implementation of the visualization tool.

6.3 Security

The honeypot relies on isolation between the target containers and the host to keep the host running reliably and securely. The security of the containers is based on the isolation provided by Docker, which is being actively discussed and researched. Bui states that Docker is relatively secure depending on how it is configured [80]. Yasrab gives examples of security mitigations for both Docker developers and end-users, such as running the containers with the lowest possible privileges [81]. One alternative to containers is virtual machines, which was discussed briefly in section 2.3.1. While virtual machines may offer better isolation between the host and guest operating systems, they did not fit the project's resource and startup time requirements.

As described in section 4.4.1, the target containers do not provide restrictions for what an adversary can do. As mentioned in section 5.1, a script was written which applies different restrictions depending on the type of traffic. While this provides basic restrictions, an argument can be made that attackers can still utilize the honeypot to attack other targets. Arguably, it would be best not to let the attacker initiate connections outbound from the honeypot at all, but this was decided against in section 4.4.1. The balance referenced there could use some refinement by conducting further research.

Another way of abusing the honeypot is to, for example, host services, mine bitcoin, or store files. The honeypot does not restrict such usage, and one solution would be to set

a time limit on how long a container may operate. Careful consideration of this limit would be required as the adversaries need to be able to carry out their actions before the container is closed.

6.4 Detectability

As mentioned in section 5.2.2, when an adversary logs in to the honeypot via SSH, the look and feel of a real system is provided. One potentially suspicious part is that the frontend has a separate IP address than the target system. It can be remedied by placing them both on the same network or, as mentioned in related work (section 1.3) have an encrypted tunnel connection between them to have them appear as being on the same network.

Another way to suspect the existence of the honeypot is if the adversary downloads a file or types a command that makes a lasting effect, leaves the honeypot, and later returns expecting to find the changes still there. Unfortunately, as mentioned in section 3.2.3 the containers containing the changes get destroyed after each use, and the changes are gone.

6.5 Information gathered

As can be seen in Figure 5.3, the majority of IP addresses that connected to the honeypot during the deployment were located in China. Similar observations have been made by Ryan J. McCaughey [82]. In the paper from 2017, McCaughey used an SSH honeypot to gather information during two and a half months. When McCaughey presents the geographic locations of the unique IP addresses that connected to the honeypot, China comes in first place with 569 IP addresses. Answering why China is at the top place is no easy task. One theory might be that the large number of unique IP addresses is only natural considering that China is a wealthy nation with many devices. This theory would also support the United States being in the second place seen in Figure 5.3. However, things are seldom this straightforward. In McCaughey's paper, Vietnam follows China in second place, which it could be theorized might be related to how cyber-security aware a country is. For example, a small country with little focus on IT security might have more infected devices attacking other devices than a bigger country with a better IT security focus.

The results received from our honeypot differ greatly from McCaughey's paper regarding login attempts received. As can be seen in the two tables 5.1a, and 5.1b, the username and password "user" top the lists, whereas in McCaughey's paper, the by far most popular username was "root", and the most popular password was "welc0me". This difference might possibly be explained by the different configurations used for the honeypot systems. In our deployment explained in section 5.1, the deployed frontends had either a 100% or a 20% login success rate, whereas McCaughey used a mixture of default-allow or default-deny for different usernames throughout his deployment phase. This might have led adversaries to conduct more login attempts in an effort to try and break into the system, whereas for our honeypot, they might have always had a 100% chance of logging in and thus not continued trying to log in with more credentials. Because root is a user that every Linux system starts out with [83], it is not surprising to see many adversaries using

this username in McCaughey's paper.

An interesting discrepancy can be seen in Table 5.3, where the first command appears around 50 more times than the second command. One theory for this discrepancy could be a poorly written computer virus continuously executing the same command. Supporting this theory is that it was observed that several adversaries connected, ran the command which failed, disconnected and repeated the process several times over. Any human adversary would realize that the command failed to run and would not attempt to run it again. A possible interesting follow-up question from this might be: What if the command they issued succeeded to run? Would the adversaries exhibit the same behavior or have left the honeypot after one successful command execution?

In Table 5.3, the commands 13, 15, and 16 appear to be related to creating a backdoor to the device. The last two commands change the password of a given user by piping the string from the `echo` command into the `passwd` command. This modification to the password could allow an adversary to log back into the device later while at the same time locking out the original owner of the device. The command with index 13 achieves the backdoor in an arguably more sophisticated manner. The command first removes OpenSSH's default directory for SSH user configuration [84] and then adds a public-key to the `authorized_keys` file inside the configuration directory. In OpenSSH's SSH server implementation, this file contains public-keys allowed to access the device through public-key authentication [84] (described in Appendix A.1).

6.6 Ethical aspects

It should be noted that the aim of a honeypot is not to broadcast its existence to anyone, but to be found by those who search for it. Information collected by a honeypot could expose an attacker or an innocent person whose device has been compromised. To protect their integrity, it was decided that this report should not include any personally identifiable information by aggregating or redacting collected information before presentation.

Since a high-interaction honeypot allows an attacker to use the honeypot's resources by executing commands, there could be troublesome actions they may attempt to perform. For example, the attacker might attempt to bruteforce other devices or might try to use our system to build a network of devices that can execute a distributed denial-of-service attack on a target of the attacker's choosing. This risk is especially prominent in our deployment, with several Gigabit connections that could be aimed toward an innocent target. Therefore, firewall rules are provided which limit the bandwidth and blocks some common protocols to avoid providing resources for the attacker. However, additional firewall rules would be needed to fully prevent bruteforce attacks. Careful placement of these limitations had to be considered. The limits were ultimately placed on the backend's physical host, out of reach for any attacker inside the target containers.

Another ethical concern was that the attackers were allowed to download files inside target containers. As it was unknown what these files might contain, it was decided that there had to be an option deciding if the storage should be persistent or not. This option defaults to not having the storage persistent.

Publishing the honeypot as free software along with this report detailing its features could be taken advantage of by adversaries. They could, for example, implement checks for detecting if a system has any identifying quirks that match our honeypot. However, assuming adversaries will not figure out these quirks by not publishing this work would be naive. By shedding light on the honeypot's implementation and our experiences we hope to contribute to a better understanding of how to develop even more advanced honeypots. Additionally, as more types of honeypots become publicly available it becomes harder for adversaries to fingerprint all honeypots, unless there is some identifiable property which is present in multiple honeypot implementations. However, with more honeypots being published, an adversary also gains more information about different kinds of honeypots to avoid.

Another aspect of publishing the honeypot is that adversaries could use the honeypot themselves to try to learn about present attacks. Using this new-found knowledge, they could possibly create more advanced malware. Furthermore, the honeypot could be deployed by an adversary and modified to proxy incoming login credentials. If the login credentials succeed on another system, an attack can proceed to be proxied to real systems they wish to attack instead of to the backend. This would allow an inexperienced adversary to easily perform attacks.

Our honeypot can be extended with new Docker images that enable imitating multiple types of systems, which gives it the ability to discover new exploits in specific systems. By analyzing these exploits, patches could be developed and rolled out to vulnerable devices in order to help mitigate potential consequences which could be anything from data theft to cryptocurrency mining. Simultaneously, it is usually best practice to publish found exploits, which leads to attackers gaining more tools to penetrate, take over, and abuse vulnerable systems.

A decrease of security vulnerabilities in consumer and commercial systems could contribute to a safer society. Large manufacturers may patch exploits, which could positively affect many companies and individuals by protecting them from, e.g., legal, economic, social, and emotional damages.

6.7 Future work

Multiple ways of improving the honeypot system have been identified. One way would be to allow the deploying user to customize settings such as the Docker image used by the target systems and the maximum number of connections to the database and frontend. Another improvement would be to address the issue of not having the communication between the frontend and backend encrypted, which would prevent any unauthorized starting and stopping of target containers.

An important aspect of security research is the analysis of the collected data, which could result in advances in the security field. A deep analysis was not within the scope of this project due to time constraints. Analyzing and finding new ways of interpreting the collected data could be a way forward in revealing attacker's motives faster, which could provide security patches quicker. Analysis of the commands being run could reveal what system identifiers attackers are looking for when probing a system. Such an analysis could

help construct better target systems that match an attacker's expectations better, further tricking the attacker into providing information to be analyzed.

As mentioned in section 4.4.3, only unencrypted information can currently be extracted from the network traffic the honeypot captures. To additionally allow encrypted traffic to be extracted, techniques for TLS session key extraction, as presented in [85]–[88], could be integrated.

7

Conclusion

The main goal of this project was to design and develop a honeypot that will be able to evolve with the ever-changing environment in computer security. As a result, a high-interaction research honeypot with support for the SSH protocol has been implemented. Information collected by the honeypot can easily be accessed from a centralized and structured database. When deployed on low-tier Azure hosts, the honeypot operated uninterrupted for 21 consecutive days accessible on the Internet.

The brief analysis of the information collected shows that the honeypot can be used to gain knowledge regarding adversaries' strategies. Nonetheless, an in-depth analysis of the collected information is required to truly know if the honeypot's data collection capabilities are enough to draw valuable conclusions for aiding security research. Furthermore, the honeypot's capability to present information could be improved.

Docker proved to be useful in many scenarios. Firstly, for achieving high-interaction by providing an environment that can be customized to imitate different systems. Secondly, to isolate the adversaries' actions in both a relatively secure and resource-efficient manner. Lastly, for easily deploying the honeypot's components on multiple machines.

The honeypot's design successfully provides modularity which has proven to aid distributed deployment and facilitates exchanging components. New systems to be imitated and protocols for an adversary to connect to can be added. However, the process has room for improvement.

The implemented honeypot's properties aid in allowing it to evolve. With this contribution, we hope to assist research in the field of computer security.

Bibliography

[1] D. Palmer. (Nov. 2018). "IoT security: Why it will get worse before it gets better," ZDNet, [Online]. Available: https://www.zdnet.com/article/iot-security-why-it-will-get-worse-before-it-gets-better/ (visited on 2021-04-29).

[2] L. Dignan. (Oct. 2016). "Dyn, a managed DNS service, hit with attack, popular sites see performance issues," ZDNet, [Online]. Available: https://www.zdnet.com/article/dyn-a-managed-dns-service-hit-with-attack-popular-sites-see-performance-issues/ (visited on 2021-04-29).

[3] P. Newman. (Mar. 2020). "The internet of things 2020: Here's what over 400 IoT decision-makers say about the future of enterprise connectivity and how IoT companies can use it to grow revenue," Insider, [Online]. Available: https://www.businessinsider.com/internet-of-things-report (visited on 2021-04-18).

[4] Telefonaktiebolaget LM Ericsson. (n.d.). "Why IoT changes everything," [Online]. Available: https://www.ericsson.com/en/internet-of-things (visited on 2021-04-18).

[5] S. Morrow. (2020). "Reactive vs. proactive security: Three benefits of a proactive cybersecurity strategy," Infosec, [Online]. Available: https://resources.infosecinstitute.com/topic/reactive-vs-proactive-security-three-benefits-of-a-proactive-cybersecurity-strategy/ (visited on 2021-05-08).

[6] F-SECURE. (2019). "Attack landscape H12019," [Online]. Available: https://blog-assets.f-secure.com/wp-content/uploads/2019/09/12093807/2019_attack_landscape_report.pdf (visited on 2021-05-11).

[7] B. Hawkes. (May 15, 2019). "Project zero: 0day 'in the wild'," Google Blogspot, [Online]. Available: https://googleprojectzero.blogspot.com/p/0day.html (visited on 2021-05-11).

[8] L. Spitzner, *Honeypots: Tracking Hackers*. Boston, USA: Addison-Wesley, 2003.

[9] A. Kyriakou and N. Sklavos, "Container-based honeypot deployment for the analysis of malicious activity," in *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, 2018, pp. 1–4.

[10] S. Eftimie and C. Racuciu, "Honeypot system based on software containers," in *Scientific Bulletin "Mircea cel Batran" Naval Academy*, vol. 19, Naval Academy Publishing House, 2016, p. 582.

[11] M. Valicek, G. Schramm, M. Pirker, and S. Schrittwieser, "Creation and integration of remote high interaction honeypots," in *2017 International Conference on Software Security and Assurance (ICSSA)*, 2017, pp. 50–55.

[12] A. Sardana and R. C. Joshi, *Honeypots: A New Paradigm to Information Security*. Enfield, NH, USA: Science Publishers, 2011, pp. 7–18.

[13] M. Mohammed and H. Rehman, *Honeypots and Routers: Collecting Internet Attacks*. Boca Raton, FL, USA: Taylor & Francis Group, 2015.

[14] C. M. Lonvick and T. Ylonen, *The Secure Shell (SSH) Protocol Architecture*, RFC 4251, Jan. 2006. DOI: 10.17487/RFC4251. [Online]. Available: https://rfc-editor.org/rfc/rfc4251.txt.

[15] C. M. Lonvick and T. Ylonen, *The Secure Shell (SSH) Connection Protocol*, RFC 4254, Jan. 2006. DOI: 10.17487/RFC4254. [Online]. Available: https://rfc-editor.org/rfc/rfc4254.txt.

[16] Ian Shields. (Jan. 2019). "Learn linux 101: Install and configure x11," IBM Developer, [Online]. Available: https://developer.ibm.com/technologies/linux/tutorials/l-lpic1-106-1/ (visited on 2021-04-29).

[17] Shodan. (n.d.). "Shodan," [Online]. Available: https://www.shodan.io/ (visited on 2021-04-29).

[18] C. M. Lonvick and T. Ylonen, *The Secure Shell (SSH) Transport Layer Protocol*, RFC 4253, Jan. 2006. DOI: 10.17487/RFC4253. [Online]. Available: https://rfc-editor.org/rfc/rfc4253.txt.

[19] C. M. Lonvick and T. Ylonen, *The Secure Shell (SSH) Authentication Protocol*, RFC 4252, Jan. 2006. DOI: 10.17487/RFC4252. [Online]. Available: https://rfc-editor.org/rfc/rfc4252.txt.

[20] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. vol. 4, Upper Saddle River, NJ, USA: Pearson Education, 2015.

[21] W. Stallings and L. Brown, *Computer Security Principles and Practice*. vol. 4, Hudson Street, NY, USA: Pearson Education, 2018.

[22] J. Gerend, D. Coulter, H. Lohr, and E. Ross. (2019). "Containers vs. virtual machines," Microsoft Corporation, [Online]. Available: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm (visited on 2021-04-21).

[23] Red Hat Inc. (n.d.). "Containers vs VMs," [Online]. Available: https://www.redhat.com/en/topics/containers/containers-vs-vms (visited on 2021-04-21).

[24] Google LLC. (n.d.). "Containers at Google," [Online]. Available: https://cloud.google.com/containers (visited on 2021-04-21).

[25] Docker Inc. (n.d.). "Docker overview," [Online]. Available: https://docs.docker.com/get-started/overview/ (visited on 2021-04-21).

[26] Docker Inc. (n.d.). "Use volumes," [Online]. Available: https://docs.docker.com/storage/volumes/ (visited on 2021-04-21).

[27] Docker Inc. (n.d.). "Accelerate how you build, share and run modern applications," [Online]. Available: https://www.docker.com/ (visited on 2021-04-21).

[28] IBM Corporation. (2010). "Database management systems on z/OS," [Online]. Available: https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-what-is-database-management-system (visited on 2021-04-28).

[29] Grafana Labs. (n.d.). "Grafana: The open observability platform," [Online]. Available: https://grafana.com/ (visited on 2021-05-10).

[30] PyPI. (n.d.). "Pypi · the python package index," [Online]. Available: https://pypi.org/ (visited on 2021-05-05).

[31] Docker Inc. (n.d.). "Overview of Docker Compose," [Online]. Available: https://docs.docker.com/compose/ (visited on 2021-05-02).

[32] solid IT. (Apr. 2021). "DB-engines ranking," [Online]. Available: https://db-engines.com/en/ranking (visited on 2021-04-21).
[33] The PostgreSQL Global Development Group. (n.d.). "About," [Online]. Available: https://www.postgresql.org/about/ (visited on 2021-04-29).
[34] The PostgreSQL Global Development Group. (n.d.). "Chapter 8. data types," [Online]. Available: https://www.postgresql.org/docs/13/datatype.html (visited on 2021-04-21).
[35] PostgreSQLWiki. (2019). "Python," [Online]. Available: https://wiki.postgresql.org/index.php?title=Python&oldid=34420 (visited on 2021-04-21).
[36] PythonWiki. (n.d.). "PostgreSQL," [Online]. Available: https://wiki.python.org/moin/PostgreSQL?action=recall&rev=45 (visited on 2021-04-21).
[37] Psycopg. (n.d.). "PostgreSQL driver for Python - Psycopg," [Online]. Available: https://www.psycopg.org/ (visited on 2021-04-21).
[38] Docker Inc. (n.d.). "Postgres," [Online]. Available: https://hub.docker.com/_/postgres (visited on 2021-04-21).
[39] Grafana Labs. (n.d.). "PostgreSQL data source," [Online]. Available: https://grafana.com/docs/grafana/latest/datasources/postgres/ (visited on 2021-04-21).
[40] B. Nikkel, "Registration data access protocol (RDAP) for digital forensic investigators," *Digital Investigation*, vol. 22, pp. 133–141, 2017. DOI: 10.1016/j.diin.2017.07.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287617301688 (visited on 2021-05-05).
[41] VirusTotal. (2020). "Search," [Online]. Available: https://developers.virustotal.com/v3.0/docs/file-search (visited on 2021-05-05).
[42] J. Nummenmaa and A. Ranta, "Databases in 137 pages," unpublished.
[43] J. Forcier. (n.d.). "Welcome to Paramiko!" Paramiko, [Online]. Available: http://www.paramiko.org/ (visited on 2021-04-22).
[44] J. Forcier, *Paramiko*, https://github.com/paramiko/paramiko, n.d. (visited on 2021-04-22).
[45] The PostgreSQL Global Development Group. (n.d.). "Chapter 3. advanced features," [Online]. Available: https://www.postgresql.org/docs/13/tutorial-transactions.html (visited on 2021-05-02).
[46] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *2012 IEEE Fifth International Conference on Cloud Computing*, IEEE, 2012, pp. 423–430.
[47] The LinuxServer.io team, *Openssh-server*, https://github.com/linuxserver/docker-openssh-server, 2021. (visited on 2021-05-10).
[48] J. Petazzoni, *Using docker-in-docker for your CI or testing environment? think twice*, Jérôme Petazzoni GitHub pages, 2020. [Online]. Available: https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/ (visited on 2021-04-23).
[49] Docker Inc. (n.d.). "Docker SDK for Python," [Online]. Available: https://pypi.org/project/docker/ (visited on 2021-05-13).
[50] M. Cotton, L. Eggert, D. J. D. Touch, M. Westerlund, and S. Cheshire, *Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry*, RFC 6335, Aug. 2011. DOI: 10.17487/RFC6335. [Online]. Available: https://rfc-editor.org/rfc/rfc6335.txt.

[51] P. Dimou, J. Fajfer, N. Müller, E. Rekleitis, and F. Střasák, "Encrypted traffic analysis — use cases & security challenges," ENISA, 2020. [Online]. Available: https://www.enisa.europa.eu/publications/encrypted-traffic-analysis (visited on 2021-04-25).

[52] The Tcpdump Group. (n.d.). "Tcpdump & libpcap," [Online]. Available: https://www.tcpdump.org/ (visited on 2021-04-25).

[53] G. Combsa, G. Ramirez, and G. Harris. (n.d.). "Wireshark," GitLab, [Online]. Available: https://gitlab.com/wireshark/wireshark (visited on 2021-04-25).

[54] D. Miller, *et al.* (n.d.). "Nmap," GitHub, [Online]. Available: https://github.com/nmap/nmap (visited on 2021-04-25).

[55] G. Turner and G. Harris. (Mar. 30, 2011). "application/vnd.tcpdump.pcap media type definition," IANA, [Online]. Available: https://www.iana.org/assignments/media-types/application/vnd.tcpdump.pcap (visited on 2021-04-25).

[56] A. Cortesi, *et al.* (n.d.). "Mitmproxy - an interactive HTTPS proxy," Mitmproxy, [Online]. Available: https://mitmproxy.org/ (visited on 2021-04-25).

[57] NETRESEC AB. (n.d.). "Polarproxy," [Online]. Available: https://www.netresec.com/?page=PolarProxy (visited on 2021-04-25).

[58] D. Roethlisberger. (n.d.). "SSLsplit - transparent SSL/TLS interception," Rö's Wiki, [Online]. Available: https://www.roe.ch/SSLsplit (visited on 2021-04-25).

[59] Mitmproxy. (n.d.). "Modes of operation," [Online]. Available: https://docs.mitmproxy.org/stable/concepts-modes/ (visited on 2021-04-25).

[60] Netfilter. (n.d.). "Netfilter firewalling, NAT, and packet mangling for linux," [Online]. Available: https://www.roe.ch/SSLsplit (visited on 2021-04-25).

[61] Mitmproxy. (n.d.). "Introduction," [Online]. Available: https://docs.mitmproxy.org/stable/howto-transparent/#3-create-an-iptables-ruleset-that-redirects-the-desired-traffic-to-mitmproxy (visited on 2021-04-25).

[62] Mitmproxy. (n.d.). "About certificates," [Online]. Available: https://docs.mitmproxy.org/stable/concepts-certificates/ (visited on 2021-04-25).

[63] Docker Inc. (n.d.). "Docker run reference," [Online]. Available: https://docs.docker.com/engine/reference/run/#network-settings (visited on 2021-04-26).

[64] P. Bogaerts. (Jan. 2017). "How to tcpdump effectively in Docker," Medium, [Online]. Available: https://xxradar.medium.com/how-to-tcpdump-effectively-in-docker-2ed0a09b5406 (visited on 2021-04-26).

[65] The Tcpdump Group. (Jan. 2021). "Man page of tcpdump," [Online]. Available: https://www.tcpdump.org/manpages/tcpdump.1.html (visited on 2021-04-26).

[66] The GraphQL Foundation. (n.d.). "A query language for your API," [Online]. Available: https://graphql.org/ (visited on 2021-04-28).

[67] J. Harmon *et al.* (Feb. 2021). "OpenAPI specification," OpenAPI, [Online]. Available: https://spec.openapis.org/oas/v3.1.0 (visited on 2021-04-28).

[68] SmartBear Software. (n.d.). "API development for everyone," [Online]. Available: https://swagger.io/ (visited on 2021-04-28).

[69] gRPC Authors. (n.d.). "gRPC," [Online]. Available: https://grpc.io/ (visited on 2021-04-28).

[70] The GraphQL Foundation. (n.d.). "Queries and mutations," [Online]. Available: https://graphql.org/learn/queries/#mutations (visited on 2021-04-28).

[71] gRPC Authors. (Dec. 2020). "Introduction to gRPC," [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction/ (visited on 2021-04-28).
[72] K. Sandoval. (Aug. 2018). "When to use what: REST, GraphQL, webhooks, & gRPC," Nordic APIS, [Online]. Available: https://nordicapis.com/when-to-use-what-rest-graphql-webhooks-grpc/ (visited on 2021-04-28).
[73] M. Nally. (Apr. 2020). "gRPC vs REST: Understanding gRPC, OpenAPI and REST and when to use them in API design," Google cloud, [Online]. Available: https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them (visited on 2021-04-28).
[74] SmartBear Software. (n.d.). "Swagger codegen," [Online]. Available: https://swagger.io/tools/swagger-codegen/ (visited on 2021-04-28).
[75] Google LLC. (Mar. 31, 2021). "Language guide (proto3)," [Online]. Available: https://developers.google.com/protocol-buffers/docs/proto3 (visited on 2021-04-29).
[76] A. Kumar, *et al.* (Sep. 13, 2020). "Status codes and their use in gRPC," GitHub, [Online]. Available: https://github.com/grpc/grpc/blob/c5c2793427c691b2663e1cf271cff04ad0ec50b3/doc/statuscodes.md (visited on 2021-04-29).
[77] Microsoft Corporation. (Mar. 2020). "Virtual machines," [Online]. Available: https://azure.microsoft.com/en-us/services/virtual-machines/ (visited on 2021-04-21).
[78] Shodan. (n.d.). "Honeypot or not?" [Online]. Available: https://honeyscore.shodan.io/ (visited on 2021-04-30).
[79] SIA Mikrotīkls. (n.d.). "Scheduler," [Online]. Available: https://help.mikrotik.com/docs/display/ROS/Scheduler (visited on 2021-05-05).
[80] T. Bui, "Analysis of Docker security," *CoRR*, vol. abs/1501.02967, 2015. arXiv: 1501.02967. [Online]. Available: http://arxiv.org/abs/1501.02967.
[81] R. Yasrab, "Mitigating Docker security issues," *CoRR*, vol. abs/1804.05039, 2018. arXiv: 1804.05039. [Online]. Available: http://arxiv.org/abs/1804.05039.
[82] R. J. McCaughey, *Deception using an SSH honeypot*, 2017. [Online]. Available: https://calhoun.nps.edu/handle/10945/56156.
[83] C. Negus, *Linux Bible*. Indianapolis, USA: John Wiley & Sons, 2020, p. 174.
[84] The OpenBSD Project, *ssh – OpenSSH remote login client*, n.d. [Online]. Available: https://man.openbsd.org/OpenBSD-6.9/ssh (visited on 2021-05-12).
[85] S. Blok, *I spy, with my little eye, something inside TLS!* YouTube, Open Information Security Foundation, Nov. 14, 2019. [Online]. Available: https://www.youtube.com/watch?v=hP0ctgD3dBM (visited on 2021-05-04).
[86] P. Wu, *SF18ASIA - 19: SSL/TLS decryption: Uncovering secrets (Peter Wu)*, YouTube, SharkFest Wireshark Developer and User Conference, Apr. 12, 2018. [Online]. Available: https://www.youtube.com/watch?v=bwJEBwgoeBg (visited on 2021-05-04).
[87] B. Taubmann, C. Frädrich, D. Dusold, and H. P. Reiser, "TLSkex: Harnessing virtual machine introspection for decrypting TLS communication," *Digital Investigation*, vol. 16, pp. 114–123, 2016. DOI: 10.1016/j.diin.2016.01.014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287616300081 (visited on 2021-05-04).
[88] R. Caragea, "TeLeScope - real-time peering into the depths of TLS traffic from the hypervisor," in *HITBSECCONF*, 2016. [Online]. Available: https://conference.hitb.org/hitbsecconf2016ams/wp-content/uploads/2015/11/D1T1-Radu-Caragea-Peering-into-the-Depths-of-TLS-Traffic-in-Real-Time.pdf.

[89] N. Freed, D. J. C. Klensin, and T. Hansen, *Media Type Specifications and Registration Procedures*, RFC 6838, Jan. 2013. DOI: 10.17487/RFC6838. [Online]. Available: https://rfc-editor.org/rfc/rfc6838.txt.

A

Appendix

A.1 Additional Secure Shell login methods

Public-key authentication

In public-key authentication a client proves his identity to a server by sending a signature created with a private key [19]. The server then verifies that the signature is valid and that the key is a valid authenticator for the user authenticating. If both conditions are met, the user is granted access.

Host-based authentication

Host-based authentication lets the server authenticate a user based on a host key [19]. It works similarly to public-key authentication, but instead of users possessing individual private keys, the host has one. In other words, while public-key authentication lets users authenticate from anywhere, given that they possess their private key, host-based authentication only lets them authenticate from a single host.

A.2 Detailed database design

While Figure 4.1 gives an overview of the database design entity-relationship model and the relationships between entities, Table A.1 describes what each entity of the model represents. Furthermore, the table shows the intended usage of each attribute and its data type (e.g., `attribute : type`). The data types `inet`, `serial` and `bytea` are specific to PostgreSQL and represent an IPv4 or IPv6 address, a strictly increasing integer and a byte array, respectively [34].

Table A.1: Entity descriptions and attributes for the honeypot's database design.

Entity	Description
NetworkSource	The network origin of a session or download identified by its IP address. • `ip_address : inet`
Session	An attacker session, identified by a generated ID, starting when connecting and ending when disconnecting from the honeypot. • `id : serial`

Entity	Description
	• protocol : text • attack_src : inet • src_port : int • dst_ip : inet • dst_port : int • start_timestamp : timestamp • end_timestamp : timestamp protocol is the name of the protocol the session corresponds to. Currently, the only type allowed is "ssh".
SSHSession	An attacker session, extending Session, with properties specific to the SSH protocol. • ssh_version : text ssh_version is the remote client SSH version sent by the attacker.
EventType	The type of an Event identified by its name. • name : text Currently, the allowed event types are: "pty_request", "env_request", "direct_tcpip_request", "x_eleven_request", "port_forward_request", "command", "ssh_channel_output", "download" and "login_attempt".
Event	An event caused by an attacker, during a Session, at a specific point in time, identified by a generated ID. All events inherit from this entity. • id : serial • type : text • timestamp : timestamp type references an EventType.
PTYRequest	An event corresponding to an SSH channel request of type "pty-req" [15] being received. • term : text • term_width_cols : int • term_height_rows : int • term_width_pixels : int • term_height_pixels : int
EnvRequest	An event corresponding to an SSH channel request of type "env" [15] being received. • channel_id : int • name : text • value : text
DirectTCPIPRequest	An event corresponding to an SSH channel open request with channel type "direct-tcpip" [15] being received. • channel_id : int • origin_ip : inet • origin_port : int

Entity	Description
	• `destination : text` • `destination_port : int`
XElevenRequest	An event corresponding to an SSH channel request of type “x11-req” [15] being received. • `channel_id : int` • `single_connection : boolean` • `auth_protocol : text` • `auth_cookie : bytea` • `screen_number : int`
PortForwardRequest	An event corresponding to an SSH global request with type “tcpip-forward” [15] being received. • `address : text` • `port : int`
Command	A shell command execution event. • `input : text`
SSHChannelOutput	An event representing a block of bytes being output over an SSH channel. • `data : bytea` • `channel : int` `data` is `bytea` to allow storing any binary channel output up to 1 GiB.
File	A downloaded file identified by its hash. • `hash : bytea` • `data : bytea` • `type : text` `hash` is `bytea` to store a 256-bit SHA-2 hash of `data`. `data` is `bytea` to allow storing any file contents up to 1 GiB. The value may be `null`. `type` is intended to store the media type [89] of the file.
Download	A file download event. • `hash : bytea` • `src : inet` • `url : text` `hash` references a `File` entity corresponding to the downloaded file. `src` and `url` are the IP address and URL where the download was retrieved from.
LoginAttempt	A login attempt event. • `username : text` • `password : text`