MD-Honeypot-SSH

Gathering Threat Intelligence Data during the SSH Hand-shake

Bart de Jonge

Master Thesis

MD-Honeypot-SSH

TU Delft

MD-Honeypot-SSH

Gathering Threat Intelligence Data during the SSH Handshake

by

Bart de Jonge

to obtain the degree of Master of Science at the Delft University of Technology, to be defended publicly on February 25th, 2022 at 10:00 AM

Student number:	4392256	
Project duration:	November 2018 – February 2022	
Thesis committee:	Dr. S.E. Verwer,	TU Delft, supervisor
	Dr. M. Aniche,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

With the amount of network connected devices every increasing, and many of them running the Secure Shell (SSH) protocol to facilitate remote management, research into SSH attacks is more important than ever. SSH honeypots can be used to act like vulnerable systems while gathering valuable data on the attacker and its methods in the meantime. The SSH handshake is a currently undervalued asset in these honeypots as a lot of data is already exchanged in this early part of the protocol. In this thesis we propose the MD-Honeypot-SSH framework that can be used to gather threat intelligence research data on the SSH handshake. We show the design choices made in the development of the framework and consider which data is useful to collect in the SSH handshake for future research. As part of the framework we modify an existing OpenSSH implementation to allow us to log any relevant branching decisions made in the server. We then use this logging data to create state machines of the server behaviour while handling a specific connection. We use these state machines to compare different connections and show, as a proof of concept, that we can group these connections based on the used client. The main contribution of this thesis is to provide the MD-Honeypot-SSH framework as a tool to future research, and we provide some recommendations for future research directions.

Preface

You are reading the thesis "MD-Honeypot-SSH: Gathering Threat Intelligence Data during the SSH Handshake" which outlines the research that went into creating a honeypot framework for future research that focuses on the SSH handshake. It was written to fulfil the requirements of the Computer Science master at the Cyber Security research group at the Delft University of Technology.

During the project I was initially supervised by Christian Doerr, but after he left his post at the TU Delft Sicco Verwer kindly stepped in as my supervisor. I want to thank them both for their ideas and guidance, especially during the struggles near the end of the project. I also want to thank the thesis students in the cyber security group for creating a nice environment to work in. A special thanks goes to my two colleague student Paul van der Knaap and Ahmet Gudek, for working together with me on parts of the core MD-Honeypot framework. Finally, I want to thank my family and friends, especially my parents and girlfriend, for keeping me motivated and sane during this long project.

I hope you enjoy reading this thesis.

Bart de Jonge
Delft, February 2022

Contents

1 Introduction

As computers are becoming more capable and more interconnected a logical next step has been to control these computers remotely over a network. One of the first widely adopted protocols that achieved this is Telnet, which was developed during the late 1960s and early 1970s, as ARPANET was slowly growing into the internet we know today. Telnet gave users a bidirectional plaintext communication channel that was mainly used to access the command-line interface of remote hosts. In these early years this form of plaintext communication, without any form of authentication or encryption, was acceptable as networks consisted of parties that all knew and trusted each other. As the networks merged and became interconnected this trust started to fade and the wish for more secure connections grew. As a response to this, the Berkely r-commands suite was developed in 1982, which included, among others, the rlogin command that allowed a similar remote shell to Telnet, but required users to login. These so-called r-commands were a major security increase and became the standard for Unix operating systems.

1.1. Secure Shell - SSH

However, there was still a major problem concerning security with this and other solutions: the plaintext communication. Anyone who could intercept and read the network packets that were used to login could read the plaintext password, which would negate any security it originally provided. Tatu Ylönen experienced this first-hand when his university network was the victim of a password-sniffing attack. He developed the first version of his Secure Shell (SSH) protocol in 1995 to counter this problem. SSH improves on older protocols like Telnet and rlogin by using public-key cryptography to authenticate the user and encrypt all data in transit. This makes it impossible to just intercept the packets and read all sent data, including passwords, and therefore offers a much higher security standard. Since then many different versions of SSH have been developed and nearly every major network environment uses a version of it to protect data in transit and manage systems remotely. It can be found running in routers, servers, managed switches, IP- and IOT cameras, among others. This widespread deployment of SSH combined with the fact that SSH access gives a user many capabilities on the host system has made SSH a major target for many different attackers. These adversaries range from automated scripts trying to brute-force passwords, botnets trying to extend their network, to targeted attacks on specific networks to potentially steal data.

1.2. Honeypots

To gather information on SSH-based attacks, researchers and network managers deploy so-called SSH honeypots. Honeypots are decoy systems that are deployed alongside real systems with the aim of being attacked instead. This both diverts attackers away from the important systems as well as giving the opportunity to study the attackers' behaviour and/or alarm the network managers of an ongoing attack. Honeypots can be divided based on their level of interaction, most commonly into high- and low-interaction. High interaction honeypots are designed to resemble an actual system as closely as possible. They generally run a full operating system (OS) and all the services a production system would run. This allows full insight into the attackers' behaviour, but is also very resource-intensive to set up and maintain. Low interaction honeypots on the other hand, run or emulate simple services that attackers are looking for, and nothing else. These systems are a lot

easier to set up and maintain, but also a lot easier for an attacker to recognise as services will not respond as they expect or not respond at all.

While high interaction honeypots are obviously a better representation of a real system than their low interaction variants, every honeypot is still a less or more sophisticated approximation of a real system which could give away information that it is not real in some way. For low interaction honeypots this could be that simple OS commands do not respond or the low amount of responding services. While for high interaction honeypots, this could be outgoing network traffic being blocked or edited files not being synced between sessions. Since the aim of these honeypots is to gain knowledge about the presence and methods of attackers, they can use these flaws to detect the honeypots and prevent giving away any information.

A famous example of this is the kill-switch in the original version of the WannaCry ransomware. The developers of the ransomware knew that when it became widespread it would be investigated in so-called sandboxes, controlled environments with no network access to prevent the malware from spreading. To still be able to investigate the malware these sandboxes usually respond to any network request with dummy responses. Since the developers of the ransomware did not want their software to be investigated and potentially reverse-engineered which would hurt their business model, they added a request to a non-existing domain. If this request would be met with a response the ransomware would conclude it might be in such a sandbox being investigated and would shut down. A fault in this implementation was that the malware analysis expert MalwareTech could just register that domain so it would answer to all new instances of the ransomware worldwide and they would all shut down. While this case in particular is not dealing with SSH honeypots, it still shows attackers are motivated to hide their methods and try to prevent giving away any information to researchers.

1.3. Current Honeypot Capabilities

When looking specifically at SSH honeypots, we get the following characteristics for the different honeypot types:

- Low-interaction: these are easy to develop, deploy and maintain but they usually only give very basic metadata (connection source/time/location) and are very easy to detect (don't respond to services or unexpected results etc). This usually means they only log incoming connections and don't respond at all, or they accept all incoming connection regardless of authentication to log the first few commands they issue.

- High-interaction: these are very time-consuming to develop, deploy and maintain. They focus mainly on collecting data on attacker behaviour once the system was entered. So username/password combination is set to be easily guessed. These systems usually try to mimic a real system as well as possible since they don't want to be detected but there are always shortcomings.

This means that to deploy a honeypot that can gather large amounts of data per attacker, while the data is not biased by the attacker easily recognizing the system as being a honeypot, we currently have to create a high-interaction honeypot that is very difficult to deploy and maintain, and might still be detected. This could be improved upon by limiting the scope of the data collection such that the system becomes easier to deploy and maintain, while still retaining the usefulness of the data by preventing attackers recognizing the system as a honeypot.

1.4. Research Questions

The field of cyber security research is a constant race with adversaries: researchers trying to detect and prevent their methods, and the adversaries developing new methods and ways to prevent detection. The same is the case with respect to SSH honeypots. To build the most effective honeypots it is essential to know as much as possible about what different types of attackers we are phasing and what methods are being used to try and detect SSH honeypots. As we dicuss in more detail in chapter 2, much research has been done on this topic with respect to the behaviour of high-interaction honeypots after an attacker has logged in, however no research has been done into the SSH handshake part of the interaction.

As we explain in more detail in chapter 2, there is a set handshake as part of the SSH protocol in which the client and server exchange information and decide on things like cryptography algorithms to use in the rest of the session. If we can gain information on how critically attackers look at this information that is exchanged in the handshake we can get a better picture of who we are dealing with and how to build better

SSH honeypots in the future. Currently, there is no clear way to collect and analyse this information. In this thesis we investigate what data can be collected during this handshake process and what is useful with respect to threat intelligence. In other words, this research aims to answer the following research questions:

RQ1. What data is useful to log during execution monitoring to conduct threat intelligence research on the SSH handshake?

RQ2. How can we collect SSH handshake data for threat intelligence?

RQ3. How can we group connections by SSH client, using only real-time, connection-specific data?

The outcome of this research will be a platform to gather the data to do further research on this topic, as well as sample analysis on sample data. We recognise this is an unusual thesis as we do not present statistically relevant conclusions based on a complete dataset. The difficulties of the development and deployment of the data gathering platform, and the limit on the available time for this project made us unable to deliver this full analysis. We argue however that this research is still relevant as we take the necessary scientific steps to deliver an useful research tool that can be used in future work.

1.5. Thesis Outline

The rest of this report presents the results of this research and the answers to the research questions. First, in chapter 2 we will cover the background information needed for the rest of the thesis, and discuss related work. Then chapter 3 will cover what data we will collect and how. A full overview of the MD-Honeypot framework will be given in chapter 4. Then chapter 5 will present a number of experiments to show what can be done with the developed framework. Finally, chapter 6 concludes this thesis and discusses limitations.

2

Background and Related Work

The previous chapter briefly discussed the SSH protocol and the concept of honeypots. In this chapter, we give a more in-depth explanation of these topics and provide the essential technical knowledge for the remaining parts of this thesis in section 2.1. After that, we give an overview of related work in section 2.2.

2.1. Secure Shell Protocol

As discussed in chapter 1, SSH was mainly developed as a more secure replacement for older protocols that allowed remote access via the network like Telnet and rlogin. The main advantage that SSH has over these older protocols, is that once the connection is established all communication is encrypted. Furthermore, no piece of authentication data, in the case of SSH meaning passwords or keys, are ever sent in plaintext over the network. Since this was the case for the older protocols this was a major security upgrade. SSH is divided into two major versions, SSH-1 and SSH-2, which are not compatible with each other, although some SSH servers support both. SSH-2 was released in 2006 and features some major security and feature upgrades and is the version that most modern applications use. Therefore, for the rest of this thesis SSH will refer to SSH-2. While logging in to remote machines and executing commands is the typical use case of SSH, it has gathered many different features over its lifespan like forwarding TCP ports, file transfer and tunneling other TCP/IP based sessions. SSH does this while providing 4 guarantees: privacy of your data, integrity of communications, authentication and authorization.

Privacy (Encryption)

Privacy with respect to networking means protecting data from disclosure. While this was not necessary when the physical network structure was secured, this has changed with growing networks and improved interconnectivity. In many cases, attackers can now intercept network traffic and read the contents if nothing is done to prevent this. SSH guarantees privacy by using end-to-end encryption with fresh encryption keys for each session. These random keys are negotiated during the handshake process at the start of the connection and destroyed afterwards. SSH supports a number of different encryption algorithms including AES, ARCFOUR, DES and triple-DES (3DES).

Integrity

It is important when trying to achieve a secure connection to a remote machine that you know that the data transmitted arrives at the other side unaltered. We call this integrity checking, and even though TCP/IP does some of this to detect and solve noise, packet loss or other network problems, this does not prevent deliberate tampering from an attacker. While its impossible for an attacker to insert accurate commands because of the end-to-end encryption of SSH, random noise or a copy of old packets could still be inserted. SSH prevents this by using keyed hash algorithms like SHA or MD5. Hashes based on negotiated keys, message content and packet sequence number are added at the end of each packet. This allows the receiver to check that the message has not been tampered with and is no replay of a previous packet and therefore guarantees integrity of the messages.

Authentication

Authentication serves the purpose of confirming the identity of the party you are talking to. This works in two directions as the client wants to know it is talking to the real server and not an imposter trying for example a man-in-the-middle attack, and the server wants to confirm the clients identity before giving access to the SSH features. The first is done by using public-private key cryptography to identify the server. The server is the only owner of the private key, while the client stores the public key after the first connection. This means that after the initial connection the client is always sure its talking to the same server that it was talking to during that first connection, but the client still has to verify the identity of the initial server. This can be done for example by exchanging public keys with other clients or connecting the server via a direct connection for the first time.

For the client authentication a lot of different options exist within SSH. Firstly a simple username/password combination to gain access. While the security level of this option fully depends on the strength and uniqueness of the password, it is still more secure than similar options in older protocols like Telnet, since the password is sent to the server in an encrypted message and not in plaintext. Another widely used, and arguably more secure, option is using client keys. For this option the client generates a public-private key pair, of which the public part is stored on the server in a list of authorized keys. The client is the only one with access to the private key and this is used during the handshake process to authenticate the clients identity. Over time SSH has added support for many third-party authentication option including Kerberos and Pluggable Authentication Modules (PAM), and the server can even require multiple authentication methods before granting access. Which method is used is determined by the client and server during the handshake process.

Authorization

Depending on the user, the server may want to grant different levels of access to the different SSH features and/or restricting certain actions. This is called authorization and can be done in SSH either serverwide or on a user-level depending on the authentication method used.

2.1.1. SSH Handshake

SSH works with a client-server model where the server is always listening for incoming connection requests and the client initiates the connection. To achieve the security guarantees laid out above, the server and client need to reach a consensus on the different algorithms used and the way to generate the session keys. This is done in the so-called handshake process which uses a standard set of messages between the client and server to reach this consensus. The handshake process consists of the following steps:

1. **TCP connection establishment:** Client establishes a TCP connection with the server.
2. **Version exchange:** Client and Server both send their respective version strings. This is mainly to establish the use of SSH1 or SSH2 for clients/servers that support both, but nowadays most software only supports SSH2. Usually the particular version of the SSH distribution is also shared which can be used to for example circumvent certain vulnerabilities in older versions. An example of a version string is `SSH–2.0–libssh–0.6.3`.
3. **Key Exchange (KEX):** This is the part where the client and server exchange some information in plaintext that then leads to a consensus on the algorithms and keys used for encryption where the keys cannot be derived by intercepting the KEX messages. The key exchange itself again consist of multiple steps:
   (a) **Key Exchange Initialization:** the first step is for the client and server to both send a `SSH_MSG_KEX_INIT` message to each other with a list of cryptographic primitives they support. These lists are ordered based on their preference, and a set algorithm chooses a primitive that both parties support and is as high on both preference lists as possible. Since the input of this algorithm is both support/preference lists and both parties use the exact same selection algorithm the outcome of this is always the same for the client and server and both know which primitives to use moving forward. The main primitives that are agreed on using this message are:
      - `kex_algorithms`: algorithm to use to exchange encryption keys, for example `curve25519–sha256` or `ecdh–sha2–nitp256`.

- server_host_key_algorithms : algorithm to verify the host key. In case of the server these are all algorithms it can supply a host key (possibly different keys for different algorithms) for, in the case of the client this is all algorithms it supports to validate the host key.
- encryption_algorithms : symmetric encryption algorithm used to encrypt all traffic after the key exchange has been completed.
- mac_algorithms : message authentication algorithm (MAC) used to provide a MAC-code to the end of each message which proves the message came from the stated sender (authenticity) and has not been changed.
- compression_algorithms : algorithm used for compression on the packets.

(b) **Key Exchange execution:** next the client initiates the actual key exchange based on the agreed upon key exchange algorithm. Currently all supported key exchange algorithms in SSH are some form of the Diffie-Helman algorithm. This is a well known algorithm that can create a shared secret between the client and server by both creating a public-private pair of numbers, exchanging the public parts and combining them with the private numbers using specific formula. The key exchange is also authenticated by a hash signature to make sure the secret is created with the intended party and not a man-in-the-middle. The following messages are sent in a handshake using the standard Diffie-Helman algorithm:

- SSH_MSG_KEXDH_INIT : the client sends his public part to the server
- SSH_MSG_KEXDH_REPLY : the server replies with its public part and the hash signature
- SSH_MSG_NEWKEYS : the client indicates a successful key exchange and all messages from now on will be encrypted using the agreed keys and algorithms

4. **Authentication**: this is initiated by the client and uses encrypted SSH2_MSG_SERVICE_REQUEST messages, which depend on the kind of authentication the client prefers and the server accepts. This can be password-based or key-based for example.

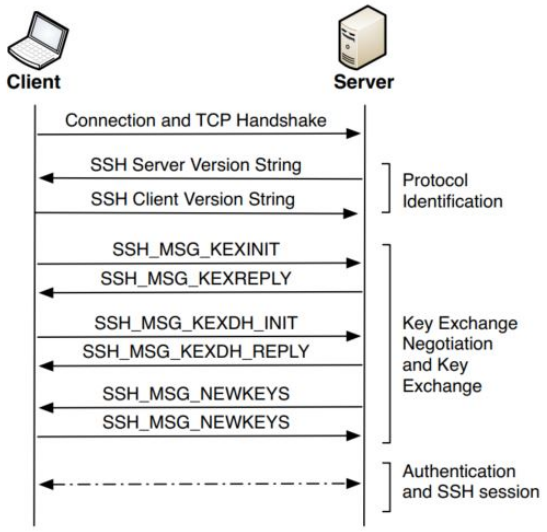

Figure 2.1: Schematic overview of the message exchanges during the SSH handshake

It is apparent that this handshake process is quite a complicated process. A schematic overview of this process is given in Figure 2.1. During the handshake, before the client has any access to the target system, quite a lot of data is already exchanged between the server and the client to initiate a secure connection. We can use this data to study attacker behaviour in this very first step of the attack process. The available data does not only exist of the content of the packets sent by the client, but also in the whole flow of the connection. We can monitor some parts of this flow by for example inspecting the state of the connection inside the server.

2.2. Related Work

In this section we provide an overview of academic research relevant to this thesis. We present the principles and findings of previous work and summarise the most important insights from these studies. Next to summarizing related work, we will discuss some research gaps which are at the basis of the research questions stated in the introduction. We first discuss related work on honeypot design followed by an overview of previous research on SSH honeypots in particular. Finally we present the research gaps we have identified and discuss how we are going to resolve them.

2.2.1. Honeypot design

The concept of honeypots, while under a number of different names, goes back to the 1990s. However, honeypots, and their design, as a research topic only started in the year 2000. In this year Roesch[31] established one of the first classifications of honeypots, separating honeypot primarily based on their field of operation: production honeypots and research honeypots. Production honeypots are, like their name suggests, mainly meant to be used in production environments of companies and are therefore focused on ease of deployment and utilization. Their intended purpose is to raise to security level within a network by deflecting and/or detecting attacks. Their focus on ease of deployment and utilization comes with a tradeof however, as this means a smaller amount of collected information. This is contrasted by research honeypots where ease of deployment is considered less important and more focus is on collecting as much information as possible. Honeypots types can be further classified by level of interaction (low-high)[22] or direction of interaction (client/server)[26].

The first broadly accepted formal definition of a honeypot was introduced by Spitzner[30]: "A honeypot is a decoy computer resource whose value lies in being probed, attacked or compromised." Nawrocki et al. give a summary of honeypot software in their survey from 2016 [23]. One of their findings is that different honeypots exist which are applied to different protocols and network types. This highlights the universality of the concept of honeypots. Another finding is that certain honeypot software overlap in their field of operation. In these cases, the quality and maintenance life time of the honeypot influence the success. Despite many different honeypots and related tools, the general trend is clear: simple proof of concepts developed into complex honeypot tools which are designed to be deployed for a long time. As the intended analysis is getting more complex, management and analysis tools emerge and even combinations with intrusion and malware detection tools are considered. The first honeypot and also the majority of available honeypot software are low-interaction server honeypots. The reason for this obversation might be the fact, that on the one hand server honeypots require less implementation effort than client honeypots as they do not have to have a sophisticated crawler engine and on the other hand the emulation of services requires less maintenance effort than providing real services on high-interaction honeypots.

Recently, Sethi et al. reviewed the use of honeypots as a way of securing networks [27]. They showed how this fairly old concept is still very relevant, but should be carefully configured. Some recommended techniques are displayed on how to best do this. While the configuration of the networks where the honeypots are deployed is very important, Yener et al. showed that there is no difference in scanning and attack behaviour when deploying identical honeypots in different ISP networks [33]. Kelly et al. showed that even deploying honeypots on cloud platforms is a valid option as they are also continually being scanned and attacked [16].

2.2.2. SSH Research

Since SSH is one of the most used and well known internet protocols there is a long history of research related to SSH. Early on different vulnerabilities were proposed in the earlier SSH versions. In 2001, Song showed that SSH was leaking information on passwords and other plaintext typed in interactive mode by the timing of character echoes [28]. Bellare proposed multiple vulnerabilities to the SSH authentication scheme in 2002 [6] and 2004 [7]. In 2009, Albrecht showed that some implementations of SSH, including the widely used OpenSSH implementation, were vulnerable to plaintext recovery attacks [5]. These exposed attack vectors have since been fixed in the maintained SSH versions and research into SSH has mostly split into two directions. First, it has been widely accepted that the main remaining weak points in the SSH protocol are the used cryptographic functions. Research into the security of these cryptographic functions therefore plays an important part in keeping SSH secure. Secondly, a lot of research is done into brute-force attacks, and how to detect and avoid them. With the rise of IOT devices, the number of badly managed network devices connected to the internet running remote control services like SSH or Telnet have skyrocketed. These devices often use default password settings [18]. Botnets search for these devices and try to gain access by using lists

of commonly used passwords to increase their reach [12]. These brute-force attacks have been analysed in a large number of studies, starting with Owens in 2008 [24], Sperotto in 2009 [29] and Vykopal in 2009 [32]. The detection of these attacks has improved and made more efficient by using NetFlow data [25]. Since then, smarter attackers have moved to distributed brute-force attacks to try and prevent the attack being detected and simply blocked. Ghiette showed that by fingerprinting clients that try to connect to an SSH server, these distributed attacks can also be detected efficiently [14].

Another branch of research into not only SSH, but many networking protocols, is protocol model learning using a method called protocol state fuzzing. Fiterău-Broştean et al. showed this for both SSH [11] and DTLS (a UDP implementation of TLS) [10]. Protocol state fuzzing is a technique that creates an input alphabet of possible protocol messages and uses that to learn a state machine of a target server by analyzing the response messages. All possible combinations of input messages are used to make this state machine as complete as possible. This state machine can then be used to check the server against protocol specifications and to find security flaws caused by logic errors.

2.2.3. SSH honeypots

While some of the research discussed above used honeypots to gather data, their aim was not specifically to improve the effectiveness of the honeypot. This is the aim in the research published by Hellemons in 2012 [15], where SSHCure is introduced. SSHCure is a mix between an intrusion detection system and a honeypot and uses NetFlows. Koniaris [19] shows how to best deploy a SSH honeypot and analyse the results. More recently, Belqruch [8] uses knowledge gained from research into honeypots that target a number of SCADA protocols and shows how SSH honeypot can be adjusted to be used in the SCADA environment to detect attacks. In 2021, Martinez Garre et al. proposed a machine learning-based approach for the detection of SSH botnet infection. This method focuses on the initial phase of the botnet lifecycle, where the focus is infection. The method uses data from high interaction honeypots to identify connections that aim to infect the honeypot [13].

While open-source honeypots like Cowrie are great way to increase honeypot availability and development, it also allows attackers to investigate the honeypot framework, aiming to find ways to detect it. This can be done with relative ease as most honeypot deployments use default configurations. Cabral et al. proposes a framework that makes configuring Cowrie deployments easier, while greatly increasing deceptiveness [9].

2.2.4. Research gap

As shown above, SSH and SSH honeypot have been the subject of many research projects. Recently, a lot of research has been conducted on how attackers can be prevented to detect they are targeting a honeypot by making the shell environment as realistic as possible. We argue that an important step is missed here as the initial interaction of an attacker with a honeypot is the SSH handshake and authentication. There is no research on how different behaviours of the server during this process impact how an attacker interacts with the honeypot. It might be possible that widely-used honeypots give away their identity during this process and certain attackers stop their efforts before reaching the interactive-shell stage. By presenting different server behaviors to the attackers and analysing their response we can get a better understanding of the intelligence of these attackers and how to build better honeypots in the future. Currently there is no easy way to gather data in order to fill this gap as honeypots focus on gathering data on password lists used on brute-force attacks or to analyse attacker behaviour once logged in. Therefore, this thesis aims to develop a platform where this data can be easily gathered in future research.

3

Data Collection

In chapter 2 we have summarised previous work related to threat intelligence research on SSH honeypot data. From this we have concluded that a research gap exists when focusing specifically on the handshake portion of the SSH protocol. To be able to do threat intelligence research on this topic we need to find out what data from the SSH handshake we need to gather and how to gather it. This chapter focuses on determining what data is useful to collect. After this, chapter 4 discusses the gathering method.

3.1. Available Data

Since our aim is to deploy a honeypot for threat intelligence on SSH attackers our only data source is the SSH server run by the honeypot. To get as much insight as possible into the different adversaries we need to extract information on the client side of the SSH connection from the server side. In this section we discuss what data we want to gather on this server side.

To make a decision on what data to gather during the SSH handshake we start by listing all data available. To not limit ourselves to any current SSH honeypot technologies we look at this from a protocol and production server point of view. This means any information available either in the protocol data or internal server data should be considered. To keep the scope of this thesis manageable, we focus on the most prevalent SSH server around the globe: OpenSSH [2].

The following connection specific information is present in a production OpenSSH server, focusing on the handshake:

1. Protocol:
   (a) Client version string; this indicates which SSH implementation a client claims to run.
   (b) Client protocols suggestion; this is often determined by the SSH implementation used but can be set manually.
   (c) Client authentication credentials; this can be a private key or a username/password pair.
   (d) Service requests by the client; SSH service requests can be sent before the client is authenticated and should therefore be considered as part of the available data.
2. Network:
   (a) TCP header fields; this includes fields like TTL, sequence number and window size.
   (b) Packet timing; time between certain packets, this can be influenced both by the client software and the network between client and server.
   (c) Traffic on other ports; this can indicate for example port scans hitting the honeypot.
3. Server:
   (a) Server logs; general output from the SSH server per connection.
   (b) Process Execution Monitoring Output; we can follow exactly what part of the OpenSSH server code is executed to catch any small difference in connection logic.

Whenever we want to gather data on adversarial behaviour, we want to collect as much data as is realistically possible as any information can help during analysis. For the mentioned protocol data points this is no issue as we have easy access to the data and the amount of data per connection is limited. For the network data points everything can be captured using so-called TCP dumps of all network traffic on the used network interface. These TCP dumps can be generated by existing and efficient applications that store all network packets, sent or received, into PCAP files that can be processed later. This can take up a reasonable amount of storage space, but this is worth it, considering we can go over the PCAP files later and extract any features we might need as we do not discard anything. The server logs are already outputted to log files and are therefore easily collected. Point 3b is more complicated and will be discussed in the next section.

3.2. SSH Server Process Monitoring Output

While we could theoretically adjust the OpenSSH server to output each line of code it is executing, this is not a viable option for practical data gathering and analysis as this would create an enormous amount of data. On top of this, the OpenSSH server code contains many lines of code that are always executed, or always executed together as a block, independent of the connection state, and therefore do not provide insights in to attacker/client behaviour. Therefore we choose not to focus on outputting executed lines of code, but instead branches and statements that depend on connection specific data. Using these relevant branches we can build up a logic flow through the server code for each connection during analysis to get insights into how the server handles different connections. These flows can also be used to catch any small difference in server behaviour which could allow us to differentiate between different clients, different versions or different distributions.

To achieve this, we forked the OpenSSH-Portable Github repository and edited the code manually [1]. OpenSSH-Portable is an open-source port of OpenBSD's OpenSSH to most Unix-like operating systems and includes a full SSH-2 server and client. We took the most current stable released version branch at the time of starting this work: `V_8_1`. The OpenSSH-Portable project provides a custom logging framework that can be accessed throughout the codebase. Different SysLog facilities can be set up to output the generated logs to different services, the most straightforward one being log files. We used this logging framework to add log statements on any important events or branching decisions in the server logic. These log statements include an identifier to easily filter on these logs, as well as identifying which source code file and what function is being processed and what specific part of that function. In Listing 3.1 we show a part of the modified code. This code is part of the main function of the sshd.c file, which is the main source file for the OpenSSH server. This specific part is placed right after the main listening process forks off to handle a new connection. The OpenSSH server has one listening process and forks of a new child process for each connection, which then handles all state management for that connection. Since a clean process is forked for each new connection, we can be sure that connections can not influence each other, or influence the OpenSSH server state in such a way that the behaviour is altered when handling new connections.

In the presented code we see that a new state variable is made for the new connection. We add a log statement in all of the if-statements that depend on this connection state `ssh` in this piece of code as they are all dependent on this connection-specific state, and can give an insight into the behaviour of the connection. We do not add a log statement for the last if statement though, as this only depends on the server's `options` variable, and the code in this if's body (`set_process_rdomain`) will either be executed for all connections or no connections, depending on the server's options, independent of the connection state. The code also contains some calls that various functions that alter that current connections, like `set_packet_set_server(ssh)` and `channel_init_channels(ssh)`. While these statements do use the connection state, they will be always be executed, no matter this state, so we do not need to add a log statement here. We do however, log any decisions in the source code of these functions that depend on the connections state.

Listing 3.1: Part of the modified OpenSSH_V_8_1 sshd.c->main() code

```
/*
 * Register our connection.  This turns encryption off because we do
 * not have a key.
 */
if ((ssh = ssh_packet_set_connection(NULL, sock_in, sock_out)) == NULL) {
    logit("[THESIS-%s-%s-6] Failed to register connection", __FILE__, __func__);
    fatal("Unable to create connection");
}

the_active_state = ssh;
ssh_packet_set_server(ssh);
check_ip_options(ssh);

/* Prepare the channels layer */
channel_init_channels(ssh);
channel_set_af(ssh, options.address_family);
process_permitopen(ssh, &options);

/* Set SO_KEEPALIVE if requested. */
if (options.tcp_keep_alive && ssh_packet_connection_is_on_socket(ssh)) {
    logit("[THESIS-%s-%s-7] Setting SO_KEEPALIVE", __FILE__, __func__);
    if(setsockopt(sock_in, SOL_SOCKET, SO_KEEPALIVE, &on, sizeof(on)) == -1) {
        logit("[THESIS-%s-%s-8] Failed to set SO_KEEPALIVE", __FILE__, __func__);
        error("setsockopt SO_KEEPALIVE: %.100s", strerror(errno));
    }
}

if ((remote_port = ssh_remote_port(ssh)) < 0) {
    logit("[THESIS-%s-%s-9] ssh_remote_port failed", __FILE__, __func__);
    cleanup_exit(255);
}

if (options.routing_domain != NULL)
        set_process_rdomain(ssh, options.routing_domain);
```

With this process we went through the source code of the OpenSSH server to add logs wherever any branching decisions are made based on the connection state. This can be because it directly uses the `ssh` variable, or it depends on the contents of a received packet, et cetera. The result is chronological list of logs that we can use to generate a state machine of the server execution for a specific connection. Two important pieces of information that the OpenSSH logging framework adds are the datetime and a connection ID. This ID can be used to group all logging statements for a single connection together. An example logging statement is given below:

```
Oct 27 09:53:42 sshd[32]: [THESIS-packet.c-ssh_packet_set_connection-1] Setting connection
```

This method of data collection gives us a list of states for each connection id. We can visualise this by rendering it as a state machine to show the flow of the server execution. The full image is rather large and can be found in Appendix A, but to illustrate, we show a small part of the state machine in Figure 3.1. Here we can see the server going through a couple of functions relating to privacy separation pre-authentication in multiple source files. We see that the numbers indicating what part of the function is executing do not simply increment. This is because certain branches are skipped, and this is what gives us an idea about the difference in behaviour depending on the client/connection. We further discuss these analyses in chapter 5.

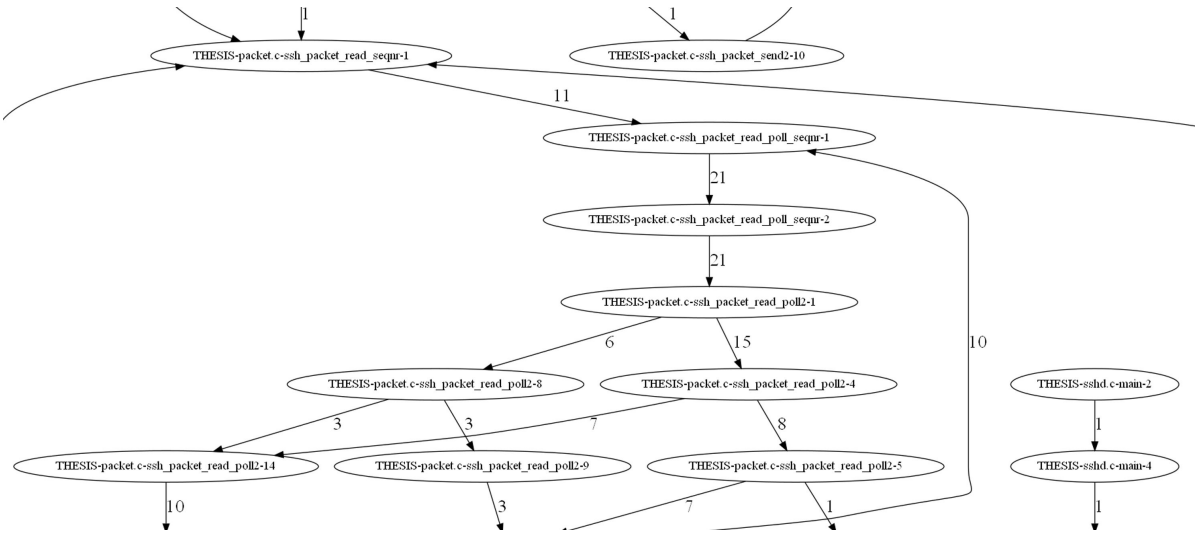

Figure 3.1: Part of OpenSSH server state machine

4

MD-Honeypot Framework

The previous chapter discussed how we can collect useful data from the OpenSSH server in order to analyse client behaviour using any data available on the server side. However, in order to apply this practically to gather data, we need a framework that deploys servers and collects their data in one place. Therefore, we developed the Massively-Distributed Honeypot Framework (MD-Honeypot). This chapter discusses the requirements and design decisions we formulated while developing this framework, outlines its features, and highlights the lessons we have learned during this process.

One of the goals of this thesis is to provide a practical framework to gather threat intelligence research data on the SSH handshake on a large scale. We have three main goals for this framework. First of all, we want our framework to be able to handle many different server agents. This provides us with the flexibility to deploy the framework in many different research contexts in the future. Secondly, the framework must support large-scale distributed deployment. The significance of our findings will increase with the amount of data collected. Also, the distributed deployment allows for comparing different server versions or geographical locations in our analysis. Lastly, the framework should allow for easy data aggregation in one spot and be easy to deploy and manage.

In some previous works, AmpPot and Honeytrap were used for data collection [3][20]. However, AmpPot was not available for use because the source code was not publicly available, and Honeytrap used different agent sensors that were all redirected towards a central agent. This poses a problem because that way traffic would be forwarded back and forth from the honeypots to the central agent. This does not allow for true large-scale distributed deployment and is a limiting factor in the long term. To the best of our knowledge, there are no other frameworks available that fulfil the requirements needed to conduct this experiment. Therefore, for this research, a new framework has been created: the MD-Honeypot framework.

Together with my colleagues Paul van der Knaap and Ahmet Gudek, we developed a new versatile honeypot framework that can facilitate A/B testing with a large number of distributed honeypots. The newly built framework supports both the TCP and UDP protocol, but this thesis only focuses on the TCP functionality as SSH is a TCP-only protocol. For additional information regarding the UDP elements of the system, the work by van der Knaap and Gudek can be consulted [17]. The core parts of the system were jointly developed. On a high-level, the following items were mostly developed by the researcher of this work: all TCP related functionality, including the TCP agent dashboard, TCP protocol stack/handler implementation, the handlers for all TCP services, a portion of the Docker/virtualized integration, a large share of the dashboard server, a large share of the Python to JAVA rewrite and a share of the data logging system and the data retrievers for the logging server. This chapter discusses the design- and development process of the framework, as well as the final product that forms the basis for the SSH specific part of the framework that this thesis proposes. The MD-Honeypot framework was constructed in several phases: first, the requirements were set, which is discussed in section 4.1. Then, the high-level framework was designed and prototyped, which is presented in section 4.2. Finally, the SSH specific part of the framework is covered in section 4.3.

4.1. Requirements

We have created a list of requirements that satisfied the needs of all three theses that used this framework. The requirements were separated into two parts: functional and non-functional requirements. These requirements were created with maintainability and future use in mind. For the requirements, the following list was created:

Functional requirements

- The framework should support at least the following modes of operation:
  - Emulated (for certain protocols): implement a protocol in the system itself so the system can handle a request of a particular protocol directly without external calls of forwarding.
  - Virtualised: The system should support running software in virtualised containers, so it is easier to encapsulate existing software to run on different machines without the need to distribute different versions of the system. It also provides security benefits as the software is contained in its own container.
  - Tunnelling (for TCP): allow incoming packets to be encapsulated and be forwarded to another honeypot so the packet can be handled there. This is necessary as it shields the system where the connection comes in from potential vulnerabilities in the set-up. This is especially useful for SSH or Telnet honeypots, where an adversary can execute commands on remote systems. A honeypot using this mode is called a forwarder.
- The honeypot should be configurable in different modes for different protocols simultaneously. For example, it should to be able to run a UDP service in emulated mode while forwarding (tunnelling) TCP SSH traffic towards another honeypot.
- Additionally, it should be possible to run different honeypot configurations on different ports simultaneously.
- The system should run on different platforms. It should run at least on Linux and Windows.
- There should be a central logging server to collect (aggregated) information about incoming packets and connections.
- There should be a central configuration server, so configurations can be (re)loaded dynamically without requiring a redistribution of the software or its configurations.
- There should be a simple dashboard to view statistics, so participants that run our software can also view information about the data they contribute to the project.
- Components: the different components of the system should run independently of each other so the components can be deployed at different systems and their resources can be scaled up when required.

Non-functional requirements

- Choice of programming language: the main programming language of the project should provide adequate performance and stability. Furthermore, the programming language should be a language we have already worked with extensively, to prevent delays while trying to master a new programming language and keep the development time relatively short.
- Choice of frameworks: if possible, choosing proven technology and frameworks is preferred over using experimental software and frameworks

4.2. Design & Development

The design and development process of the MD-Honeypot framework consisted of a number of phases. We used the list of requirements to create a system design that satisfied all those requirements.

4.2.1. System Design

The system design we constructed based on our list of functional requirements over a number of iterations is shown in Figure 4.1. The components are split into core components and optional components. While the core components are vital to be able to gather data with the framework, the optional components are mainly for added usability and insights.

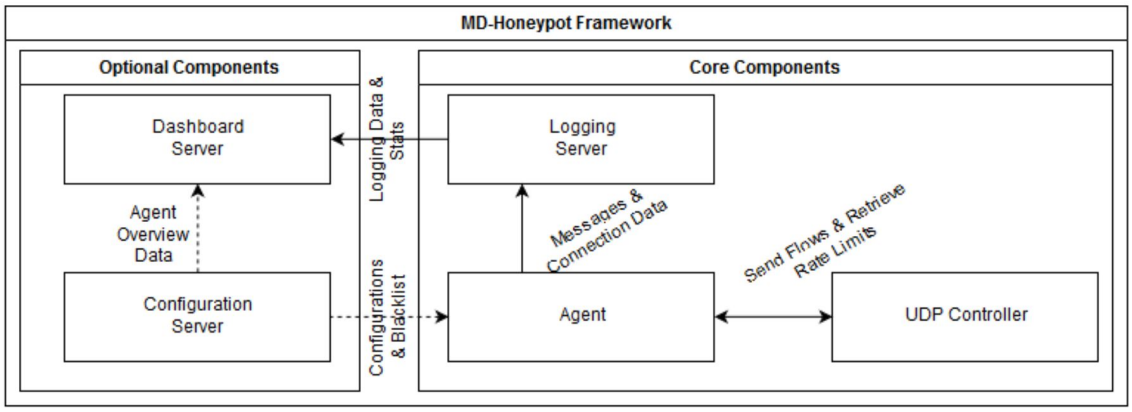

Figure 4.1: Schematic overview of the MD-Honeypot Framework

The final design consists of five components:

- **Agent**: this component runs the actual honeypot server, using either the UDP or the TCP protocol. The agent can either retrieve a configuration from the configuration server, or be started using a local configuration file. The agent sends all logging data to the logging server and has no capability to store any logging data itself to make sure we follow the principle of separate dependencies. For testing purposes, a logging server can be run on the same machine as the agent. The agent also has the option to run in tunnelling mode, in which case all traffic is forwarded to another agent. This works like a proxy server and can be used to avoid handling potential malicious traffic in unsecured network environments.
- **Logging Server**: this component accepts logging data from the agents and stores it in a database for later analysis. The logging server can also be queried for specific logging data and statistics.
- **UDP Controller**: this component is used in Paul van der Knaap's work on DDOS attacks to actively monitor the bandwidth used by the agents and limit their rates if needed.
- **Configuration Server**: this optional component can be used to set specific configurations for agents, or set up A/B testing groups which can be automatically allocated to different agents. The dashboard also receives the current status of connected agents and can be queried for an overview of running agents and their statistics.
- **Dashboard Server**: this optional component gives an overview of the entire system. It can show agent statistics queried from the configuration server or statistics on the incoming connections queried from the logging server.

4.2.2. Development & Testing

The MD-Honeypot framework has been implemented using a combination of Python servers and JAVA agents, as well as a MariaDB storage backend. The initial prototype was tested during the Network Security course at the TU Delft, taught by Dr. C. Doerr. This first prototype included the full implementation of all components except the service-specific agents. The agents that were used included a log-only UDP service that did not reply to any traffic and a simple SSH honeypot that logged connection origin and username/password attempts. The students of the course were asked to run an agent for 72 hours, either at home or using virtual private servers (VPSs). The agents were run in tunnelling mode to not handle any potential malicious traffic in the home networks of the students. The traffic was forwarded to agents we ran on VPSs on an external network. The students then completed a small assignment in which the data their agent created had to be analysed. During the initial development and this testing phase we learned a number of valuable lessons with respect to developing a distributed honeypot framework.

Home network deployment
Normally we would run such a large-scale distributed deployment on cloud-based services where we have control over all network and firewall settings. The test agents ran by students allowed us to try a home network deployment. To facilitate this we automated deployment using the Python package manger and port forwarding using the Universal Plug and Play (UPnP) protocol [4]. While we followed the appropriate standard practices and used a verified implementation of the protocol, it still caused many issues during this test. This was either caused by the UPnP settings not properly propagating through a multi-layered network, or certain routers not applying the correct port forwarding, even though UPnP was enabled. The lesson learned here that it is very difficult to get this to work reliably in a large amount of different network and system environments. We therefore advice future deployments to use custom VPS images that run on cloud services.

System resource management & scalability
We performed various scalability tests to ensure the framework could handle the high load that comes when deploying a large-scale distributed system. During these tests we learned that when dealing with big flows of smaller packets, the Linux kernel runs into problem when running the program on a single core that should be more than capable. This problem causes the CPU to be disproportionately loaded by networking processes in the kernel when the number of packets increases. This problem is very relevant for our application as the SSH protocol uses a lot of fairly small packets during the handshake phase. The only bigger packets that are being sent are the packets that contain the algorithm proposals. This niche problem was also discussed in a Cloudflare post by Majkowski, which gives some recommendations on improving performance[21]. The main lesson here is that these large-scale networking frameworks that have a focus on smaller packets should always make sure to support multiple cores, independent on individual core performance. This caused us to rewrite the entire agent implementation from Python to JAVA, as Python does not support true multi-core execution. Python does support multi-threading, but all threads are still interlaced on the same physical core. While JAVA does not support raw sockets, which could be used to manipulate TCP flags if future research would need this, this is a trade-off we needed to make. Other system resources proved to be no significant problem and could be scaled when needed.

4.3. MD-Honeypot-SSH
Aside from the general MD-Honeypot framework that was jointly developed, this thesis individually proposes the MD-Honeypot-SSH agent. This is an implementation of the agent component of the MD-Honeypot framework that is designed to work with the other parts of the framework to allow the ease of distributed deployment and data collection, and this combination should answer our second research question. For the development of the MD-Honeypot-SSH agent, a list of requirements specific to the MD-Honeypot-SSH agent was set up.

4.3.1. MD-Honeypot-SSH Requirements
The aim of these requirements is to gather data during the SSH handshake process with minimal chance to be detected as a honeypot. As we know, attackers do not want to share their methods if they can avoid it, meaning they might try to detect if they are targeting a honeypot and abort when they detect one. This can be done through various different signals, like something as simple as looking at the SSH server banner that might include a string unique to a certain honeypot, or something more complex like looking at the TCP header values. In any case, it is impossible to know all the signals that attackers may be looking for at this moment, or may look at in the future. Therefore, we start with a legit OpenSSH implementation, which is the most used SSH server [2], so that we know for sure it cannot be detected as a honeypot and modify that to suit our needs. As a result, the following list of requirements was formulated:

- The MD-Honeypot-SSH agent should:
  - use a widely-used OpenSSH server version as a base to handle the connections.
  - forward incoming connections to this OpenSSH server by interfering with the connection as little as possible, meaning minimal changes in for example timing and TCP header values.
  - save pcap dumps from all network traffic without interfering with the connection.
  - minimise the introduced overhead to keep high performance.

  - have the capability to change the SSH server behaviour if needed by using a configuration file or the configuration server. These configurations could alter the SSH banner text or support key exchange algorithms for example.

- The used OpenSSH server code can be modified to allow as much logging as needed as long as it does not interact with the connection or alters the state of the server thread handling the connection.

4.3.2. MD-Honeypot-SSH Development & Testing

To satisfy these requirements, it was decided to run the modified OpenSSH server in a Docker container and let the MD-Honeypot-SSH agent forward the packets from the (potential malicious) clients to the Docker container and back. This allows the MD-Honeypot-SSH to still store all required metadata like connection source and pcap dumps to be used in combination with the rest of the MD-Honeypot framework. While this does not alter the contents of the TCP packets in any way, it does change the TCP header values as they are sent as a new packet when forwarded. This could be solved by copying all TCP header values except for the source/destination addresses using raw sockets, but as discussed before this was not possible with our JAVA implementation. Our testing however showed that the only differing values, when looking at a single connection, were the TCP window size and the TCP sequence numbers. However, when looking at a larger sample-size for the used TCP sequence numbers and window sizes there is no significant difference between the used ranges, and they are just randomized for each connection. This gives us enough confidence that this will not significantly impact the gathered data.

We tested the framework extensively in a local environment, but this thesis does not provide the results of a full large-scale deployment. This would need a lot of systems to deploy the agents on in different parts of the world. While this can be made a lot easier by using VPSs from different cloud providers we could not secure the funding to do a significant large-scale deployment. This thesis therefore presents the completed system and does a number of small-scale experiments that are discussed in the next chapter.

4.3.3. MD-Honeypot-SSH Future Improvements

In the future, a number of things could be improved upon with respect to the MD-Honeypot-SSH agent and the MD-Honeypot framework as a whole. Firstly, agent code could be ported to a language that has both access to raw sockets and provides enough performance for large-scale deployment. This would allow us to forward the traffic to the docker container without changing the TCP headers, or modify any TCP headers as needed for future research. As discussed this should not have any significant impact on the collected data, but it is always wise to mitigate any potential influences. Secondly, the system should be deployed on a large-scale to truly test the capabilities and to provide a very valuable dataset to analyse. Finally, the current framework provides the implementation of a single OpenSSH server version for data collection as the support requires a significant amount of manual work per version. If this process could be automated in the future, we could do research on the impact of different SSH server versions on attacker behaviour. To conclude, this framework in combination with the SSH agent is a fully working data gathering platform that can be used for all kinds of future research.

5

Initial Results

This chapter outlines the results of some initial tests done with the MD-Honeypot-SSH framework. The experiments outlined have all been performed in a designated test environment with the connections/attacks being generated by ourselves. The aim of these experiments is to show that the framework produces the desired data and is a valuable asset for future research. We will discuss a number of experiments that show how we can identify equal, different, and similar connections.

5.1. Data

As discussed before in chapter 3, the MD-Honeypot-SSH framework generates a lot of valuable data for each incoming connection. We want to group these connections into similar users/clients as a way of categorising them. This can be useful to blacklist/whitelist certain clients with specific characteristics. There are two main ways to do this:

1. Only look at the real-time, connection-specific data. This means we group the connection based on data that is immediately locally available, which allows for quick processing. The drawback is that we cannot use any aggregate information we have gathered that may be related to this connection.

2. Look at all available data related to the connection. This means also taking into account for example all connections from the same IP-address or subnet, the combined list of client authentication credentials for these connections, port scans from the same IP-address or subnet, or similarities in timing between connections. The advantage here is that we have a lot more data available and we can pool data from multiple distributed honeypots together to get the best clustering possible. The drawback however, is that this takes a lot longer to process and is more difficult to set up.

For these initial experiments with the framework we decided to use only the real-time, connection-specific data. Looking at the list of available data points this means we will be using:

- Client version string: this should be the same for connections from the same client.

  ```
  Example: SSH-2.0-OpenSSH_8.5
  ```

- Client protocols suggestion: again this should be the same for the same client. Note that this is a long string (1500 characters depending on client configuration) so we will be creating a MD5-digest from this string and comparing this digest.

  ```
  Example suggestion:
  "curve25519-sha256,curve25519-sha256@libssh.org,diffie-hellman-group-exchange
  -sha256,diffie-hellman-[...],zlib@openssh.com;none,zlib,zlib@openssh.com;;"
  MD5-digest: A29608DF28E5F4F2F1AD92AACC999B11
  ```

- State machine generated from logfiles. These state machine include the information from the general server logs as well as the service requests made by the client. An example of such a state machine can be found in Appendix A. To compare these state machines between two different connections, we have developed an algorithm that identifies which states and transitions are common between the two state machines, and which states and transitions are only present in one of the state machines. The algorithm also identifies if the amount of transitions between the same states differs from one state machine to the next. It then generates a visual of the state machine where the differences are coloured green or red. Each colour represents the states and transitions that are specific to one of the state machines. A full example of such a comparison state machine can be found in Appendix B. With this method we can identify how much two state machines differ and in which way, just more transitions, or entire new states.

We will not be using the TCP header fields, packet timing and traffic on other ports as this requires analysis over multiple connections. We will also not be using the provided client authentication credentials as this can change each connection from the same client, for example during a brute-force attack.

5.2. Experiments

In these experiments we will compare a number of connections that are listed in Table 5.1. These connections are made from two different clients (Git bash terminal on Windows 10 and Raspbian Jessie terminal). Depending on the connection we enter no, 1 or 3 password(s) before the connection is terminated. With no or 1 password the client terminates the connection, while the server automatically terminates the connection after 3 failed login attempts. We will use the identifier to refer to the connections in this text, while the process ID is used in the log files to separate the logs per connection.

Identifier	Process ID	Client	Username	Passwords
A	179	Git bash Windows 10	test	[test1, test2, test3]
B	181	Git bash Windows 10	test	[test1, test2, test3]
C	183	Raspbian Jessie	test	[test1, test2, test3]
D	185	Raspbian Jessie 10	test	[test1, test2, test3]
E	187	Git bash Windows 10	test	[test]
F	189	Git bash Windows 10	test	-

Table 5.1: Experiment connections

5.2.1. Equal connections

Connection pairs (A, B) and (C, D) are made from the same client, with the same input. The only differences in the generated data are the process ID, timestamps and source port (which is generated randomly for each new connection a client makes). We do not use these fields to group the connections though, and Table 5.2 shows the relevant data fields. This shows indeed that the pairs can be grouped together based on the resulting data. When running the state machine comparison algorithm, no differences are detected within the pairs, and the output is the same as generating a state machine from a singular connection.

Identifier	ClientVersionString	ProtocolsSuggestionDigest
A	SSH-2.0-OpenSSH_8.5	A29608DF28E5F4F2F1AD92AACC999B11
B	SSH-2.0-OpenSSH_8.5	A29608DF28E5F4F2F1AD92AACC999B11
C	SSH-2.0-OpenSSH_6.7p1 Raspbian-5+deb8u3	A463904C8337B3F13B3D29D1C814289E
D	SSH-2.0-OpenSSH_6.7p1 Raspbian-5+deb8u3	A463904C8337B3F13B3D29D1C814289E
E	SSH-2.0-OpenSSH_8.5	A29608DF28E5F4F2F1AD92AACC999B11
F	SSH-2.0-OpenSSH_8.5	A29608DF28E5F4F2F1AD92AACC999B11

Table 5.2: Connections from the same client

5.2.2. Different connections

Next we compare two connections with the same input, but from different clients. Connections A and C both have 3 passwords entered, but one is made from the Windows Git bash client and the other from a Raspbian Jessie terminal. In Table 5.2 we see that both the client version string and the protocols suggestion digest differ

between the two connections. However, these are configurable settings in the client and could be overwritten. When looking at the generated comparison state machine we see a number of differences between the two connections. The full comparison state machine is available in Appendix B, but we will discuss a smaller part here. Figure 5.1 shows a subset of the differences between the state machine for connection A and C. We see that, while the input is the same, the behaviour of the server differs between the two connections because the clients behave slightly different. Connection A (green) does an extra pass through the packet-read sequence, while connection C (red) adds two entirely new states in this sequence that are not present for connection A. This behaviour is not configurable and dependent on the client, and is therefore a reliable way to differentiate between two clients.

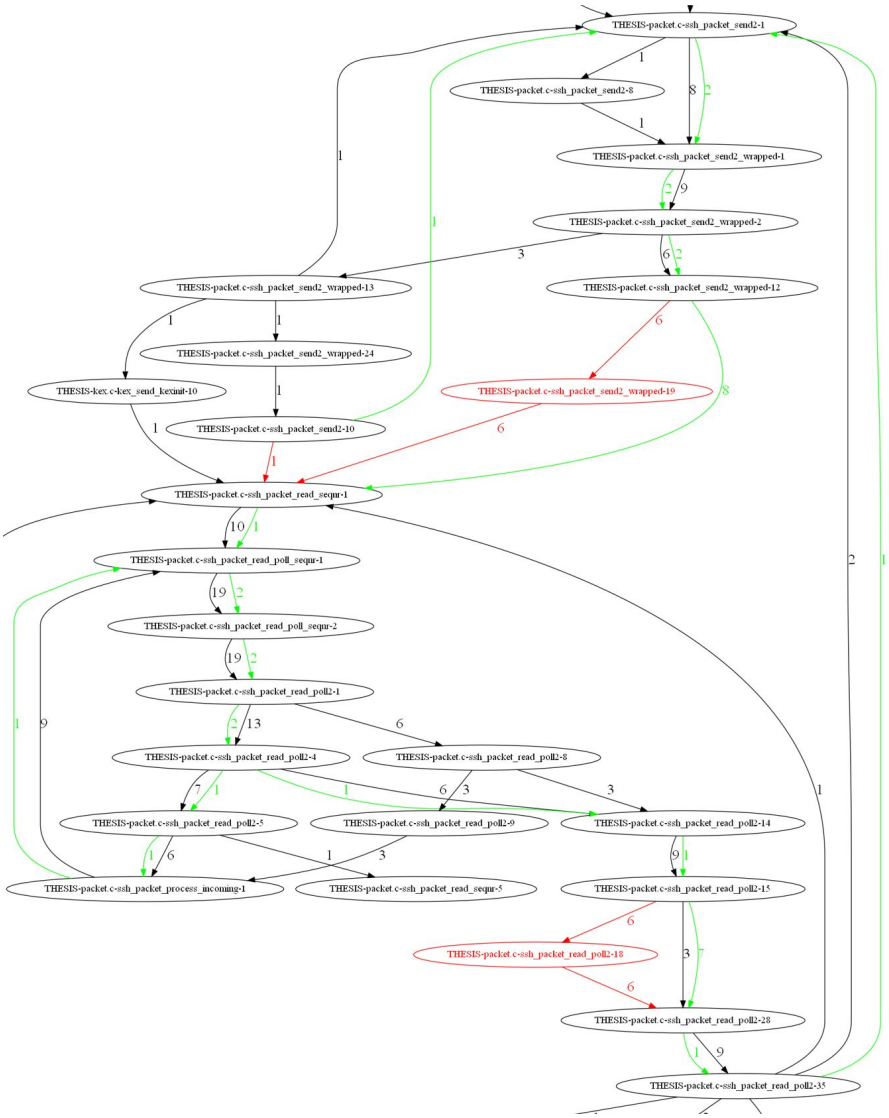

Figure 5.1: Part of comparison state machine for connection A and C

5.2.3. Similar connections

Finally we compare two connection from the same clients, but with different inputs. Connections E and F are both made from the Windows Git bash client, however E has a single password entered before terminating the connection, while F is terminated immediately without entering any password. We see in Table 5.2 that again the client version string and protocols suggestion digest are the same, as is expected. When running the state machine comparison algorithm however, we see some clear differences. In the full comparison state machine, that is available in Appendix C, we see that in connection E (with password) a number of transitions and a few states are added, but connection F has no unique transitions or states. In Figure 5.2 we see a smaller part of this comparison state machine. We see that an extra pass is made over the packet-read sequence, but

also three added states:

- THESIS-auth2-passwd.c-userauth_passwd-1
- THESIS-auth.c-auth_log-1
- THESIS-monitor.c-monitor_child_preauth-7

As the names of these states already show, these are all states specific to authorization. This is expected as the authorization flow differs between the two connections. All other states are the same, which show the behaviour of the client is the same when disgarding the authentication behaviour. If we discard these authentication-specific differences, we can still group these connection together.

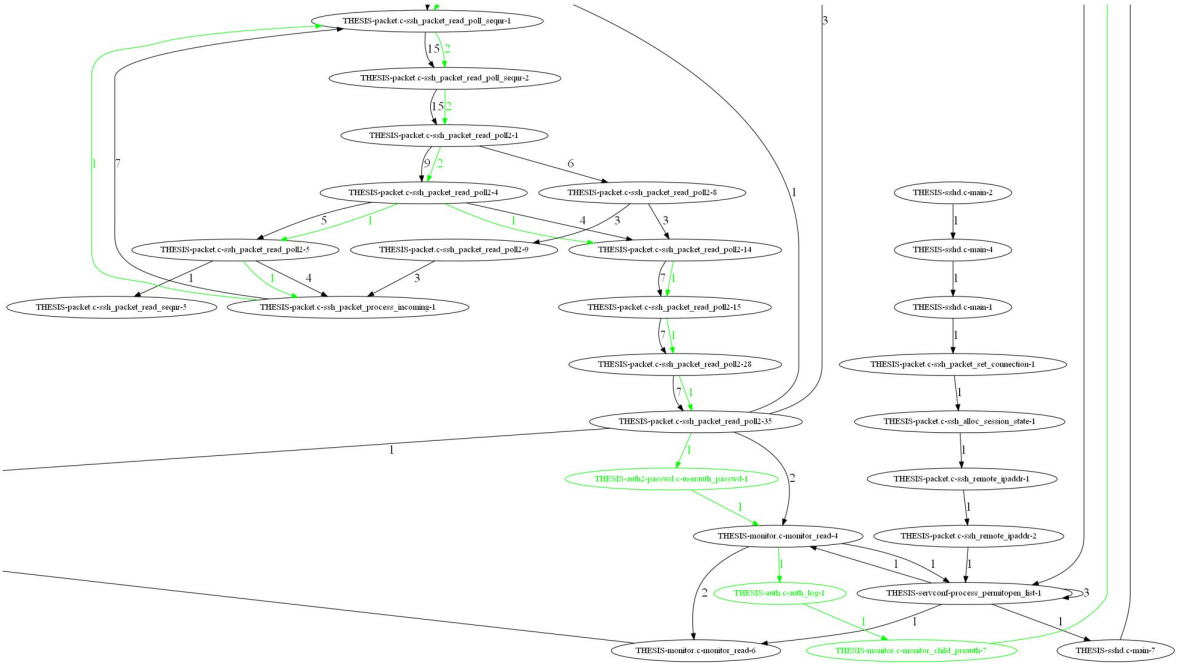

Figure 5.2: Part of comparison state machine for connection A and C

To conclude, we have shown that we can group connections together by the used client, by only using the real-time connection-specific data. This can be used in the future to white- or blacklist certain clients that are commonly used by attackers for example. This also shows the capabilities of the MD-Honeypot framework.

6

Evaluation & Conclusion

In this thesis we have investigated the current limitations of SSH honeypots, identified a research gap and proposed the MD-Honeypot-SSH framework to help close this gap. Furthermore, we performed some initial experiments with the framework to show it solves the identified problems. This chapter concludes our research findings in section 6.1. section 6.2 discusses the limitations of this thesis. After this, recommendations for future work are discussed in section 6.3. Finally, the contributions of this thesis are evaluated in section 6.4.

6.1. Conclusions

This section gives an overview of our research by answering our research questions.

RQ1: What data is useful to log during execution monitoring to conduct threat intelligence research on the SSH handshake?

As discussed in chapter 4, the aim of the framework is to collect every bit of data that could be useful for analysis in the future, that is feasible to collect when deploying on a large scale. We showed that all available data points, except a full line-by-line trace of the server code falls within this feasibility constraint and are therefore stored. With regards to the trace of the server code, we performed a manual pass through the code to log any connection-specific data and branching decisions. We showed that by doing this the amount of data collected falls within the feasibility constraint for large-scale deployments. We also showed that no information was lost by using this method.

RQ2: How can we collect SSH handshake data for threat intelligence?

To answer this research question we propose the MD-Honeypot-SSH framework. This framework allows for distributed, large-scale deployment of SSH honeypots. We developed the SSH honeypots to target the gathering of handshake data and do not allow any logins to lower the risks. In chapter 5 we showed that the data gathered using the framework can be successfully used to identify clients from only connection-specific data.

RQ3: How can we group connections by SSH client, using only real-time, connection-specific data?

We have shown in chapter 5 that by looking at the client version string, the protocols suggestion digest, and the generated statemachine, we can successfully identify when connections are made from equal or different clients. By discarding differences in the authentication flow this also works when clients provide different credentials when trying to connect.

6.2. Limitations

In this section we discuss limitations of this thesis. Some of these limitations can be tackled in future work.

The first limitation of this thesis is that the SSH implementation of the MD-Honeypot framework has not been tested in a real-world large-scale deployment. While the core parts of the framework has been used in a successful research in Paul van der Knaap's work[17], his work focused on UDP services. Since SSH works with TCP, this is no guarantee that no problems will arise during a large-scale deployment of the SSH implementation. It does however prove the potential of the framework in real-world scenarios.

The second limitation is that currently the SSH implementation of the MD-Honeypot framework can only gather handshake data. While this was the aim of this thesis, the framework would be more versatile if it could also allow logins to collect data when an adversaries has access to the honeypot. While this is not an easy task, this functionality could be added to the framework in a future work.

6.3. Future Work

The MD-Honeypot-SSH framework that was presented in this thesis provides a solid basis for future research into SSH using honeypots, specifically research aimed at the SSH handshake. Therefore the findings in this thesis form a solid basis for future research, both in using the framework to gather data and test hypotheses, and in improving the framework further. We have identified the following pointers for future work:

- The MD-Honeypot-SSH framework could be deployed on a large scale to gather data on the SSH handshake in a real world scenario. This data can then be used to test hypothesis on attacker behaviour and could then be used to improve security measures.
- The ability to group connections together based on connection-only data that we showed in chapter 5 could be used to implement a real-time white- or black-listing feature that could dynamically block the fingerprints of common attackers for example.
- The MD-Honeypot-SSH framework could be extended to also allow logins and present the attackers with some emulated shell. This way a more end-to-end type of research could be conducted with the framework instead of focusing on the handshake.
- The MD-Honeypot-SSH framework could be extended to support raw sockets. This way research could be conducted on how certain TCP flags influence the behaviour of attackers. While this is very unlikely to make any significant difference, it is the only piece of information the framework is currently unable to gather with respect to the SSH handshake. Extending it to support the gathering of the raw socket data would allow for truly exhaustive data gathering.
- Currently the MD-Honeypot-SSH framework only supports a singular OpenSSH version (V8.1). Future research could extend this to support multiple versions or even dynamically generate a framework agent for each new OpenSSH version. It could also be useful to extend the research to other common SSH servers.

6.4. Contributions

The main contributions this thesis makes are the following:

- We have shown which data is available to an SSH server during the SSH handshake, and we have determined which data points are useful to collect in a large-scale distributed honeypot framework.
- We have designed and implemented the MD-Honeypot-SSH framework that allows us to gather the SSH handshake data on a large scale.
- We have showed that this framework can be used to detect SSH clients by grouping incoming connections together with just the handshake data.
- We have proposed some ideas on how to use this framework in future research.

A significant part of these contributions is the ability of the MD-Honeypot-SSH framework to collect the state-logging data and generate the connection state machines which can later be used to, for example, group connections. While something similar can be done using the techniques shown by Fiterău-Broştean et al.[11],

the MD-Honeypot-SSH framework has a few distinct features that are valuable additions to the SSH honeypot research landscape. The main difference is that the tools developed by Fiterău-Broştean et al. are not designed as a honeypot, and therefore can not be deployed into a live environment to collect data. Only the types of the messages are considered and not the contents (authentication user/password, KEX proposal, et cetera). Furthermore, the state machines that are generated by Fiterău-Broştean et al. are purely learned by the response messages that the server in question provides. This does not take into consideration any internal decisions that do not result in different message types, but might result in different message content. Since the MD-Honeypot-SSH framework is directly plugged into the SSH server it has access to all internal state information, and can therefore be more useful for later threat intelligence research.

To conclude, the MD-Honeypot-SSH framework provides a robust way to gather a large amount of honeypot data on the SSH handshake, which can be used in future research to study the behaviour of attackers during this critical phase of the SSH protocol.

A

Full OpenSSH State machine

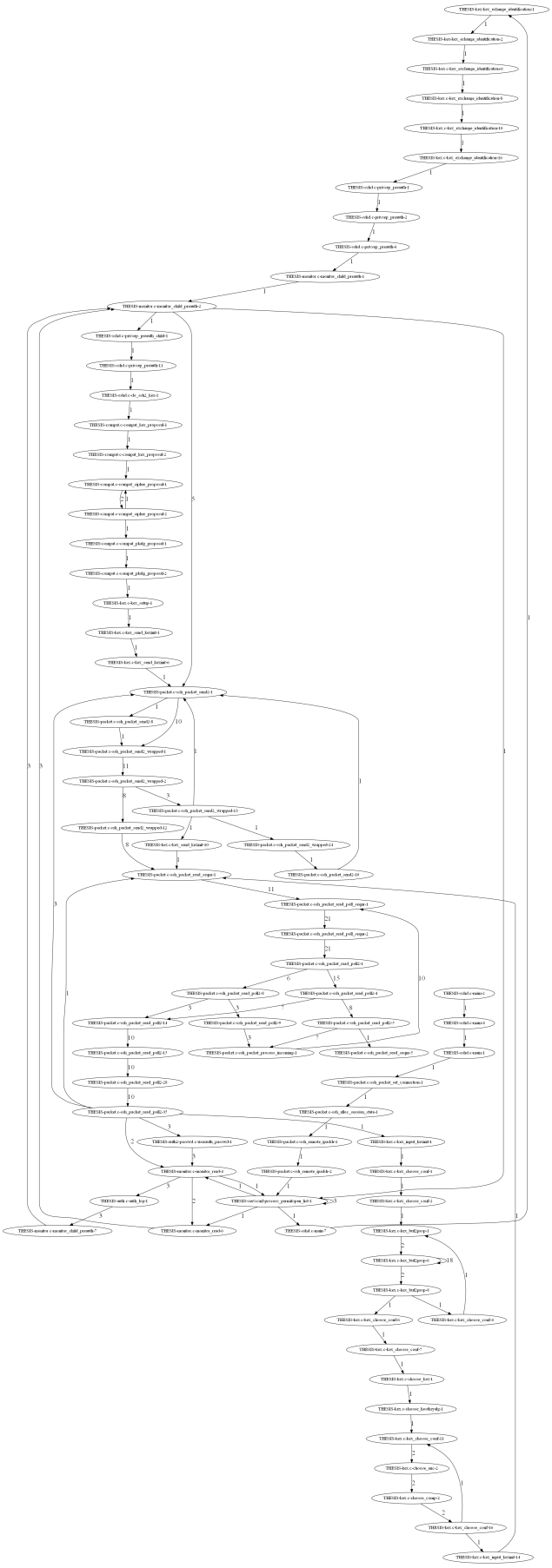

Figure A.1: Full OpenSSH Server State machine

B

Example Comparison OpenSSH State machine

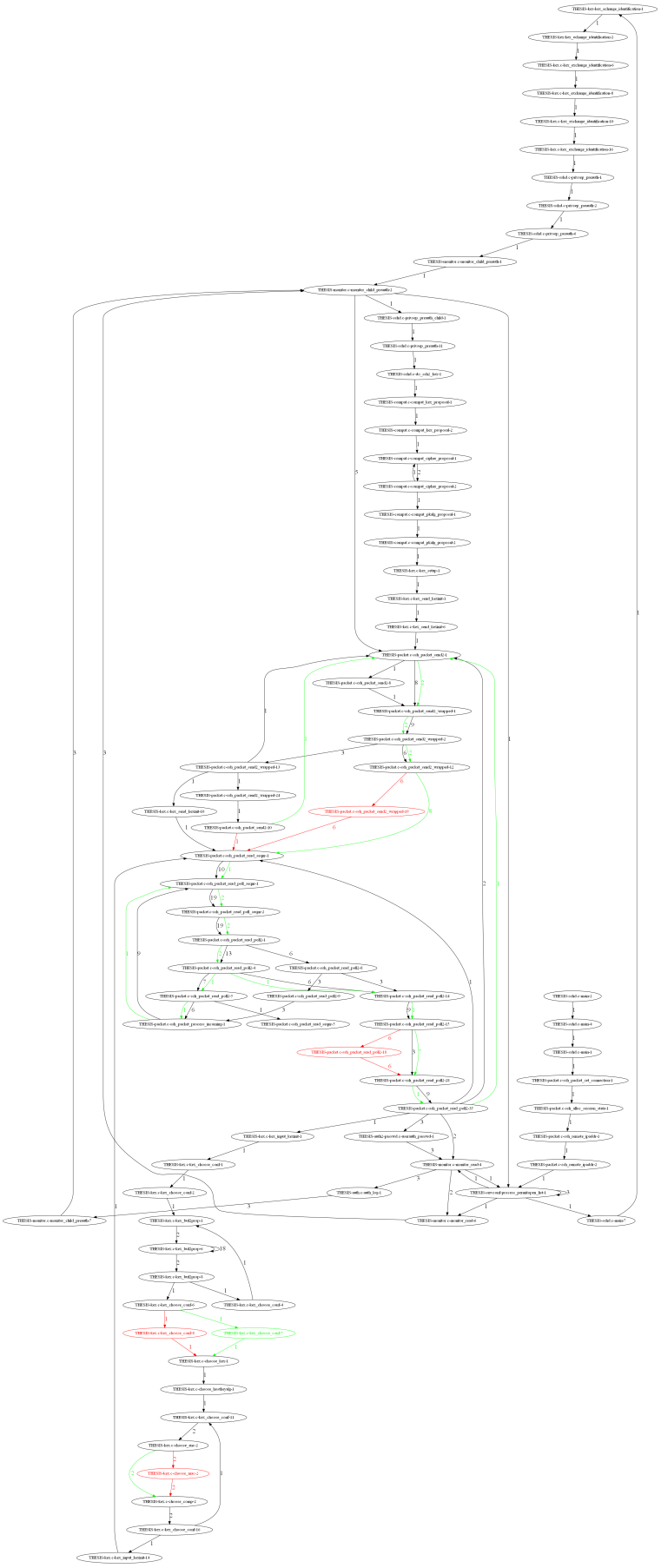

Figure B.1: Example Comparison OpenSSH Server State machine - same input, different clients

C

Comparison OpenSSH State machine Authentication

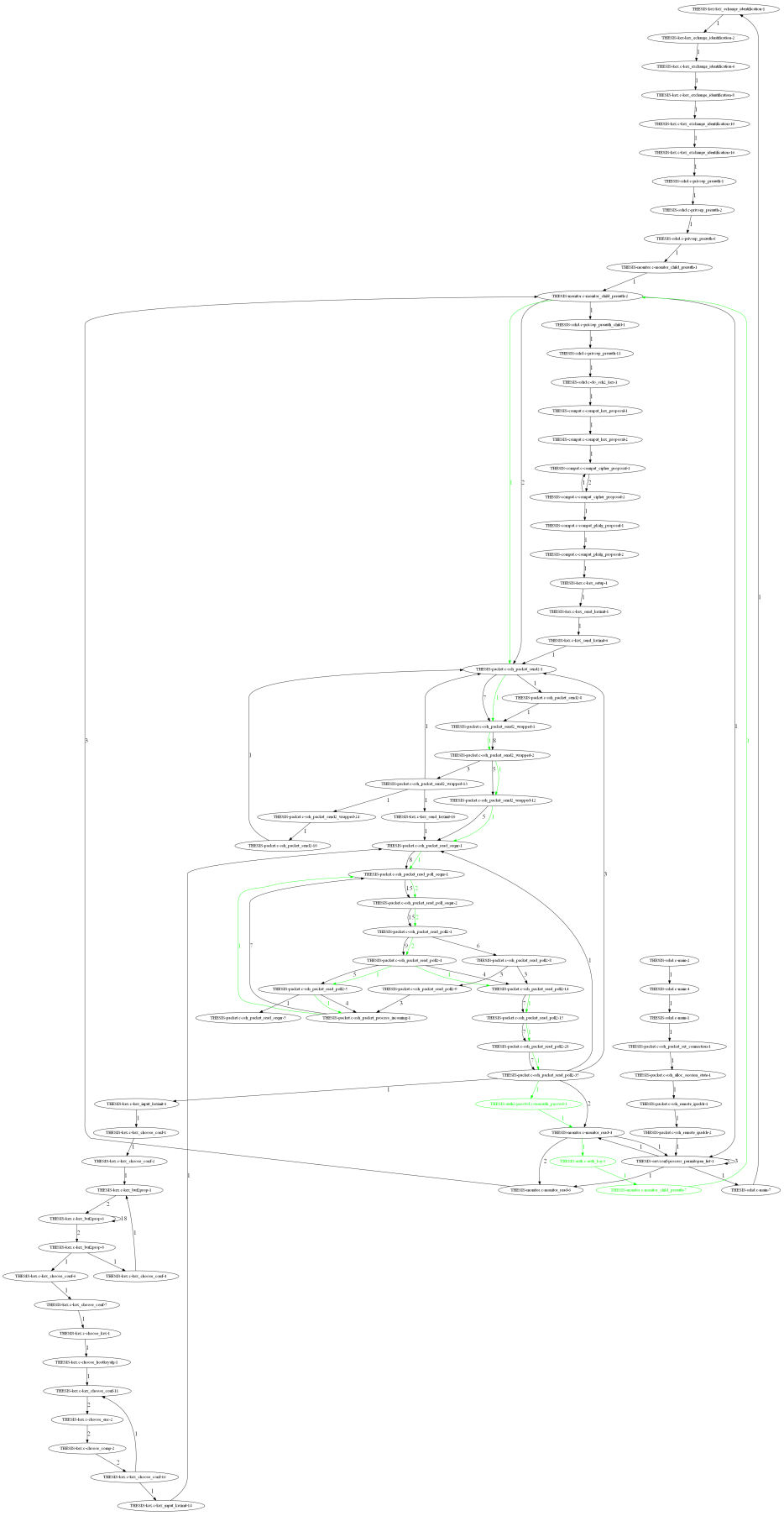

Figure C.1: Comparison OpenSSH Server State machine - same client, one with password, one without

Bibliography

[1] Openssh-portable. URL https://github.com/openssh/openssh-portable. (Accessed: 19 November 2021).

[2] Comparison of ssh servers. URL https://en.wikipedia.org/wiki/Comparison_of_SSH_servers. (Accessed: 19 November 2021).

[3] Honeytrap. URL https://github.com/honeytrap/honeytrap. (Accessed: 19 November 2021).

[4] Universal plug and play. URL https://en.wikipedia.org/wiki/Universal_Plug_and_Play. (Accessed: 19 November 2021).

[5] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against ssh. IEEE, 2009. ISBN 978-0-7695-3633-0. doi: 10.1109/SP.2009.5.

[6] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated encryption in ssh: provably fixing the ssh binary packet protocol. ACM Press, 2002. ISBN 1581136129. doi: 10.1145/586110.586112.

[7] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and provably repairing the ssh authenticated encryption scheme. *ACM Transactions on Information and System Security*, 7, 2004. ISSN 1094-9224. doi: 10.1145/996943.996945.

[8] Amine Belqruch and Abdelilah Maach. Scada security using ssh honeypot. ACM Press, 2019. ISBN 9781450366458. doi: 10.1145/3320326.3320328.

[9] Warren Z. Cabral, Craig Valli, Leslie F. Sikos, and Samuel G. Wakeling. Advanced cowrie configuration to increase honeypot deceptiveness, 2021.

[10] Paul Fiterău-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analsysis of dtls implementations using protocol state fuzzing. pages 2523–2540. USENIX Association, 2020.

[11] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of ssh implementations. pages 142–151. ACM, 7 2017. ISBN 9781450350778. doi: 10.1145/3092282.3092289.

[12] Daniel Fraunholz, Daniel Krohmer, Simon Duque Anton, and Hans Dieter Schotten. Investigation of cyber crime conducted by abusing weak or default passwords with a medium interaction honeypot. IEEE, 2017. ISBN 978-1-5090-5063-5. doi: 10.1109/CyberSecPODS.2017.8074855.

[13] José Tomás Martínez Garre, Manuel Gil Pérez, and Antonio Ruiz-Martínez. A novel machine learning-based approach for the detection of ssh botnet infection. *Future Generation Computer Systems*, 115: 387–396, 2 2021. ISSN 0167739X. doi: 10.1016/j.future.2020.09.004.

[14] Vincent Ghiette, Harm Griffioen, and Christian Doerr. Fingerprinting tooling used for ssh compromisation attempts. pages 61–71. USENIX Association, 2019. ISBN 978-1-939133-07-6. URL https://www.usenix.org/conference/raid2019/presentation/ghiette.

[15] Laurens Hellemons, Luuk Hendriks, Rick Hofstede, Anna Sperotto, Ramin Sadre, and Aiko Pras. Sshcure: A flow-based ssh intrusion detection system. 2012. doi: 10.1007/978-3-642-30633-4_11.

[16] Christopher Kelly, Nikolaos Pitropakis, Alexios Mylonas, Sean McKeown, and William J. Buchanan. A comparative analysis of honeypots on different cloud platforms. *Sensors*, 21:2433, 4 2021. ISSN 1424-8220. doi: 10.3390/s21072433.

[17] P C F Van Der Knaap. Md-honeypot on adversarial choices in drdos attacks, 2020. URL http://repository.tudelft.nl/.

[18] Brandon Knieriem, Xiaolu Zhang, Philip Levine, Frank Breitinger, and Ibrahim Baggili. An overview of the usage of default passwords. 2018. doi: 10.1007/978-3-319-73697-6_15.

[19] Ioannis Koniaris, Georgios Papadimitriou, and Petros Nicopolitidis. Analysis and visualization of ssh attacks using honeypots. IEEE, 2013. ISBN 978-1-4673-2232-4. doi: 10.1109/EUROCON.2013.6624967.

[20] Lukas Krämer, Johannes Krupp, Daisuke Makita, Tomomi Nishizoe, Takashi Koide, Katsunari Yoshioka, and Christian Rossow. Amppot: Monitoring and defending against amplification ddos attacks, 2015.

[21] Marek Majkowski. How to receive a million packets per second, 2015. URL https://blog.cloudflare.com/how-to-receive-a-million-packets/. (Accessed: 19 November 2021).

[22] Iyatiti Mokube and Michele Adams. Honeypots: concepts, approaches, and challenges. page 321. ACM Press, 2007. ISBN 9781595936295. doi: 10.1145/1233341.1233399.

[23] Marcin Nawrocki, Matthias Wählisch, Thomas C. Schmidt, Christian Keil, and Jochen Schönfelder. A survey on honeypot software and data analysis. 2016. URL http://arxiv.org/abs/1608.06249.

[24] Jim Owens and Jeanna Matthews. A study of passwords and methods used in brute-force ssh attacks, 2008.

[25] Akihiro Satoh, Yutaka Nakamura, and Takeshi Ikenaga. Ssh dictionary attack detection based on flow analysis. IEEE, 2012. ISBN 978-1-4673-2001-6. doi: 10.1109/SAINT.2012.16.

[26] C. Seifert, I. Welch, and P. Komisarczuk. Honeyc-the lowinteraction client honeypot. *Proceedings of the 2007 NZCSRCS, Waikato University, Hamilton, New Zealand*, 2007.

[27] Tanmay Sethi and Rejo Mathew. A study on advancement in honeypot based network security model. pages 94–97. IEEE, 2 2021. ISBN 978-1-6654-1960-4. doi: 10.1109/ICICV50876.2021.9388412.

[28] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh, 2001.

[29] Anna Sperotto, Ramin Sadre, Pieter-Tjerk de Boer, and Aiko Pras. Hidden markov model modeling of ssh brute-force attacks. 2009. doi: 10.1007/978-3-642-04989-7_13.

[30] L. Spitzner. The honeynet project: trapping the hackers. *IEEE Security and Privacy*, pages 15–23, 3 2003. ISSN 1540-7993. doi: 10.1109/MSECP.2003.1193207.

[31] Lance Spitzner and Marty Roesch. The value of honeypots, part one: Definitions and values of honeypots, 2001. URL https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=a8da0d16-65ae-405a-abeb-325af33a393d&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments. (Accessed: 20 june 2021).

[32] Jan Vykopal, Tomas Plesnik, and Pavel Minarik. Network-based dictionary attack detection. IEEE, 2009. ISBN 978-0-7695-3567-8. doi: 10.1109/ICFN.2009.36.

[33] Devran Yener, Tansel Ozyer, and Emin Kugu. Analysis and comparison of honeypot activities on two isp networks. pages 1–6. IEEE, 12 2021. ISBN 978-1-6654-0759-5. doi: 10.1109/IISEC54230.2021.9672371.