

JAMJAR: a Unix / Linux Honeypot

Anna Eisner,
Jani Gabriel,
Malte Schulten,
Oliver Werner

IT Sec Lab Work

Course of Studies: Enterprise- and IT-Security

Department of Media
Offenburg University

11.08.2024

Tutor

Prof. Dr. rer. nat. habil. Dirk Westhoff

Note on Gender-Neutral Pronouns

In this paper, the generic masculine form is used for better readability. Female and other gender identities are explicitly included, insofar as it is necessary for the statement.

Statutory Declaration

We hereby declare that this project work was independently completed by us without any unauthorized external assistance. In particular, we confirm that all sections taken verbatim, nearly verbatim, or in essence from publications, unpublished documents, and conversations are identified as such at the appropriate points within the work through citations, where the extent of the original quotations is also indicated. We are aware that a false declaration will have consequences.

Offenburg, 11.08.2024

Anna Eisner,
Jani Gabriel,
Malte Schulten,
Oliver Werner

Contents

1	Introduction	1
1.1	Motivation	1
1.2	What is a Honeypot?	1
1.3	Scope of this Project	3
1.4	Code	3
2	Considerations for the design of the system	4
2.1	Justification for this "budget" honeypot approach	4
2.2	How is the "fake" system simulated?	5
2.2.1	Using a different file system root	5
2.2.2	Docker	6
2.2.3	Journal / Internal	6
2.3	Activation / Deactivation of the System	6
2.3.1	When does it activate?	6
2.3.2	How does it deactivate?	7
2.4	Different Command Modification and Tracing Approaches	8
2.4.1	execve()	8
2.4.2	eBPF	9
2.4.3	ptrace()	9
2.4.4	Changing the original binary or using different binaries	10
2.5	Which commands are changed / intercepted?	10
2.5.1	Relationship between commands	14
2.6	Ways to hide honeypot processes and files	15
2.7	Logging	16
3	Implementation	18

3.1	Overview	18
3.2	Command intercept	19
3.2.1	Tracing commands via eBPF	19
3.2.2	Intercepting and Killing Processes	22
3.2.3	Challenges and Takeaways	24
3.3	Command handler	25
3.3.1	Directory	25
3.3.2	Network	28
3.3.3	Process	31
3.3.4	System	34
3.4	Logging	35
3.5	Additional Commands	35
3.5.1	cd	35
3.5.2	Piped Commands	38
3.5.3	iptables	40
3.5.4	cat	42
3.5.5	bash	42
3.5.6	echo	45
4	Results	48
4.1	Conclusion	48
4.2	Evaluation	48
4.3	Future Work	48
	List of Tables	V
	List of Figures	VI

1 Introduction

1.1 Motivation

The choice of focusing on "UNIX/Linux Honeypot" for our project work stems from its strategic importance in modern cybersecurity landscapes. UNIX systems are omnipresent and the standard in both enterprise and cloud computing environments, serving as the backbone for numerous critical services and applications. However, they are also prime targets for malicious actors seeking to exploit vulnerabilities and gain unauthorized access.

By exploring the realm of UNIX/Linux honeypots, we aim to address several key motivations. Firstly, these platforms represent a significant portion of the computing infrastructure worldwide, making them a priority for security professionals and attackers alike. Understanding the specific challenges and threats faced by UNIX/Linux systems is essential for developing robust defensive strategies.

Secondly, the open-source nature of UNIX/Linux distributions provides ample opportunities for customization and experimentation in honeypot deployment. This flexibility allows us to tailor our honeypot solution to match the intricacies of UNIX/Linux environments, enhancing its effectiveness in luring and monitoring attackers.

Moreover, UNIX/Linux honeypots offer unique insights into the tactics, techniques, and procedures employed by adversaries targeting these platforms. By analyzing the interactions between attackers and our honeypot infrastructure, we can gain valuable intelligence to inform threat detection, incident response, and vulnerability management efforts.

1.2 What is a Honeypot?

A honeypot is a tool used in IT security, designed to detect, deflect, or counteract unauthorized access or attacks on a network by luring potential attackers into a trap.

At its core, a honeypot in general is essentially a decoy system or network resource deliberately deployed to attract attackers and monitor their activities. Unlike traditional security measures that primarily focus on fortifying defenses, honeypots operate under the premise of

deception, enticing attackers into revealing their tactics, techniques, and motives. By mimicking legitimate services, applications, or data, honeypots serve as bait, enticing malicious actors to interact with them.

Honeypots come in various forms, ranging from low-interaction to high-interaction deployments. Low-interaction honeypots simulate only basic services and protocols, offering limited functionality to potential attackers. They are relatively easy to deploy and maintain but provide less insight into attackers behavior. On the other hand, high-interaction honeypots emulate entire systems or networks, allowing extensive interaction with attackers. While more complex to set up and manage, high-interaction honeypots yield richer data about attacker's methodologies and intentions.

There are many benefits in using honeypots. Firstly, they provide intelligence about emerging threats, attack vectors, and malicious activities. By analyzing the interactions between attackers and honeypots, defenders can gain insights into new attack techniques and vulnerabilities, enabling them to fortify defenses of real systems proactively. Additionally, honeypots can serve as early warning systems, alerting organizations to potential security breaches before they escalate. Also honeypots can divert attention of an attacker away from critical assets and infrastructure, buying some time for defenders to respond.

Deploying honeypots has some challenges and considerations. One major concern is the risk of inadvertently attracting legitimate users or automated scanning tools, leading to false positives and unnecessary alarms. Therefore, careful planning and segmentation are essential to ensure that honeypots do not interfere with normal operations. Furthermore, maintaining the authenticity and credibility of honeypots is crucial to deceive sophisticated attackers effectively. Regular updates, realistic configurations, and plausible data are vital to sustaining the illusion.

In general, honeypots offer a proactive and dynamic approach to cybersecurity, enabling organizations to gather actionable intelligence, detect threats early, and enhance their overall security posture. By using honeypots as part of their security arsenal, organizations can stay one step ahead of adversaries.

In conclusion, honeypots play a pivotal role in modern IT security by leveraging deception to thwart cyber threats effectively. Through their ability to lure, monitor, and analyze attacker's behavior, honeypots provide valuable insights into emerging threats and vulnerabilities. While challenges such as false positives and maintenance complexities exist, the benefits of honeypots in bolstering cybersecurity defenses outweigh these concerns. As organizations strive to protect their digital assets and infrastructure, integrating honeypots into their security strategies is indispensable in mitigating cyber risks and maintaining a robust security posture.

1.3 Scope of this Project

In this project, we focus on deploying a honeypot on a productively used server, for example a mail server. The idea is that a separate honeypot system is expensive in purchase, running costs and maintenance. Therefore using the already existing and managed server has cost benefits for the enterprise.

However this does not come without challenges and compromises. For example we have to think about some general techniques and how we want to implement them, that a regular, dedicated honeypot as described does not have to deal with.

Legitimate users and administrators have to be able to manage the productive part of the server and therefore have to be able to use regular Linux commands. Therefore it is necessary to decide when to trigger the honeypot behavior and also when and how do deactivate it.

Another consideration is the fact that we cannot just present the whole file system to an adversary as this contains real data that is valuable to the organization. A file system has to be simulated, using techniques such as docker or a journal.

Last but not least we could think about some criterion to lock an adversary out. The primary goal of this honeypot project is to get insight into an attackers motivation and behavior. This can be gathered for example using the first 20 or 30 commands he tries to execute on the server. During his operations, the simulated system will be more likely to be recognized and an attacker might try to break out of this dedicated, secured environment. Therefore a hard-coded limit of commands that can be executed might improve the overall security of this approach.

1.4 Code

All code can be found on GitHub: <https://github.com/00Jam00/JamJar-Projekt>

2 Considerations for the design of the system

When designing a honeypot system various approaches can be used to achieve a similar goal. Different approaches have a range of advantages and disadvantages. In the following chapter, multiple concepts for different design questions will be shown and discussed. A target architecture will be determined.

2.1 Justification for this "budget" honeypot approach

As we already discussed in the introduction, deploying a honeypot on a server used in production comes with some challenges and limitations. On the one hand, we do not want to interfere with commands from a legitimate administrator and make things unnecessarily difficult to use. On the other hand, an attacker on the machine should not have access to real data and therefore, the real file system has to be hidden in a way that it is not too obvious.

A possible use case for this budget-approach might be in embedded systems. For example an electricity supplier might operate an electrical grid spread over a large area. This grid is constantly monitored at hundreds and thousands of locations to make sure that everything works as expected and within given limits. These sensors produce measurement data and send these data to a some central hub, which means they are somehow connected to the internet, most likely over some virtual private network. On the other hand, these sensor platforms are not under manual administration very often. They are deployed in bulk and should operate for many years without needing maintenance. Therefore these embedded systems might be a potential use case for the budget honeypot approach. Legitimate administrators do not constantly use them and might lock themselves out and also these systems are connected to some central instance, enabling automatic sharing of intelligence about an attacker.

Of course this comes with some limitations. Most embedded systems do not contain a lot of computing power, as they are designed to only gather and transmit some sensor data. This would have an impact on the technologies that can be used in the honeypot system. For example a virtual environment might not be possible. Also in the vent of an attacker escaping the honeypot simulated system, the company would have to disconnect the compromised de-

vice entirely from their systems and potentially drive several hours to the location to replace the infected system.

To put it all in a nutshell, this approach might not have an ideal use case at the moment. But this project is not about inventing the latest state-of-the-art honeypot technology but rather about learning techniques to intercept commands and fake a file structure on a Linux-based machine.

2.2 How is the "fake" system simulated?

When creating a honeypot system that is supposed to create the impression or illusion of a real system, it is essential to simulate the effects of various commands on the real operating system. This means if an attacker executes the command `rm folder1` he expects this directory to not show up the next time he uses the `ls` command. Since all commands are intercepted by the honeypotting system, the effects of commands used by the attacker must be processed and kept track of. To achieve this goal, various approaches are conceivable.

2.2.1 Using a different file system root

The first idea was to utilize already existing features of Linux. One of these features is the `chroot` command. It allows the user to define a custom system root for the current process. Changing the root not only switches the file system to a new directory but also runs following commands with binaries present in the new system root. Since the system root is changed, all directories containing binaries like `/usr/bin/bash` are changed as well. This would open the possibility to directly redirect commands run by an adversary to a separated system, while preserving all functionality of these commands. The `rm` command for example would delete a file in the replicated filesystem without altering the real data. This kind of approach was also used in the past to realize a similar honeypotting approach. Bill Cheswick used the `chroot` method as early as 1991 to track the behavior of an adversary [cheswick1992evening].

Even tho this method provides a lot of nice properties for the realization of the project it also comes with a few serious disadvantages. Firstly, in contrast to many statements found online, `chroot` is not a jail or sandbox in the regular sense. Even though there are possibilities for a defender to hide the fact that an attacker is currently inside a `chroot` environment, an experienced adversary can still figure out the situation. Furthermore once an attacker is aware of the `chroot` environment, it is easy for him to escape. Another negative point is the fact, that simulating directory or filesystem related commands is very simple to achieve

via this method, other commands that are network or process related cannot be implemented easily.

2.2.2 Docker

The second approach utilizes a similar method like `chroot`. This time docker containers are used to redirect an adversary to a virtualized system. Compared to the previous idea, this provides a better level of abstraction and separation to the actual production system. Escaping a well set up docker container is hard to impossible and even network and process related commands can be run inside the container. This is due to the fact that a docker container simulates an operating system as a whole, compared to only the directory structure from `chroot`. Unfortunately a docker container is also easy to spot for a well trained attacker. Furthermore this approach not only requires additional software, it also needs further computing resources like CPU and RAM. In accordance to the **budget idea** (see section 2.1) this contradicts the original purpose of using limited resources to realize a honeypot.

2.2.3 Journal / Internal

The final approach does not use some kind of "virtualization" technology, but rather tracks and simulates all system behaviors at runtime. A base structure is defined and loaded at the start of the honeypot program. All changes and commands as well as applications are tracked and the results mapped onto the base data. Every action is written into a data structure. This could be in the form of a journal. A journal consists of ordered list of commands an adversary used on the system. If a "fake" output should be provided to the attacker, the program "re-runs" all commands currently in the journal. This approach does not consume a lot of resources, which is a big advantage given our scenario. It also is not persistent and can be reset quickly, as the simulated environment is generated and then kept in memory during program execution. Simply restarting the process resets the environment. Therefore this approach seems to be the most promising and will be used.

2.3 Activation / Deactivation of the System

2.3.1 When does it activate?

Another question to ask is, whether the honeypot system is activated by a particular trigger and if so, what kind of trigger could be used? In contrast the second approach would be to make the honeypot active by default. This has the advantage, that no attacker will be

missed. However, this constant activation may inconvenience legitimate users interacting with a system, as they would have to use some kind of deactivation, which is discussed in the next section.

As for the method to activate the honeypot through a specific trigger. This could be an access to a specific file or location, such as monitoring access to `"/etc/passwd/"`. This type of trigger could be chosen depending on the most commonly accessed locations during an attack.

For this first proof-of-concept version of the honeypot system, we might also think of activating the honeypot system on a user bases. For example, we might secure all legitimate accounts on the system with strong passwords and state-of-the-art Two Factor Authentication except for one specific user account that we want an attacker to find and use. Therefore, our system could be limited to only intercept commands from this one specific user account, although other triggers based on some rules might be used later.

2.3.2 How does it deactivate?

When activating the honeypot system, it's important to ensure that legitimate users can still access the genuine functionalities of the command line. One method is to globally deactivate the honeypot with password protection, enhancing security even if attackers discover how to deactivate it. A similar approach would be allowing only certain commands and disallowing them after they are not needed anymore. Both these approaches expose the system to attacks temporarily.

Another strategy involves using different commands that are created to replace the forged command. These commands provide the legitimate functionality of the forged command but have a different name. However, this requires developing and learning a new command scheme, which entails a certain level of inconvenience and could be confusing for new and inexperienced users. To mitigate this, a prefix or suffix could be added to the original command (e.g allow `ls` or `ls2`). Although using different command names provides a clear distinction between forged and real commands, it can be easily detected and bypassed by an attacker, if the attacker is aware of the new names.

Additionally, it's essential to address scripts that rely on system commands. If the original commands are replaced and don't function anymore, scripts using these commands will also fail. One solution is to modify the commands within the scripts to match the new ones. Alternatively, if the approach is chosen to temporally deactivate the honeypot, the scripts will resume functioning, which could be a feasible solution depending on the use case. Another option would be adding a user who is excluded from the honeypot's effects, allowing them to execute the original commands needed to run the scripts.

As discussed in the previous section, these problems can be avoided when using the honeypot technology on a user basis. Legitimate users do not interfere with the honeypot system whereas an attacker that uses the least secured, specifically and obviously placed account is trapped.

2.4 Different Command Modification and Tracing Approaches

The following section describes techniques that can be considered to modify linux system cli commands so that they do not serve their original purpose but emulate the original system behavior. This way a potential intruder will not be misguided, while his actions can be monitored without him actually doing harm or changing the system.

2.4.1 `execve()`

The "execve" or "execute program" syscall within UNIX operating systems executes a new program within a current process. There is a whole set of commands starting with `exec*`. They provide almost the same functionality but differ in the way they expect and handle arguments [`execve`]:

Command	Description
<code>exec*</code>	Replaces the current process image with a new process image.
<code>execl*</code>	Expects a list of arguments in the form of pointers to null-terminated strings.
<code>execv*</code>	Expects a vector of arguments in the form of pointers to null-terminated strings.
<code>exec*p</code>	Just like the shell, the program will look for the executable file in the list of directories specified in the PATH environment variable.
<code>exec*e</code>	Allows to specify the environment for the new process. This is done using the <code>envp</code> -argument which is an array of pointers to null-terminated strings.

Table 2.1: Difference of `exec*`-commands

Once a user executes a command within the UNIX command line e.g. "bash", the process of the bash is then forked via the "fork" system call as a child process of the main "bash" process [`fork`]. Afterwards, the "execve" system call then replaces the process image of said child process with that of the actual command program which is then executed to return the program output to the user [`execve`]. It is important to note that not all UNIX commands can be traced with "execve" for example if they directly interact with the kernel or are builtin shell

commands. Commands such as "kill" directly interfere with processes by sending signals to said processes without using the "execve" system call [kill].

2.4.2 eBPF

Extended Berkeley Packet Filter (eBPF) is a technology which enables programming of extensions for the linux kernel functionality without changing the source code of the kernel or loading kernel modules. eBPF can be utilized to run sandboxed programs within the operating system to add additional capabilities to the operating system at runtime. In the particular case of the honeypot eBPF can be used to run custom code once specific pre-defined kernel hooks such as system calls, function entries/exits, kernel tracepoints etc. are passed. In case that there are no pre-defined hooks present, it is also possible to create so called kernel probes (kprobe) or user probes (uprobe) to attach the program nearly anywhere in the kernel. This way the eBPF capabilities can be utilized for probing specific system calls such as "execve" to trace UNIX commands and extract details such as process IDs and command parameters [bcc].

The implementation is done via the python framework bcc which enables the coding of eBPF C-programs embedded into python source. The following illustration depicts this architecture in detail and shows the eBPF interacting with the System Call Interface (SCI).

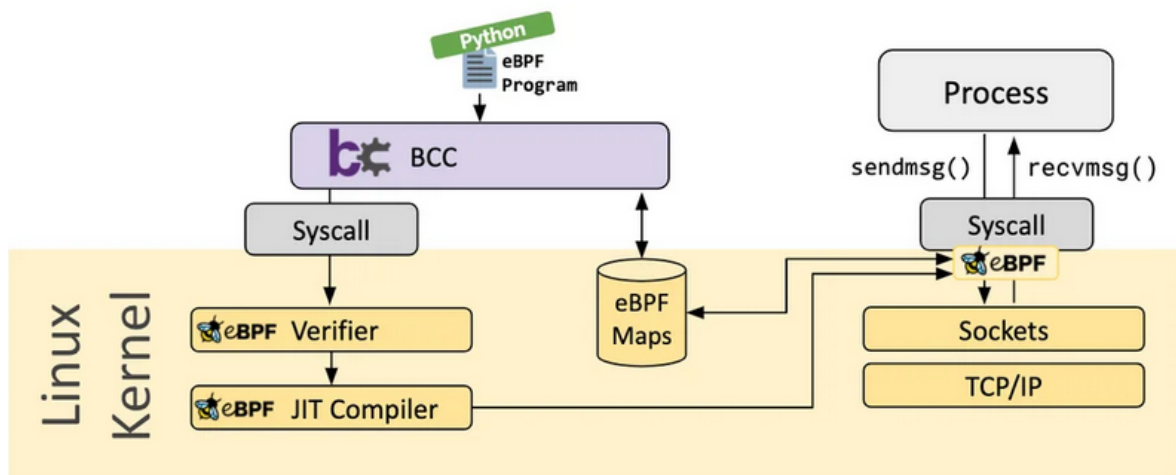


Figure 2.1: eBPF with python bcc [bcc]

2.4.3 ptrace()

"ptrace" is a system call that enables us to control the execution of another process by attaching and adding breakpoints for debugging purposes [ptrace]. Therefore, "ptrace" can

be utilized to debug processes and halt the execution flow. In combination with eBPF this would allow for modification and interception of processes that were traced.

2.4.4 Changing the original binary or using different binaries

As already discussed we do not want to change binaries or introduce additional binaries to the system that are named slightly different. This does hinder normal system users from administrating the machine and might interfere with execution of scripts. As the goal is to implement this honeypot approach on an already in-use system, an administrator would have to be aware of the honeypot running during all of his actions on the system, which is hard when administrating many servers in a network during the day. Also some scripts or third party programs for network monitoring or malware detection might not work as expected when we mess with the standard system binaries. Therefore this option is not further considered.

2.5 Which commands are changed / intercepted?

Another critical aspect of designing this honeypot system is selecting the commands to be modified. The objective is to alter the behavior of commands that are commonly utilized by potential attackers. This can be approached in various ways. One approach would be examining the most popular general commands used in the Linux command line interface by regular users. Another approach is considering the most popular commands used in an attack scenario.

Numerous websites offer lists of commonly used Linux commands to assist beginner users in familiarizing themselves with the operating system. These serve as a starting point for our selection process. For instance, a popular educational platform like Geeks for Geeks recommends these 25 fundamental Linux commands:

ls, pwd, mkdir, cd, rmdir, cp, mv, rm, uname, locate, touch, ln, cat, clear, ps, grep, echo, wget, whoami, sort, cal, wheris, df, wc [noauthor_25_2023].

In a 2007 experiment conducted by Ramsbrock et al., researchers examined the specific actions taken by an attacker and the order in which they occurred. To gain more information about the systems software attackers used the commands:

w, id, whoami, last, ps, cat /etc/*, history, cat .bash_history, php -v.
To install a programm attackers used the following commands:
tar, unzip, mv, rm, cp, chmod, mkdir.

When downloading a file wget, ftp, curl, lwp-download were used the most. In order

to run files the `./` was used. Some attackers modified the path variable to run programs without the `./` notation. Some also used perl scripts, so they also included `perl` and `*.pl`. When changing the password of a compromised account the command `passwd` was used. Commands for checking the hardware configuration include `uptime`, `ifconfig`, `uname`, `cat /proc/cpuinfo`. Lastly, `export`, `PATH=`, `kill`, `nano`, `pico`, `vi`, `vim`, `sshd`, `useradd`, `userdel` were used to change the system configuration. Certain routine commands made up 34.08 percent of all used commands. These commands include: `cd`, `ls`, `bash`, `exit`, `logout`, `cat` [4272962].

Knöchel and Wefel 2022 compiled the following list of most utilized commands from their findings observing the behaviour of attackers on a high-interaction Linux honeypot:

`ls`, `df`, `ifconfig`, `w`, `who`, `netstat`, `ps`, `top`, `uname`, `lscpu`, `curl`, `wget`, `ftp`, `git clone`, `scp`, `perl`, `python`, `bash`, `sh`, `chpasswd`, `passwd`, `useradd`, `rm`, `cat /dev/null >`, `crontab -r`, `history -c`, `kill`, `pkill`, `killall` [9943718].

In 2013, Kheirkhah et al. conducted an experimental study showing a list of the most used commands entered by attackers during their honeypot experiments:

`w`, `uname`, `wget`, `id`, `ls`, `cat/proc/cpuinfo`, `uptime`, `ps`, `ls-a`, `passwd`, `whoami`, `halt`, `help`, `history`, `netstat`, `php-v`, `ifconfig` [Kheirkhah13].

The final list consists of a combination of general commands and commands used in attacks that might be useful to intercept and forge in a honeypot system. Commands that have no impact on the system, such as "clear" have been excluded. The final prototype will have a exemplary implementation of a small number of these commands.

To better understand the influences the commands have on each other, they can be categorized into different types. When analyzing the order of commands used in attacks, various approaches can be chosen. For instance, the Cyber Kill Chain provides a structured framework for dividing attacks into distinct phases. It is a attack modeling technique, which defines an attack as a chain of actions. It has seven steps described by Al-Mohannadi et al. [7592703]:

1. Reconnaissance: Gathering information before the attack.
2. Weaponization: Creating a malicious payload to send to the victim.
3. Delivery: Sending the malicious payload to the victim.
4. Exploitation: Victim needs to download the payload.
5. Installation: Executing malware on the infected system.
6. Command and control: creating a command and control channel to access the internal assets of the victim
7. Action on objective: Attacker achieves his goal on the victims system

When analyzing the cyber kill chain, the actual execution of commands only starts in stage six "command and control" and continues in the seventh stage "Action on objective". This makes it harder to map our commands into different stages of the attack. Alternatively, another approach proposed by Ramsbrock et al. could be used. The found commands are divided into seven states that represent the typical observed actions [4272962]:

1. Check Software Configuration: The attacker gains more information about the systems software and it's users.
2. Install a program: The attacker installs software on the system
3. Download: The attacker downloads files remotely.
4. Run a program: The attacker runs a program that was not originally part of the system
5. Change the account password: Changing the password for the compromised account
6. Check the hardware configuration: The attacker gains more information about the systems hardware(uptime, network, CPU speed)
7. Change the system configuration: The attacker changes the state of the system permanently.

This approach is simplified and used in our command overview. States 1 and 6 are combined into one state: "Finding out information about the system". State 3,4, and 5 are combined into: "Download, install, and run programs". The last state is called "changing the state of the system" and consists of states 7 and 5.

2 Considerations for the design of the system

Basic commands	cd ls rmdir mkdir	Change directory List directory contents Remove empty directory Create directory
1. Finding out information about the system	cat df history php-v uname locate ps grep whoami df w id last uptime ifconfig crontab nmap ping arp traceroute	Concatenate and print files Check details of file system Show command history Show php version Get basic information about the OS Find a file in the database Display processes Search for a specific string in an output Displays current username Check details of a file system Information about current users Find out user and group names and ids Display the history of last logged in users Find out how long the system is active Display current network configuration information Submit, edit, list, or remove cron jobs Network scanning Send packets of data Display and modify arp entries Trace IP packet path
2. Download, install and Run programs	cp mv rm wget touch ln ftp curl unzip chmod Nano, pico, vi, vim	Copy files Rename and replace file Delete files Download files Create empty file Create shortcuts Ftp connection Download and upload data to a server Unzip file Set or modify a file's permissions File Editor
3. Changing the of the system	kill, pkill, killall useradd userdel halt passwd	Terminate process Add user account Delete user account Stop all CPU functions Set and change passwords for users

Table 2.2: Final List of commands

2.5.1 Relationship between commands

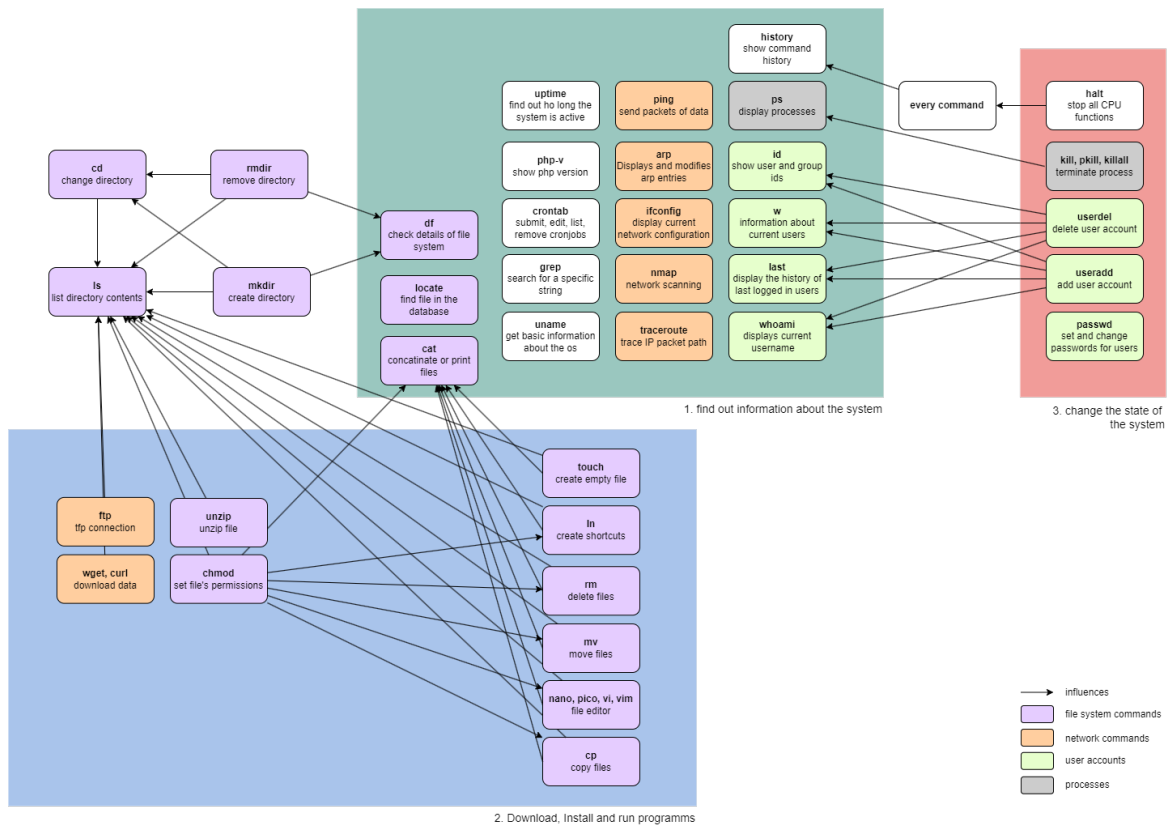


Figure 2.2: Relationship between Linux commands

The graph illustrates the relationships between selected commands and their influences. For instance, the `cd` command is influenced by `rmdir`. Since `rmdir` deletes directories, it restricts `cd` from navigating into those directories. Additionally, there are commands, such as `history`, influenced by all commands, and commands like `ifconfig` which are not influenced by any other command. It is important to note that this overview only contains the base commands without considering different flags or endings like `-a` for simplification purposes.

The figure suggests splitting commands into three groups:

1. Commands to find out information about the system
2. Commands to download, install and run programmes
3. Commands to change the state of the system

Furthermore commands in these groups can be split into classes:

- File system commands
- Network commands
- Commands that influence user accounts
- Commands that influence processes and the overall system state

When intercepting these four classes of commands, we notice that we have to define an environment and accordingly a data structure to simulate parameters that can be used by the classes of commands. For example, we have to define a virtual file system stored in a data structure that can be viewed and manipulated using file system commands such as `cd`, `mkdir` and `touch`. We also need an independent data structure providing a virtual network that can be read and used using network commands such as `ping`, `arp` and `nmap`.

These are the two main environments we have to simulate. On top, interactions like altering and creating user accounts and processes pose another challenge. Commands that interact with user accounts require dedicated structures to simulate user-related operations, such as creating, modifying, or managing accounts dynamically. Similarly, commands influencing processes or retrieving system state, such as information about active users or sessions, necessitate an additional abstraction layer. This layer complements the virtual file system and network structures to fully simulate the environment.

2.6 Ways to hide honeypot processes and files

In order to deceive the attacker inside the honeypot certain processes and files must be hidden. Linux provides several methods for hiding processes and files, some of which are outlined below.

Regular Linux user accounts typically have the ability to view a Process ID (PID) listing using commands such as `ps`, `pgrep`, and `pidof`. However, to hide processes from specific users, the `hidepid` feature can be used. By mounting the `/proc` file system with the `hidepid=2` option, process files become invisible to non-root users. It's worth noting that although the existence of a process can still be learned through other means, this approach limits visibility to unauthorized users [emad-al-mousa_how_2022]. Additionally, there are also tools available for hiding processes, commonly used to hide malicious programs. For instance, `Xhide` can be used to mask processes by giving them the appearance of running as another process. Another tool called `Libprocesshider` can hide processes from commands like `ps`, `top`, and `ls` by editing the source code and including the program that needs to be hidden [noauthor_difference_2020]. Modifying the source code of commands like `ps` could also be a way to conceal processes [noauthor_difference_2020, borello_hiding_2014]. Another potential approach is to alter the `readdir()` function within the `libc` library and incorporate code to filter out access to certain `/proc` files [borello_hiding_2014]. The last approach could be intercepting and modifying the system calls directly in the kernel with a custom module [borello_hiding_2014].

Similarly, there are also multiple options to hide files in the Linux file system. Naturally, it is possible to create hidden files by adding a period (.) to the beginning of the file name or uti-

lizing the "hidden" attribute. However these files still remain visible to everyone, as a simple `ls -a` will reveal them. Another straightforward method involves altering directory permissions to restrict the access to files. There are also stenographic approaches that hide files by compressing them and concatenating them with other files [sk_steganography_2019].

In our honeypot approach, the system mimics the output of commands such as `ps` and replicates the entire file system. This ensures that attackers are unable to detect any files or processes they are not supposed to see.

2.7 Logging

In order to later learn from the attackers command patterns in a attack scenario each executed command should be logged. This could be done in different ways.

One possible way is using a journal. `Journalctl` is a tool used to query and display logs from `journald`, the logging service of `systemd` [noauthor_journalctl_2023]. One of the primary benefits of using `journald` is its ability to provide a detailed and clear timeline of events. This feature is valuable for forensic analysis, allowing administrators to trace the actions of an attacker step by step. `Journald` stores logs in a binary format, which is proprietary. This can pose challenges in terms of compatibility and flexibility, as specialized tools (like `journalctl`) are required to read these logs. There are also questions regarding the scalability. As the volume of log data increases, performance issues may arise. Especially in our approach, performance and resources of the host are limited.

Another common method is simple logging in plain text files. This method is straightforward but comes with its own set of pros and cons. It is easy to implement and does not require additional software or complex configurations. Similar to `journald`, using custom formats or structures in file-based logging can lead to compatibility issues. As with `journald`, plain text file logging may face scalability issues as the volume of logs grows. Additionally, since the logs are stored on the same system, they can be found and destroyed by an attacker. This makes it a less secure option compared to methods that store logs off-system.

Network Service or Security Information and Event Management (SIEM) solutions offer a more advanced and secure approach to logging. By storing logs off-system, these solutions make the logs immutable. An attacker cannot easily tamper with or destroy the logs. SIEM solutions also often come with powerful analytics and correlation capabilities, enabling proactive threat detection and response. Con of that approach are that implementing a network service or SIEM solution requires additional systems and resources. This includes dedicated servers, network bandwidth, and potentially significant financial investment. These solutions can also be complex to configure and maintain, requiring specialized knowledge and skills.

In our honeypot system, simple logging in plain text files seems to be appropriate. As we hide the real system from an attacker, he does not have direct access to this log in order to destroy it. Also the small impact on performance is a strong argument in favor of this approach.

3 Implementation

After discussing the most important design aspects of the honeypot system, we now implemented a prototype that can simulate a basic environment and might help gather information about an attacker's intentions.

3.1 Overview

The following diagram illustrates the final pipeline that handles commands.

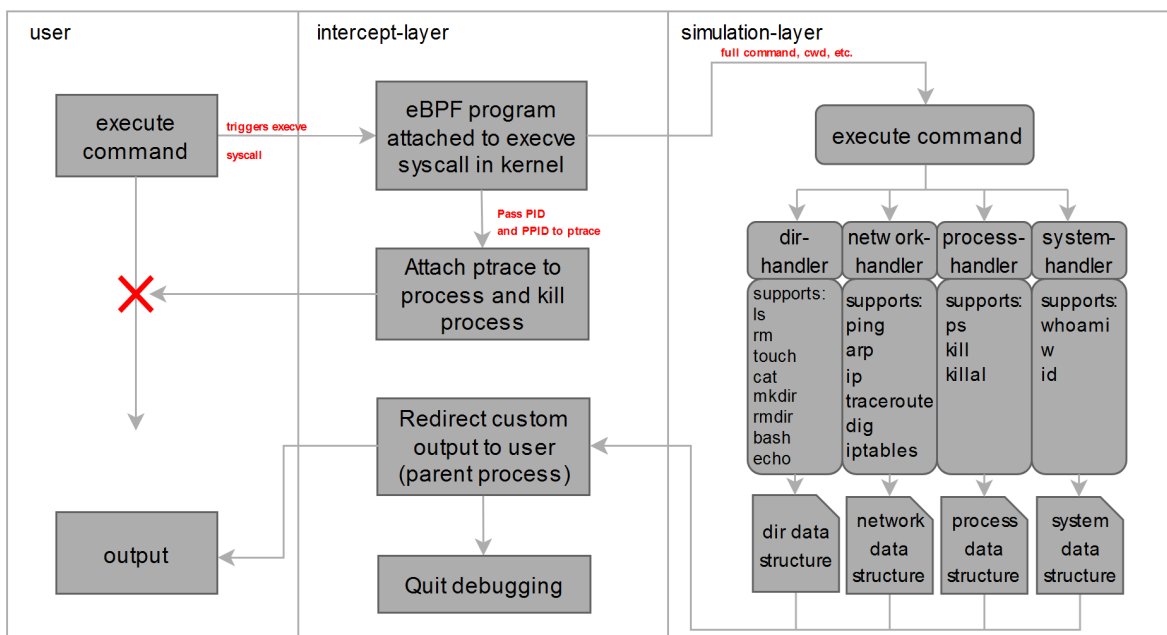


Figure 3.1: Command pipeline

As shown in the graph, we use an intercept-layer to block the immediate execution of a command that is provided by a user. This layer is capable of calling APIs and deciding, which type of command it just intercepted. It can distinguish between directory-commands, network-commands, process-commands and system-commands. Examples for the different classes will be shown later. After intercepting a command, it will call the corresponding

API and deliver the command itself, all provided arguments and the current directory as a payload.

In the next step, the four handlers act as API endpoints and can react to a given input. For example a user might execute the ls-command. The "dir-handler" generates a simulated file system on program start using the data structure and can therefore apply a simulated ls-command on the virtual file system. The output is returned to the calling intercept-layer which will present it to the user. In theory the interception of commands and the simulated output can not be detected by the user.

3.2 Command intercept

As described in the previous section, the intercept-layer has two basic tasks. On the one hand, it has to trace and intercept all commands a user wants to execute. These commands have to be classified according to the four categories we defined (directory, network, process, system) and the corresponding API has to be called. On the other hand, the output of the called API has to be presented to the user in a way he can not distinguish it from a real output of the system.

3.2.1 Tracing commands via eBPF

To trace commands that are issued by the user we implemented and modified a eBPF program based on the exemplary "execsnoop.py" program that comes with the python bcc library [execsnoop]. This program enables us to trace execution details of the "execve" system call that is triggered once the user commands are executed. Within the c-based eBPF program we use a structure to collect data such as the process ID, the parent process ID, the user id, the actual command as well as the arguments and other details like the return value. Furthermore, the user identifier 1000 of the user that is traced by the honeypot is hardcoded into the eBPF program.

```
1 # Define BPF program
2 bpf_text = """
3 #include <uapi/linux/ptrace.h>
4 #include <linux/sched.h>
5 #include <linux/fs.h>
6
7 #define ARGSIZE 128
8 #define UID_FILTER 1000
9
10 enum event_type {
11     EVENT_ARG,
12     EVENT_RET,
13 };
14
```



```
15 struct data_t {  
16     u32 pid;  
17     u32 ppid;  
18     u32 uid;  
19     char comm[TASK_COMM_LEN];  
20     enum event_type type;  
21     char argv[ARGSIZE];  
22     int retval;  
23 };
```

To store events that can be read from userspace we make use of a BPF map.

```
1 BPF_PERF_OUTPUT(events);
```

Within the eBPF program we also utilize a function "syscall__execve" to trace the "execve" system call and capture the already mentioned parameters and store them in the BPF map. Helper functions such as "__submit_arg" and "submit_arg" are responsible for reading and submitting arguments towards the map. Using the "BPF" class we then initialize the implemented eBPF program.

```
1 b = BPF(text=bpf_text)
```

Afterwards, the function name of the "execve" system call needs to be resolved since these names can vary depending on the underlying kernel version. This ensures that we are tracing the correct system call.

```
1 execve_fname = b.get_syscall_fname("execve")
```

We then attach the eBPF program to the system call by utilizing bcc's "attach_kprobe" which will execute the prior defined function "syscall__execve" once the "execve" system call is triggered.

```
1 b.attach_kprobe(event=execve_fname, fn_name="syscall__execve")
```

Lastly, we also attach a "kretprobe" which will capture returns of the system call and then execute the defined "do_ret_sys_execve" eBPF function which simply captures the return value of the system call.

```
1 b.attach_kretprobe(event=execve_fname, fn_name="do_ret_sys_execve")
```

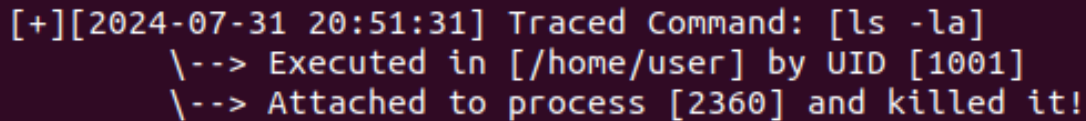
All of the prior mentioned code samples are implemented within the main program "jam-jar.py". Whenever the system call is triggered we execute the same routine which is implemented within the "proc_event" function. This function processes the data that is returned by the eBPF program and therefore parses the process ID, which is required to intercept and kill the processes as well as the full commands which are needed for generating the fake output that is later returned to the user. Note that the "ptrace" debugger is attached to the target process as soon as the process ID has been parsed. Since the simulation layer also requires other data such as the current working directory for the command issued by the user, we derive such details from the file system entries for the traced processes within "/proc/<pid>/cwd".

```

1 def proc_event():
2     event = b["events"].event(data)
3
4     if event.type == EventType.EVENT_ARG:
5         argv[event.pid].append(event.argv)
6     elif event.type == EventType.EVENT_RET:
7         target_process = attach_ptrace(event.pid)
8         argv_text = b' '.join(argv[event.pid]).replace(b'\n', b'\\n')
9         cwd = os.readlink(f"/proc/{event.pid}/cwd")
10        tty = os.readlink(f"/proc/{event.pid}/fd/0").replace("/dev/", "")
11        username = pwd.getpwuid(event.uid).pw_name
12        # Getting cleaned command
13        full_command, cmd_wo_args = cleaup_cmd(event.comm, argv_text)
14
15        # Print event to console
16        print_event(full_command, cwd, event.uid)
17        # Handle commands
18        event_handler(cmd_wo_args, event.pid, full_command, cwd, tty, event.ppid, username,
19                      target_process)

```

Once all the required details are parsed they are passed to a function "print_event" which simply prints information regarding the traced command to the screen for logging purposes.



```

[+][2024-07-31 20:51:31] Traced Command: [ls -la]
  --> Executed in [/home/user] by UID [1001]
  --> Attached to process [2360] and killed it!

```

Figure 3.2: JamJar console output

Furthermore, the parsed data is also passed to the function "event_handler" which is responsible to call different subroutines which are implemented within "Subroutines.py" depending on the type of command that was issued by the user differentiating between directory, network and process commands.

```

1 match comm:
2     case ("ls"|"rm"|"touch"|"cat"|"echo"):
3         PtraceSubroutines.dir_routine(pid,ppid,full_cmd,cwd,target_process)
4         DEBUGGER.quit()
5     case ("ping"|"arp"|"ip"|"traceroute"|"dig"|"iptables"):
6         PtraceSubroutines.network_routine(pid,ppid,full_cmd,target_process)
7         DEBUGGER.quit()
8     case ("ps"|"kill"|"killall"):
9         PtraceSubroutines.process_routine(pid,ppid,full_cmd,tty,username,target_process)
10        DEBUGGER.quit()
11    case ("whoami"|"w"|"id"):
12        PtraceSubroutines.system_routine(pid, ppid, full_cmd, username, target_process)
13        DEBUGGER.quit()

```

To constantly monitor for system call events we make use of a simple while-loop which then calls the "proc_event" function once an event is triggered.

```

1 b["events"].open_perf_buffer(proc_event)
2 while True:
3     try:

```

```

4         b.perf_buffer_poll()
5     except KeyboardInterrupt:
6         exit()

```

3.2.2 Intercepting and Killing Processes

As mentioned in the previous section, as soon as the process id for the traced command is parsed we make use of the "ptrace" python library to attach to said process and put it in a halted state. This is required since processes for the traced commands exit rather quickly. Once "ptrace" is utilized to attach to the target process and all the required details are parsed within "jamjar.py" we move into the subroutines for the different commands within "Ptrace-Subroutines.py". This is where we kill the processes for the commands issued by the user and also pass the details to the simulation layer to generate the fake data and return it to the user. The different routines for the directory, network and process commands are very similar with the exception of commands such as "ping" or "traceroute" as they require time intervals for different sequences of output that is returned to the user. To establish a basic understanding of such routines we now detail the routine for network commands "network_routine". This function is called and passed with the parameters process id (pid), the parent process id (ppid), the full command (command) and the ptrace context (running_process).

```

1 def network_routine(pid,ppid,command,running_process):
2     ...

```

To first retrieve the fake output, we invoke a call to the simulation layer passing the full command.

```

1 cmd_output = CMD.invoke_network(command)

```

Now that the fake data has been generated we make sure to verify if the returned data is of type list, this would be the case if we need to delay the output for commands such as "ping" or "traceroute". If this is the case, we then call the helper function "write_to_proc" for each item in the list with a delay of one second. The helper function "write_to_proc" simply writes the fake output to the parent process of the issued user command which in this case is the tty of the user that issued the command.

```

1 if type(cmd_output) == list:
2     for n,item in enumerate(cmd_output):
3         write_to_proc(item+"\n",str(ppid))
4         if n < 5:
5             time.sleep(1)
6     # Write modified output to target process
7 else:
8     write_to_proc(check_linebreak(cmd_output),str(pid))
9 kill_and_quit(running_process,str(pid))

```

Once the fake output is returned to the user, we then call our second helper function "kill_and_quit" which takes the "ptrace" context for the target process to then kill said process by sending a signal "SIGTERM". This then concludes the process of the intercept layer to trace and intercept user commands.

```
1 def kill_and_quit(process,pid):
2     # Kill the process using SIGTERM
3     process.kill(signal.SIGTERM)
4     print(f"\t\\--> Attached to process [{pid}] and killed it!")
```

3.2.3 Challenges and Takeaways

While implementing the intercept layer we came across multiple challenges. Originally we had in mind to perform this whole process by utilizing only eBPF programs. eBPF generally seems to be mainly used for its tracing capabilities and is therefore not meant to write within the kernel or modify system call return values and behavior to provide security within the kernel [bcc]. Though it seems to be possible in certain edge cases it is not recommended as it would require changing certain kernel configurations which would put the availability of the operating system at risk.

While we opted to implement the actual interception and the termination of the processes via "ptrace" we originally attached to the command processes within "PtraceSubroutines.py" as it looked more reasonable at the time. This led to the issue of race conditions as the processes for the user commands oftentimes already finished executing before we were able to attach to them. Therefore we decided to attach and halt the process as soon as we parsed the process ID via eBPF, this minimized the occurrence of race conditions significantly. Nevertheless, race conditions are still relevant as an attacker could simply use programs written in compiler languages such as C to outrun the execution flow of our python implemented honeypot.

Finally, we managed the termination of the user command processes by sending the "SIGTERM" signal. This would normally return signal messages such as "Terminated" to the user which would indicate that something went wrong while executing the command. To solve this we turned off the monitor mode for the target user by disabling it within the user's bash profile ".bashrc".

```
1 export PROMPT_COMMAND='set +m'
```

3.3 Command handler

Next we have a look at the four command handlers. As already mentioned in the theory-chapter, we can distinguish between four types of commands. **Directory** related commands, **Network** related commands, **Process** related commands and **System** related commands. Each command corresponds to a dedicated handler in the python code:

- Directory-commands -> `dir_handler`
- Network-commands -> `network_handler`
- Process-commands -> `process_handler`
- System-commands -> `system_handler`

Each handler acts as an API-endpoint and needs some context to "execute" commands. All four handlers define and build a simulated environment that can be used to execute the commands. This environment is semi-persistent. This means it is generated in a pre-defined state at program start and is completely discarded once the program stops. So in case it is broken, all that has to be done is restart the program. This could also be done periodically, for example every hour.

3.3.1 Directory

Simulating a directory is the most challenging part of the three handlers. The most crucial aspect is designing a file system that on the one hand provides all the information that might be needed if a user executes a command (e.g `ls -a`). On the other hand, commands like `rm` alter the file system, which needs to be taken into account and tracked. Further more, this simulated file system has to be efficient, as a delay of the output might be suspicious to an adversary.

To simulate the filesystem, we use Python objects. Basis is the class **file**, which is used to represent a single "virtual" file. Each file object has several attributes, such as permissions, owner and name. One key aspect is the attribute "parent", which can be used to reference the parent object (a directory) of a file. The init-method provides some default-values each time a new object of the class "File" is generated.

Directories can be seen as a special kind of files, because they share a lot of the same attributes. Therefore we defined a new class "Dir" in "dir.py" which extends the class "File". It also adds a new attribute to all objects of the class which is called "content". This content-attribute holds a dictionary of all files and directories that are currently located in the directory. In this case the key of the dictionary is the name of the object, the value is the object itself. We used this approach to store the content rather than a simple list, because it allows for a quick and efficient navigation through the simulated filesystem.

The last file that is needed before we focus on the directory-handler is called "helper.py". This file contains helper-methods for all four handlers. For example the method "create-fake-dir-data-helper" is used to create an initial fake directory structure with some data. The dir-handler itself will call this function on initialization and set its root-point to the root of the generated fake file system.

Currently there are eight directory-commands with some common arguments supported.

Command	Description	Supported Arguments
ls	list directory contents	a / l / r
rm	remove files or directory	r / f
touch	create file or change file timestamp	-
cat (see ch. 3.5.4)	list file content	-
echo (see ch. 3.5.6)	displays text or command output	-
mkdir	create directory	-
rmdir	remove directory	-
cd (see ch. 3.5.1)	change directory	-

Table 3.1: Currently supported directory-commands

Each command has a source-directory, where it is executed from (passed from the interception layer), and a target-directory (extracted from the command), where the command should be executed. Here are some examples:

Command	Source directory	Target directory
user@comp: \$ ls	/home/user	/home/user
user@comp: \$ ls /etc	/home/user	/etc
user@comp:/home ls user	/home	/home/user

Table 3.2: Possible context of the command "ls"

Because of the many possible ways to execute a simple command such as `ls`, we have to think of a way to extract different sections of a command in a way that it is represented in a unified structure that can then be used to get the desired output. The `ls` command for example can be split into four main points of interest:

`ls -al /home/user` executed in `/home`

1. The command itself (`ls`)
2. Optional arguments (`-al`)
3. Optional path. Can be an absolute or a relative path (`/home/user`)
4. The directory the command was executed in (`/home`)

Each of these sections, can hold a variety of different inputs, which change the output of the command. Therefore, we need to look at each section and act accordingly.

To simplify this process, we first look at what variables are needed at the end to decide the output of the command.

For `ls` they are:

1. The source directory
2. The target directory

With these two variables, we can always walk through the filesystem and find the directory the user wants to see.

By definition, every intercepted command is handed over with the source-directory as a parameter. Therefore this is fixed and can be used to find out the target-directory. In the easiest case, the command does not contain a target-directory, meaning we can set the `src-dir` as the `target-dir`. In case there is a parameter given as the target-directory, we are only interested in the object that represents the last folder in the given path. For example, if the target-directory is given as `/a/b/c/d`, we only need the object of `d` to access its attributes like the `"content"`, containing a dictionary of all files in that virtual folder. To achieve this, the following small algorithm is used:

```
1 for layer in src_dir_list:
2     src_obj = src_obj.content[layer]
```

This algorithm on the virtual structure allows for an easy and efficient usage of the virtual file system.

If every aspect of the command has been taken into account, it is easy to determine the expected output. For example if we know the source and target object of an `ls` command, we can find the corresponding object in our data-structure, which holds the desired data. Finally the content of this object must be returned:

```
1 for file in target_obj.content:
2     output += file + " "
```

Additionally at this point further instructions and arguments that alter the display of the output would be taken into account. This could for example be reversing the output string in case of the `"-r"` argument being present in the command.

Likewise when using the data-structure and helper functions to dissect a command, implementing further commands like `rm` and `touch` is very straight forward.

For example when using the data-structure and correct preprocessing of the command, a file can be deleted like the following. This, of course, only represents the simplest case without any checks and arguments.

```
1 src_obj.content.pop(target_file)
```



```

drwxr-x--- 1 root root 4096 Jul 31 23:24 .
drwxr-xr-x 1 root root 4096 Jul 11 12:29 ..
-rw----- 1 user user 4096 Jul 13 12:31 .bash_history
-rw-r--r-- 1 user user 4096 Jul 13 12:31 .bash_logout
drwx----- 1 user user 4096 May 20 21:13 .cache
drwx----- 1 user user 4096 May 20 21:13 .bashrc
drwxrwxr-x 1 user user 4096 Jul 1 00:25 .local
-rw-r--r-- 1 user user 4096 Jul 7 14:10 .profile
drwx----- 1 user user 4096 Jun 13 19:14 .ssh
-rw-r--r-- 1 user user 4096 Jun 15 11:12 test_file
-rw-r--r-- 1 user user 4096 Jun 15 11:13 .test_file_hidden

```

Figure 3.3: Simulated output of "ls -al /home/user"

3.3.2 Network

Currently there are six network-commands with some common arguments supported.

Command	Description	Supported Arguments
ip	interface related information	a / r / add / del
ping	ping a host	c
arp	view arp entries	d
traceroute	see routers in the route to a host	-
dig	retrieve DNS records for domains	x / v / A / AAAA
iptables (see ch. 3.5.3)	manage Linux network traffic rules	L / h / A / D / F

Table 3.3: Currently supported network-commands

On startup of the program we have to create a fake arp-table as well as a fake list of interfaces. This information is currently hard-coded but could be semi-randomized in the future. When the program is started, the methods "create-fake-arp-data-helper" and "create-fake-interface-data-helper" are executed which load the initial values for the arp-table and the list of interfaces from the file "interface.py" and "arp.py". After that, the environment is ready to handle network commands.

As already mentioned, we currently support the commands ping, arp, ip, traceroute, dig and iptables with some flags that are used often by a malicious actor.

On execution of the ping-command, our program will first check whether an ip-address is given as a parameter or a domain name. In the easiest case, an ip-address is given. The program will check, whether this ip-address is located in the same subnet as the simulated interfaces using the python-library "ipcalc". If it is in the same subnet, the output of a normal ping-command will be handed back using a random value for the roundtriptime between 10 and 50 ms for each ping. By default, four pings are sent, but the user can adjust this number with the -c flag, specifying the desired number of pings.

```
PING google.com (142.250.185.78) 56(84) bytes of data.  
64 bytes from 142.250.185.78: icmp_seq=1 ttl=56 time=45.4 ms  
64 bytes from 142.250.185.78: icmp_seq=2 ttl=56 time=31.2 ms  
64 bytes from 142.250.185.78: icmp_seq=3 ttl=56 time=17.4 ms  
64 bytes from 142.250.185.78: icmp_seq=4 ttl=56 time=25.5 ms  
--- google.com ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 119.5  
rtt min/avg/max/mdev = 12.578/13.047/13.555/0.273 ms
```

Figure 3.4: Simulated output of "ping google.com"

For each target that is pinged by the attacker, an entry in the arp-table should be established. Therefore we need to distinguish between local ip-addresses in the same subnet and addresses outside our subnet. If the ip-address is in the local subnet, an arp-entry for this ip-address is created. If the ip-address is outside the local network, an entry for the standard gateway is created.

The function also calculates realistic roundtrip time (RTT) statistics, including the minimum, average, maximum, and standard deviation of the RTT values. These computed statistics are displayed in the final output along with the RTT values for each individual ping.

Last but not least, if an attacker does not ping an ip-address but uses a domain name as a parameter, this address is translated into a real ip-address using the library "socket".

Similar to ping, traceroute shows information about the reachability of network devices, just with other faked data.

The arp-command can print the current arp-table or delete entries from this table. On startup, only the local address of the host is included in the arp-table. Whenever a user tries to ping a target as described in the previous section, this address is dynamically added to the arp-table. Of course these entries can also be deleted by the user.

The **ip** command is the most extensive command in the network handler. It supports two **main arguments** which can be seen as two separate commands.

1. **ip a**: Shows information about the network interface.
2. **ip r**: Shows information about set network routes.

Both of these sub-commands support **add** and **del** as arguments. Similar to the directory commands, a structure of objects is used in the back-end to store all necessary information about interfaces such as ip-address, gateway, hw-address, etc.

```

1: lo: <LOOPBACK,UP,LOWER_UP mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1 scope host lo
       valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens18: <BROADCAST,MULTICAST,UP,LOWER_UP mtu 1500 qdisc noqueue state UP group default qlen 1000
   link/ether 42:f6:3a:54:ad:bd brd ff:ff:ff:ff:ff:ff
   altname enp0s18
   inet 192.168.0.12 metric 100 brd 192.168.0.255 scope global dynamic ens18
       valid_lft forever preferred_lft forever
   inet6 fe80::ef52:de12:d4ee:139a/64 scope link
       valid_lft forever preferred_lft forever

```

Figure 3.5: Simulated output of "ip a"

The `dig` command simulates the functionality of the widely used DNS querying tool, providing the ability to retrieve domain name system records and perform both forward and reverse lookups. The command accepts a domain name or IP address as input and processes it to generate a realistic DNS response. It supports a few flags to tailor the query, such as specifying whether to retrieve IPv4 (A) or IPv6 (AAAA) records or conducting reverse lookups for IP addresses.

When a domain name is queried, the program performs a forward lookup, resolving the name to its associated IP addresses. Depending on the flags used, the results include either IPv4 or IPv6 addresses, with simulated metadata such as TTL (Time To Live) and DNS query times. In cases where the domain cannot be resolved, the output indicates an appropriate error status, such as *SERVFAIL*, replicating real-world DNS behavior. The generated response mirrors the typical output of the `dig` tool, including headers, sections for questions, answers, and additional information, along with metadata like the server address and the timestamp.

For reverse lookups, the command processes the provided IP address into the reverse DNS format (e.g., *1.0.0.127.in-addr.arpa*) and attempts to resolve it back to a domain name. If the IP address can be resolved, the output includes a PTR record with the corresponding domain name. If the resolution fails, the result indicates a failure status, such as *NXDOMAIN*, and includes relevant authority section data to simulate real-world behavior.

```

; <<>> DiG 9.18.28-0ubuntu0.22.04.1-Ubuntu <<>> -x 142.250.184.195
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 50959
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;195.184.250.142.in-addr.arpa.                IN      PTR

;; ANSWER SECTION:
195.184.250.142.in-addr.arpa. 3600 IN    PTR    fra24sll-in-f3.1e100.net.

;; Query time: 8 msec
;; SERVER: 127.0.0.53#53 (127.0.0.53) (UDP)
;; WHEN: Thu Jan 09 16:05:17 EST 2025
;; MSG SIZE rcvd: 471

```

Figure 3.6: Simulated output of "dig -x 142.250.184.195"

3.3.3 Process

Simulating process commands does also need a default list of simulated processes that run on the system. This initial list is copied from a running linux system in order to be as authentic as possible. Currently the process list with all its parameters is hard-coded but could be semi-randomized in the future. For example process-ids or cpu- and memory-usage could be given a random value on each program start.

We identified the following commands to be important in an attack scenario:

Command	Description	Supported Arguments
ps	Display processes	-a, -e, -A, -f, -C, -u, a, u, x
kill, killall	Terminate process	-

Table 3.4: Currently supported process-commands

The most difficult one to implement is ps. It reports a snapshot of the current processes. The version of ps that is mimicked has three different syntax options: UNIX-options that can be grouped and use a dash in front of the letter; BSD options which also can be grouped but do not use a dash and GNU options that use two dashes[noauthor_ps1_2024]. Our implementation includes some Unix options and some BSD options.

Every ps call from the intercept layer includes the current terminal and the current user ID. With this information an accurate representation of the processes can be generated and returned. Without a given parameter, ps hands back a list of all processes currently running by the user who executes the command and the current terminal the command is executed in[noauthor_ps1_2024]. In our implementation all process objects are filtered by the user id (UID) and the terminal (TTY) that is send by the intercept layer:

```

1  if not args_str:
2      for process in self.output: # none
3          if (process.tty == tty or process.tty == "pts/0")
4              and process.uid == uid:
5                  processes.append(process)

```

The process objects that fit that criteria are selected and their information is turned into an output that looks similar to the original output using .format():

```

1  for process in processes:
2      process_list.append([process.pid, process.tty, process.stat,
3                          process.time[-4:], process.ucmd])
4
5      process_list = [["PID", "TTY", "STAT", "TIME", "COMMAND"]] +
6                      sorted(process_list, key=lambda x: x[0])
7
8      for row in process_list:
9          output+= "{:>6} {:<8} {:<4} {:<4} {:<4}\n".format(*row)

```

The process information that is presented by ps without a parameter includes the process ID (PID), the terminal associated with the process (TTY), the accumulated CPU time in [DD-

]hh:mm:ss format (TIME) and the executable name (CMD)[noauthor_ps1_2024]. As seen in Figure 3.7 the layout of the simulated output resembles the real output, but it is possible to manipulate the processes that are shown. The following options follow a similar algorithm simply changing the filtered attributes and the format of the output for each parameter.

PID	TTY	TIME	CMD	PID	TTY	TIME	CMD
1673	tty2	00:00:00	gdm-x-session	14	tty1	00:00:00	bash
1675	tty2	00:00:00	Xorg	47	tty1	00:00:00	ps
1682	tty2	00:00:00	gnome-keyring-d				

Figure 3.7: Simulated output of "ps" (left) and real output (right)

The Unix options that are included in our implementation are "-a", "-e/-A", "-f", "-C" and "-u" and most combinations of multiple parameters. `ps -a` displays all processes but excludes all session leaders. Session leaders have the same process Id and Session ID. `ps -e` or `ps -A` display all processes of the system without any filters. `ps -C` is able to filter all processes by the process name[noauthor_ps1_2024].

PID	TTY	TIME	CMD	PID	TTY	TIME	CMD
2587	pts/0	00:00:00	bash	14	tty1	00:00:00	bash

Figure 3.8: Simulated output of "ps -C bash" (left) and real output (right)

Another filter option is `ps -u`. This parameter filters all processes by a user Id and only displays processes belonging to that user[noauthor_ps1_2024].

The last Unix parameter that is implemented is "f". It is able to display additional information about the processes by adding additional columns. The information includes the user ID, process ID, parent process ID, CPU utilization in percent, terminal associated with the process, elapsed CPU utilization time for the process and the name of executable command[noauthor_ps1_2024].

UID	PID	PPID	C	STIME	TTY	TIME	CMD
user1	1673	2503	0	12:31	tty2	00:00:00	gdm-x-session
user1	1675	0	0	12:32	tty2	00:00:00	Xorg
user1	1682	0	0	12:32	tty2	00:00:00	gnome-keyring-d

Figure 3.9: Simulated output of "ps -f"

This option only changes the format of the output and not the displayed selection of processes. It can also be combined with other parameters. For example it is possible combine

"-f" and "-u" to search for a specific user ID and display the selected with more information as seen in Figure ??.

UID	PID	PPID	C	STIME	TTY	TIME	CMD
user1	1673	2503	0	12:31	tty2	00:00:00	gdm-x-session
user1	1675	0	0	12:32	tty2	00:00:00	Xorg
user1	1682	0	0	12:32	tty2	00:00:00	gnome-keyring-d
user1	2587	0	0	12:32	pts/0	00:00:00	bash

Figure 3.10: Simulated output of "ps -fu user1"

The BSD-options that are implemented include "a", "x", "u". `ps a` displays all processes in the current terminal but has a different layout additionally displaying the process state codes (STAT) and the long name of the executable command [noauthor_ps1_2024] as seen in Figure 3.11.

PID	TTY	STAT	TIME	COMMAND
1673	tty2	S	0:00	/usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu
1675	tty2	S	0:00	/usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -back
1682	tty2	S	0:00	/usr/libexec/gnome-session-binary --systemd -systemd -session=ubuntu
2587	pts/0	S	0:00	bash

Figure 3.11: Simulated output of "ps a"

The option `ps x` shows all processes of the current user, but combined with "a" it displays all processes in the system. The option "u" displays a user-oriented format with additional information like the used ID, cpu utilization of the process in "##.%" format, ratio of the process's resident set size to the physical memory on the machine, virtual memory size of the process in KiB, resident set size and the start time of the process. The popular combination of all three `ps aux` shows all processes with additional information [noauthor_ps1_2024] as seen in Figure 3.12.

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	TIME	COMMAND
root	1	0.2	0.0	170260	11888	?	Ss	0:09	/sbin/init splash
root	2	0.0	0.0	0	0	?	S	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	I<	0:00	[rcu_gp]
root	4	0.0	0.0	0	0	?	I<	0:00	[rcu_par_gp]
root	5	0.0	0.0	0	0	?	I<	0:00	[slub_flushwq]
root	6	0.0	0.0	0	0	?	I<	0:00	[netns]
root	8	0.0	0.0	0	0	?	I<	0:00	[kworker/0:0H-events_highpri]
root	10	0.0	0.0	0	0	?	I<	0:00	[mm_percpu_wq]
user1	1673	0.0	0.0	0	0	tty2	S	0:00	/usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu
user1	1675	0.8	0.0	0	0	tty2	S	0:00	/usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -back
user1	1682	0.0	0.0	0	0	tty2	S	0:00	/usr/libexec/gnome-session-binary --systemd -systemd -session=ubuntu
user1	2587	0.0	0.0	0	0	pts/0	S	0:00	bash

Figure 3.12: Simulated output of "ps aux"

The commands `kill` and `killall` can be used to terminate processes[kill][noauthor_killall1_2023]. This is done by simply removing entries from the list of simulated processes. `kill` is used with a process ID to remove the process with this ID[kill]. `killall` uses the executable command to terminate a process[noauthor_killall1_2023].

```

1 for process in self.output:
2     if process.pid == target_process:
3         self.output.remove(process)
4         valid_arg = True

```

To check whether the process was removed a `ps` command can be submitted before and after submitting `kill` or `killall`. If the argument is not valid an error message is displayed that resembles a real error message.

3.3.4 System

The system command handler is responsible for simulating common system-related commands such as `whoami`, `id`, and `w`. These commands are used to retrieve information about the current user and system status, often employed in reconnaissance or debugging scenarios. While their current implementation is hardcoded to provide consistent and predictable outputs, they could be extended in the future to include semi-randomized elements for greater realism.

Currently there are only three basic system-commands supported.

Command	Description	Supported Arguments
<code>whoami</code>	displays the current username	-
<code>id</code>	shows user and group IDs	-
<code>w</code>	lists active user session details	-

Table 3.5: Currently supported system-commands

The `whoami` command simply returns the current username, which is passed to the event handler upon initialization. The `id` command provides details about the user's ID, group ID, and associated groups, with static values reflecting a typical Linux environment. The `w` command simulates an active session, displaying system uptime, load averages, and user activity, including login times, idle durations, and the currently executed command. Although the `w` command currently uses fixed data, attributes such as system uptime, load averages, or idle times could be randomized in the future to enhance authenticity.

```

ubuntu@ubuntu-2204:~$ whoami
ubuntu
ubuntu@ubuntu-2204:~$ id
uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu)
ubuntu@ubuntu-2204:~$ w
 17:08:49 up 51 min,  1 user,  load average: 0.00, 0.01, 0.05
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU WHAT
ubuntu    ttyl     -                12:57   51:15   0.00s   0.00s -bash

```

Figure 3.13: Simulated output of system commands

The underlying implementation relies on a unified interface for command execution. When a command is issued, the handler identifies the requested command, extracts the relevant parameters, and invokes the corresponding helper function to produce the simulated output. This modular approach allows for flexibility and extensibility, making it easy to add additional system commands in the future.

3.4 Logging

Every command intercepted and processed by the handlers is logged in a central log-file. This is done using the python-library "logging". The basic configuration of the logging is executed once on program startup. It is persistent, so even if the program ends and is run again, the new log entries will be appended to the existing file. It needs the following configuration parameters:

- **level:** Sets the threshold for this logger. Messages which are less severe than the given level, in this case "INFO", will be ignored. Using this level-model given by the library means we can mark some dangerous commands in the log-file.
- **format:** The format is set as "Year-Month-Day Hours:Minutes:Seconds Log-Level Message", where the message-parameter contains the whole intercepted command including all parameters.
- **datefmt:** The date format described in the format-section.
- **filename:** As this project is about a honeypot-system, the log-file got the related name "jamjar".

```
1 logging.basicConfig(  
2     level=logging.INFO,  
3     format="% (asctime)s %(levelname)s %(message)s",  
4     datefmt = "%Y-%m-%d %H:%M:%S",  
5     filename="jamjar.log"  
6 )
```

3.5 Additional Commands

3.5.1 cd

Like described in section 3.2.1, the interception of commands is currently achieved primarily by monitoring `execve` system calls using eBPF code. While this approach works well for commands that spawn new processes, it does not cover built-in commands like `cd`, which do not trigger an `execve` call. This limitation causes, among other things, issues when

navigating directories within the honeypot. For example, if a user creates a fake directory using `mkdir` and subsequently attempts to navigate into it using `cd`, the command is executed on the host system rather than being properly handled within the honeypot environment. This results in errors, as the directory only exists in the honeypot's simulated file system.

To address this, the first step was analyzing the behavior of the `cd` command to understand which system calls it invokes. This was done using tools like `strace`, which traces the system calls made by a bash process executing for example the `cd` command. The output revealed that `cd` does not call `execve`, but instead relies on the `chdir()` system call to change the current working directory.

```
1 $ strace -f bash -c "cd /home"
2 ...
3 chdir("/home")          = 0
4 ...
```

The above output, captured with `strace`, demonstrates that `chdir()` is the key system call responsible for directory changes. To enhance the honeypot's capabilities, we extended our eBPF code to intercept `chdir()` calls. Below is the code snippet used for this purpose:

```
1 int syscall__chdir(struct pt_regs *ctx, const char __user *argv) {
2     struct data_t data = {};
3     struct task_struct *task;
4
5     u32 uid = bpf_get_current_uid_gid() & 0xffffffff;
6     if (uid != UID_FILTER) {
7         return 0;
8     }
9
10    data.pid = bpf_get_current_pid_tgid() >> 32;
11    data.uid = uid;
12
13    task = (struct task_struct *)bpf_get_current_task();
14    data.ppid = task->real_parent->tgid;
15    bpf_get_current_comm(&data.comm, sizeof(data.comm));
16    data.timestamp = bpf_ktime_get_ns();
17    data.type = EVENT_CD;
18    bpf_probe_read_user(&data.argv, sizeof(data.argv), argv);
19    events.perf_submit(ctx, &data, sizeof(data));
20    return 0;
21 }
```

This probe allows the honeypot to detect when a directory change is attempted, capturing both the target directory and other relevant data for logging or simulation purposes.

To simulate the behavior of the `cd` command, we also needed to determine the current working directory of the process executing the command. This is crucial to validate whether the target directory is reachable from the current directory within the honeypot's virtual file system. Although this functionality is not yet fully implemented, the foundation is in place to enable future enhancements. Logs already provide feedback on whether a directory change would be successful in the simulated environment.

Another consideration is the terminal prompt itself. In a real Linux environment, the prompt reflects the current working directory, such as:

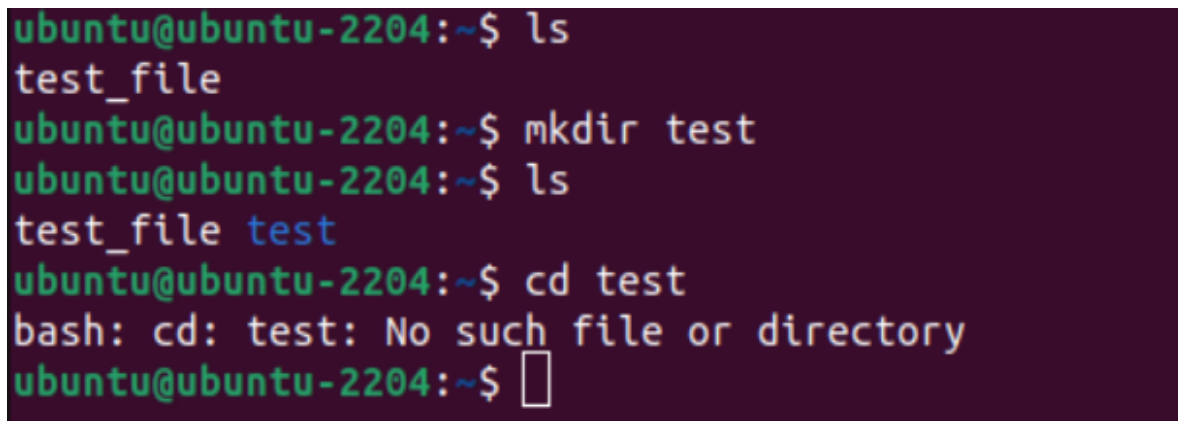
```
ubuntu@ubuntu-2204: /home$
```

One possible approach to addressing this issue could be to fake the prompt within the honeypot to align with the simulated environment. This could be achieved by modifying the PS1 variable in the `.bashrc` file. However, such changes should be applied dynamically to ensure they only affect sessions running within the honeypot.

While the current implementation serves as a foundational layer, it highlights several areas for future work, including:

- Fully integrating the `cd` command into the honeypot's virtual file system.
- Simulating or manipulating the terminal prompt to reflect the honeypot's state dynamically.
- Handling edge cases, for example symlinks, to improve authenticity.

To illustrate the issue with the `cd` command, we created a directory within the honeypot's simulated file system. From the attacker's perspective, the directory creation appears successful, but attempting to change into the newly created directory fails. This happens because the `cd` command is still executed on the host system, where the directory does not exist. Below is a screenshot showing the attacker's terminal output during this process:

A terminal window with a dark purple background and light green text. The prompt is 'ubuntu@ubuntu-2204:~\$'. The user enters 'ls', and the output is 'test_file'. Then the user enters 'mkdir test', and the prompt returns. The user enters 'ls' again, and the output is 'test_file test'. Finally, the user enters 'cd test', and the output is 'bash: cd: test: No such file or directory'. The prompt returns to 'ubuntu@ubuntu-2204:~\$' with a cursor.

```
ubuntu@ubuntu-2204:~$ ls
test_file
ubuntu@ubuntu-2204:~$ mkdir test
ubuntu@ubuntu-2204:~$ ls
test_file test
ubuntu@ubuntu-2204:~$ cd test
bash: cd: test: No such file or directory
ubuntu@ubuntu-2204:~$
```

Figure 3.14: Attacker's perspective: Attempt to change into a fake directory fails.

In contrast, the honeypot's logs accurately reflect the expected behavior within the simulated environment. These logs confirm that the directory change would have been successful if it had been handled entirely within the honeypot. The logs include details about the intercepted `chdir()` system call, the target directory, and whether the operation was valid. The screenshot below shows the corresponding log entry:

```
[+][2025-01-10 14:03:50] Traced Piped Command: [mkdir test]
  \--> Executed in [/home/ubuntu] by UID [1000]
  \--> Attached to process [3851] and killed it!
[+][2025-01-10 14:03:52] Traced Piped Command: [ls]
  \--> Executed in [/home/ubuntu] by UID [1000]
  \--> Attached to process [3852] and killed it!
[ACTION: CHDIR] PID=3754, PPID=3489, UID=1000, COMM=cd, FROM=/home/ubuntu, TO=/home/ubuntu/test, RETVAL=0, RESULT=
Changed directory to home/ubuntu/test
```

Figure 3.15: Honeypot logs: Successful directory change simulation.

This example highlights the current limitation of executing `cd` on the host system rather than fully within the honeypot and demonstrates how the extended eBPF code and logging provide accurate simulation data for future enhancements.

3.5.2 Piped Commands

The goal of implementing piped commands in the honeypot was to simulate Linux-like behavior when executing multiple commands chained together using the pipe symbol (`|`). However, due to the nature of how piped commands operate and the honeypot's current architecture, several challenges arose, requiring additional analysis and foundational improvements for future enhancements.

Piped commands execute each command in the chain as a separate process, with the output of one command serving as the input to the next. For example, a command like `touch test | ls` spawns two distinct processes:

1. `touch test`, which creates a file named `test`.
2. `ls`, which lists the contents of the directory, including the newly created `test` file.

The pipe ensures that the output of the first process (`touch test`) is directly connected to the input of the second process (`ls`). This sequential data flow makes it difficult to handle piped commands in the honeypot, as the current implementation processes one command at a time without considering the interdependence of outputs and inputs in a pipeline.

Using `strace`, the behavior of piped commands was analyzed. The command produced output revealing a `pipe2` system call, which is used to establish the communication channel between commands in the pipeline:

```
1 $ strace -f bash -c "touch test | ls"
2 ...
3 pipe2([3, 4], 0)           = 0
4 ...
```

This discovery highlighted that the honeypot needed to intercept `pipe2` system calls in addition to the existing `execve` monitoring. To address this, the eBPF code was extended as follows:

```
1 int syscall__pipe2(struct pt_regs *ctx, int __user *pipefd)
2 {
3     struct data_t data = {};
4     u32 uid = bpf_get_current_uid_gid() & 0xffffffff;
5     if (uid != UID_FILTER) {
6         return 0;
7     }
8
9     data.pid = bpf_get_current_pid_tgid() >> 32;
10    data.uid = uid;
11    data.timestamp = bpf_ktime_get_ns();
12    bpf_get_current_comm(&data.comm, sizeof(data.comm));
13    data.type = EVENT_PIPE;
14    events.perf_submit(ctx, &data, sizeof(data));
15    return 0;
16 }
```

This enhancement enabled the honeypot to detect when a pipeline was initiated, allowing further logic to handle piped commands.

Despite detecting the pipe2 system call, several challenges remained:

1. **Command Sequencing:** Since each command in the pipeline spawns a separate process, there is no straightforward way to determine the total number of commands in the pipeline or their exact order. The `execve` calls are processed one by one, meaning the honeypot would process and output each command individually, breaking the sequential flow required for a proper pipeline simulation.
2. **Simulating Outputs and Inputs:** The honeypot needs to store the output of each command in the pipeline and pass it as input to the next. Initially, a piped flag was introduced, which, when set to `True`, prevented the immediate output generation. Instead, command outputs were temporarily stored in a buffer. The idea was to wait until the final command in the pipeline was reached before processing the output.
3. **Identifying the Final Command:** A significant challenge was determining when the last command in the pipeline had been reached. Since the honeypot processes each `execve` call independently, there was no clear indicator marking the end of the pipeline. The `pipe2` system call does not provide information about how many commands are connected via pipes. To address this, a counter was implemented to track the number of commands. For simplicity, an initial predefined maximum of two commands was set, and once this limit was reached, the pipeline processing logic was triggered.
4. **Re-execution and Ordering:** Another approach involved collecting all processes in the pipeline, sorting them for example by process ID (PID) to ensure the correct order, and then re-executing the commands in sequence. This method faced issues as many processes had already terminated by the time the re-execution logic was applied, making it impossible to write outputs back to their respective `/proc/<pid>/fd/1` files.

The current implementation focuses on handling up to two commands in a pipeline. Outputs are buffered, and the pipeline's results are written to the final process in the chain. However, due to the nature of piped commands and the limitations of the current architecture, several issues persist:

- Processes often terminate too quickly, making it almost impossible to reliably attach to them using `attach_trace`. As a result, capturing their outputs or integrating them into the pipeline simulation frequently fails.
- The reordering of processes is error-prone and does not always align with real-world behavior.
- Simulating complex pipelines with more than two commands remains infeasible due to these challenges, as the current implementation struggles to handle interdependent processes in a coordinated manner.

The earlier approach of re-executing all processes in the correct order would theoretically allow the honeypot to handle pipelines with an arbitrary number of commands. However, due to the issues with terminated processes and inaccessible file descriptors, this method was ultimately abandoned in favor of the simpler two-command limit.

Due to these challenges, parts of the code have been commented out and are currently non-functional. These sections, however, serve as a foundation for future improvements. They highlight key areas where the logic could be refined to achieve a more reliable and accurate simulation of piped commands. While the current logic is not yet robust enough for consistent and trustworthy results, it provides valuable insights and a framework for developing a fully functional implementation.

3.5.3 iptables

The `iptables` command simulates the basic functions of the Linux command `iptables`, which makes it possible to use network filters. Therefore, a central class, `Iptables`, is used to manage and store the groups (chains) and the related rules. The predefined chains in this implementation are `INPUT`, `FORWARD`, and `OUTPUT`. These chains are managed by a class located in the helper folder, specifically in the `iptables.py` file. This class is designed to efficiently handle the storage, retrieval, and management of rules.

The class contains the following functions:

- `add_rule(self, chain, source, destination, protocol, source_port, destination_port, action)`: This function makes it possible to create new rules for the various chains.
- `remove_rule(self, chain, source, destination, protocol, source_port, destination_port, action)`: This function allows you to delete certain rules.

- `list_rules(self)`: This function lists all existing rules together with their chains.
- `clear_chain(self, chain)`: This function can be used to empty a chain.
- `clear_all(self)`: This function can be used to remove all existing rules.
- `rule_exists(self, chain, source, destination, protocol, source_port, destination_port, action)`: This function can be used to check whether a specific rule exists or not.

It supports various arguments like `-L`, `-h`, `-help`, `-A`, `-D` and `-F`. The functions of these arguments are explained below:

- `-L`: This argument allows you to output the current rules and chains.
- `-h` & `-help`: This argument can be used to output the help overview.
- `-A`: This argument can be used to create a new rule.
- `-D`: This argument allows you to delete a rule. It checks whether the rule to be deleted does exist.
- `-F`: With this argument, either all chains or only a specific chain can be emptied by passing a chain name.

For example you can display all chains and rules with the command `iptables -L`:

```
Chain INPUT (policy ACCEPT)
target     prot opt source                destination
ACCEPT     tcp  --  192.168.1.1            anywhere             dport:22
DROP       udp  --  10.0.0.0/8            anywhere             sport:53
Chain FORWARD (policy ACCEPT)
target     prot opt source                destination
ACCEPT     tcp  --  172.16.0.0/16         192.168.0.0/16      dport:80
Chain OUTPUT (policy ACCEPT)
target     prot opt source                destination
```

Figure 3.16: iptables output

To block outgoing ICMP traffic and prevent pings, you can use the following approach:

For example, executing the command `iptables -A OUTPUT -p icmp -j DROP` will add a rule to the OUTPUT chain that blocks all outgoing ICMP packets. As a result, if you try to ping a domain, like `ping google.com`, the request will fail because the ICMP packets are dropped.

```
PING google.com (172.217.18.14) 56(84) bytes of data.
--- google.com ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 37 ms
```

Figure 3.17: ping fail

3.5.4 cat

The `cat` command replicates the core functionality of the Linux `cat` command, enabling the content of a file to be displayed directly in the terminal. In this implementation, simulated files and their contents are represented as file objects created using the `touch` command. The contents of each file are stored in the `file_content` variable within the corresponding class, ensuring efficient access and management of file data.

When executing the `cat` command, the process begins by checking whether a file name has been provided as an argument. If an argument is supplied, the command then verifies whether the specified file exists. Once confirmed, it determines if the argument refers to a file or a folder. If the argument equals a folder, an error message is displayed, indicating that directories cannot be processed. However, if the argument corresponds to a valid file, the command reads and displays its content directly in the terminal.

For instance, if you use the `cat` command to display the contents of a script, the output will show the file's text as follows:



```
#!/bin/bash
ps
iptables -A OUTPUT -p icmp -j DROP
iptables -L
ping google.com
iptables -L
iptables -D OUTPUT -p icmp -j DROP
ping google.com
touch test2
echo 'ls -la ping google.com iptables -L' > test2
echo 'abcdef' >> test2
```

Figure 3.18: Simulated output of cat command

3.5.5 bash

The `bash(self, args, scr_dir)` function provides the capability to execute scripts with various contents in a secure and controlled environment. To run a script, the command `./filename` is used. When executed, the function first checks whether the specified file exists. If the file is not found, an appropriate error message is displayed. However, if the file exists, the function verifies whether it is an executable file and whether the required permissions are in place for execution.

If the necessary permissions are granted, the script's execution begins. The script's content is processed line by line, with each line sent for execution individually. If the first line of the script contains a shebang (e.g., `#!/bin/bash`), it is recognized and skipped to prevent errors during execution. Once the shebang is processed, each command in the script is passed sequentially to the `execute_command(self, command, src_dir)` function, which is placed in the `scripts.py` file within the helper folder. This modular approach ensures a robust and error-free execution of scripts.

Additionally, commands like `echo` can include text inside quotation marks (' or "), which might contain line breaks. These line breaks are treated as part of the text, and everything inside the quotation marks is processed as a single unit to avoid errors.

```
def execute_command(handler, cmd, src_dir=""):
    comm = cmd.split(" ")[0]
    match comm:
        case ("ls"|"rm"|"touch"|"cat"|"echo"|"mkdir"|"rmdir"):

            dir = PtraceSubroutines.CMD.invoke_dir(cmd, src_dir)
            if isinstance(dir, list):
                dir = "\n".join(dir)
            return dir
        case ("ping"|"arp"|"ip"|"traceroute"|"dig"|"iptables"):
            network = PtraceSubroutines.CMD.invoke_network(cmd)
            if isinstance(network, list):
                network = "\n".join(network)

            return network
        case ("ps"|"kill"|"killall"):
            process = PtraceSubroutines.CMD.invoke_process(cmd,0,0)
            if isinstance(process, list):
                process = "\n".join(process)

            return process
        case ("w"|"whoami"|"id"):
            system = PtraceSubroutines.CMD.invoke_system(cmd,0,0)
            if isinstance(system, list):
                process = "\n".join(system)

            return system
        case _:
            print(f"[!] Subroutine for command {comm} is not implemented yet!")
            return ""
```

Figure 3.19: Execute Command function

The `execute_command` function is designed to process individual commands without creating a separate process, returning the result of each command as a string. These output strings are collected by the `bash` function, merged into a single output, and then displayed in the terminal.

The `execute_command` function takes two arguments: `cmd` (the command to execute) and an optional `src_dir` (the source directory, if required). The function operates by analyzing the main command (the first part of `cmd`), identified by splitting the string at the first space.

Based on this, the command is routed to the right subroutine corresponding to its type. This makes sure that each command is handled properly and efficiently, and its output is combined smoothly into the final result provided by the bash function.

For example, if you execute a script that has the following content:

```
1 #!/bin/bash
2 iptables -A OUTPUT -p icmp -j DROP
3 iptables -L
4 ping google.com
5 iptables -D OUTPUT -p icmp -j DROP
6 iptables -L
7 ping google.com
```

The individual commands are executed sequentially. First, a new iptables rule is added to block all ICMP packets. Afterward, the current iptables rules are listed to confirm that the new rule has been successfully created. Next, a ping to google.com is attempted, demonstrating that the rule is functioning correctly by dropping the ping request. Finally, the rule is removed, the updated rules are displayed, and a subsequent ping is performed to verify that ICMP traffic is working again.

The output appears as follows:

```

Chain INPUT (policy ACCEPT)
target     prot opt source                               destination
ACCEPT     tcp  --  192.168.1.1                           anywhere           dport:22
DROP       udp  --  10.0.0.0/8                           anywhere           sport:53
Chain FORWARD (policy ACCEPT)
target     prot opt source                               destination
ACCEPT     tcp  --  172.16.0.0/16                        192.168.0.0/16     dport:80
Chain OUTPUT (policy ACCEPT)
target     prot opt source                               destination
DROP       icmp --  anywhere                             anywhere
PING google.com (172.217.18.14) 56(84) bytes of data.
--- google.com ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 37 ms
Chain INPUT (policy ACCEPT)
target     prot opt source                               destination
ACCEPT     tcp  --  192.168.1.1                           anywhere           dport:22
DROP       udp  --  10.0.0.0/8                           anywhere           sport:53
Chain FORWARD (policy ACCEPT)
target     prot opt source                               destination
ACCEPT     tcp  --  172.16.0.0/16                        192.168.0.0/16     dport:80
Chain OUTPUT (policy ACCEPT)
target     prot opt source                               destination
DROP       icmp --  anywhere                             anywhere
PING google.com (172.217.18.14) 56(84) bytes of data.
64 bytes from 172.217.18.14: icmp_seq=1 ttl=56 time=46.1 ms
64 bytes from 172.217.18.14: icmp_seq=2 ttl=56 time=13.3 ms
64 bytes from 172.217.18.14: icmp_seq=3 ttl=56 time=16.5 ms
64 bytes from 172.217.18.14: icmp_seq=4 ttl=56 time=23.1 ms
--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 99 ms

```

Figure 3.20: Simulated output of "script"

3.5.6 echo

The echo command is not fully implemented as it is a shell build-in command by default. Built-ins are processed directly by the shell itself and are not executed as independent executable files in the file system. That's why the echo command does not work completely. In contrast, `/bin/echo` works because it is an independent executable file that is located directly in the file system. When using `/bin/echo`, a new process is started that executes system calls such as `execve`, allowing the command to be monitored and traced. However, using only echo results in the shell's built-in being executed, which is processed internally and does not generate comparable system calls.

If the echo command is implemented in a script, the built-in echo also works reliably. This is because executing the bash function simulates the individual commands within the script without creating a process.

The echo function starts with a check of the arguments (args). If no arguments are specified, a line break (\n) is simply returned, as is the case with the original echo command.

It is then checked whether redirection characters such as > or » are contained in the arguments. If no redirection is specified, the function simply returns the arguments as a linked string. If redirection characters are found, the function attempts to write the text to the specified file. If one of these arguments is found, the function checks if the target file exists. If the file exists, the new text is written to the contents of the file. If the file does not exist, the function returns an error message: -bash: '{target_file}': No such file or directory.

```
def echo(self, args, src_dir):
    target_file = ""
    if len(args) == 0:
        return "\n"
    else:
        if '>>' in args or '>' in args:
            if len(args) > 2 and args[-2] in [ ">", ">>" ]:
                new_file_content = " ".join(arg.replace("'", '').replace('"', "") for arg in args[:-2])
                src_dir_list = helper.path_to_list_helper(src_dir)
                target_file, args_str = helper.get_main_arg_helper(args)
                target_file = args[-1]
                if "/" in target_file:
                    target_file, src_dir_list = helper.target_dir_is_path_helper(target_file, src_dir_list)
                src_obj = self.root
                if src_dir_list != []:
                    for layer in src_dir_list:
                        src_obj = src_obj.content[layer]
                if args[-2] == ">":
                    try:
                        target_obj = src_obj.content[target_file]

                        target_obj.filecontent = new_file_content
                        return ""
                    except KeyError:
                        return f"-bash: '{target_file}': No such file or directory"
                elif args[-2] == ">>":
                    try:
                        target_obj = src_obj.content[target_file]

                        target_obj.filecontent = target_obj.filecontent + "\n" + new_file_content
                        return ""
                    except KeyError:
                        return f"-bash: '{target_file}': No such file or directory"
            else:
                return "-bash: syntax error near unexpected token `newline'"
    return " ".join(arg.replace("'", '').replace('"', "") for arg in args) + "\n"
```

Figure 3.21: Echo function

Problems:

During development, problems often occurred with the ptrace call. A common issue was the error No such process, which appeared when trying to trace a process.

```
Exception ignored on calling ctypes callback function: <function PerfEventArray._open_perf_buffer.<locals>.raw_cb_ at 0x73691e7ea660>
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/bcc/table.py", line 992, in raw_cb_
    callback(cpu, data, size)
  File "/home/ubuntu/Desktop/test2/jamjar/src/eBPF/prod/jamjar.py", line 190, in proc_event
    target_process = attach_ptrace(event.pid)
  File "/home/ubuntu/Desktop/test2/jamjar/src/eBPF/prod/jamjar.py", line 180, in attach_ptrace
    process = DEBUGGER.addProcess(pid, False)
  File "/usr/lib/python3/dist-packages/ptrace/debugger/debugger.py", line 82, in addProcess
    process = PtraceProcess(self, pid, is_attached,
  File "/usr/lib/python3/dist-packages/ptrace/debugger/process.py", line 172, in __init__
    self.attach()
  File "/usr/lib/python3/dist-packages/ptrace/debugger/process.py", line 189, in attach
    ptrace_attach(self.pid)
  File "/usr/lib/python3/dist-packages/ptrace/binding/func.py", line 177, in ptrace_attach
    ptrace(PTRACE_ATTACH, pid)
  File "/usr/lib/python3/dist-packages/ptrace/binding/func.py", line 155, in ptrace
    raise PtraceError(message, errno=errno, pid=pid)
ptrace.error.PtraceError: ptrace(request=16, pid=3514, 0x(nil), 0x(nil)) error #3: No such process
```

Figure 3.22: error

After analyzing this error, we quickly came to the conclusion that the problems were caused by the short lifespan of the processes. Probably the processes are terminated too quickly so that the ptrace call can no longer be called in time. To check this, we adapted the attach_ptrace function that throws the error and added some checks:

```
def attach_ptrace(pid):
    if not os.path.exists(f"/proc/{pid}"):
        print(f"[DEBUG] Process {pid} does not exist anymore.")
        return None
    try:
        process = DEBUGGER.addProcess(pid, False)
        return process
    except Exception as e:
        print(f"[ERROR] Failed to attach to PID {pid}: {e}")
        return None
```

Figure 3.23: ptrace function

This checks whether the process to be terminated still exists at the time of the attach_ptrace call. If it no longer exists, a suitable message is displayed in the console. If the process is found, the attach_ptrace function is processed normally.

The results of this adjustment confirmed the assumption, as in some cases the process no longer existed at the time of the attach_ptrace call.

```
[DEBUG] Skipping process 2944 as attach failed.
[DEBUG] Process 2944 does not exist anymore.
```

Figure 3.24: ptrace error

4 Results

4.1 Conclusion

The main goal of this project work was to create a honeypot system for Linux/unix systems that can run on an already-in-use system such as a mailserver or IoT-device. Our solution does provide honeypot capabilities on a system while using only limited resources. Of course this project could not implement all possible unix commands so an attacker will find boundaries and become suspicious when spending enough time on the system. But the open architecture and APIs of JamJar make it easy for others to expand the capabilities by implementing new commands.

4.2 Evaluation

During the presentation the system was shown as a live demo. While a real user can still access the whole system and is not influenced or hindered by JamJar, a potentially breached user account was locked inside the simulated environment and could not easily tell the difference. As already discussed, currently the simulated environment does contain some abnormalities. For example all files and directories contain the same timestamp and size and all simulated processes use the same amount of CPU and memory. Randomizing these values might be an effective step to make this system more stealthy. Nevertheless JamJar works and because of the lightweight architecture, no noticeable delay between command execution and the simulated response occurs. Also the performance of the host system is not really restricted.

4.3 Future Work

As this project was only the start of JamJar, there are a lot of points and ideas for future work on the topic. For example it might be interesting to be able to replace the simulated system that is loaded into memory on program start in an easy way. We could think of some configuration file stored in JSON format that contains system parameters like the directory

structure, running processes or network parameters. Outsourcing these parameters in the configuration file would make it easy to swap them and deploy JamJar on a lot of systems fast. Also a separate program could be implemented to generate this configuration from another machine simply by executing it on the running system. This would allow JamJar to generate some sort of fingerprint of machines that can be used as the configuration file.

Another interesting approach might be the simulation of a windows system. Although the currently used commands and technologies such as eBPF are not as easily accessible on Windows machines, there is also a need for honeypot systems in windows environments.

List of Tables

2.1	Difference of exec*-commands	8
2.2	Final List of commands	13
3.1	Currently supported directory-commands	26
3.2	Possible context of the command "ls"	26
3.3	Currently supported network-commands	28
3.4	Currently supported process-commands	31
3.5	Currently supported system-commands	34

List of Figures

2.1	eBPFs with python bcc	9
2.2	Relationship between Linux commands	14
3.1	Command pipeline	18
3.2	JamJar console output	21
3.3	Simulated output of "ls -al /home/user"	28
3.4	Simulated output of "ping google.com"	29
3.5	Simulated output of "ip a"	30
3.6	Simulated output of "dig -x 142.250.184.195"	30
3.7	Simulated output of "ps" (left) and real output (right)	32
3.8	Simulated output of "ps -C bash" (left) and real output (right)	32
3.9	Simulated output of "ps -f"	32
3.10	Simulated output of "ps -fu user1"	33
3.11	Simulated output of "ps a"	33
3.12	Simulated output of "ps aux"	33
3.13	Simulated output of system commands	34
3.14	Attacker's perspective: Attempt to change into a fake directory fails.	37
3.15	Honeypot logs: Successful directory change simulation.	38
3.16	iptables output	41
3.17	ping fail	41
3.18	Simulated output of cat command	42
3.19	Execute Command function	43
3.20	Simulated output of "script"	45
3.21	Echo function	46
3.22	error	47
3.23	ptrace function	47
3.24	ptrace error	47