HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

BACHELORARBEIT

Herr
Christian Roth

Developing of a methodology for evaluation of targeted cyber attacks using exploits on ARM-based Industrial IoT devices

Mittweida, 2020

Fakultät: Angewandte Computer- und Biowissenschaften

BACHELORARBEIT

Entwicklung einer Methodik zur Erforschung von zielgerichteten Angriffen mittels Exploits auf ARM-basierte Industrial IoT-Geräte

Autor:
Herr

Christian Roth

Studiengang:
Angewandte Informatik IT-Sicherheit

Seminargruppe:
IF16wI-B

Erstprüfer:
Prof. Dr. rer. pol. Dirk Pawlaszczyk

Zweitprüfer:
Heiner Winkler, M.Sc.

Einreichung:
Kulmbach, 31.08.2020

Faculty of Applied Computer- and Biology Sciences

BACHELOR THESIS

Developing of a methodology for evaluation of targeted cyber attacks using exploits on ARM-based Industrial IoT devices

author:
Mr.

Christian Roth

course of studies:
Applied Computer Science IT-Security

seminar group:
IF16wI-B

first examiner:
Prof. Dr. rer. pol. Dirk Pawlaszczyk

second examiner:
M.Sc. Heiner Winkler

submission:
Kulmbach, 31.08.2020

Bibliografische Beschreibung:

Christian Roth:

Entwicklung einer Methodik zur Erforschung von zielgerichteten Angriffen mittels Exploits auf ARM-basierte Industrial IoT-Geräte

2020 - 103 Seiten.

Mittweida, Hochschule Mittweida (FH), University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften, Bachelorarbeit, 2020

Referat:

Diese Arbeit beschäftigt sich mit der Entwicklung einer Methodik / eines Konzeptes um zielgerichtete Angriffe auf IIoT / IoT Geräte zu analysieren. Aufbauend auf den recherchierten Grundlagen über Honeypots, Fileless Malware und Injection Techniken wird eine Methodik erstellt, die zu einem Konzept eines Honeypot Analysesystems führt. Dieses System dient dem spezialisierten Detektieren und Analysieren von neuartigen Gefahren wie Fileless Attacks die häufig von Advanced Persistent Threats genutzt werden. Ein Teil dieses Systems wird implementiert und durch einen simulierten Angriff welcher Fileless Attacks nutzt getestet. Die entsprechende Effektivität der Implementierung wird bewertet und diskutiert.

Bibliographic description :

Christian Roth:

Developing of a methodology for evaluation of targeted cyber attacks using exploits on ARM-based Industrial IoT devices

2020 - 103 Seiten.

Mittweida, Hochschule Mittweida (FH), University of Applied Sciences,
Faculty of Applied Computer- and Biology Sciences, Bachelor Thesis, 2020

Abstract:

This thesis deals with the development of a methodology / concept to analyse targeted attacks against IIoT / IoT devices. Building on the established background knowledge about honeypots, fileless malware and injection techniques a methodology is created that leads to a concept of a honeypot analyzation system. The system is created to analyse and detect novel threats like fileless attacks which are often utilized by Advanced Persistent Threats. That system is partially implemented and later evaluated by performing a simulated attack utilizing fileless attacks. The effectiveness is discussed and rated based on the results.

Content

Content I

Table of figures III

List of tables IV

Index of Abbreviations V

1 Introduction 1

1.1 Motivation *1*

1.2 Goals *2*

1.3 Chapter overview *3*

2 Background 5

2.1 Internet of Things and Industrial Internet of Things *5*

2.2 Different ways to compromise *7*

2.2.1 Traditional Malware 7
2.2.2 Fileless Attacks / Malware 9
2.2.3 Advanced Persistent Threats 17

2.3 Post Exploitation utilizing Injection Techniques *17*

2.3.1 Process Hollowing 18
2.3.2 Injection and Persistence via Registry Modficiations 18
2.3.3 Hook Injection via SetWindowsHookEX 19
2.3.4 IAT Hooking and Inline Hooking 19
2.3.5 Anonymous File Creation through memfd_create() 19
2.3.6 Reflective DLL injection 20
2.3.7 Ptrace injection 21
2.3.8 Doppelgänging Injection 22

2.4 Forensic Investigation / Detection of Fileless Attacks *22*

2.5 Honeypots *26*

2.6 Intrusion Detection Systems *30*

3 Methodology 32

3.1 Layer 1 32

3.2 Layer 2 42

3.3 Layer 3 47

4 Layer 1 Implementation 49

4.1 Implementation 49

4.2 Attack simulation 54

4.3 Results 57

5 Discussion / Conclusion 62

5.1 Result Interpretation 62

5.2 Restrictions and Limitations 64

5.3 Future Work 65

5.4 Conclusion 66

Bibliography 67

Appendices 73

Appendix 1 .... A-I

Appendix 2 .... A-VII

Selbstständigkeitserklärung

Table of Figures

Figure 1: SCADA Components (cf. United States General Accounting Office, 2002).....7

Figure 2: The four different monitoring modules of the Raspberry Pi Honeypot.......33

Figure 3: Windows paths that should be monitored (cf Johnson, et al., 2017)............43

Figure 4: Fileless malware honeypot detection concept..48

Figure 5: The layout of the Layer 1 implementation test...49

Figure 6: /etc/audit/audit.rules ruleset...53

Figure 7: Wireshark capture of SSH bruteforce 55

Figure 8: Syscall number memfd_create 57

Figure 9: Wireshark intercepting OpenSSH handshake 58

Figure 10: ausearch -f /etc/passwd log results...59

Figure 11 auditd systemcall log for memfd_create 60

Figure 12 console forensics to show memfd files 60

Figure 13 module.dwarf error upon loading the profile..61

List of Tables

Table 1: WMI objects used for reconnaissance (cf. Graeber, 2015) 12

Table 2: Linux blind files post exploitation (Adrian, 2019) 15

Table 3: Useful ptrace operations (Chester, 2017)... 21

Table 4: Useful WMI Objects for forensic investigation (Buddy, 2019) 24

Table 5: characteristics of honeypot interaction levels ....
(Nawrocki, Wählisch, Schmidt, Keil, & Schönfelder, 2016) 27

Table 6: Important Linux directories for monitoring attacks (cf. Smith, 2017)........... 36

Index of Abbreviations

IoT	Internet of Things
IIoT	Industrial Internet of Things
APT	Advanced Persistent Threat
ICS	Industrial Control System
LPWAN	Low Power Wide Area Network
SCADA	Supervisory Control and Data Acquisition
ASLR	Address space layout randomization
DEP	Data Execution Prevention
WMI	Windows Management Instrumentation
DoS	Denial of Service
SSH	Secure Shell
WQL	Windows Management Instrumentation Query Language
DLL	Dynamic Link Library
IDS	Intrusion Detection System
API	Application Programming Interface
ELF	Executable and Linkable Format
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ICMP	Internet Control Message Protocol
URL	Uniform Resource Locator
OS	Operating System
PLC	Programmable Logic Controller

HIDS Host-based Intrusion Detection System

GCC GNU Compiler Collection

1 Introduction

In this chapter the motivation and goals of this thesis are discussed. A small overview over the individual chapters is also presented.

1.1 Motivation

During recent years cyberattacks on IoT[1] / IIoT[2] devices have increased rapidly. In 2019 alone reports from F-Secure claimed a surge of Cyberattacks targeting IoT devices by 300% within a single year (cf. Doffman, 2019).
In 2017 FIREEYE detailed in a blog post the malware TRITON[3] which targeted ICS[4] Systems (cf Johnson, et al., 2017). The attack could have led to potential disruptions of critical infrastructure. It was one of the many APT[5] attacks targeting industrial networks. One of the reasons for the popularity of IoT cyberattacks is that existing security systems are often not compatible with IoT. In addition, the resource constrained nature makes it difficult to create reliable endpoint / network security systems. Furthermore, software and protocols in use are often outdated or have simply not been designed to fulfil security requirements. ICS often lack cybersecurity measures, as they were designed with isolated environments in mind. Protocols like Modbus[6] and Profinet[7] are missing important features providing secure authentication and detection of unusual behaviour. Deployment of devices / technologies from different vendors introduces different security levels leading to potential risks too. (cf. Trend Micro, 2020)

[1] IoT means Internet of Things and is the interconnection of various devices over the Internet

[2] IIoT is a more specialized description of IoT devices, which are primarily deployed in industrial networks

[3] TRITION is an ICS Attack Framework malware https://www.fireeye.com/blog/threat-research/2017/12/attackers-deploy-new-ics-attack-framework-triton.html

[4] ICS describes various kinds of control systems used to operate and monitor industrial processes

[5] APT are extremely well funded attacks that usually persist for a long time and are executed with extreme caution utilizing modern techniques and undiscovered vulnerabilities

[6] Modbus is a Protocol used to establish client-server communication between intelligent devices, sensors and instruments for field device monitoring using PCs and HMIs https://modbus.org/faq.php

[7] Profinet is an advanced industrial ethernet protocol used to exchange data between controllers and devices https://us.profinet.com/technology/profinet/

As IoT is a rather novel technology, it also lacks the years of research from traditional system security.

In recent years especially fileless malware attacks have gained popularity because of their powerful and stealthy characteristics. A security report of Kaspersky Lab in 2017 revealed that over 140 enterprises in 40 different countries were affected by this kind of malware (cf. Kaspersky GREAT Global Research & Analysis Team, 2017). Recently researchers from the MobiSys19 conference created the HoneyCloud[8] platform to analyse fileless attacks[9]. They reported that up to 9.7% of successful breaches captured within their honeypots[10] could be identified as fileless attacks. Fileless attacks are especially dangerous as they leave no footprint, can be extremely stealthy and thus are hard to detect with existing security solutions making them a reliable APT tool. Systems with weak credential authentication are especially prawn to be targeted.
(cf. Fan, et al., 2019)

As the popularity of such attacks keeps increasing, more complex and sophisticated attacks will emerge, endangering the security of industrial sectors. New defensive measures must be created and evaluated to secure the future of IIoT.

1.2 Goals

This thesis deals with potential modern IoT / IIoT threats focusing on APT utilizing fileless malware and fileless attack trends. Possible attack techniques in this category are described with a special focus on memory injection for post exploitation.

The goal is to create a concept for a security platform which can be used to analyse fileless attacks of advanced persistent actors and gather intel for possible further investigations of new attack patterns or trends. Especially the detection and functionality of memory injection plays a major role as it serves as a distinguishment from basic threats. For this the necessary background knowledge is gathered from literature about the components employed in the system and the nature and techniques of fileless attacks. The security platform is created with extensibility in mind to open the possibility of adding any number of honeypots that suit the necessary situation. Due to the high complexity and big scope of the thematic only the first layer of the proposed concept is implemented utilizing a Raspberry Pi device acting as a honeypot. The honeypot

[8] A honeypot system created to analyse IoT attacks based on Linux devices https://honeycloud.github.io/

[9] Fileless attacks / Fileless malware utilizes techniques that don't require persistent files unlike traditional malware

[10] Honeypots are systems deceiving intruders into believing they are legitimate to extract information about newly arising attack patterns or simply detect threats

will monitor various characteristics to deduce malicious activity from changes that are easily observable in an isolated state. Afterwards a possible attack is showcased to assess the effectivity of the system to monitor fileless attack activity. At the end, the effectivity of the implementation is discussed while comparing it to current research on fileless attacks.

1.3 Chapter overview

After the chapter overview the background is presented consisting of current research information about IoT / IIoT, the nature of fileless attacks, post exploitation injection techniques, forensic investigation of fileless malware, honeypots and Intrusion Detection Systems. Upon this research a Methodology is created step by step which leads to the final concept of the honeypot platform at the end of the chapter. One layer of the proposed concept is implemented to assess the effectivity of the proposed methods and afterwards evaluated by simulating an attack in the following chapter. The resulting monitored activity is used to further discuss the effectivity of the implementation. Finally, the results and their implications are discussed at the end, as well as limitations and future possibilities.

2 Background

To understand the components used in in the analyzation platform and the modern threats endangering IoT / IIoT networks and systems this chapter describes the necessary elemental knowledge needed.

2.1 Internet of Things and Industrial Internet of Things

The IoT can be summarized as a mass of smart devices which interact with their surrounding environments by sensing and then processing and transmitting the data back to the environment. All those objects are interconnected through the internet and lead to sustainability and safety of industries and society. The IIoT is a subset of the IoT, focusing specifically on the machine-to-machine interaction as well as industrial communication technologies. IIoT requires to have a support of very large number with cost efficiency in mind, work with resource constrained devices and be energy efficient. Low latency and reliability are important to ensure the timeless and guaranteed execution of critical procedures. While IoT and IIoT are closely related in nature, there are key differences. When referring to the regular IoT mostly the consumer IoT is meant. Such devices are mostly used to improve human awareness of the surrounding environment and can be classified as machine-to-user interaction. IIoT on the other hand tries to connect the entirety of all industrial assets with each other interconnecting machines, control systems, information systems and business processes. Data acquired this way can be used for analytics optimization of industrial operations. Important activities include monitoring/supervision, closed-loop control and interlocking control[11]. Closed-loop control[12], inter-locking and control applications require minimal delay and very high reliability in communication. Depending on the use case huge amounts of data are being transmitted in IIoT networks.
(cf. Sisinni, Saifullah, Han, Jennehag, & Gidlund, 2018)

A popular approach to describe IIoT architecture is the three-tier pattern. The first domain within this architecture is the Enterprise, which encompasses business and application domains, providing analytics, management options, archives as well as User

[11] Interlocking Systems are mostly used in rail operations to ensure safety by checking if sections are free and determining possible routes https://www.mobility.siemens.com/global/en/portfolio/rail/automation/interlocking-systems.html

[12] Closed loop control systems do not require any manual input regulating processes by themselves

Interfaces. Monitoring, control and safety is contained by the edge domain. IIoT components interact with one another within this domain, consisting of various components like sensors, controllers and actuators within multiple independent local area networks. Edge gateways connect the smaller local networks to larger ones. Finally, the Platform tier creates a link between the other tiers, providing a secure and shared message bus. The lower layers of the IIoT network stack describes the exchange of physical signals as well as the protocols used to link the devices together. One well accepted standard is based on IEEE802.15.4[13] compliant radio, however it was not designed with supporting many devices in mind. Another such solution is LPWAN[14] which allows communication over long distances at very low transmission power. The upper layer aims to ensure interoperability using common data structures and fixated rules for information exchange. It consists of the transport layer exchanging variable length messages among the individual devices and the framework layer transferring structured data with higher abstraction levels. Messaging protocols are utilized for Horizontal integration. (cf. Sisinni, Saifullah, Han, Jennehag, & Gidlund, 2018)

One of many key technologies for IIoT are Supervisory Control and Data Acquisition (SCADA) systems. SCADA is being used to control many critical infrastructures such as electrical power generation and transmission, water treatment, mass transit and manufacturing. Figure 1 shows the components of such a system. The Supervisory control and monitoring station consist of a redundant application server, an engineering workstation as well as a human-machine interface which logs information acquired by remote stations and sends commands in response to events. Alarms and status information is also displayed by the human-machine interface. Local stations have remote terminal units, programmable logic controllers or other controllers build into them, which receive signals from local sensors and transmits information to the control equipment. The enterprise network is often interconnected with the control systems. (cf. United States General Accounting Office, 2002)

[13] IEEE802.15.4 is a low data rate solution with extensive battery life and low complexity http://www.ieee802.org/15/pub/TG4.html

[14] LPWAN means Low Power Wide Area Network and consist of wireless technologies providing large coverage of areas, low bandwidth, small packet size and long battery life https://tools.ietf.org/id/draft-ietf-lpwan-overview-09.html

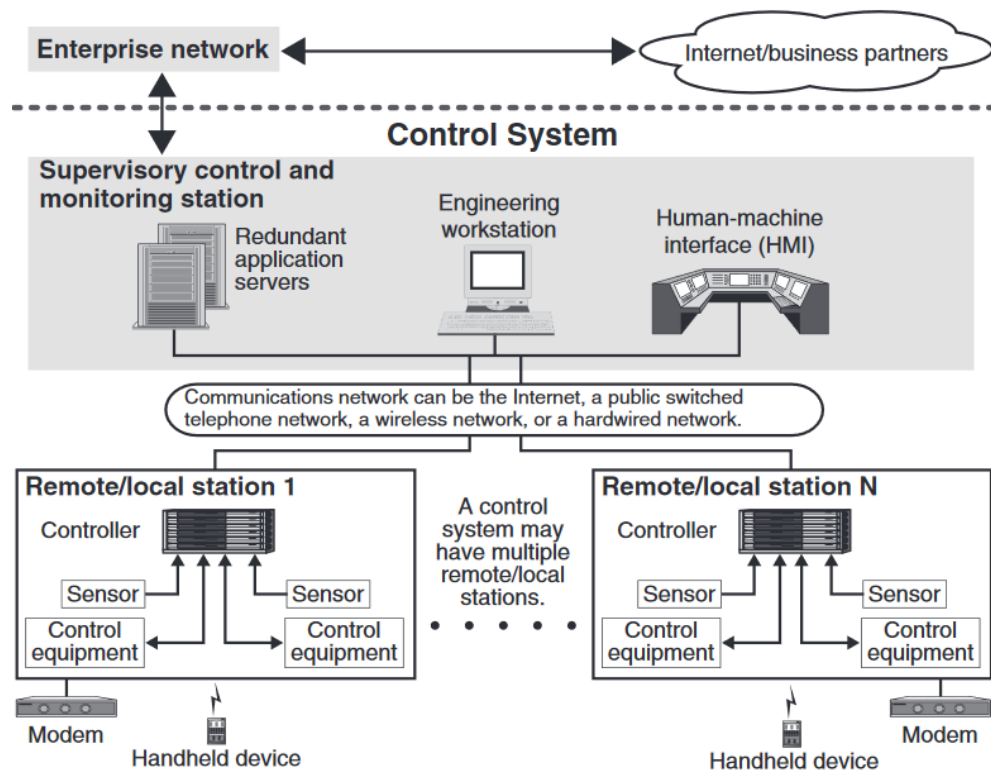

Figure 1 SCADA Components (cf. United States General Accounting Office, 2002)

2.2 Different ways of compromise

2.2.1 Traditional malware

Malware is short for malicious software and can consist of various types which is harmful to systems. A short overview of common different categories will be given in this chapter.

A backdoor is malicious code intended to give an attacker hidden access to a computer without rightful authentication, by installing itself and thus allowing to execute commands. (cf. Sikorski & Honig, 2012)

Botnets are very similar to backdoors. However, they are mostly utilized to receive certain single commands from command-and-control servers. (cf. Sikorski & Honig, 2012)

Downloaders function as a first step tool to download the actual payload once they are installed / executed. This allows the attacker to first infect the system with a very small footprint file, making it easier to successfully compromise a system.
(cf. Sikorski & Honig, 2012)

Information stealing malware, also called **spyware**, secretly collects private information and transmits it to the attacker's system. Examples are keyloggers, password hash

grabbers and sniffers. After extracting the information, the attacker can use the stolen information to get access to password protected services or sell the stolen intellectual property / private data. (cf. Sikorski & Honig, 2012)

Launchers have a similar concept to downloaders except that they simply execute another piece of malware. This technique is used to increase the stealth, making it harder to target down the main threat or to start processes in higher privileged modes. (cf. Sikorski & Honig, 2012)

Rootkits are y designed to especially remain stealthy making detection a cumbersome process. They are usually paired with different kind of malware to conceal their existence. (cf. Sikorski & Honig, 2012)

Any malware trying to scare a user into paying a ransom or buying something is labelled **Scareware**. They are often designed in a way to intimidate the user into making irrational decisions. Buy scaring the user with messages that their PC has been compromised, they attempt to make the user buy software to remove the seeming virus. In the end the bought software either does nothing or simply removes the scareware. (cf. Sikorski & Honig, 2012)

Spam-sending malware uses innocent systems to send spam to services. It is a way to generate income for bad actors by attempting to phish / scam people. (cf. Sikorski & Honig, 2012)

Worms and Viruses copy themselves to infect additional computers. The main difference is that a virus infects a host file to propagate itself while the worm propagates through a network on its own. (cf. Sikorski & Honig, 2012)

Ransomware is a modern version of Scareware. The difference is that ransomware utilizes encryption algorithms to encrypt entire drives. Afterwards a ransom is required to receive the private key necessary to decrypt the drive again. Without backups the encrypted data is forever gone, forcing a payment. However, as such malware is utilized by bad actors, there is no guarantee that the drive will be unlocked on payment. (cf. Grimes, 2019)

Trojan horse malware masquerade themselves to deceive users into thinking they are using legitimate software. They also must be executed by the victim to work. They are mostly distributed by tricking users into installing them, thus security software and measurements are only limitedly effective against this kind of threat. (cf. Grimes, 2019)

Adware is one of the most harmless categories which is used to display unwanted, potentially malicious advertising to the user. Most often browsers are infected, which can be used to redirect users onto sites with malicious ads or product promotions. (cf. Grimes, 2019)

Exploits use weaknesses in legitimate software to infect computers. They are mostly utilized by cyber criminals to infiltrate an organizations network. Upwards of 90% of reported data breaches are used in at least some steps within the attack chain. Multiple mitigation techniques such as ASLR and DEP have been developed to counteract exploits, however attacking techniques also develop further each year. (cf. SOPHOS, 2018)

Finally, **fileless malware** utilizes techniques so no persistent files must be used. The malicious code is directly injected into memory by either exploiting legitimate processes or loading malicious code directly into memory. In addition, legitimate tools already available on the system can be utilized for an attack. For persistence operating system objects, registries, APIs and scheduled tasks can be manipulated.
(cf. Grimes, 2019)

Overall, there are various categorizations of malware. As stated in the beginning of the thesis, fileless malware has been on the raise in recent years and is one of the top threats. For that reason, fileless malware is the specific focus of upcoming captures and is explained in depth in the next chapter.

2.2.2 Fileless Attacks / Malware

Characteristics

Fileless Malware uses legitimate processes, namely Living off the Land Binaries[15] and various built-in tools provided by operating systems to compromise a system. Neither do they download any malicious files nor is any content written to non-volatile memory. The attack vectors are vulnerabilities in applications which are used to inject the code directly into volatile memory, simple authentication brute forcing or phishing campaigns. The life cycle can be divided into three different steps. First, an attack vector which is used to initially target the victim is found. After that, the initial malicious attack alters system settings to achieve persistence or invoke instances of a shell. By creating a new instance of a shell / command line / etc. attackers open the ability of injecting malicious programs / code into legitimate process memory. No files are being downloaded to the file system. The entire attack only alters RAM making the attack harder to detect. Those characteristics make it unlikely for Anti-virus to detect such a threat. In addition, their nature of being designed to behave like a benign process leads to evasion of most behaviour-based detection mechanisms. The biggest difference between ordinary malware and fileless malware is the lack of methods to achieve persistence (although it is possible) and the lack of actual files / source code. However, both

[15] Living of the land binaries are typically legitimate admin tools being repurposed by an attacker
https://www.paloaltonetworks.com/cyberpedia/what-are-fileless-malware-attacks

traditional malware and fileless malware utilize advanced obfuscation to hide the intent from any analysts and avoid detection. (cf. Sudhakar & Kumar, 2020)

Attack lifecycle

The lifecycle can be divided into four different stages. In the first stage the attacker utilizes exploits, password brute forcing or social engineering to gain initial access to the victim's system. Two different strategies can be used to deploy the attack. Either the malicious files are directly downloaded into memory to be executed, avoiding signature-based detection or trusted / whitelisted applications are used so security software will not inspect the application. PowerShell[16] is a powerful tool on windows for fileless attacks as it supports functionalities for memory-based download and execution.
The second stage tries to achieve persistence as any memory within RAM will be erased after a restart. Naturally, such attacks are short-lived, however techniques exist for persistence. Malicious code can be stored in unusual locations associated with valid operating system files or common utilities. Those utilities include the Windows Registry[17], WMI[18] Store, SQL tables and Scheduled Tasks. Activities coming from those processes are generally seen as legitimate improving evasion.
Thirdly, windows internal tools such as PowerShell and Macro execution of office documents are used for stage execution. As most tools provide countless of privileges and functionality, fileless attacks offer a wide variety of possibilities.
Lastly, achieving the objective is the goal which can range from reconnaissance and credential harvesting to cyberespionage and damage.
(cf. Sanjay, Rakshith, & Akash, 2018)

Categories

Fileless Malware is divided into RAM-resident and Script-based. This is just a broad distinction, a more in-depth taxonomy especially on IoT focused Fileless Malware follows in the next subchapter. Executing exclusively in RAM avoids verification of digital signatures and malware signatures. By avoiding any writes to disk or changes to the system the malware avoids any trigger-based detection. Run-time solutions might be able to detect abnormalities, however usually evidence of an attack is only subtle and not unfailingly. False positives could lead to the shutdown of legitimate processes leading to

---

[16] PowerShell consists of a command-line shell and scripting language built on top of the .NET Common Language Runtime https://docs.microsoft.com/en-us/powershell/

[17] The Registry is a central hierarchical database storing information about configurations from users, applications and hardware devices https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page

[18] WMI is an infrastructure for management data and operations on Windows systems https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page

data loss and data damage. Background processes of applications are exploited to ensure longer persistence, as those keep running even after the application is closed. An in-depth review of memory injection techniques is presented in following sub-chapters. Scripts exploit existing systems and applications which can run them. In the windows landscape Visual Basic[19] Scripts were a popular tool for script-based malware. However, windows operating system entities limited the possibilities and made it difficult to compromise a system. A modern alternative is PowerShell, a powerful command line utility shell developed to help system administrators. Access to sensitive memory like registries, files, kernel configuration and even digital signature certificates is possible. (cf. Sanjay, Rakshith, & Akash, 2018)

In Linux IoT fileless attacks access is often gained through weak authentication. As IoT devices often lack appropriate patching and security measures, exploits can also open an entry gate. Afterwards the established connection makes it possible to use a shell with privileged access, which can be used to issue versatile commands like “**kill**” to shutdown monitoring services and query system parameters to gather information to build up more advanced attacks afterwards. (cf. Fan, et al., 2019)

IoT taxonomy

The researchers behind HoneyCloud have divided the taxonomy of fileless malware into eight different types based on their findings.
The first type is occupying end systems by altering password files, securing future access or preventing access from other parties.
The second type damages system data by removing or altering files or programs via the use of commands.
Thirdly, prevention of system monitoring/auditing services is ensured by killing watchdog processes / services or disabling firewalls / security software.
Type 4 encompasses any kinds of attempt to extract hardware or system information via commands. This intel can be used to develop further advanced attacks against the system.
The fifth type steals valuable information such as passwords or config files. Obtained password hashes are attempted to be cracked.
The next sixth type launches network attacks of different types to launch DoS[20] attacks. Other typical attacks such Heartbleed[21] and SQL Injection have also been abused.

---

[19] Visual Basic is a programming / scripting language to create .NET apps https://docs.microsoft.com/en-us/dotnet/visual-basic/

[20] DoS is short for Denial of Service which are attacks aiming to disrupt a service from being accessible / operating

[21] Heartbleed is a vulnerability in the OpenSSL cryptographic library which allowed attackers to steal protected information https://heartbleed.com/

Type 7 cannot be directly classified as they issue other different shell commands without any clear reasons for the motive.
The last type conducts attacks with no shell commands in use. A typical example is SSH Tunnelling[22] by port forwarding to conceal the original IP and make it seem like the attack was launched from the compromised device. (cf. Fan, et al., 2019)

Windows

The execution mechanism behind fileless attacks on windows relies mostly on invocation of a PowerShell instance using a WMI object with VBscript or Javascript. WMI stands for windows management instrumentation and is an excellent tool for reconnaissance, Anti-Virus / Virtual Machine detection, code execution, lateral movement, covert data storage and persistence. (cf. Sudhakar & Kumar, 2020)

Following fileless attacks utilizing WMI techniques are discussed based on Graeber's Blackhat paper. Some of the advantages of using WMI are that it is installed on all versions dating back to Windows 98, it is stealthier than psexec, permanent event subscriptions run as System, unawareness about WMIs opens capabilities to launch multi-purpose attacks, nearly every operating system action is capable of triggering WMI events and no payloads touch the disk.
WMI can be used for common reconnaissance by quiring the following objects:

Win32_OperatingSystem, Win32_ComputerSystem (OS information)
CIM_DataFile (File and directory listings)
Win32_Volume (Disk volume listings)
StdRegProv (Registry operations)
Win32_Process (Running processes)
Win32_Service (Service listing)
Win32_NtLogEvent (Event log)
Win32_LoggedOnUser (Logged on accounts)
Win32_share (Mounted shares)
Win32_QuickFixEngineering (Installed patches)

Table 1 WMI objects used for reconnaissance (cf. Graeber, 2015)

With the WQL Query
Select * FROM AntiVirusProduct
it is possible to gather information about installed Anti-Virus products, as those often register themselves in WMI via the **AntiVirusProduct** class, which is contained in either

[22] SSH tunnelling stands for Secure Shell tunnelling and is a method of transporting arbitrary networking data over an encrypted SSH connection https://www.ssh.com/ssh/tunneling/

root\SecurityCenter or root\SecurityCenter2. In PowerShell the command
Get-WmiObject -Namespace root\SecurityCenter2 -Class AntiVirusProduct
returns the necessary information. It will display the name, the path to the signed product exe, instanceGuid and various other meta data.
The next query is a generic way to evaluate if any sandbox environment is present. It assesses if the physical memory is limited to less than 2GB or if only a single processor core is available. Those conditions are unlikely to be true except for very outdated systems.
Select * FROM Win32_ComputerSystem WHERE TotalPhysicalMemory < 214748348
Select * FROM Win32_ComputerSystem WHERE NumberOfLogicalProcessors < 2

Additional Query to detect VMware:

```
SELECT * FROM Win32_NetworkAdapter WHERE Manufacturer LIKE
"%VMware%"SELECT * FROM Win32_BIOS WHERE SerialNumber LIKE
"%VMware%"SELECT * FROM Win32_Process WHERE Name="vmtoolsd.exe"
SELECT * FROM Win32_NetworkAdapter WHERE Name LIKE "%VMware%"
```

By using the **Win32_Process** it is possible to spawn a process locally or remotely, which is the equivalent of **psexec**. This PowerShell command would remotely spawn "notepad.exe" on a machine with the given IP address:

```
Invoke-WmiMethod -Class Win32_Process -Name Create – ArgumentList 'note-
pad.exe' -ComputerName 192.168.72.134 -Credential 'WIN-B85AAA7ST4U\Ad-
ministrator'
```

In addition, event consumers can lead to arbitrary remote code execution. It works by creating a permanent WMI event subscription. By choosing fitting routines the attacker opens new possibilities. For example, **"__IntervalTimerInstruction"** which would fire an event after a given time or **"Win32_ProcessStartTrace**" triggering upon the creation of a **"LogonUI.exe"** process signifying a user locking their screen.
Various attacks like the **"Push Attack"** can utilize remotely created WMI classes to store file data. In the next step PowerShell is used to access the file data and drop them to a remote file system. A detailed review of specific attacks is out of scope and can be read in the original paper used to describe this section. (cf. Graeber, 2015)

Wueest Candid and Anand Himanshu from Symantec describes in their report various fileless malware techniques. For example, Poweliks modifies the registry for persistence using embedded JavaScript code. The complete malware is contained within the registry and is later extracted on demand. A non-ASCII character is used as a name to

obfuscate and confuse tools. Certain access rights are modified to hinder the removal of the malware. The Powerliks registry run key consists of a call to **rundll32** with the following arguments:

rundll32.exe javascript: "\..\mshtml, RunHTMLApplication ";alert('payload');

Rundll32.exe uses **LoadLibrary** to load **mshtml.dll** after several tries to load other combinations of the arguments. **RunHTMLApplication** is started as entry point. This leads to a search of the JavaScript protocol handler. The first part is ignored as it is a string and the payload after the "," is executed. A script started this way can do the next step of the attack by decrypting another registry key. Powerliks uses PowerShell loading yet another DLL[23], which also is stored as an encrypted string within the registry.

By using Dual-use tools it is easier to mask the activities as legitimate because it is harder to distinguish them from malicious activities. Sometimes software cannot be blacklisted, and often commands are being used by legitimate administrators. A simple example is the use of the following commands:

```
net user /add [username] [password]
net localgroup administrators [username] /add
```

This snippet creates a new user and adds it to the group of administrators. It's hard to distinguish a legitimate from a malicious use, as this command can be used for valid reasons like an system admin creating a new admin user An extensive list of more commands and tools used by attackers can be found within the ISTR Special Report created by Wueest Candid and Anand Himanshu from Symantec cited in this section. (cf. Wueest Candid, 2017)

Linux

During exploitation various blind files are available for first information:

/etc/resolv.conf	Contains DNS, unlikely to trigger IDS
/etc/motd	Message of the Day
/etc/issue	Current version of distro

[23] DLL means Dynamic Link Library and can be loaded by more than one program and at runtime https://support.microsoft.com/en-us/help/815065/what-is-a-dll

/etc/passwd	List of local users
/etc/shadow	List of users' passwords' hashes
/home/xxx/.bash_history	Directory context

Table 2 Linux blind files post exploitation (Adrian, 2019)

Various files and commands also exist to delete history and check for individual distributions, system information, installed package lists, networking information, configuration files, accounts, credentials, important file, etc. Furthermore, certain commands are useful for creating reverse shells, check suid 0 related things and to cover tracks. Those can be seen in the Blackhat presentation of Adrian Hendrick. (cf. Adrian, 2019)

Useful frameworks for automation are Metasploit[24], Cobalt Strike[25] and various open source frameworks and tools. An example of a powerful Meterpreter command is the module **"checkvm",** which can detect virtual machines. (cf. Adrian, 2019)

Evasion

Documents can function as containers embedding malicious files which are written in languages like JavaScript. Social engineering can be used to make the victim click on the malicious embedded file. This is an intended design choice making it harder to detect if an embedded script is malicious or valid. Often documents also support scripting capabilities which opens another attack vector for bad actors. Software launched by such malicious files or scripts can be directly executed in memory resulting in fileless infection.

Compared to compiled malicious files, scripts are harder to detect for anti-malware vendors. Due to the high flexible nature it is possible to split malicious logic across several processes decreasing the likelihood of behaviour-based detection. Still, they can just as ordinary malware be obfuscated to slow down any potential analysts. Obfuscation can also possibly lead to evasion of detection engines. Bohannon Daniel created a useful tool compassing most obfuscation techniques for PowerShell commands

[24] Metasploit is a framework for penetration and security testing available at https://github.com/rapid7/metasploit-framework

[25] Cobalt Strike is software for Adversary Simulations and Red Team Operations https://www.cobaltstrike.com/

and scripts[26]. Windows includes multiple script interpreters for PowerShell, VBScript and JavaScript making it a reliable attack vector. (cf. Sanjay, Rakshith, & Akash, 2018)

Built in windows tool are used by adversaries to make it difficult to differentiate between legitimate and malicious use. For a comprehensive listing of tools and built-in binaries, libraries and scripts see Oddvar Moe's LOLBAS project[27].

Injecting malicious code directly into RAM hinders common endpoint security solutions, as their main strength are files existing on the drive. This gives the malware the opportunity to remain in hidden spots and change its shape.
(cf. Sanjay, Rakshith, & Akash, 2018)
Various injection techniques are described in the following chapter.

Detection

Sandboxing is an approach to monitor API calls wrapped by the sandbox and blocking all susceptible and dangerous calls. Another approach is using the open source code of PowerShell for emulation. This way it is possible to verify and deobfuscate scripts before they are run on the actual host. While it is not entirely reliable, the amount of malicious attacks can be scoped down by heuristics. Scripts / attacks utilizing rather ubiquitous calls can be filtered out and be deemed suspicious. An especially important factor is the starting point of the PowerShell process. Documents and browsers are very suspicious candidates likely indicating malicious intent. A combination of heuristics, behaviour-based detection and memory scanning can detect fileless threats reliably. (cf. Sanjay, Rakshith, & Akash, 2018)
As existing solutions have only limited capabilities to detect such threats, honeypots are a perfect tool to detect and analyse fileless attacks. Honeypots can be fully isolated systems that are rarely accessed. As a result, any kind of activity can be instantly deemed suspicious and it is easier to monitor changes. Almost any kind of activity can be deemed to be evil and thus complicated algorithms and heuristics deciding the likelihood of a true positive can be omitted, simplifying the system (cf. Nawrocki, Wählisch, Schmidt, Keil, & Schönfelder, 2016). Processes and memory can be scanned for any changes during their runtime pointing towards injections. Environment variables, registries and similar configuration entities should also mostly remain constant unless an attacker actively modifies them. All shell and command line commands can be intercepted as any potential ones must be coming from a malicious actor. The most important step is monitoring all potential attack vectors rather than deciding the

[26] https://github.com/danielbohannon/Invoke-Obfuscation

[27] LOLBAS is a collection of Living Off The Land Binaries and Scripts (and also Libraries)
https://github.com/api0cradle/LOLBAS available on 30.08.2020

likelihood of an attack. IDS[28] can further help detect abnormalities within the network packages like outgoing DoS attacks.

2.2.3 Advanced Persistent Threats

APTs are usually groups of advanced attackers who are well-funded by organizations or governments. Usually the time the intent is to gain information about an adversary. Advanced means that the attackers are well founded, giving them access to advanced tools and methods. This leads to more attack vectors and the exploitation of more secure systems. They are also highly persistent and determined. Attacks are conducted over long periods, trying to occupy systems as long as possible. Several evasive techniques to elude detection systems and intrusion detection are utilized. The approach is “low and slow”, taking a lot of caution to increase the chance of success. Threat stems from possible damage which could be the loss of sensitive data or loss or impediment of critical components or missions. Many organizations and nations see this as rising threats. The 5 stages that represent almost any APT are Reconnaissance, Establish Foothold, Lateral Movement, Exfiltration, Post-Exfiltration. First data is gathered to better understand the target, increasing the chance of success. Next, the attacker enters the targets system / computer network. By this stage a foothold is established within the victim’s network. After that, they laterally move within the network trying to evade security / detection systems while searching for sensitive / critical data or their mission accomplishment. During Exfiltration attackers send the stolen information to their command and control centre. Destroying or disabling critical components can also be part of this stage. In the final stage exfiltration is continued and further critical components are disabled / destroyed. Importantly the attackers will try to delete any evidence for a clean exit. (cf. Alshamrani, Myneni, Chowdhary, & Huang, 2019)

2.3 Post Exploitation utilizing Injection Techniques

Process / Memory Injection is an important tool of fileless attacks, as they directly inject the malicious code into already running and possibly trusted processes or memory leaving no footprint on the drive. An overview of the most common techniques is given in this chapter. Hosseini Ashkans blogpost about the ten most common and trending injection techniques is used as a basis for the first four injection techniques in this chapter (Hosseini, 2018).

[28] IDS are systems monitoring the network according to rules or utilizing AI to detect evil activity

2.3.1 Process Hollowing

This technique is named after the functionality of the injection in which the malware unmaps (hollows out) the legitimate code from memory to overwrite the memory space of the target process with a malicious executable. On Windows this is done by calling **CreateProcess** and setting the Process Creation Flag to **CREATE_SUSPENDED**. Till the **ResumeThread** function is called the new process will remain in a suspended state. **ZwUnmapViewOfSection** or **NtUnmapViewOfSection** unmaps the memory of the process pointed to by a section. Next **VirtualAllocEx** allocates new memory, which can now be used by **WriteProcessMemory** for writing the individual malware sections into memory. **SetThreadContext** changes the entry point, which is necessary as the contents have changed. Lastly, **ResumeThread** resumes the process out of the suspended state. (cf. Hosseini, 2018)

2.3.2 Injection and Persistence via Registry Modifications

Useful registry keys for malware which are both used for persistence and injection:

HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_Dlls

HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_Dlls

HKLM\System\CurrentControlSet\Control\Session Manager\AppCertDlls

HKLM\Software\Microsoft\Windows NT\currentversion\image file execution options

The libraries in the **AppInit_Dlls** key are loaded into each process loading **User32.dll**. As it is a common library most processes will load the malicious library. It can be used simply by calling **RegCreateKeyEx** and modifying the value by calling **RegSetValueEx.** (cf. Hosseini, 2018)

AppCertDlls injection approach is almost the same, the only difference is that the registry keys are loaded upon a process calling the Win 32 API functions **CreateProcess**, **CreateProcessAsUser**, **CreateProcessWithLogonW**, **CreateProcessWithTokenW** and **WinExec**. (cf. Hosseini, 2018)

2.3.3 Hook Injection via SetWindowsHookEX

SetWindowsHookEx is called to install a hook routine. Four arguments are necessary for the call, the first being the event which describes the hook type. The second argument points to a function which will be invoked upon the event triggering. As third argument the module containing the function itself is used. LoadLibrary and GetProcAddress are usually called to achieve this. As last argument the thread associated with the hook procedure is given. CreateToolhelp32Snapshot and Thread32Next can be used to target a single thread for less noise. If the last argument is 0, then all threads perform the procedure, which is not optimal to remain undiscovered.
(cf. Hosseini, 2018)

2.3.4 IAT Hooking and Inline Hooking

This technique is used to change the import address table, so when a valid application calls an API located in the specific DLL, the malicious function is executed instead of the original one. In difference to inline hooking (which is omitted due to lower effectivity) this technique modifies the API function directly. FinFisher[29] achieved this by modifying **CreateWindowEx** pointers. (cf. Hosseini, 2018)

2.3.5 Anonymous File Creation through memfd_create()

“**memfd_create()** creates an anonymous file and returns a file descriptor that refers to it. The file behaves like a regular file, and so can be modified, truncated, memory-mapped, and so on. However, unlike a regular file, it lives in RAM and has a volatile backing storage.” (Kerrisk, memfd_create(2) - linux man-pages, 2020)

Stuart, a professional “red teamer” explains such an attack in his blog. The so created anonymous memory-only file is only visible in the filesystem as symlink in **\prco\<PID>\fd**, which can be used by **execve()**. Perl’s **syscall()** can be used to avoid dropping any binaries. The binary of the code to be injected into memory is contained within the script. Perl’s **syscall()** lacks access to libc, but the necessary raw system call number for **memfd_create()** and numeric constant for **MEMFD_CLOEXEC** can be both found in the header files in **/usr/include**. The final Perl script would look like this:

[29] FinFisher also called FinSpy is legally sold spyware for Android, iOS, Windows, macOS and Linux https://www.kaspersky.com/blog/finspy-commercial-spyware/27606/

```
My $name = "",
my $fd = syscall(319, $name, 1);
if(-1 == $fd) {
   die "memfd_create: $!";
}
```

319 is the system call number on a x64 Linux operating system and 1 represents the **MFD_CLOEXEC** constant. After executing this script, a file descriptor to the anonymous RAM file is saved in **$fd**.
In the next step the anonymous file needs to be filled with actual ELF[30] data. The following Perl script can be used to open a file:

```
open(my $FH, '>&=' .$fd) or die "open: $!";
select((select($FH), $|=1)[0]);
```

This turn the already-open file descriptor into a file handler. Now chunks of binary data can be written to the anonymous file by using a command like:

```
perl -e '$/=\32;print"print \$FH pack q/H*/, q/".(unpack"H*")."/\ or die
qq/write: \$!/;\n"while(<>)' ./elfbinary
```

This Perl print statement creates multiple prints that each write binary data into the file.

Finally **exec()** in perl is called, which is very similar to the native **execve()** system call:

```
exec {"/proc/$$/fd/$fd"} "name", "-kvl", "4444", "-e", "/bin/sh" or die "exec:
$!";
```

The first parameter is the file being passed as a string (**$$** returns the pid of the Perl process and **$fd** contains the file descriptor). The other parameters are parameters that are passed to the binary which was injected. The result is an anonymous file running in RAM without touching the disk. The only differences to a normal file is the **/proc/<PID>/exe** symlink and that **memfd_create()** calls are sticking out during "**strace -f**" system call monitoring. (cf. Stuart, 2020)

FireELF[31] is a useful framework which utilizes this injection technique.

2.3.6 Reflective DLL injection

Reflective DLL injection creates a DLL that maps itself into memory without using the windows loader. Import addresses, relocations and **DllMain** entry point are all handled by a component within the DLL. This technique leverages large RWX memory sections that are easy to detect. The overall technique works very similar to shellcode injection,

[30] ELF means Executable and Linkable Format and is the standard file format for executable files on linux

[31] FireELF can be downloaded at https://github.com/rek7/fireELF

which uses **OpenProcess** to allocate memory within the opened process by using **VirtualAllocEx**. Next **WriteProcessMemory** is used to write the DLL to memory and finally execute by calling **CreateRemoteThread**. (cf. Desimone, 2019)

2.3.7 Ptrace Injection

Ptrace is exposed by the kernel to offer debuggers the ability to interfere with running processes. The following table sums up several useful operations possible:

PTRACE_ATTACH	Allows one process to attach itself to another for debugging, pausing the remote process
PTRACE_PEEKTEXT	Allows the reading of memory from another process' address space
PTRACE_POKETEXT	Allows the writing of memory to another process address' space
PTRACE_GETREGS	Read the current set of processor registers from a process
PTRACE_SETREGS	Writes to the current set of processor registers of a process
PTRACE_CONT	Resumes the execution of an attached process

Table 3 Useful ptrace operations (Chester, 2017)

By calling **ptrace(PTRACE_ATTACH, pid, NULL, NULL)** with the specific PID of the process targeted ptrace is going to get attached. A **SIGSTOP** is sent resulting in the pausing of the execution. To ensure a clear resuming afterwards, the current state of the processor registers is saved using **PTRACE_GETREGS** and saving them into a **user_regs_struct** struct. Next a place which accepts write operations for future code injection needs to be found, "**/proc/PID/maps**" can be parsed to display sections. The data also needs to be backed up and this can be done using **PTRACE_PEEKTEXT**. Overwriting is achieved by using **PTRACE_POKETEXT**. Both peektext and poketext deal with 1 word of data at a time, thus requiring multiple calls to achieve the goal. Once done,

the processes instruction pointer registers need to be pointed towards the injected code. The earlier saved registers are next written back using **PTRACE_SETREGS**. Finally, **PTRACE_CONT** resumes execution. For a more thorough explanation visit the cited blog post. (cf. Chester, 2017)

2.3.8 Doppelgänging Injection

First a transaction is opened with **hTransaction = CreateTransaction()**. A clean file is opened transacted by calling **CreateFileTransacted("svchost.exe", GENERIC_WRITE | GENERIC_READ, ..., hTransaction, ...)** and the returned value saved in **hTransactedFile**. Next the file is overwritten with malicious code by calling **WriteFile(hTransactedFile, MALICIOUS_EXE_BUFFER, ...)**.
NTCreateSection(&hSection, ..., PAGE_READONLY, SEC_IMAGE, hTransactedFile) creates a section from the transacted file which points to the malicious executable. **RollbackTransaction(hTransaction)** removes any changes from the file system. Now **NtCreateProcessEx(&hProcess, ..., hSection, ...)** and **NtCreateThreadEx(&hThread, ..., hProcess, MALICIOUS_EXE_ENTRYPOINT, ...)** are used to create a thread object. Process Parameters are created by calling **RtlCreateProcessParametersEx(&ProcessParams, ...)**. Now the created process parameters are copied to the newly created process's address space by first using:
VirtualAllocEx(hProcess, &RemoteProcessParams, ..., PAGE_READWRITE), then **WriteProcessMemory(hProcess, RemoteProcessParams, ProcessParams, ...)** and finally **WriteProcessMemory(hProcess, RemotePeb.ProcessParamters, &RemoteProcessParams, ...)**. Lastly, **NtResumeThread(hThread, ...)** starts the execution of the injected process.
This Injection technique is very resistant against even advanced forensic tools and advanced versions can avoid detection from anti-virus engines while simultaneously working on all versions since Vista. No "unmapped code" is left and the entire process is entirely fileless. (cf. Liberman & Kogan, 2017)

2.4 Forensic Investigation / Detection of Fileless Attacks

The most popular tool to investigate memory related incidents is Volatility[32]. It's an open source memory analysis tool. Multiple operating systems such Windows, Linux, Mac and Android are supported. Virtual Machine images can be inspected as well as raw dumps, crash dumps and others.

[32] Volatility is a memory forensics framework and can be downloaded at https://www.volatilityfoundation.org/releases

Windows

Buddy Tancio SANS paper that shows the investigation process of fileless malware is used in this chapter to describe the used techniques to unveil the malicious code. The first fileless ransomware SOREBRECT[33] utilizes process hollowing. First **svchost.exe** revealed to lack a parent process. Upon further investigation with Volatility it became apparent that the parent process has already been unlinked. The **malfind** plugin detected an irregularity as the **PAGE_EXECUTE_READWRITE** flag was set, which allows malware to inject and overwrite process memory. The **shimcache** plugin can detect post execution artifacts, showing the execution of **PSEXESVC.exe**. Finally, the process was dumped so it could be further investigated by a reverse engineer. (cf. Buddy, 2019)

Another malware investigated is the Dridex Banking Trojan[34]. This time **winver.exe** process was injected with malicious code. Upon investigating the **pstree** with Volatility it shows that the parent process **drd4.exe** has exited right after spawning **winver.exe**. In addition, **malfind** reveals once again a **PAGE_EXECUTE_READWRITE** flag. The plugins **ldrmodules** an **hollowfind** were ineffective in both variants, probably due to new API hooking techniques. (cf. Buddy, 2019)

Later on, a Meterpreter shell injection attack is analysed as Meterpreter uses reflective dll injection. The **netscan** Volatility command shows an injection into **spoolsv.exe** to port 4444, which is the default port number for Meterpreter shells. **Malfind** once again shows the **PAGE_EXECUTE_READWRITE** flag being set. This showcases that volatility is an excellent tool to monitor changes within memory to detect injections. **Malfind** is an immensely useful tool, as a process can only be injected to if the right memory flags are set, which are revealed by this plugin. As it is compatible with different operating systems, it can also be used to detect process injections in Linux although those naturally differ in their execution. (cf. Buddy, 2019)

To detect persistence methods of fileless malware multiple possible vectors must be searched. The Windows registry hive is the first important location. The registry hives can be found in the following locations:

---

[33] SOREBRECT is a fileless and code-injecting ransomware https://blog.trendmicro.com/trendlabs-security-intelligence/analyzing-fileless-code-injecting-sorebrect-ransomware/

[34] The Dridex Banking Trojan caused millions of dollars’ worth of damage and continues to adapt and attack successfully https://www.kaspersky.com/about/press-releases/2017_the-dridex-banking-trojan-an-ever-evolving-threat

```
C:\Windows\system32\config\system
C:\Windows\system32\config\<name>
C:\Windows\system32\config\security
C:\Windows\system32\config\software
C:\Users\UserName\NTUSER.dat
C:\User\UserName\AppData\Local\Microsoft\Windows\UsrClass.dat
```

The Kovter[35] trojan for example issued the registry entries to achieve persistence: **HKEY_CURRENT_USER\Software\Classes\{random key}\shell\open\command** which contains the first malicious script, **HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run** containing the second script and **HKEY_CURRENT_USER\Software\Classes\.{random extension}** containing the third script. The registry can be analysed with registry explorer[36] or Volatility.

regsvr32, http, javascript, mshta, rundll32.exe

Those key words are a good indicator of fileless compromise. (cf. Buddy, 2019)

Next the WMI is an important toolset of fileless attacks on windows as described earlier. The following WMI objects are important:

__EventFilter	Holds conditions to trigger event
__EventConsumer	Containts persistence payload and instructions to execute malicious script. CommandLineEventConsumer and ActiveScriptEventConsumer hold executable scripts
__FilterToConsumerBinding	Connects classes and instances together

Table 4 Useful WMI Objects for forensic investigation (cf. Buddy, 2019)

PowerShell is a powerful tool investigate WMI entries. The WMI database is stored in

[35] Kovter is a constantly evolving malware that has evolved multiple times throughout the years https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/kovter-an-evolving-malware-gone-fileless

[36] Registry Explorer is a free tool intended to replace regedit https://ericzimmerman.github.io/#!index.md

%System%\System32\webm\Repository

In the example a cryptocurrency mining malware is investigated and the following command is issued:

Get-WMIObject -Namespace root\Subscription -Class __EventFilter

This command queries **EventFilter**, which reveals the malicious entry Windows Event Filter. The following Query is stored within the malicious entry using a specified time interval to trigger the event:

SELECT * FROM __InstanceModificationEvent WITHIN 5600 WHERE Target-Instance ISA 'Win32_PerfFormattedData_PerfOS_System'

The malicious script is in a **CommandLineEventConsumer** object in **ROOT\subscription** namespace. This persistence payload is executed whenever the condition is met.

Get -WMIObject -Namespace root\Subscription -Class __EventConsumer

This query reveals the PowerShell script installing the crypto-miner. The malware uses the **__FilterToConsumerBinding** object to associate **__EventFilter** with **__EventConsumer** classes. This step is needed to ensure that **__EventConsumer** is executed upon the condition of **__EventFilter** being met.
Lastly, this query shows which objects are linked in the **__RELPATH** section.

Get- WMIObject -Namespace root\Subscription -Class __FilterToConsumerBinding

(cf. Buddy, 2019)

Sysmon[37] is a tool to monitor events happening within a windows system. It displays system process chains, network activity and file modification / creation time. In a blog-post Perez Carlos explains how WMI can be monitored in Sysmon (cf. Perez, 2017).

Windows Task Scheduler located in **C:\Windows\System32\Tasks** is another useful tool, as it shows the corresponding triggers to each task and displays which action is then executed. In the case of the crypto-miner malware **WindowsLogTasks** and **System Log Security Check** both have triggers set that respawn a malicious PowerShell script using **regsvr32**. Windows Event Logs is yet another tool which can potentially log PowerShell scripts and WMI logs. (cf. Buddy, 2019)

[37] Sysmon can be downloaded at https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon

Shimcashe, Muicache, userassists and **prefetech** contain execution artifacts that can indicate fileless malware. Some useful keywords are **powershell.exe, wscript.exe, cscript.exe, scrons.exe, cmd.exe, regsvr32.exe, mshta.exe.** (cf. Buddy, 2019)

Linux

Memfd_create() attacks can be detected with command line forensics.

```
ls -alR /proc/*/exe 2> /dev/null | grep memfd:.*\(deleted\)
```

The following command searches for all running processes within the **/proc** directory and checks for the form **memfd: (deleted).** In a normal running system this command should not return any results. Anything returned is highly suspicious and should be investigated.

Netstat can be used afterwards to check for suspicious ports. By using the command **ps**, the suspicious process should show up somewhere in the list which matches the PID. By using **cd/proc/<PID>** the directory can be explored. A simple **ls -al** shows information like data stamps when the process was started, current working directory where the user started the process and exe link points which usually points to non-existing binary locations. Cat and strings can be used to check the contents of cmdline. If the binary name doesn't match or is oddly weird then it could indicate malware / an attack. Cat maps shows the binary names and other library files it is using while running. A reference to the actual binary should be contained within the first part. A reference to **/memfd: (deleted)** is suspicious. By checking **environ** with strings, artefacts of the process can be found as most user started processes leave information in this extensive list of environment variables. (cf. Sandfly Security, 2020)

2.5 Honeypots

Honeypots are decoy systems which solely exist to be probed, attacked or compromised. They complement other solutions such as IDS and dynamic firewalls. The goal is to spot zero-day attacks / techniques and get insight on attackers' actions as well as their motivations. (cf. Nawrocki, Wählisch, Schmidt, Keil, & Schönfelder, 2016)

Honeypots can be categorized in different interaction levels. Low interaction honeypots lack the ability to launch attacks to an external system. This limits the possible damage but makes it also easier to detect. Medium interaction level honeypots implement a limited amount of services. A certain degree of intelligence is built into them to avoid immediate detection. High interaction honeypots have a full operating system and an entire range of functionality available. Such systems can be used to launch further attacks and pose a possible security threat.

The following table signifies the different qualities of the interaction levels

	Low interaction honeypots	Medium interaction honeypots	High interaction honeypots
Real operating System	No	No	Yes
Risk of compromise	Low	Mid	High
Wish of compromise	No	No	Yes
Information gathering	Low	Mid	High
Knowledge to deploy	Low	Low	High
Knowledge to develop	Low	High	High
Maintenance time	Low	Low	Very high

Table 5 characteristics of honeypot interaction levels (Nawrocki, Wählisch, Schmidt, Keil, & Schönfelder, 2016)

Furthermore, a categorization into three deployment modes is present. Deception mode are used to attract hackers, being designed in a way to force attackers to use a big repertoire of their weapons / tools. Responses are manipulated in a way to deceive the hacker into thinking they are coming from a legitimate system. Intimidation mode honeypots are strictly secured, and the attacker might even get a message warning about the monitoring. The goal of this is to scare away novice hackers to mostly detect techniques of experienced hackers. Such information is more valuable than data from average attacks. Lastly, the reconnaissance mode is specifically used to capture and record new attacks on a system. The tools and techniques by the attacker are to be determined and later used to for example implement heuristic-based rules for IDS systems. Both internal and external attacks are monitored. The main difference between intimidation and reconnaissance mode is that the intimidation mode tries to keep the attacker on the system as long as possible by feeding back engaging information.

Finally, the Deployment category splits honeypots into research and production honeypots. Research honeypots are used to gather intel on possible new attacks to develop new security measures, while production honeypots are mostly used to detect attacks and protect the network from those. (Campbell, Padayachee, & Masombuka, 2015)

As honeypots are systems that are not really connected to any meaningful procedures, the collected data is not polluted and thus valuable for investigation and analysis. As relatively isolated systems they don't have to stem the workload from common production systems. Catching zero-day-exploits and unknown strategies is a huge benefit, as they monitor any activity coming from an attacker. For that reason honeypots also have reduced false positives and negatives. Activity on server-honeypots is an anomaly. Also, honeypots are very flexible concepts, as various versions exist to tackle specific tasks, reducing redundant load. However, Server-honeypots suffer from a limited field of view, as they can only detect attacks if attackers actively send packages to them. If no anti-fingerprinting measures are in place, honeypots can be detected by attackers / malware rendering them useless. Lastly, a compromise of a honeypot brings a certain risk to the environment making it important to put up the necessary security measures to protect the rest of the network. (cf. Nawrocki, Wählisch, Schmidt, Keil, & Schönfelder, 2016)

Honeypot detection can be avoided by certain approaches. First an automatic honeypot redeployment of low interaction honeypots upon a certain criterion (for example a drop of ICMP packets below a certain threshold or timers). Another important improvement is the reduction of delay introduced by the honeypot systems. By differentiating the delay of execution and communication an attacker can detect a honeypot. Either optimization or adjusting code / settings this can be avoided. Next a lack of transparency can also lead to detection. Hybrid honeypot systems often reveal their deceptive design due to a lack of transparency. For example, if a honeypot frontend redirects traffic into a honeypot backend, certain measures must be taken. In the case of TCP sessions, a TCP replaying approach can be used to transfer the session from frontend to backend. This makes the honeypot much stealthier although slowing down performance. In general honeypots need to hide modified sequences of events that are not realistic in a way that makes it hard for the attacker to deduce their existence. An easy but costly approach is using dedicated hardware. First, this reduces software delays naturally, as an attacker is interacting with the real version of the system. Second, they provide more security compared to other honeypot variants. Finally, dynamic intelligence changing honeypots utilizing machine learning and artificial intelligence is a possibility. The main assets are a very adaptable honeypot that is hard to detect. Reconfiguration is not performed, rather the system dynamically changes based on reinforced learning. (cf. Nawrocki, Wählisch, Schmidt, Keil, & Schönfelder, 2016)

An important aspect of honeypots is to increase the likelihood of deceiving an attacker and certain modification can increase that chance. Attackers or malware often attempt

to fingerprint systems in order to identify honeypots. Artificially crafted packages can be sent to various ports to analyse the response which can contain operating system data and alike information. TCP[38]/IP protocol are often utilized for this. Because every Operating System replies differently, such information can be used to deduce significant information. The first possible solution is to identify abnormalities in the TCP options field which is utilized by most fingerprinting tools. The field is used during the SYN and SYN/ACK handshake process. The following attributes should be monitored for abnormalities: Maximum Segment Size (MSS), Window Scaling, Selective Acknowledgements, Timestamps and Nop.

Furthermore, TCP flags are often utilized for fingerprinting.

The first version of such an attack is FIN Probing. FIN indicates the end of a connection. By probing with a FIN packet, the attacker can find out if a port is closed or not. As opened ports will ignore a single FIN packet without prior established connections and an RST packet will be sent back. This technique is also fairly evasive dodging most firewalls, IDS and packet filters.

The second version utilizes FIN/SYN packets. Those flags are mutually exclusive and should never occur in an ordinary connection. Linux replies to such a combination with a FIN/SYS/ACK packet.

Thirdly, URG/PSH/FIN probing can detect closed ports as those usually return an RST/ACK packet. It should be noted that only systems conforming to RFC 793 are vulnerable

The fourth attack uses NULL packets, which sends packets without any flags set. If the port is closed an RST packet will be sent back.

Reserved Bit Probing is the fifth version using the 3 reserved bits of the TCP header.

Sixth, ECN-Echo Probing utilizes the explicit congestion notification which is an extension to TCP packets.

UDP[39] ports will send an ICMP[40] error if they are closed such as “Destination Unreachable”. Setting the DF bit can also cause an ICMP error response. Packet size, checksum and payload should be monitored for abnormalities.

ICMP Echo Requests, ICMP Router Solicitation Requests and ICMP Timestamp requests are all used in fingerprinting attacks. Certain fields have predetermined values such as Code 0 in ICMP Echo Requests. Deviations indicate malformed packages for scanning purposes. Such abnormalities must be monitored for all different ICMP types. Furthermore, Packet sizes are predictable depending on the operating system, thus making it

---

[38] TCP is short for Transmission Control Protocol and is used for highly reliable host-to-host connections between packet-switched communication networks https://tools.ietf.org/html/rfc793

[39] UDP is short for User Datagram Protocol and is transaction oriented making delivery and duplicate protection not guaranteed https://tools.ietf.org/html/rfc768

[40] ICMP is a protocol that is an integral part of IP typically reporting errors in the processing of datagrams https://tools.ietf.org/html/rfc792

possible to derive an attack from packet size abnormalities.
The paper suggests five different prediction severity levels with the following attributes: Lower >= 1 abnormalities, Moderate >= 5 abnormalities, Higher >= 10 abnormalities, Serious >= 15 abnormalities and Imminent >= 20 abnormalities. Depending on the needs a different level can be chosen for a system defending against fingerprinting attacks. (cf. Naik & Jenkins, 2018)

By modifying the responses to behave in a certain way an attack can be deceived by the fingerprinting results. This is important so an attacker does not immediately dismiss a simulated honeypot as reconnaissance is usually one of the first steps involved in an elaborate attack.

Important programs that monitor the honeypot can be coupled with rootkit capabilities to stop attackers from not realizing that their activities are being scanned. Developing fully functional rootkits is highly complicated, however as full control is guaranteed it simplifies the whole process as the rootkit does not have to bypass security measures and can instead just be whitelisted or deployed while disabling security mechanisms. Certain rootkit techniques can be adopted in a honeypot system to evade detection from an attacker. Processes that monitor activity can utilize process hiding to avoid being listed by system utilities like "**ps**". This technique works because the kernels task scheduler and process accounting utilities consult different process lists. Process descriptors of all running tasks are contained in the linked list "**all-tasks**". It is represented by the data structure **init_tasks->next_task**. Meanwhile the scheduler uses the linked list "**run-list**", which is represented in the kernel by **run_queue_head->next** structure. By removing the process descriptor from all-tasks, a process won't be shown by tools utilizing this specific list. Because the kernel uses a different linked list the process will just transparently continue running without any issues.
(cf. Baliga, Ganapathy, & Iftode, 2011)

By deploying vulnerable applications / operating systems or creating accounts / users with weak credentials attackers can be attracted. However, depending on the use-case it might be necessary to only deploy vulnerabilities that are complex to exploit. This ensures that captured attacks are of high quality coming from experienced attackers rather than collecting a massive amount of low-profile malware data.

2.6 Intrusion Detection Systems

Intrusion detection systems aim to defend a system by issuing alarm when certain conditions are met that indicate an attack. They are generally first divided into network based (NIDS) and host-based systems (HIDS). Modern IDS can be categorized into anomaly based, signature based and compound detectors. Anomaly based IDS check the traffic for any suspicious abnormalities. By comparing the traffic to what is deemed normal and deciding how much it derivates from that, the software can decide if an

attack is dangerous. Those can either be self-learning systems powered by techniques such as the hidden Markov model, stochastic models, various other machine learning algorithms, AI or programmed by a user who teaches the system to detect anomalous events. Signature detection is based upon signatures, which must be created by a user. They define what constitutes legal and illegal behaviour. Different approaches like state-modelling encompassing different states that must be present during an intrusion, expert systems which reason about the state of a system according to given rules, string matching and simple rule based systems are in use. Compound detectors operate by checking an intrusion against the background of normal traffic in the system. This enables them to be more accurate and mostly capture the truly interesting events, since they know the patterns of intrusive behaviour and can check them against the normal behaviour of the system. (cf. Axelsson, 2000)

A popular signature-based IDS is Snort[41]. The usage is explained using the officially provided Snort cheat sheet. It works by creating rules specifying certain actions upon finding a packet that matches the rule criteria. The general syntax is:

[action] [protocol] [sourceIP] [sourceport] - > [destIP] [destport]
([Rule options])

Rule options are the key to snort's intrusion detection engine being both flexible and powerful. General rule options are:

Message outputs a simple text string upon a rule match.
Flow is used in conjunction with TCP stream reassembly and narrows down the direction of flow that fires up the rule.
Reference can include external sources of information .
Classtype displays what the effect of a successful attack would be.
Sid/rev is a unique identifier helping plugins to identify rules easily.

Furthermore following detection options exist:

Content sets rules searching for specific content in the packet. Within this option **distance/offset** specifies the relative beginning and **within/depth** specifies how far forward it will search relative to the end of a previous content match.
PCRE allows perl compatible regular expressions for more complex content matching.
Byte_test compares a number of bytes against specific values in binary.

More information can be found on the official snort homepage[37] or on the cheat sheet that is used to describe this section. (cf. Snort, 2016)

---

[41] Snort is a free open source signature based intrusion detection and prevention system capable of real-time traffic analysis and packet logging https://www.snort.org/

3 Methodology

In this chapter a concept is created step by step. The concept consists of three different layers that encompass different sub systems that interact with each other. The main idea is to have multiple honeypots that are aware of each other and possibly lead to a multi staged compromise from an attacker. The first layer encompasses an easy entry point for an attacker from an IoT device while the second layer consists of SCADA workstation honeypots and the possibility to attach any further honeypots. To interfere the network traffic an IDS is used. Due to the nature of the whole system being isolated and thus normally not accessed it is easy to monitor incoming and outcoming traffic (cf. Nawrocki, Wählisch, Schmidt, Keil, & Schönfelder, 2016). The last and third layer is secured behind firewalls being the central point of the whole system. The central control server that evaluates connectivity, can restart honeypots and receives any malicious detected activity is the core of the third layer. Following each layer is described and established. Finally, the concept is visualized in a computer network diagram. Due to the high complexity of the system and hardware constraints only the first layer of the system is implemented afterwards.

3.1 Layer 1

The first Layer is supposed to be an easy to exploit entry point within the network. It is important to note that the honeypot is connected to the production network and not directly to the internet, as the goal is to monitor and capture potential advanced attacks which already managed to infiltrate the network in some capacity. This gives a good balance between being easy to infiltrate and only attracting advanced attackers. A Raspberry Pi is used for this first layer IoT honeypot. There are multiple different potential operating systems that could be deployed, however due to the compatibility of the later used tools and techniques Ubuntu Server[42] is chosen. Ubuntu Server offers support for Raspberry Pi 2, Raspberry Pi 3 and Raspberry Pi 4 with long term support guaranteed, which is another important factor considering that such a system would be deployed for long periods of time. Furthermore, as this OS image is specifically adapted to the Raspberry Pi there is no need for concern regarding performance issues. The Raspberry Pi honeypot consists of multiple modules that scan different aspects of the system for any malicious activity. The Raspberry Pi are booted over USB Sticks, which are cheap and disposable in case of serious

[42] Ubuntu Server is a Raspberry Pi compatible Linux distribution https://ubuntu.com/download/raspberry-pi

compromise. Figure 2 visualizes the concept of the honeypot and the four different modules that monitor the activity.

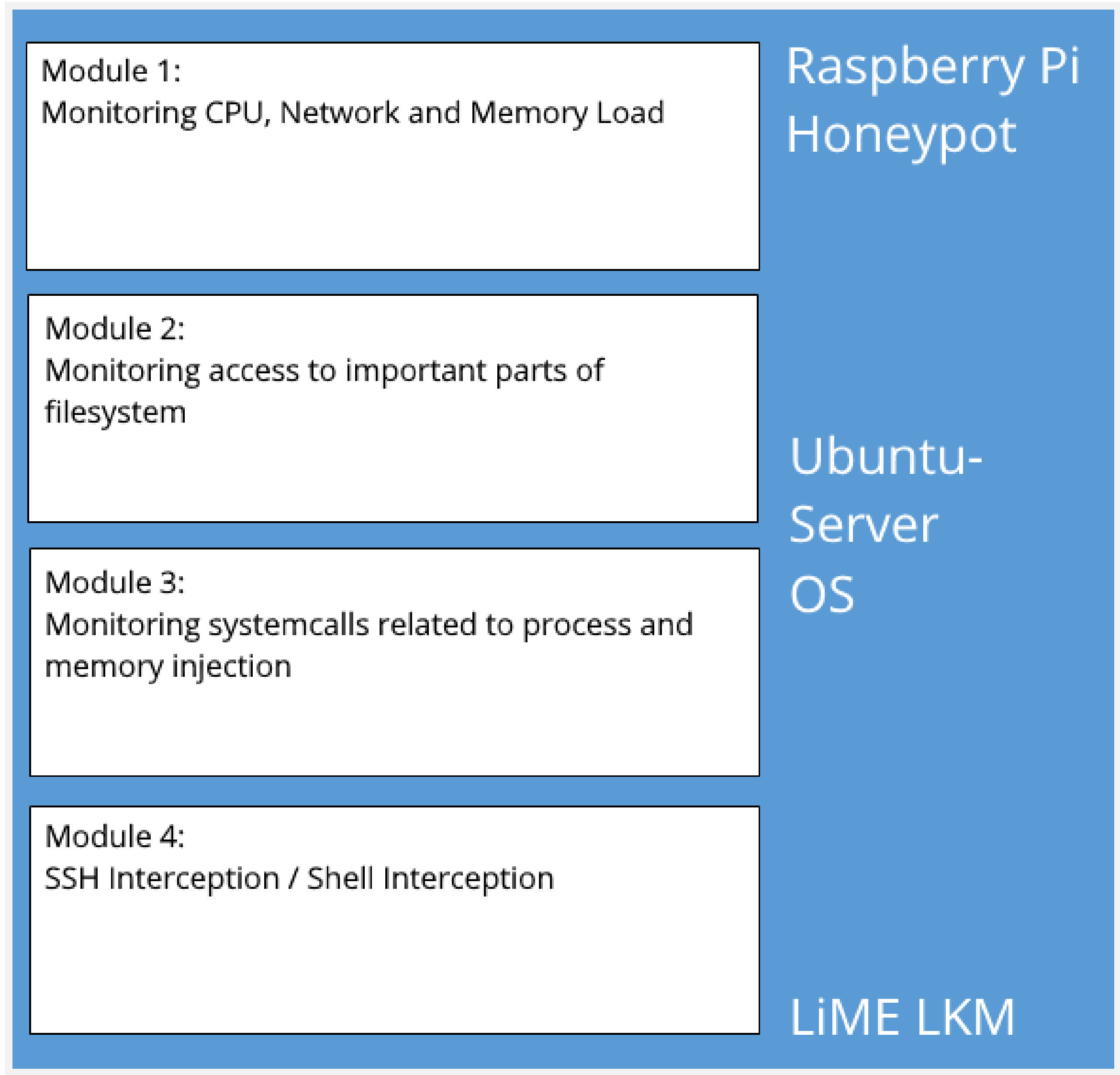

Figure 2 The four different monitoring modules of the Raspberry Pi Honeypot

Module 1 is used to derive any activity on the honeypot by observing the CPU and network load. As the system is isolated and thus remains in a rather constant state any potential activity can be deemed malicious. The CPU load can be derived in percentages by reading in the **/proc/state** file from the Linux file system and calculating the difference in Jiffies[43] over a certain period of time. The file consists of multiple information,

[43] Jiffies are typically a hundredths of a second, the precise numbers depend on the HZ value configuration in the kernel https://man7.org/linux/man-pages/man7/time.7.html

such as the amount of time the CPU has spent performing different tasks in Jiffies / USER_HZ. Those tasks are devided into **user, nice[44], system, idle, iowait, irq, softirq, steal, guest and guest_nice**. By summing up all the values at different times and calculating the difference the overall amount Jiffies elapsed can be calculated. By adding up user, nice and system the actual work over a period is calculated. With those two values the percentage of CPU workload can be derived by dividing the work by the amount of total jiffies times 100.

CPU workload = work difference / total difference * 100

This formula summarized the calculation of the current given CPU load.
(cf. Tasoulas, 2014) (Kerrisk, proc(5) - linux man-pages, 2020)

In addition, other details are also included in this file such as stats about pages, swap pages, interrupted services, disk input / output, context switches, time passed since boot, number of forks since boot, number of processes in runnable state, number of processes blocked waiting for I/O to complete and softirq. Especially the number of forks since boot is also an important value to observe, as this value should stay rather constant in an isolated system. (cf. Kerrisk, proc(5) - linux man-pages, 2020)

Similarly, the network load can be observed by parsing the **/proc/net/dev** file, which contains information about interfaces. It displays the received and transmitted bytes and packets as well as other various information. As the honeypot only interactions over a network whenever it detects malicious activity, the overall amount of transmitted and received data should remain mostly consistent. The file can be scanned for any anomalies such as sudden spikes and sudden increase in load. It needs to be noted that detected malicious activity will lead to communication between the honeypot and server and thus lead to false positives. A way to avoid this is to communicate in big chunks and disable this monitoring functionality during the short communication period. (cf. Kerrisk, proc(5) - linux man-pages, 2020)

The memory region can be monitored for anomalies in the **/proc/meminfo** file which displays various statistics about the RAM such as totally memory, free memory, cached memory etc. (cf. Kerrisk, proc(5) - linux man-pages, 2020)

Module 2 monitors important directories of the filesystem upon any changes. Table 6 summarizes the most important directories, although it is important to mention that due to the constraints of the Raspberry Pi combined with the complexity of an overall filesystem it is not possible to fully scan every single part of the system. The Linux Kernel >2.6 contains an audit subsystem to monitor filesystem activity. It consists of

[44] The nice value of a process determines favourability of the scheduling with smaller numbers indicating higher priority https://linux.die.net/man/3/nice

multiple components that can be used for different areas. The ability for remote / centralized logging is also provided, which is hugely beneficial as it adds no extra work like rewriting the client. The two important areas from this software for this thesis are the ability to log system calls as well as the file system. The userspace component to search and view the logs is called auditd[45]. The following components are present within the auditing system: Auditd handling the filtering rules and writing logs to disk, Audisp transmitting logs to remote systems, Auditctl to add / remove rules, report status and enable / disable the auditing system, Ausearch to search the audit logs, Aureport for summary reports from audit records and Aulast as an audit system of the "last" command. (cf. Kennel, 2018)
The auditctl module can be used to add rules to monitor file system access. The basic syntax to observe a file for changes is:

Auditctl -w <path> -p <permissions that trigger the event> -k <key>

This rule would fire if the file / directory at <path> is being used with the given permissions. The key is just a unique identifier for this certain rule to distinguish which rules create which logs. For a more thorough explanation of possible options visit the man page. (cf. Grubb, audit.rules(7) - linux man-pages, 2019)

To make sure that log entries are sent to the remote server **/etc/audit/auditd.conf** can be edited in the following way to omit logging on the local machine:

Log_format = NOLOG

Next the audispd-plugins package must be installed and the value **active** must be set to **yes** in **/etc/audisp/plugins.d/au-remote.conf.**
Finally **/etc/audisp/audisp-remote.conf** needs to be edited:

remote_server = server2.hl.local
port = 60

The receiving server only needs to audit **/etc/audit/auditd.conf:**

tcp_listen_port = 60

As the last step port 60 should be enabled and made accessible through the firewall. (cf. Peng, 2016)

[45] User component of the linux auditing system https://linux.die.net/man/8/auditd

/etc/lilo.con **/boot/grub/grub.conf**	Important files that store configuration for the bootloaders grub / lilo
/proc/cmdline	Contains kernel parameters
/etc/system.d	Daemons and services
/etc/rc.* **/etc/init**	Run commands
/etc/crontab **/etc/cron.*/** **/var/spool/cron/** **/etc/profile** **~/.bash_profile** **~/.bash_login** **~/.profile./home/user/.bashrc** **/etc/bash.bashrc, /etc/pro-file.d/**	Important directories to monitor as they can be used to automatically start up scripts on shell launch
/etc/hosts **/etc/resolv.conf**	Network and DNS settings
/etc/passwd **/etc/group** **/etc/gshadow**	Important authentication and access right
/bin **/sbin**	System binaries
/tmp	Temporary files
/etc/audit/audit.rules **/etc/audit/auditd.conf**	Contains configuration and rule settings of the auditd daemon

Table 6 Important Linux directories for monitoring attacks (cf. Smith, 2017)

Module 3 intercepts the SSH connection that enables access in the first place for an attacker. As ssh server Dropbear[46] is being used. The source code is slightly modified to allow intercepting commands / packets respectively before they get encrypted prior to being transmitted and after they get decrypted after being received. The file that contains the final encrypting and first decrypting routine is packet.c. The decrypt routine is called at the end of the read_packet() function call. write_packet() functions a bit differently and thus encrypt_packet() has to be intercepted instead. By modifying the source code at the end of those functions to store the data or transmit it over a network connection, the connection can be monitored. The following code is copied from the original source code given in the footnote and displays the location in which the code must be inserted. The buffer that contains the actual contents of the transmitted packages is the ses variable.

```
void read_packet() {

   int len;
   unsigned int maxlen;
   unsigned char blocksize;

   TRACE2(("enter read_packet"))
   blocksize = ses.keys->recv.algo_crypt->blocksize;

   if (ses.readbuf == NULL || ses.readbuf->len < blocksize) {
      int ret;

      ret = read_packet_init();

      ...

   maxlen = ses.readbuf->len - ses.readbuf->pos;

   ...

   if ((unsigned int)len == maxlen) {
      /* The whole packet has been read */
      decrypt_packet();
      /* The main select() loop process_packet() to
       * handle the packet contents... */
   }

   <Insert code to intercept ses buffer right after decrypt packet>

   TRACE2(("leave read_packet"))
}
```

[46] Dropbear is a lightweight SSH server and client https://github.com/mkj/dropbear

Intercepting outgoing packages is slightly different, as the **write_packet()** function writes out an already encrypted packet out. Thus **encrypt_packet**() is modified. The code sample is not be copied in this case as the code to read out the ses buffer can be directly placed at the beginning of the function after the variable assignments.

In the sessions.h header it can be seen that **ses** is a structure of type **sshsession**:

```
/* Global structs storing the state */
extern struct sshsession ses;
```

The struct is rather big with various different variables and further structs, the following are the most important field relevant for the interception:

```
struct sshsession {

    time_t connect_time;
    int sock_in;
    int sock_out;

    buffer *writepayload; /* Unencrypted payload to write - this is used
                          throughout the code, as handlers fill out this
                          buffer with the packet to send. */
    struct Queue writequeue; /* A queue of encrypted packets to send */
    unsigned int writequeue_len; /* Number of bytes pending to send in
    writequeue */
    buffer *readbuf; /* From the wire, decrypted in-place */
    buffer *payload; /* Post-decompression, the actual SSH packet.
                     May have extra data at the beginning, will be
                     passed to packet processing functions positioned past
                     that, see payload_beginning */
    unsigned int payload_beginning;
    unsigned int transseq, recvseq; /* Sequence IDs */

    /* Packet-handling flags */
    const packettype * packettypes; /* Packet handler mappings for this
                                    session, see process-packet.c */

    ...
}
```

By accessing the pointers to the buffers the whole ssh session connection packets can be intercepted and later analysed. By identifying the packettype from the **packettypes** pointer it is possible to understand the payload. It is the structure used to redirect to the corresponding handler in the **process_packet()** function:

```
if (ses.packettypes[i].type == type) {
            ses.packettypes[i].handler();
            goto out;
        }
```

The start of the payload is given in **payload_beginning**. Following are the different packet types and their corresponding handler functions:

```
static const packettype svr_packettypes[] = {
    {SSH_MSG_CHANNEL_DATA, recv_msg_channel_data},
    {SSH_MSG_CHANNEL_WINDOW_ADJUST,
     recv_msg_channel_window_adjust},
    {SSH_MSG_USERAUTH_REQUEST, recv_msg_userauth_request},
    {SSH_MSG_SERVICE_REQUEST, recv_msg_service_request},
    {SSH_MSG_KEXINIT, recv_msg_kexinit},
    {SSH_MSG_KEXDH_INIT, recv_msg_kexdh_init},
    {SSH_MSG_NEWKEYS, recv_msg_newkeys},
    {SSH_MSG_GLOBAL_REQUEST, recv_msg_global_request_remotetcp},
    {SSH_MSG_CHANNEL_REQUEST, recv_msg_channel_request},
    {SSH_MSG_CHANNEL_OPEN, recv_msg_channel_open},
    {SSH_MSG_CHANNEL_EOF, recv_msg_channel_eof},
    {SSH_MSG_CHANNEL_CLOSE, recv_msg_channel_close},
    {SSH_MSG_CHANNEL_SUCCESS, ignore_recv_response},
    {SSH_MSG_CHANNEL_FAILURE, ignore_recv_response},
    {SSH_MSG_REQUEST_FAILURE, ignore_recv_response},
    {SSH_MSG_REQUEST_SUCCESS, ignore_recv_response},
    #if DROPBEAR_LISTENERS
    {SSH_MSG_CHANNEL_OPEN_CONFIRMATION,
    recv_msg_channel_open_confirmation},
    {SSH_MSG_CHANNEL_OPEN_FAILURE, recv_msg_channel_open_failure},
    #endif
    {0, NULL}
};
```

The most important packet types are **SSH_MSG_CHANNEL_DATA**, **SSH_MSG_CHANNEL_WINDOW_ADJUST**, **SSH_MSG_USERAUTH_REQUEST** and **SSH_MSG_SERVICE_REQUEST.** The corresponding functions in the same order are **recv_msg_channel_data**, **recv_msg_channel_window_adjust**, **recv_msg_userauth_request** and **recv_msg_service_request**.

To further make interception even easier the handler functions can be the interception points, making it clear what kind of data is being used and thus only making it necessary to extract the payload. As not all data is relevant this is the preferred option, leading to more organized information and less useless network traffic and delay. The three functions of particular interest are **common_recv_msg_channel_data()**, **send_msg_channel_data()** and **recv_msg_channel_window_adjust().** The receiving handler copies the payload into the **cbuf** buffer which can be accessed to get all the necessary information about channel data.

```
void common_recv_msg_channel_data(struct Channel *channel, int fd,
        circbuffer * cbuf) {

    unsigned int datalen;
    unsigned int maxdata;
    unsigned int buflen;
    unsigned int len;
    unsigned int consumed;
    int res;

    ...

    datalen -= consumed;
    buf_incrpos(ses.payload, consumed);


    if (res == DROPBEAR_SUCCESS) {
        len = datalen;
        while (len > 0) {
            buflen = cbuf_writelen(cbuf);
            buflen = MIN(buflen, len);

            memcpy(cbuf_writeptr(cbuf, buflen),
                    buf_getptr(ses.payload, buflen), buflen);
            cbuf_incrwrite(cbuf, buflen);
            buf_incrpos(ses.payload, buflen);
            len -= buflen;
        }
    }

    <Intercept cbuf buffer for entire payload of a received channel message>

    TRACE(("leave recv_msg_channel_data"))
}
```

The sending channel data can be intercepted by saving the contents read from the shell to another buffer simultaneously.

```
static void send_msg_channel_data(struct Channel *channel, int isextended) {

    ...

    /* read the data */
    len = read(fd, buf_getwriteptr(ses.writepayload, maxlen), maxlen);
    <intercept the written data to ses.writepayload at the specific offset>

    ...
}
```

The contents of the window adjust handler are not particularly interesting, however it would be an asset to be informed each time the handler is being called. The reason is that calls of this function very likely indicate human activity rather than machine interaction (cf. Fan, et al., 2019).

Module 4 monitors systemcalls that could indicate memory or process injection. As this is a Linux system particularly **memfd_create()** and **ptrace()** which are both calls that are frequently used to inject code into memory / processes. The monitoring is done with the same software as module2, auditd. The general form of such a system call is:

```
-a action,list -S syscall -F field=value -k keyname
```

Actions specify what happens after the event triggers, list is a rule list it is going to be appended to, syscall specifies the call that is going to be monitored, field=value can further specify and finetune the rule and keyname is an unique identifier for this certain rule. The resulting logs can as well be transmitted over the network to the honeypot main server. (cf. Grubb, audit.rules(7) - linux man-pages, 2019)

Lastly LiME[47] is deployed on the Raspberry Pi to create memory dumps upon certain events for further manual investigations. Just like anything else those dumps are also meant to be transmitted to the honeypot server. Memory dumps should be created whenever the modules detect any anomalies in the process lists, network / CPU / memory load and when system call rules are triggered as those are most likely to indicate changes within the RAM. The purpose is to investigate possible malicious code injected into live processes / memory to further broaden the understanding of the techniques utilized by the attackers. The earlier mentioned Volatility forensic software can be used to analyse the resulting memory dumps. As memory dumps slow down the system for an ineligible period of time the memory dumps are supposed to be taken manually from the remote honeypot server, which can send the command over the serial communication channel.

To attract an intruder Dropbear utilizes weak credentials that can easily be brute forced / guesses for an admin access. The goal is to make the first entry point an easy effort and further honeypots in the second layer require more sophisticated exploits. This is an attempt to improve the motivation to compromise systems with a gradual increase in difficulty to overtake them and slowly unveil a bigger part of the repertoire of an attacker.

An operating system that performs greatly on Raspberry Pi is OpenWRT[48], as it has specifically been designed for embedded devices and performance with low resources in mind. Furthermore, it is a software mostly used for routers, which could open new

[47] A forensic Loadable Kernel Module tool to create memory dumps from Linux and Linux-based devices https://github.com/504ensicsLabs/LiME

[48] OpenWrt is a Linux operating system targeting embedded devices and available at https://openwrt.org/

possibilities like deceiving an attacker into thinking it is a routing device. However, as Ubuntu Server properly supports auditd and has specific images for Raspberry Pi this operating system is used.

The earlier mentioned rootkit technologies can be used to an extent to hide potential processes from showing up to an attacker. As popular hooking frameworks and techniques are not necessarily supported by most embedded operating systems and it would require tremendous efforts to repurpose the kernel / system calls into accepting those, the effort to implement this is omitted. This is further discussed in the limitations section of the thesis.

As the Honeypot is naturally based on real embedded hardware there is no need to specifically change any simulations or software to behave more like a real system. This was one of the main concerns and priorities when creating the concept, as it is not only easier to deceive attackers, but it also prevents unnecessary effort. Usually even after sophisticated changes small artifacts still disclose the real nature to an attacker if they decide to dig deep enough, in this case it is only necessary to hide the few software modules and possibly encrypt any network connection.

3.2 Layer 2

Before talking about Layer 2 it should be mentioned that the IDS scanning the network is connected to the switch that connects the devices from the different layers and thus can be considered between Layer 1 and 2. The purpose is to catch any network fragments that haven't been monitored by the system. With resource constraints on an embedded system and security software in general it is nearly impossible to capture every kind of activity on a system, especially when undisclosed vulnerabilities are in use. However, IDS can still detect potential network traffic that has lead to a compromise and in the best case the captured event may even showcase the payload or any information that can give hints towards the motivation or intent. As mentioned before multiple times, the whole honeypot network is supposed to be rather isolated. This means that aside from communication between the server and honeypot there should almost be no other traffic. This makes it much easier to monitor for any potential malicious network activity, as all that is required is to whitelist certain activity between the honeypots and the server and deem everything else malicious. This can simply be done in SNORT by creating rules that target the honeypots as destination and source IP exempting activity coming from and to the server if it fits the actively used ports. The IDS should also monitor any potential attacks and in emergency situations notify the main server with an alert appealing to shut down the whole system. Such a situation could arise when an attacker loses interest in the honeypot systems and starts trying to further exploit / infiltrate the production network or uses denial of service attacks. It can be summed up that on one hand SNORT is being used as an alert system for the honeypot server and secondly logging network traffic for further analysis.

The actual second layer is supposed to consist of further honeypots. In this elementary concept those honeypots are SCADA workstations running on various operating systems such as Linux and Windows. The purpose is to provide a second stage for an attacker. As those are not mere IoT devices but workstations that seemingly control ICS, there is more of a motivation to compromise such a system. However, it is not a real SCADA system but rather a simulated one. Depending on the needs an ICS simulation or real ICS system can be connected to the workstation. The SCADA software can be an emulation to intercept the communication and analyse the attack patterns. Together with Layer 1 the real resemblance of an IIoT network is created. These workstation honeypots mostly scan for potential attacks that could affect a SCADA system, hooking techniques for potential persistence, Registry / WMI objects and powershell / bash commands. SCADASim[49] is a potential project that can be used to setup a minimalistic working SCADA environment in virtual machines. It encompasses the modbus protocol, Human-Machine-Interfaces and PLC[50] devices.

Important directories that should be monitored under windows are provided by FireEye after their analysation of the TRITON malware:

```
C:\Windows\system32\inetsrv\
C:\Windows\temp\
C:\Windows\SysWOW64\wbem
C:\Windows\SysWOW64\drivers
C:\Windows\SysWOW64
C:\Windows\system32\wbem\
C:\Windows\system32\drivers\
C:\Windows\system32\
C:\Windows\
C:\Users\Public\Libraries\
C:\Users\administrator\appdata\local\temp\
C:\ssh\
C:\perflogs\admin\servermanager\ssh\
C:\perflogs\admin\servermanager\
C:\perflogs\admin\
C:\perflogs\
C:\cpqsystem\
C:\hp\hpdiags\
C:\hp\bin\log\
```

Figure 3 Windows paths that should be monitored (cf Johnson, et al., 2017)

The directories can be monitored by creating a service utilizing the File **SystemWatcher** class. Further SCADA specific directories used to communicate within the system can

[49] SCADASim is a configurable SCADA exercise environment https://github.com/cmu-sei/SCADASim

[50] PLCs are Programmable Logic Controllers controling manufacturing processes

also be observed using this class object. (cf. Microsoft, n.d.)
Specific protocols used to communicate with Programmable Logic Controllers and other devices such as Modbus should also be strictly monitored. To monitor the outgoing traffic from the workstations within the ICS an open source HIDS can be used. OSSEC[51], an open source and free HIDS that is available on windows and can perform a multitude of scanning such as monitoring ports, the registry and file integrity is a possible choice. The earlier mentioned hooking techniques need to be monitored as well to possibly detect any injections. A potential tool to trace system calls is drstrace[52], which can monitor certain applications for the system calls that result out of them. Child processes are traced as well. As it is counterproductive to disable the security measures that prevent modifications of trusted binaries hooking techniques are not as useful as on Linux. A trusted security kernel device driver could be created to monitor system call activity, however this is a very difficult and complex task. The WMI objects can be monitored with Sysmon as mentioned earlier. Network activity, system process chains and file modification can also be monitored with this tool.
In an official Microsoft dev blogpost methods to enable protected event logging are described. It works by public key cryptography storing the private key at a central event log collector. The Group Policy has to be edited to enable event logging in **Windows Components -> Administrative Templates -> Event Logging**. An encryption certificate is necessary for example in the form of a base-64 encoded x.509 certificate[53]. This way logs stay encrypted and secure from a potential attacker. It also is possible to log PowerShell script blocks that it processes. To enable this feature the Script Block Logging feature in the Group Policy has to be enabled at Windows **Components -> Administrative Templates -> Windows PowerShell.**
(cf. Microsoft PowerShell Team, 2015)

Furthermore, it is possible to edit / replace standard system binaries, so they omit any returns involving the modules. As the honeypot is in full control of the defender, any changes can be made to the system before deployment. A more advanced approach could be copying the stealth capabilities of a rootkit cryptocurrency miner analysed by Trend Micro. Files and network traffic can be hidden by hooking various functions such as **readdir, fopen, fopen64, lstat, lxstat, open, rmdir, stat, stat64, __xstat, __xstat64, unlink, unlinkat, opendir, readdir, readdir6**. By hooking the function and returning a "no such file or directory error" upon receiving a string that is equal to the module name stealth can be achieved. (cf. Remillano II & Malagad, 2019)

[51] OSSEC is a multi-platform, open source Host-based Intrusion Detection System https://www.ossec.net/about/

[52] Drstrace is a system call tracing tool for Windows https://dynamorio.org/drmemory_docs/page_drstrace.html

[53] X.509 is a public key infrastructure standard https://docs.microsoft.com/en-us/windows/win32/seccertenroll/about-x-509-public-key-certificates

It is very unlikely that an attacker will use advanced forensic methods to search a system for possible rootkits, thus improving the possibility of deception.

On x64 Linux systems system call hooking techniques can be utilized to improve the detection and stealth of the honeypot systems. As altering a kernel brings security risks, possible stability issues and is very hard to maintain a good alternative is the usage of ftrace[54] kernel hooking. This technique is described in a blogpost by Lozovsky and Stepanchuk. The following structure summarizes each hooked function:

```
struct ftrace_hook {
    const char *name;
    // The name of the hooked function
    void *function;
    // Address of wrapper function that is called instead
    void *original;
    // pointer to the place where the address of the hooked function will be
       stored
    unsigned long address;
    // address of the hooked function
    struct ftrace_ops ops;
    // ftrace service information, initialized by zeros

    // Only name, function and original needs to be used
}
```

By using kallsyms the address of the needed function is found. An error message checks if there are any unresolved symbols and finally the address of the hooked function is stored in the pointer **original**.

```
static int resolve_hook_address(struct ftrace_hook *hook)
{
        hook->address = kallsyms_lookup_name(hook->name);

        if (!hook->address) {
                pr_debug("unresolved symbol: %s\n", hook->name);
                return -ENOENT;
        }

        *((unsigned long*) hook->original) = hook->address;

        return 0;
}
```

Following the ftrace_ops structure is initialized within the **fh_install_hook()** function call.

[54] Ftrace is a kernel tracing framework

```
int fh_install_hook(struct ftrace_hook *hook)
{
	int err;
	err = resolve_hook_address(hook);
	if (err)
		return err;

	hook->ops.func = fh_ftrace_thunk;
	hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS |
	FTRACE_OPS_FL_IPMODIFY;
	err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
	if (err) {
		pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
		return err;
	}

	err = register_ftrace_function(&hook->ops);
	if (err) {
		pr_debug("register_ftrace_function() failed: %d\n", err);
		/* Don't forget to turn off ftrace in case of an error. */
		ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
		return err;
	}
	return 0;
}
```

fh_ftrace_thunk is the callback that ftrace will call. The flags ensure a consistent save and restore of the processor's registers. Next, **ftrace_set_filter_ip()** turns on the ftrace utility for the specific function and **register_ftrace_function()** gives ftrace the necessary permissions to call the callback.

Now by changing the register %rip an unconditional jump can be forced from the current function to take over control.

```
static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long parent_ip,
		struct ftrace_ops *ops, struct pt_regs *regs)
{
	struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);

	/* Skip the function calls from the current module. */
	if (!within_module(parent_ip, THIS_MODULE))
		regs->ip = (unsigned long) hook->function;
}
```

The address of **ftrace_hook** is obtained by the usage of the macro **container_of()**[55] and **ftrace_ops** is embedded within **ftrace_hook** with the pointer ***ops**.

[55] This macro makes it possible to find the container of a given field of a structure. The second line of the macro finds the real location in memory of the struct https://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/067/6717/6717s2.html

The **%rip** register is then accordingly set to point to the handler's address. By specifically checking if the call originates from the same module recursion is avoided. For repeated calls, the **parent_ip** argument will not point to some place in the kernel anymore and instead point to inside the wrapper thus failing the check. This technique can eventually also be used in the Layer 1 honeypots, nonetheless adjustments must be made as registers are different on ARM architecture. Also, there is no guarantee that this technique can even properly function on an embedded ARM operating system. (cf. Lozovsky & Stepanchuk, 2018)
The Second layer can be extended by any number of honeypots to further improve the honeypot system. For example, virtualized honeypots and architectures such as MIPS can be employed to broaden the potential to detect attacks against specific architectures.

3.3 Layer 3

Layer 3 only encompasses the central server of the whole system secured behind a firewall. The purpose of the server is to collect log data from all honeypot devices deployed across the system, provide a heartbeat to assess if honeypots are still accessible and able to forcefully shutdown systems. This central entity can be accessed remotely by a security expert to further analyse collected log data and captured activity for example to investigate memory dumps for novel malicious code. Communication is ensured over the usual network connection, but the heartbeat utilizes serial communication to be stealthier and avoid a single point of failure, providing a potential emergency way to shutdown systems and test connectivity continuously with less detectable network footprint. In addition, the serial communication can be used to issue commands such as ordering memory dumps. The firewall needs to be set accordingly to only allow necessary communication between the honeypot and the server. The Server also needs to be properly hardened and secured to avoid infiltration. Unnecessary ports and services must be disabled. Encrypted communication between the honeypot and server can veil the log and activity transmission to a certain degree.

Figure 4 summarizes the whole concept in a visual computer network diagram with special emphasize on the particular layers and their contents. The switch inbetween Layer 1 and Layer 2 is capable of hosting multiple additional honeypots that can further improve the broadness of covered operating systems and architectures. The overall goal of the system is to detect and monitor fileless attack activity of specialized attackers, be expandable and hide the true honeypot nature from any attacker. The deployed honeypots are a mixture between deception and reconnaissance high interaction honeypots for both research and production detection. On one hand they gather intel on novel attack patterns and techniques on the other hand they also notify any potential breaches in the ongoing production network. Relatively seen the honeypot is mostly fit to monitor APT activity during the fifth stage as it is deployed within a production network and thus a perfect target for further reconnaissance and exploitation.

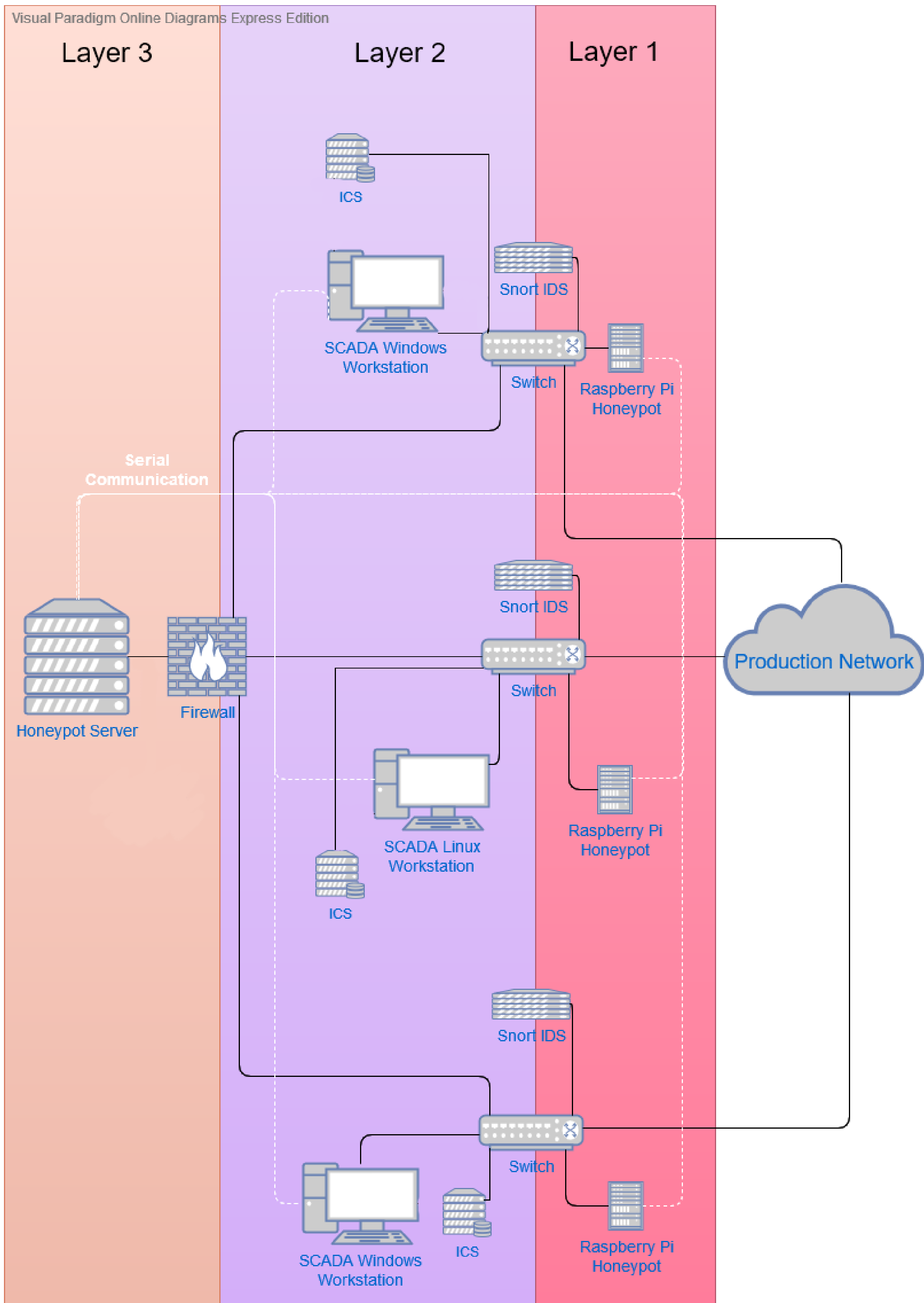

Figure 4 Fileless malware honeypot detection concept. Made with Visual Paradigm https://online.visual-paradigm.com/diagrams/solutions/free-network-diagram-software

4 Layer 1 Implementation

4.1 Implementation

In this chapter Layer 1 is implemented for test purposes to partially see the effectivity of the concept. Due to limitations in hardware and configuration the implementation is a bit different from the original concept, however it still demonstrates the main purpose well enough. It features a computer with a Raspberry pi on the same local network. The internet connection is monitored with Wireshark[56]. The goal is to implement the earlier described modules of the Layer 1 honeypot and evaluate the effectivity. As an operating system for the honeypot Ubuntu Server is used as the audit kernel subsystem still has stability issues on the current OpenWRT release. Furthermore, Ubuntu Server opens the ability to compile directly on the Raspberry Pi making the setup easier. Wireshark demonstrates a similar concept to Snort as the main purpose of the IDS was to sniff packets. The following graphic shows the setup:

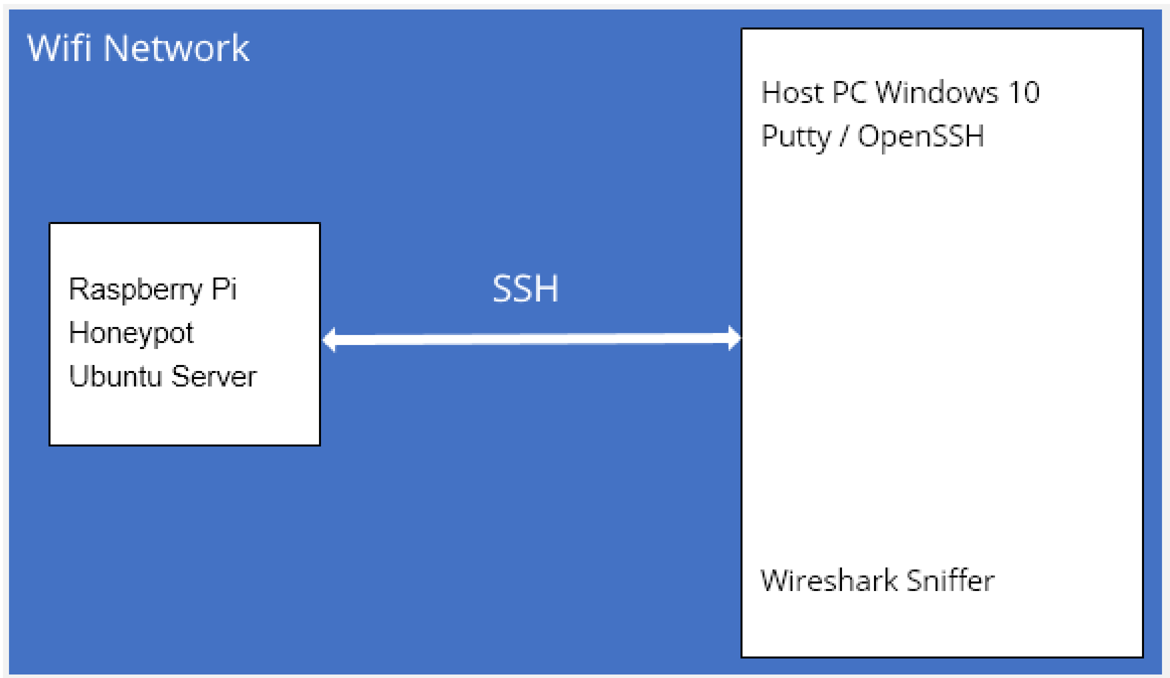

Figure 5 The layout of the Layer 1 implementation test

[56] Wireshark is a free and open-source packet analyzer https://www.wireshark.org/

In the first step the official Ubuntu Server image is downloaded from the official Ubuntu homepage[57]. To flash the image onto the USB Stick Etcher[58] is used. Win32Diskimager[59] is used to create any necessary backups. Etcher has a lot faster performance but it lacks a backup feature, thus making it necessary to use both software. After successfully booting and setting up the Raspberry Pi Ubuntu Server operating system, **/etc/netplan/50-cloud-init.yaml** must be edited to utilize Wi-Fi to update and install necessary software. The syntax is the following:

```
Network:
    wlan0:
        optional: true
        access-points:
            “Wifi”:
                password: <Passw>
        dhcp4: true
```

It is important to not use tabs but intend with 4 spaces each level, else the syntax does not function properly. After rebooting the machine, it should automatically connect to the Wifi. (cf. Lubos, 2020)

GCC[60] can be simply used to compile any necessary sources on the Raspberry Pi itself. The first step is modifying dropbear and installing it from source. Like discussed in the previous chapter, certain files are edited to intercept traffic. Dropbear functions by associating packets of a certain type to handler functions that are called upon receiving such packets. The ones of interest are any that carry actual payload that is send to a shell channel or connection information such connection attempts. Both cli.session.c and svr-session.c define the packettypes and their handler functions. The important ones are **recv_msg_channel_data, recv_msg_channel_window_adjust, recv_msg_userauth_request.** The first one carries the actual payload and upon investigation ends up calling a function called **common_recv_msg_channel_data** within **common-channel.c.** This function uses the current channel (shell) to write the payload from the **ses** structure. The final function called is **writechannel** which results in a call to **writev**[61]. By doing a second custom **writev** to a log file it is possible to log any commands that are issued to the shell. This simplifies the matter so that it is not necessary to parse the earlier mentioned **ses** structure.

---

[57] Ubuntu Server Raspberry Pi Image available on https://ubuntu.com/download/raspberry-pi

[58] Etcher is a software to flash OS images to SD cards & USB drives https://www.balena.io/etcher/

[59] Win32Diskimager is a software that can write and read bootable ISO images https://win32diskimager.download/

[60] The GNU Compiler Collection http://gcc.gnu.org/

[61] The system call writev() writes iovcnt buffers of data described by iov to the file descriptor

In **recv_msg_channel_window_adjust** not much has to be changed, a simple write to a log file notifying of the call is enough, as the purpose of hooking this function is to deduce human activity. Finally, **recv_msg_userauth_request** is located in **svr-auth.c.** This function has a variable username which contains the sent login name. Respectively before **send_msg_userauth_sucess** and **send_msg_userauth_failure** the username variable can be intercepted to see (un)successful login attempts and the corresponding name. Furthermore, **svr_auth_password** from **svr-authpasswd.c** is called. In this function the used password can be intercepted from **password.** All the intercepted results are simply written to a log file. As changes have been made to dropbear the code has to be compiled from source. First **./configure** needs to be run, then **make** while specifying the wanted modules like **make PROGRAMS="dropbear dbclient dropbearkey dropbearconvert scp"** and finally an **install** command. Apparently there seem to be issues with the current installation files. Either the configure, make or install files are missing certain specifications for Ubuntu Server on an ARM architecture. The instillation finishes successfully but it does not install successfully as a service and all config files are missing. This is not the case when dropbear is installed from the APT package manager. As there is no real indication what causes the issue and it is a highly complex task to analyse the misbehaviour further implementation of this module is omitted. Instead the module that monitors usage of cpu, memory and network also monitors the bash history. After exiting a bash shell all commands are saved to a history file located in either **/root/.bash_history** or **/home/usr/.bash_history** (cf. GNU, 29)**.** As the goal is intercepting all incoming commands, **/root/.bashr**c and **/home/usr/.bashrc** have to be edited to persist all changes immediately. By adding the following line this can be achieved:

```
export PROMPT_COMMAND="history -a; $PROMPT_COMMAND"
```

history -a appends the current lines from the current session to the history file of the given user. PROMPT_COMMAND[62] leads to the persisting after each command is issued. In addition, the size of the history file can be edited in this file.
(cf. LinuxSecurityFreak; Pablo R., 2010)

Lastly, the profile file in the same directory was missing upon installation, which is necessary to use .bashrc. By simply creating this file and pasting the following line the problem is fixed:

```
if [ "$BASH" ]; then
  if [ -f ~/.bashrc ]; then
    . ~/.bashrc
  fi
fi
```

[62] The value in PROMPT_COMMAND is executed as a command before the printing of any primary prompt http://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html

Now any history from any session of a single user should be saved within the respective history file. It is important to do this step for every user if there are multiple ones. The next step is creating a small c program that runs in the background and periodically checks all the parameters. The program reads certain system files to monitor potential attackers which are:

```
/proc/meminfo
/proc/net/dev
/proc/stat
/root/.bash_history
/home/<user>/.bash_history
```

The first, second, third, fourth, twentieth and twenty-second line in **proc/meminfo** are read to respectively get the total memory, free memory, available memory, buffered memory, nmap shared memory and anonymous nmap shared memory. Next, **proc/net/dev** contains network usage. Even in an isolated machine memory constantly changes, however bigger than usual changes can indicate potential malicious activity. The incoming and outgoing packages from the **wlan0** interface are read. Any changes in here indicate incoming or outgoing packages, which should stay rather low without any incoming connections. The CPU load is calculated as mentioned earlier by taking the difference between two reads and comparing the idle time to processing time. In addition the **/proc/stat** file lists all currently running and number of started processes since boot. Those numbers fluctuate slightly, big changes in small timeframes also indicate activity. (cf. proc(5) - linux man-pages, 2020)
Finally, the size of **.bash_history** is constantly checked and if it changes the new added commands are written into the log file. The code periodically checks all those files for changes and if there are any changes the program writes them into a log with a timestamp. The implementation of the code is found in Appendix 1.

Next auditd is installed throught the apt package manager. To create persistent rules rather than temporary ones the **/etc/audit/audit.rules** file has to bedited. The same syntax as mentioned before is applied here except of omitting auditctl. Directories and files are observed by adding a rule with the **-w <path> -p wa -k <tag> syntax**. The **-w** simply specifies a watch on a file or directory, **-p** specifies the operations that are operated (in this case write and attribute changes) and the tag which is an identifier for a rule. Read monitoring is omitted as it leads to too much noise without a sophisticated ruling. System calls are monitored with **the -a <action> -S <systemcall> -k <tag>** syntax. Actions are used to determine what actions to take when the event is triggered. In this specific case **always,exit** is used to signify an event should always be created on exit of the system call. The reason for exit is that on entrance not all information is available for proper logging. (cf. Grubb, auditctl(8) - linux man-pages, 2018)
The following rules are all added to the file to monitor important directories which can be of interest for an attacker and system calls used for direct memory injection. All of them were mentioned earlier in the Background chapter:

```
-w /etc/passwd -p wa -k passwd
-w /etc/lilo.con -p wa -k lilo
-w /proc/cmdline -p wa -k cmdline
-w /etc/system.d -p wa -k system.d
-w /etc/rc0.d -p wa -k rc
-w /etc/rc1.d -p wa -k rc
-w /etc/rc2.d -p wa -k rc
-w /etc/rc3.d -p wa -k rc
-w /etc/rc4.d -p wa -k rc
-w /etc/rc5.d -p wa -k rc
-w /etc/rc6.d -p wa -k rc
-w /etc/rcS.d -p wa -k rc
-w /etc/init -p wa -k init
-w /etc/crontab -p wa -k cron
-w /etc/cron.d -p wa -k cron
-w /etc/cron.daily -p wa -k cron
-w /etc/cron.hourly -p wa -k cron
-w /etc/cron.monthly -p wa -k cron
-w /etc/cron.weekly -p wa -k cron
-w /var/spool/cron -p wa -k cron
-w /etc/profile -p wa -k profile
-w /home/ubuntu/.bashrc -p wa -k bash
-w /home/ubuntu/.profile -p wa -k bash
-w /root/.bashrc -p wa -k bash
-w /root/.profile -p wa -k bash
-w /etc/bash.bashrc -p wa -k bash
-w /etc/profile.d -p wa -k bash
-w /etc/hosts -p wa -k network
-w /etc/resolv.conf -p wa -k network
-w /etc/audit/audit.rules -p wa -k audit
-w /etc/audit/auditd.conf -p wa -k audit
-a always,exit -S memfd_create, ptrace -k mem_inject
```

Figure 6 /etc/audit/autit.rules ruleset

Upon persisting the changes, the auditd daemon must be restarted by issuing **sudo service restart auditd** and afterwards the success can be tested with the command **sudo auditctl -l** which lists all currently loaded rules.
(cf. Grubb, audit.rules(7) - linux man-pages, 2019)

Both system calls should be rarely used in an isolated environment and thus the existence of log entrances can be deemed highly suspicious. It is possible to configure auditd in a way that the logs are sent to a remote server, which is not goal of the test implementation, but still part of the actual concept proposed in the methodology.

In the last step lime is setup on the target device so memory dumps can be created. Lime is simply copied from the official GitHub repository[63] which also states that it is possible to create memory dumps to remote machines. The source code simply has to be created with the make command and then a new LKM is created. This module can be loaded to create a dump using the **insmod** command in the following way:

```
insmod <lime.ko LKM> "path=<outfile | tcp:<port>>
format=<raw|padded|lime> [digest=<digest>] [dio=<0|1>]"
```

Afterwards the memory dump can be analysed with the Volatility framework.
(cf. Sylve, 2020)

In the future concept it is intended that the honeypot server can send a request over the serial communication to run this command and then receive the dump over the network. However, as this implementation only evaluates the first layer the memory dump is taken manually if investigation of logs shows malicious activity that could have led to memory injection.

Wireshark is used as a packet sniffer tool instead of Snort due to the more dynamic nature which is more suitable for testing purposes. By running Wireshark and specifying **tcp.port == 22** in the filter only SSH packets are being displayed.

OpenSSH[64] is used as an entry gate as the software is already installed by default and due to the issue that dropbear cannot be properly deployed currently.

Furthermore, the deployment of exploitable software is not possible as the same issues arise as with dropbear. Configuration and instillation processes must be adjusted to make source code instalments more feasible. The same holds true for the rootkit capabilities which are supposed to hide the monitoring processes and functionalities from an attacker. A good potential starting point would be Diamorphine[65] but it would first need to be adjusted to function with ARM processors. Due to the complexity this step is omitted and later proposed in the future work section.

4.2 Attack simulation

In this subchapter an attack against the implementation is simulated to evaluate the effectivity of the proposed solutions, although it needs to be mentioned that the goal

---

[63] A forensic Loadable Kernel Module tool to create memory dumps from Linux and Linux-based devices https://github.com/504ensicsLabs/LiME

[64] OpenSSH provides a large suite of secure tunnelling capabilities https://www.openssh.com/

[65] Diamorphine is a LKM rootkit that can hide processes, directories and files https://github.com/m0nad/Diamorphine

is simply to evaluate the implementation rather than explaining a sophisticated attack. Thus, screenshots are omitted as the honeypot should intercept the activity in the form of logs which are later presented. It is assumed that the attacker is already aware of the existence of the system and that the username has been found out through reconnaissance. First the system is infiltrated by cracking the credentials using Ncrack[66]. The tool can be used by using the following command in the directory in which Ncrack is installed:

```
ncrack -v -u ubuntu -P default.pwd 192.168.178.49 -p ssh
```

This command tries to gain ssh access on the 192.168.178.49 system trying to access the user ubuntu. Default.pwd is a password list containing various in common use password. Any password list can be used as long as the formatting is compatible. The attack is visible in Wireshark as dozens of TCP / SSH protocols show up continuously which should not be the case especially right after the handshake that establishes the connection. The following screenshot shows what a brute force looks like in Wireshark.

No.	Time	Source	Destination	Protocol	Length	Info
10015	426.769852	192.168.178.49	192.168.178.37	SSHv2	106	Server: Encrypted packet (len=52)
10016	426.769971	192.168.178.37	192.168.178.49	SSHv2	202	Client: Encrypted packet (len=148)
10017	426.775327	192.168.178.49	192.168.178.37	TCP	64	22 → 53577 [ACK] Seq=1378 Ack=1801 Win=64128 Len=0
10018	426.776056	192.168.178.49	192.168.178.37	TCP	64	22 → 53576 [ACK] Seq=1378 Ack=1801 Win=64128 Len=0
10019	426.776056	192.168.178.49	192.168.178.37	SSHv2	98	Server: Encrypted packet (len=44)
10020	426.776056	192.168.178.49	192.168.178.37	SSHv2	98	Server: Encrypted packet (len=44)
10021	426.776173	192.168.178.37	192.168.178.49	SSHv2	202	Client: Encrypted packet (len=148)
10022	426.776226	192.168.178.37	192.168.178.49	SSHv2	202	Client: Encrypted packet (len=148)
10023	426.780384	192.168.178.49	192.168.178.37	TCP	64	22 → 53569 [ACK] Seq=1474 Ack=2097 Win=64128 Len=0
10024	426.781111	192.168.178.49	192.168.178.37	TCP	64	22 → 53577 [ACK] Seq=1422 Ack=1949 Win=64128 Len=0
10025	426.783410	192.168.178.49	192.168.178.37	SSHv2	106	Server: Encrypted packet (len=52)
10026	426.783501	192.168.178.37	192.168.178.49	SSHv2	202	Client: Encrypted packet (len=148)
10027	426.788218	192.168.178.49	192.168.178.37	SSHv2	106	Server: Encrypted packet (len=52)
10028	426.788308	192.168.178.37	192.168.178.49	SSHv2	202	Client: Encrypted packet (len=148)
10029	426.792365	192.168.178.49	192.168.178.37	TCP	64	22 → 53574 [ACK] Seq=1422 Ack=1949 Win=64128 Len=0
10030	426.796057	192.168.178.49	192.168.178.37	TCP	64	22 → 53570 [ACK] Seq=1474 Ack=2097 Win=64128 Len=0
10031	426.796779	192.168.178.49	192.168.178.37	TCP	64	22 → 53571 [ACK] Seq=1474 Ack=2097 Win=64128 Len=0
10032	426.808139	192.168.178.49	192.168.178.37	TCP	64	22 → 53573 [ACK] Seq=1422 Ack=1949 Win=64128 Len=0
10033	426.815280	192.168.178.49	192.168.178.37	SSHv2	106	Server: Encrypted packet (len=52)
10034	426.815396	192.168.178.37	192.168.178.49	SSHv2	138	Client: Encrypted packet (len=84)
10035	426.822059	192.168.178.49	192.168.178.37	TCP	64	22 → 53572 [ACK] Seq=1474 Ack=1969 Win=64128 Len=0
10036	426.828232	192.168.178.49	192.168.178.37	TCP	64	22 → 53576 [ACK] Seq=1422 Ack=1949 Win=64128 Len=0

Figure 7 Wireshark capture of SSH bruteforce

In the time column it is visible that many packets appear continuously. Especially on an isolated system this is very suspicious, as an ordinary ssh connection starts of rather slow. This becomes apparent later when another graphic is shown capturing an ordinary login attempt. Generally, as SSH establishes an encrypted connection after the handshake the capturing of such packets is only useful for seeing any incoming connection attempts. This is however useful, as currently there is no module that monitors incoming connections on the Raspberry Pi itself.

[66] Ncrack is a high-speed network authentication cracking tool compatible with Windows https://nmap.org/ncrack/

In the next step after gaining access the reconnaissance starts to gather information about the device, operating system, configuration, etc. First a command like **ps -aux** is used to list all currently running processes and gain more information about the current running tasks. Next the **/etc/passwd** file is printed with **cat** and afterwards edited with **nano** to add a new user. The **/etc/shadow** file is also edited with **nano** to finalize the new user. By issuing **uname -a** extended information about the processor architecture, system hostname and the version of the kernel is printed. Next, **lsb_release -d** prints a description of the linux distribution, **cat /etc/os-release** and **cat /etc/issue** prints information about the operating system in use. As Ubuntu is based on Debian the command **cat /etc/debian_version** gives further details on the current running major release version. Next **sudo -l** checks if sudoers is readable. For networking information **hostname -f** and **ifconfig** are issued. Other paths and commands to gather information about the networks are **/proc/net*, /etc/network/interfaces** and **netstat -r.** Finally, **kill -9 $$** and **touch ~/.bash_history** are used to kill the history in the shell and invasively delete the history. This closes the reconnaissance section as the attacker has gathered enough information to cross compile a binary for memory injection for further post exploitation. (cf. Adrian, 2019)

In this attack the **memfd_create** injection is demonstrated now to assess the system call monitoring capabilities. The technique described in chapter 2.3.4 is used, however this time like in the live demo example a perl script pertaining the necessary code is created rather than issuing the commands step by step. The script is taken from the same blog post of Stuart and adjusted to function on the targeted system and slightly shortened. The script contains the following code:

```
use warnings;
use strict;

$|=1;
my $name = "";
my $fd = syscall(279, $name, 1);
if (-1 == $fd) {
   die "memfd_create: $!";
}
print "fd $fd\n";

open(my $FH, '>&='.$fd) or die "open: $!";
select((select($FH), $|=1)[0]);

/*
perl print statements to write binary code into the memory...
*/

exec {"/proc/$$/fd/$fd"} "hello";
```

The necessary syscall number can be found in **/usr/include** by running the command **egrep -r '__NR_memfd_create|MFD_CLOEXE'** which returns the following information.

```
root@ubuntu:/usr/include# egrep -r '__NR_memfd_create|MFD_CLOEX' *
aarch64-linux-gnu/bits/syscall.h:#ifdef __NR_memfd_create
aarch64-linux-gnu/bits/syscall.h:# define SYS_memfd_create __NR_memfd_create
aarch64-linux-gnu/bits/mman-shared.h:# ifndef MFD_CLOEXEC
aarch64-linux-gnu/bits/mman-shared.h:#  define MFD_CLOEXEC 1U
asm-generic/unistd.h:#define __NR_memfd_create 279
asm-generic/unistd.h:__SYSCALL(__NR_memfd_create, sys_memfd_create)
linux/memfd.h:#define MFD_CLOEXEC                 0x0001U
```

Figure 8 Syscall number memfd_create

On the Ubuntu Server Raspberry Pi image memfd_create has a syscall number of 279 and MFDCLOEXEC is defined as 1. Now just like in the code example memfd_create can be called over a syscall in perl and the result saved in a file descriptor. The open line simply redirects the open file descriptor into a file handle by using **>&=.** The double call to select creates an autoflush on the new handle. This works by first setting autoflush to true with **$|=1** on the memfd file and then accessing the original file descriptor with **[0]** as **select** returns the previous default filehandle, which is then again fed into another **select** to reinstate the original state. Now the binary file to be injected must be compiled for the target architecture. Afterwards the binary is converted into perl print statements by calling:

```
perl -e '$/=\32;print"print \$FH pack q/H*/, q/".(unpack"H*")."/\ or die
qq/write: \$!/;\n"while(<>)' ./elfbinary
```

This creates many lines in the form of **print $FH pack q/H*/, q/...hexcode.../ or die qq/write: $!/;** which sequentially writes the binary data into the anonymous mem_fd file. Finally, after the binary data has been injected the program just has to be started with an **exec {"/proc/$$/fd/$fd"} "test";** call. Now the injected binary is running in the memory and no data has been written to the disk during the process making it an entirely fileless injection. In the case of a serious attack this could be a backdoor, rootkit, keylogger or anything similar. In this test a harmless loop performing addition was injected. The injection can be simply performed by piping the script output to a ssh section like:

```
cat perlscript.pl | ssh ubuntu@192.168.178.49 perl
```

Afterwards the injected program is directly running within an anonymous memfd file. (cf. Stuart, 2020)

4.3 Results

The results in form of log entries are shown and discussed in this chapter as well as the general workflow of further forensic analysation of any potential memory injections. Some date/time stamps are not correct due to issues of the time resetting wrongly upon boot. However, this does not matter too much as the correct sequence is still given. First of all, the logs created by the C program that monitors the history and

various usages is inspected. As the log entry is quite verbose and long it is added in Appendix 2 with important single sections shown in this chapter. The system did not see enough changes within the system usages to display anything upon the first connection, the traffic was intercepted by Wireshark though.

	Time	Source	Destination	Protocol	Length	Info
36	9.128537	192.168.178.37	192.168.178.49	TCP	66	52781 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
37	9.223021	192.168.178.49	192.168.178.37	TCP	66	22 → 52781 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1 WS=128
38	9.223137	192.168.178.37	192.168.178.49	TCP	54	52781 → 22 [ACK] Seq=1 Ack=1 Win=131328 Len=0
39	9.225148	192.168.178.37	192.168.178.49	SSHv2	87	Client: Protocol (SSH-2.0-OpenSSH_for_Windows_7.7)
40	9.232094	192.168.178.49	192.168.178.37	TCP	64	22 → 52781 [ACK] Seq=1 Ack=34 Win=64256 Len=0
41	9.278868	192.168.178.49	192.168.178.37	SSHv2	95	Server: Protocol (SSH-2.0-OpenSSH_8.2p1 Ubuntu-4ubuntu0.1)
42	9.282715	192.168.178.37	192.168.178.49	SSHv2	1374	Client: Key Exchange Init
43	9.287437	192.168.178.49	192.168.178.37	SSHv2	1110	Server: Key Exchange Init
44	9.289319	192.168.178.37	192.168.178.49	SSHv2	102	Client: Diffie-Hellman Key Exchange Init
45	9.293182	192.168.178.49	192.168.178.37	TCP	64	22 → 52781 [ACK] Seq=1098 Ack=1402 Win=64128 Len=0
46	9.336619	192.168.178.49	192.168.178.37	SSHv2	562	Server: Diffie-Hellman Key Exchange Reply, New Keys, Encrypted packet (len=228)
47	9.340901	192.168.178.37	192.168.178.49	SSHv2	70	Client: New Keys

Figure 9 Wireshark intercepting OpenSSH handshake

Figure 9 shows the start of a TCP handshake to establish the connection and afterwards an OpenSSH handshake to start the encrypted connection. The encrypted packets are useless without the encryption key, nonetheless the Source, Destination and Info fields are of interest as they show the IP and port that has connected to the system. In this case the device with the **IP 192.168.178.37** connected to the OpenSSH service on **port 22** at **192.168.178.49** which is the Ubuntu Server honeypot. It can not be deduced if the handshake was successful as the connection is fully encrypted.

The first command issued in the honeypot has been instantly logged:

--

Date: Sat Aug 29 00:41:49 2020

New commands detected:

o ps -aux

--

This shows that the intercepting of incoming commands works successfully, as it can be seen in the appendix every single command except kill -9 $$ has been logged. The reason this one is missing is the nature of the command killing the history and session, thus it cannot be logged by the used technique. Pretty much any logged comment in this system can be deemed to be a fileless attack as no activity should ever occur on the shell. Furthermore, whenever multiple commands were issued in succession the system detected changes above a threshold in memory and network usage.

Date: Sat Aug 29 00:42:29 2020

mem_total:929996 mem_free:409540 mem_avail:707300 mem_drive:29836 mem_mapped:63200 mem_shmem:4044

packet_out:98   packet_in:98

Here it can be seen that the packet traffic and memory difference were significant enough to appear suspicious and be logged. The goal of those message is to indicate that something might be going on in the system even if nothing of substance was logged or monitored. To find better thresholds that avoid false positives and fire on mostly any activity the system must be monitored for longer periods of time to observe the natural changes in an isolated environment. Due to time constraints only rough estimates have been taken in this experiment that avoided false positives observed over a span of half an hour.

As the log indicates that the **etc/passwd** file has been edited those changes should be recorded by the auditd daemon. The command **ausearch -f /etc/passwd** displays any logs created for any events that are associated with the given path.

```
ubuntu@ubuntu:/home/lime$ sudo ausearch -f /etc/passwd
----
time->Sat Aug 29 00:42:47 2020
type=PROCTITLE msg=audit(1598661767.687:146): proctitle=6E616E6F002F6574632F706173737764
type=PATH msg=audit(1598661767.687:146): item=1 name="/etc/passwd" inode=74922 dev=b3:02 mode=0100644 ouid=0
metype=NORMAL cap_fp=0 cap_fi=0 cap_fe=0 cap_fver=0 cap_frootid=0
type=PATH msg=audit(1598661767.687:146): item=0 name="/etc/" inode=61 dev=b3:02 mode=040755 ouid=0 ogid=0 rde
ENT cap_fp=0 cap_fi=0 cap_fe=0 cap_fver=0 cap_frootid=0
type=CWD msg=audit(1598661767.687:146): cwd="/home/ubuntu"
type=SYSCALL msg=audit(1598661767.687:146): arch=c00000b7 syscall=56 success=yes exit=3 a0=ffffffffffffff9c a
1 a3=1b6 items=2 ppid=1645 pid=1646 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts
exe="/usr/bin/nano" key="passwd"
ubuntu@ubuntu:/home/lime$
```

Figure 10 ausearch -f /etc/passwd log results

As it can be seen in figure 10 auditd successfully logged the write changes to the directory. The **exe="/usr/bin/nano"** path specifies which program has been used to edit the file. In **cdw="/home/ubuntu"** the current working directory at that given point is displayed. Various other details are also given like the **pid**, **ppid**, if the call was successful, the syscall used in the event, etc.

Similarly, the memfd injection can be specifically searched for by writing ausearch -k mem_injection with -k specifying the tag associated with the rule. Figure 11 reveals that a memory injection indeed happen using **/usr/bin/perl**. The syscall 279 is **memfd_create** as earlier stated and **success=yes** indicates the attacker successfully used the call. The call to the perl script has not been logged by the other modules because it was directly piped into perl upon starting the session.

```
time->Sat Aug 29 00:53:55 2020
type=PROCTITLE msg=audit(1598662435.392:187): proctitle="perl"
type=SYSCALL msg=audit(1598662435.392:187): arch=c00000b7 syscall=279 success=yes exit=3 a0=aaaaea40a0f0 a1=1
 a2=1 a3=4403 items=0 ppid=1839 pid=1840 auid=1000 uid=1000 gid=1000 euid=1000 suid=1000 fsuid=1000 egid=1000
 sgid=1000 fsgid=1000 tty=(none) ses=8 comm="perl" exe="/usr/bin/perl" key="mem_injection"
```

Figure 11 auditd systemcall log for memfd_create

As stated in chapter 2.4 memfd injection can potentially be seen by listing the directory **ls -alR /proc/*/exe 2> /dev/null | grep memfd:.*\(deleted\)** which goes through all directories listed under /proc and then prints the respective exe directory. If any result contains the string that is typical for memfd files the result is printed**.**
(cf. Sandfly Security, 2020)

Figure 12 shows the use of the command on the honeypot after the attack.

```
ubuntu@ubuntu:~$ ls -alR /proc/*/exe 2> /dev/null | grep memfd:.*\(deleted\)
lrwxrwxrwx 1 ubuntu          ubuntu          0 Aug 29 00:53 /proc/1840/exe ->
/memfd: (deleted)
```

Figure 12 console forensics to show memfd files

Just as described in chapter 2.4 there is also other potential ways to investigate for this kind of injection, in very sophisticated cases it is necessary to analyse the memory dump from the get-go. After finding indications of a successful indication a memory dump of the machine should be created with LiME. This memory dump can later be analysed in volatility after creating a fitting profile. This can either be done on the machine itself or a similar machine which uses the same operating system and architecture.

First dwarfdump[67] has to be installed, which in turn installs a file called **module.dwarf**. This file needs to be zipped together with the **system.map** file that can be found in /boot. The following command sums this up:

```
sudo zip ./<zip name>.tip <module.dwarf path> <system.map path>
```

This profile can now be loaded with volatility to analyse a memory dump created on a similar system. The dump could sadly not be analysed as there seems to be an issue with the current **module.dwarf** program being used for Ubuntu Server Raspberry Pi. The following error message appears upon loading the profile and there does not seem to be a fix at the given time. (cf. Huebotter, 2019)

[67] dwarfdump dumps DWARF ELF object debug information http://manpages.ubuntu.com/manpages/xenial/man1/dwarfdump.1.html

```
  File "/home/forensic/volatility/volatility/dwarf.py", line 163, in feed_line
    self.process_statement(**parsed) #pylint: disable-msg=W0142
  File "/home/forensic/volatility/volatility/dwarf.py", line 237, in process_statement
    self.id_to_name[statement_id] = [self.base_type_name(data)]
  File "/home/forensic/volatility/volatility/dwarf.py", line 126, in base_type_name
    return self.tp2vol[data['DW_AT_name'].strip('"')]
KeyError: '__int128 unsigned'
```

Figure 13 module.dwarf error upon loading the profile

5 Discussion / Conclusion

5.1 Result Interpretation

In this chapter the results of the thesis are discussed and interpreted. First a concept was proposed how a system needs to be designed to detect fileless attacks of advanced persistent actors. The concept is based upon the theoretical research posed in chapter 2 and is mostly based on the factor that fileless malware is very difficult to detect but using an isolated environment as a honeypot simplifies the process as almost any activity is suspicious. Due to the broad nature of fileless attacks it is not possible to present all possible attack patterns and vectors, rather the common occurrences were discussed as well as advanced techniques such as memory injection which are a key factor to inject malicious binaries without any drive contact. While memory injection has been widespread in windows operating systems for longer times, linux memory injection is a rather novel trend. The main question when it comes to the concept is how the single honeypot points can be implemented and if the modules they encompass are effective. Because of the complexity of the system the implementation focuses on the first layer which deploys easy to exploit ARM Raspberry Pi honeypot devices as well as capabilities to sniff the network. A small-scale attack was performed to evaluate the implementation of the honeypot which encompasses multiple modules to monitor various characteristics. The honeypot produced various logs during the attack and showed to effectively log all entered bash commands as well as indicate suspicious activity by comparing the memory, network and CPU usage periodically. The usage monitoring is somewhat inconsistent though as even the isolated system runs background programs and is attached to a network. The log entries remained empty without activity. However, they also did not always trigger during the attack. This shows that the thresholds are most likely too high in the current configuration and need to be adjusted during further analysis of running the system while monitoring the usages. Some commands and activity triggered those modules and the program logged the usage with a time stamp. Changes on directories and files monitored by the autitd daemon were also successfully and reliably logged. The implication is that this setup can be successfully used to intercept potential malicious activity coming from a SSH connection to monitor attackers. Furthermore, the autitd daemon also monitored system call activity. An attempt to use system calls to perform memory injection was logged as well, meaning it is possible to detect advanced injection techniques using this service. It is questionable though if the proposed implementation is advanced enough to capture truly sophisticated and advanced attackers. The system first needs to be finalized by fixing the omitted steps that showed to be problematic and then be tested in a harsher environment.

Compared to the honeycloud paper which also focused on detecting fileless malware on IoT devices, the approach was fundamentally different here. In the paper it is specifically stated that the setup is not suitable to detect any APT threats as the system is designed around restarting itself after infections and monitoring shell and SSH parameters. In this thesis the concept is created with the idea in mind of running the honeypot platform for long period of times to evaluate the movement and techniques of advanced persistent threats and their usage of fileless malware. The difference mostly consists of the honeypots being deployed within a production network rather than being directly connected to the internet, which only makes them accessible after an already occurring infiltration. This way uninteresting ordinary occurring malware is very unlikely to infect the system. Furthermore, the honeypots of the systems are designed to monitor memory injection techniques which are one of the key techniques to success in executing malicious binary code completely fileless. In this sense the approach is an extension of the honeycloud system to monitor more intrusive and different approaches. In addition, the simulation of the attack shows that the proposed taxonomy of the cited paper is lacking a category for more advanced fileless malware that utilizes memory injection. (cf. Fan, et al., 2019)
Memory injection is especially important when it comes to monitoring APT as they do not only utilize complex techniques but usually have goals in mind and try to persist on a system after infiltration if possible (cf. Alshamrani, Myneni, Chowdhary, & Huang, 2019). After such an injection attack is detected the honeypot system is memory dumped and then investigated with forensic memory tools.
Compared to other works that try to find solutions to detect fileless attacks on systems by understanding their nature (e.g Sanjay, Rakshith, & Akash, 2018) this work rather abuses the isolated environment characteristics to easily distinguish the malicious traffic. Because of that it is not necessary to deploy complex algorithms that first must identify if activity is malicious or not. It should be mentioned though that this approach is only suitable to monitor and detect attackers rather than securing existing systems. The intel gathered from the logging and investigations of the memory dumps can be used to improve the security of existing systems by understanding the common attack vectors and patterns of APT. For example, the generated logs show the chronological order of issued bash commands which can suggest certain attack execution flows, how much data is gathered during reconnaissance or how long it takes for attackers to start post exploitation.
Overall during the implementation it also became clear that ARM itself does not really affect fileless attacks much, the main difference is the compilation of binary code. Attackers have to cross-compile or get access to similar hardware and operating systems to create functional binaries for the target. For the honeypots it arises various issues as the compilation and instillation process can be riddled with different problems, as well as existing techniques not properly working on ARM architectures based operating systems yet. This can be seen especially on the ftrace hooking technique mentioned in the Methodology of Layer 2, as it utilizes registers only present on x86 architecture. Overall, the Architecture mostly complicates the process but does not change the

overall content of either attacks utilizing fileless attacks or defences against fileless attacks.
The first layer proves to be an effective way to provide an entry gate for attackers to further infiltrate the rest of the honeypot system. Auditd logs are extremely helpful if evaluated to potentially uncover any memory injections. This implementation lays the first stepping stone to showing that the proposed honeypots system to monitor fileless attacks from advanced persistent actors on IIoT networks has potential, although it first needs to be fully implemented to make any meaningful conclusions that fully answer the research question of this thesis. Layer 1 alone still lacks sophistication such as PLC controllers and SCADA systems to appear as a worthwhile target for malicious actors targeting IIoT networks rather than a random IoT device. Thus it is necessary to create the implementation of the remaining concept to assess the effectivity in detail. Certain limitations and new questions have appeared during the process which are discussed in the following sub chapters.

5.2 Restrictions and Limitations

Following are the restrictions that clarify the scope of the thesis. Naming all possible attack vectors and techniques is not part of this thesis, as the scope is way too broad and it is almost impossible to narrow down fileless attacks entirely due to the nature of the various tools, commands and injection techniques in existence. Creating a sophisticated attack is also not part of this work due to the complexity and early stages of the actual implementation.

Due to the complexity, time constraints and broad range of the topic itself certain limitations exist for this work. First, only the first Layer was implemented and only for testing purposes, not in a real production environment. Some of the limitations of the current layer 1 are the lack of stealth and single point of errors. Due to the incompatibility of most rootkit techniques the running monitoring programs, log files and auditd daemon are visible to an attacker who can simply shut them down to avoid being logged. This issue is not easy to tackle as the honeypot system exposes root functionalities. Also, the issues during compilation / cross compilation hinder the deployment of altered software like a modified dropbear server for interception. Due to this important information about the SSH connection itself rather than just the transmitted commands are lacking in the current state. As most hooking techniques to monitor or modify system calls are usually only documented under x86 and x86_64 architectures it is not yet possible to implement those in addition to the audit daemon. This leads to a single point of failure, even with rootkit capabilities the failure of the auditd component means a lack of awareness when system calls are being issued for injection techniques. Missing those injection techniques would be critical as they are part of the sophisticated nature of advanced fileless attacks. It also cannot yet be accurately said how effective the whole concept is till the whole implementation is fully done and tested. Furthermore, the current implementation only intercepts the BASH history, but

most systems have multiple different shells in use. An attacker can bypass the shell history logging by switching to a different shell after gaining access. Lastly, dwarfdump seems to have an issue with the current build of Ubuntu Server for Raspberry Pi and does not work properly to create Volatility profiles, which makes the investigation of the LiME dumps not possible. This is a major problem as complex techniques that inject malicious code into RAM are one of the key factors of this thesis and fileless attacks in general. Not being able to investigate the injected code properly leads to a lack of understanding the monitored behaviour properly. This problem leads to the next sub chapter which discusses future possibilities and newly opened questions about the thematic.

5.3 Future Work

The encountered problems lead to new questions that need to be answered before the actual research question of this thesis can be fully answered rather than just partially. First there seems to be various issues for compiling software from source on IoT Linux architectures even though they are or at least should be supported. The configuration, compilation and installation process must be analysed and a proper solution and workflow established to fix such issues for further deployments of exploitable or modified software. Furthermore, the lack of working hooking techniques on ARM processor architecture opens the question about how feasible rootkit techniques are. Existing methods must be either adjusted to support the architecture or if not possible, new ones researched to make it possible to hide monitoring modules and log files from any potential attacker. Most importantly, the other two layers must be fully implemented so the whole system can be assessed. As stated earlier, the concept is designed to support any kind of honeypots, so there are many different possibilities to extend the base concept to suit any specific needs. However, operating systems and architectures differ quite a bit, thus it is necessary to create each new additional honeypot system from ground up. For those similar modules must be developed that can monitor usage and system calls on different operating systems. The proposed software solutions and possible implementations from the Methodology chapter for Windows also need to be evaluated and fully implemented, as windows process and memory injection and PowerShell usage make fileless attacks differ quite a bit. In addition, Windows has additional attack vectors such as WMI and the registry, making it necessary to create a broader range of modules utilizing concepts discussed in the methodology. Especially the honeypot server which acts as a central entity within the system to control all honeypot stations needs to be implemented. After the server is implemented the honeypots should be adjusted to communicate monitored activity. SCADA systems need to be deployed and proper modules developed on the workstation honeypot to ensure a good range of monitoring specifically catered to protocols such as Modbus. The SCADA systems are important to properly attract attackers that aim to infiltrate industrial networks and gather intel about their used techniques to achieve this. Layer

1 can also be further developed to monitor more sophisticated parameters such as all currently running services as well as their memory characteristics.

5.4 Conclusion

To summarize, first the necessary background knowledge about honeypots, fileless malware and specifically injection techniques for post exploitation was researched and stated. This knowledge was used to build up a Methodology leading to a multi layered honeypot system concept. The concept is deployed as an isolated system in which most activity can be deemed malicious and thus easily analysed. The goal is to monitor and analyse fileless attacks from advanced threats targeting IIoT networks. To assess the concept partially the first layer was implemented and then evaluated by simulating a small-scale attack. The implemented methods proved to be effective, although the implementation is still lacking sophistication to be considered fully effective. Finally, the results were discussed and connections to current research was drawn.

Bibliography

Adrian, H. (2019). Fileless Malware and Process Injection in Linux: (Linux post-exploitation from a blue-teamer's point of view) [Conference session]. hack.lu 2019, Luxembourg. Retrieved July 24, 2020, from http://archive.hack.lu/2019/Fileless-Malware-Infection-and-Linux-Process-Injection-in-Linux-OS.pdf

Alshamrani, A., Myneni, S., Chowdhary, A., & Huang, D. (2019). A Survey on Advanced Persistent Threats: Techniques Solutions, Challenges, and Research Opportunities. *IEEE Communications Surveys & Tutorials vol. 21, no. 2*, 1851-1877. doi:10.1109/COMST.2019.2891891

Axelsson, S. (2000). Intrusion Detection Systems: A Survey and Taxonomy. Unpublished manuscript, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden. Retrieved July 25, 2020, from http://www1.cs.columbia.edu/~locasto/projects/candidacy/papers/axelsson00intrusion.pdf

Baliga, A., Ganapathy, V., & Iftode, L. (2011). Detecting Kernel-Level Rootkits Using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing, vol. 8, no. 5*, 670-684. doi:10.1109/TDSC.2010.38

Buddy, T. (2019, 26 April). *Hunting for Ghosts in Fileless Attacks.* Retrieved July 24, 2020, from SANS Institute Information Security Reading Room: https://www.sans.org/reading-room/whitepapers/malicious/hunting-ghosts-fileless-attacks-38960

Campbell, R. M., Padayachee, K., & Masombuka, T. (2015). A Survey of Honeypot Research: Trends and Opportunities. The 10th International Conference for Internet Technology and Secured Transactions. *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)* (pp. 208-212). London: IEEE. doi:10.1109/ICITST.2015.7412090

Chester, A. (2017, April 19). *Linux ptrace introduction AKA injecting into sshd for fun [Blog post]*. Retrieved July 2020, 24, from https://blog.xpnsec.com/linux-process-injection-aka-injecting-into-sshd-for-fun/

Desimone, J. (2019, June 13). *Hunting In Memory [Blog post]*. Retrieved July 2020, 24, from https://www.elastic.co/blog/hunting-memory

Doffman, Z. (2019, September 14). *The IIoT Threat Landscape: Securing Connected Industries. Forbes*. Retrieved July 20, 2020, from https://www.forbes.com/sites/zakdoffman/2019/09/14/dangerous-cyberattacks-on-iot-devices-up-300-in-2019-now-rampant-report-claims/

Fan, D., Zhenhua, L., Yunhao, L., Ennan, Z., Qi, A. C., Tianyin, X., . . . Jingyu, Y. (2019). Understanding Fileless Attacks on Linux-based IoT Devices with HoneyCloud. *Conference on Mobile Systems, Applications and Services (MobiSys19)*, (pp. 482-493). Seoul. Retrieved August 30, 2020, from https://www.ics.uci.edu/~alfchen/fan_mobisys19.pdf

GNU. (29, August 2020). *Bash Reference Manual*. Retrieved from https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Bash-History-Facilities

Graeber, M. (2015). *Abusing Windows Management Instrumentation (WMI) to Build a Persistent, Asyncronous, and Fileless Backdoor.* Retrieved July 23, 2020, from https://www.blackhat.com/docs/us-15/materials/us-15-Graeber-Abusing-Windows-Management-Instrumentation-WMI-To-Build-A-Persistent%20Asynchronous-And-Fileless-Backdoor-wp.pdf

Grimes, R. A. (2019, May 1). *9 types of malware and how to recognize them.* Retrieved July 21, 2020, from https://www.csoonline.com/article/2615925/security-your-quick-guide-to-malware-types.html

Grubb, S. (2018, Aug). *auditctl(8) - linux man-pages*. Retrieved August 16, 2020, from https://www.man7.org/linux/man-pages/man8/auditctl.8.html

Grubb, S. (2019, Jan). *audit.rules(7) - linux man-pages*. Retrieved August 2020, 29, from https://www.man7.org/linux/man-pages/man7/audit.rules.7.html

Hosseini, A. (2018, July 18). *Ten process injection techniques : A technical survey of common and trending process injection techniques [Blog post]*. Retrieved July 24, 2020, from https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process

Huebotter, G. (2019, November 2). *Creating A Volatility Profile For Linux [Blog post]*. Retrieved August 2020, 29, from https://www.introverted-analyst.com/creating-a-volatility-profile-for-linux/>,

Johnson, B., Caban, D., Krotofil, M., Scali, D., Brubaker, N., & Christopher, G. (2017, December 14). *Attackers Deploy New ICS Attack Framework "TRITON" and Cause Operational Disruption to Critical Infrastructure*. Retrieved July 20, 2020, from https://www.fireeye.com/blog/threat-research/2017/12/attackers-deploy-new-ics-attack-framework-triton.html

Kaspersky GREAT Global Research & Analysis Team. (2017). *Fileless attacks against enterprise networks.* Retrieved July 20, 2020, from https://media.kaspersky.com/en/business-security/fileless-attacks-against-enterprise-networks.pdf

Kennel, D. (2018, September 15). *All-Seeing Eye or Blind Man? Understanding the Linux Kernel Auditing System.* Retrieved August 15, 2020, from SANS Institute Information Security Reading Room: https://www.sans.org/reading-room/whitepapers/linux/all-seeing-eye-blind-man-understanding-linux-kernel-auditing-system-38605

Kerrisk, M. (2020, June 09). *memfd_create(2) - linux man-pages*. Retrieved July 24, 2020, from https://man7.org/linux/man-pages/man2/memfd_create.2.html

Kerrisk, M. (2020, August 13). *proc(5) - linux man-pages*. Retrieved August 15, 2020, from https://man7.org/linux/man-pages/man5/proc.5.html

Liberman, T., & Kogan, E. (2017). Lost in Transaction: Process Doppelgänging. Black Hat Europe, London. Retrieved July 7, 2020, from https://www.blackhat.com/docs/eu-17/materials/eu-17-Liberman-Lost-In-Transaction-Process-Doppelganging.pdf

LinuxSecurityFreak; Pablo R. (2010, August 26). *Reply to question "Preserve bash history in multiple terminal windows*. Retrieved August 29, 2020, from Stackexchange: https://unix.stackexchange.com/a/1292

Lozovsky, A., & Stepanchuk, S. (2018, July 05). *Hooking Linux Kernel Functions, Part 2: How to Hook Functions with Ftrace [Blog post].* Retrieved August 15, 2020, from https://www.apriorit.com/dev-blog/546-hooking-linux-functions-2

Lubos, R. (2020, May 13). *Ubuntu 20.04: Connect to WiFi from command line [Blog post].* Retrieved August 29, 2020, from https://linuxconfig.org/ubuntu-20-04-connect-to-wifi-from-command-line

Microsoft. (n.d.). *FileSystemWatcher Class*. Retrieved August 29, 2020, from https://docs.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher?view=netcore-3.1

Microsoft PowerShell Team. (2015, June 9). *PowerShell ♥ the Blue Team [Blog post].* Retrieved August 16, 2020, from https://devblogs.microsoft.com/powershell/powershell-the-blue-team/

Naik, N., & Jenkins, P. (2018). Discovering Hackers by Stealth: Predicting Fingerprinting Attacks on Honeypot Systems. *2018 IEEE International Systems Engineering Symposium (ISSE)* (pp. 1-8). Rome: IEEE. doi:10.1109/SysEng.2018.8544408

Nawrocki, M., Wählisch, M., Schmidt, T., Keil, C., & Schönfelder, J. (2016). A Survey on Honeypot Software and Data Analysis. *arXiv e-prints*, arXiv:1608.06249. Retrieved July 25, 2020, from https://ui.adsabs.harvard.edu/abs/2016arXiv160806249N/abstract

Peng, L. (2016, August 6). *Setting up centralized logging with auditd [Blog post].* Retrieved August 16, 2020, from https://luppeng.wordpress.com/2016/08/06/setting-up-centralized-logging-with-auditd/

Perez, C. (2017, October 18). *Sysinternals Sysmon 6.10 Tracking of Permanent WMI Events [Blog post]*. Retrieved July 25, 2020, from https://www.darkoperator.com/blog/2017/10/15/sysinternals-sysmon-610-tracking-of-permanent-wmi-events

Remillano II, A., & Malagad, R. (2019, May 07). *CVE-2019-3396: Exploiting the Confluence Vulnerability [Blog post].* Retrieved August 11, 2020, from https://blog.trendmicro.com/trendlabs-security-intelligence/cve-2019-3396-redux-confluence-vulnerability-exploited-to-deliver-cryptocurrency-miner-with-rootkit

Sandfly Security. (2020, July 9). Retrieved July 24, 2020, from Detecting Linux memfd_create() Fileless Malware with Command Line Forensics: https://www.sandflysecurity.com/blog/detecting-linux-memfd_create-fileless-malware-with-command-line-forensics/

Sanjay, B. N., Rakshith, D. C., & Akash, R. B. (2018). An Approach to Detect Fileless Malware and Defend its Evasive mechanisms. *2018 3rd International Conference on Computational Systems and Information Technology for Sustainable Solutions (CSITSS)* (pp. 234-239). Bengaluru: IEEE. doi:10.1109/CSITSS.2018.8768769

Sikorski, M., & Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* San Francisco: No Starch Press.

Sisinni, E., Saifullah, A., Han, S., Jennehag, U., & Gidlund, M. (2018). Industrial Internet of Things: Challenges, Opportunities, and Directions. *IEEE Transactions on Industrial Informatics, no. 11. 14(11)*, pp. 4724-4734. IEEE. doi:10.1109/TII.2018.2852491

Smith, R. F. (2017, September 25). *The Most Important Linux Files to Protect (and How) [Blog post]*. Retrieved August 15, 2020, from https://www.beyondtrust.com/blog/entry/important-linux-files-protect

Snort. (2016). *SNORTOLOGY 101: THE ANATOMY OF A SNORT RULE.* Retrieved August 3, 2020, from https://snort-org-site.s3.amazonaws.com/production/document_files/files/000/000/116/original/Snort_rule_infographic.pdf

SOPHOS. (2018, March). Retrieved August 1, 2020, from Exploits Explained: Comprehensive Exploit Prevention: https://www.sophos.com/en-us/medialibrary/Gated-Assets/white-papers/Sophos-Comprehensive-Exploit-Prevention-wpna.pdf

Stuart. (2020, March 31). *In-Memory-Only ELF Execution (Without tmpfs) [Blog post]*. Retrieved July 24, 2020, from https://magisterquis.github.io/2018/03/31/in-memory-only-elf-execution.html

Sudhakar, & Kumar, S. (2020). An emerging threat Fileless malware: a survey and research challenges. *Cybersecurity 3*, Article 1. doi:https://doi.org/10.1186/s42400-019-0043-x

Sylve, J. (2020, Jul 7). *504ENSICS Labs LiME Documentation*. Retrieved August 29, 2020, from https://github.com/504ensicsLabs/LiME/tree/master/doc

Tasoulas, V. (2014, April 29). *Reply to question "Accurate calculation of CPU usage given in percentage in Linux?"*. Retrieved August 29, 2020, from https://stackoverflow.com/a/23376195

Trend Micro. (2020, March 18). *The IIoT Threat Landscape: Securing Connected Industries*. Retrieved July 20, 2020, from https://www.trendmicro.com/vinfo/de/security/news/internet-of-things/the-iiot-threat-landscape-securing-connected-industries

United States General Accounting Office. (2002, July). *CRITICAL INFRASTRUCTURE PROTECTIONFederal Efforts Require a More Coordinated and Comprehensive Approach for Protecting Information Systems.* Retrieved July 20, 2020, from https://www.gao.gov/new.items/d02474.pdf

Wueest Candid, A. H. (2017). *Internet Security Threat Report: Living off the land and fileless attack techniques.* Mountain View, California: Symantec. Retrieved July 24, 2020, from https://www.infopoint-security.de/media/symantec-istr-living-off-the-land-and-fileless-attack-techniques-en.pdf

Appendices

Appendix 1 .... A-I

Appendix 2 .... A-III

Appendix 1

The following is the C code implementation of the modules that monitor the usage of network, memory, cpu, number of processes and bash history.

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argC, char *argV[])
{

   struct stat buffer;
   time_t rawtime;
   const char delim[2] = " ";
   char *seperate;
   char ignore[100];
   int status;
   bool tag_print1 = false, tag_print2 = false, tag_print3 = false, tag_print4 =
false, tag_print5 = false, tag_print6 = false;
   bool first = true;

   //Memory Usage

   char mem_str[100];
   int mem_total = 0, mem_free = 0, mem_avail = 0, mem_drive = 0,
mem_mapped = 0, mem_shmem = 0;
   int old_mem_free = 0, old_mem_avail = 0, old_mem_drive = 0,
old_mem_mapped = 0, old_mem_shmem = 0;

   //Network Usage

   char net_str[100];
   int packet_out = 0, packet_in = 0;
   int old_packet_out = 0, old_packet_in = 0;

   //CPU Usage

   bool old = false;
   char cpu_str[100];
   float cpu_perc = 0;
   float old_cpu_perc = 0;
   float total_period = 0, usage_period = 0;
   int i, j = 0;
   int usage = 0, old_usage = 0;
```

```
int total = 0, old_total = 0;
int proc_numtotal = 0, proc_running = 0;
int old_proc_running = 0, old_proc_numtotal = 0;

//BASH History

char bash_str[100];
int size_history = 0, size_root_history = 0;
int old_size_history = 0, old_size_root_history = 0;
int rem_line = 0, rem_root_line = 0;
int line = 0;

while (1)
{
   //Memory Usage

   FILE *mem_info = fopen("/proc/meminfo", "r");

   if (mem_info == NULL)
   {
      exit(0);
   }

   for (j = 0; j < 21; j++)
   {
      fgets(mem_str, 100, mem_info);
      seperate = strtok(mem_str, delim);
      seperate = strtok(NULL, delim);
      switch (j + 1)
      {
         case 1:
            mem_total = atoi(seperate);
            break;
         case 2:
            mem_free = atoi(seperate);
            break;
         case 3:
            mem_avail = atoi(seperate);
            break;
         case 4:
            mem_drive = atoi(seperate);
            break;
         case 20:
            mem_mapped = atoi(seperate);
            break;
         case 21:
            mem_shmem = atoi(seperate);
            break;
      }
   }

   fclose(mem_info);
   tag_print1 = false;

   if (mem_free - old_mem_free > 150 || mem_avail - old_mem_avail > 150
|| mem_drive - old_mem_drive > 25 || mem_mapped - old_mem_mapped > 25
|| mem_shmem - old_mem_shmem > 0)
```

```
{
   tag_print1 = true;
}

old_mem_free = mem_free;
old_mem_avail = mem_avail;
old_mem_drive = mem_drive;
old_mem_mapped = mem_mapped;
old_mem_shmem = mem_shmem;

//Network Usage

FILE *net_info = fopen("/proc/net/dev", "r");

if (net_info == NULL)
{
   exit(0);
}

for (j = 0; j < 5; j++)
   fgets(net_str, 100, net_info);
fgets(net_str, 100, net_info);
fclose(net_info);
seperate = strtok(net_str, delim);

for (j = 0; j < 10; j++)
{
   seperate = strtok(NULL, delim);

   //At position 2 are all outgoing packets recorded
   if (j == 1)
      packet_out = atoi(seperate);
}

//At position 10 are all ingoing packets recorded
packet_in = atoi(seperate);
tag_print2 = false;

if (packet_in - old_packet_in > 1 || packet_out - old_packet_out > 1)
{
   tag_print2 = true;
}

old_packet_in = packet_in;
old_packet_out = packet_out;

//CPU Usage and processes

FILE *cpu_info = fopen("/proc/stat", "r");
if (cpu_info == NULL)
{
   exit(0);
}

fgets(cpu_str, 100, cpu_info);
seperate = strtok(cpu_str, delim);
```

```
    i = 0;
    while (seperate != NULL)
    {
       seperate = strtok(NULL, delim);
       if (i >= 8 || i < 0) break;
       if (i < 3)
       {
          usage = usage + atoi(seperate);
       }

       total = total + atoi(seperate);

       i++;
    }

    if (old_total > 0 && old_usage > 0)
    {
       usage_period = usage - old_usage;
       total_period = total - old_total;
       cpu_perc = (usage_period / total_period) *100;

       if (total_period > 0)
       {
          if (cpu_perc - old_cpu_perc > 0.25 || old_cpu_perc - cpu_perc < -
0.25)
          {
             tag_print3 = true;
          }
          old_cpu_perc = cpu_perc;
       }
    }

    //Get actively running and total processes

    for (j = 0; j < 11; j++)
       fgets(cpu_str, 100, cpu_info);
    strtok(cpu_str, delim);
    seperate = strtok(NULL, delim);
    proc_numtotal = atoi(seperate);
    fgets(cpu_str, 100, cpu_info);
    strtok(cpu_str, delim);
    seperate = strtok(NULL, delim);
    proc_running = atoi(seperate);
    fclose(cpu_info);
    tag_print4 = false;

    if (proc_numtotal - old_proc_numtotal > 2 || proc_running - old_proc_run-
ning > 1)
    {
       tag_print4 = true;
    }

    old_proc_numtotal = proc_numtotal;
    old_proc_running = proc_running;
    old_usage = usage;
    old_total = total;
```

```
//BASH History

status = stat("/home/ubuntu/.bash_history", &buffer);
if (status == 0)
{
   size_history = buffer.st_size;
}

status = stat("/root/.bash_history", &buffer);
if (status == 0)
{
   size_root_history = buffer.st_size;
}

line = 0;

if (old_size_history != size_history)
{
   tag_print5 = true;
}

line = 0;

if (old_size_root_history != size_root_history)
{
   tag_print6 = true;
}

//If any changes detected print them

if ((tag_print1 || tag_print2 || tag_print3 || tag_print4 || tag_print5 ||
tag_print6) && first == false)
{
   time(&rawtime);
   FILE *log_file = fopen("/home/log.txt", "a+");
   fprintf(log_file, "Date: %s", ctime(&rawtime));
   if (tag_print1)
      fprintf(log_file, "mem_total:%i mem_free:%i mem_avail:%i
mem_drive:%i mem_mapped:%i mem_shmem:%i\n", mem_total, mem_free,
mem_avail, mem_drive, mem_mapped, mem_shmem);
   if (tag_print2)
      fprintf(log_file, "packet_out:%i   packet_in:%i\n", packet_out, pa-
cket_in);
   if (tag_print3)
      fprintf(log_file, "Usage of cpu = %.2f%%\n", cpu_perc);
   if (tag_print4)
      fprintf(log_file, "proc_numtotal:%i  proc_running:%i\n", proc_numto-
tal, proc_running);
   if (tag_print5)
   {
      line = 0;
      FILE *history_info = fopen("/home/ubuntu/.bash_history", "r");
      fprintf(log_file, "\nNew commands detected:\n");
      while (fgets(bash_str, 100, history_info) != NULL)
      {
         if (line >= rem_line)
            fprintf(log_file, "o %s", bash_str);
```

```
            line++;
         }
         fflush(log_file);
         fclose(history_info);
         old_size_history = size_history;
         rem_line = line;
      }
      if (tag_print6)
      {
         line = 0;
         FILE *history_info = fopen("/root/.bash_history", "r");
         fprintf(log_file, "\nNew root commands detected:\n");
         while (fgets(bash_str, 100, history_info) != NULL)
         {
            if (line >= rem_root_line)
               fprintf(log_file, "o %s", bash_str);
            line++;
         }
         fflush(log_file);
         fclose(history_info);
         old_size_root_history = size_root_history;
         rem_root_line = line;
      }

      fprintf(log_file, "\n----------------------------------------------------\n\n");
      fclose(log_file);
   }
   tag_print1 = false;
   tag_print2 = false;
   tag_print3 = false;
   tag_print4 = false;
   tag_print5 = false;
   tag_print6 = false;
   sleep(5);
   if (first == true)
      first = false;
  }
}
```

Appendix 2

This are the logs that were generated by the c monitoring program during the attack simulation. They all have a timestamp and the whole duration was recorded.

Date: Sat Aug 29 00:41:49 2020

New commands detected:
o ps -aux

Date: Sat Aug 29 00:42:09 2020

New commands detected:
o cat /etc/passwd

Date: Sat Aug 29 00:42:14 2020
mem_total:929996 mem_free:411116 mem_avail:708144 mem_drive:29804
mem_mapped:62092 mem_shmem:4044

New commands detected:
o nano /etc/passwd

Date: Sat Aug 29 00:42:29 2020
mem_total:929996 mem_free:409540 mem_avail:707300 mem_drive:29836
mem_mapped:63200 mem_shmem:4044
packet_out:98 packet_in:98

Date: Sat Aug 29 00:42:49 2020
mem_total:929996 mem_free:409792 mem_avail:707584 mem_drive:29860
mem_mapped:62116 mem_shmem:4044

New commands detected:
o sudo nano /etc/passwd

Date: Sat Aug 29 00:45:04 2020
proc_numtotal:1658 proc_running:1

Date: Sat Aug 29 00:45:49 2020
mem_total:929996 mem_free:412816 mem_avail:710656 mem_drive:29900
mem_mapped:62124 mem_shmem:4044

Date: Sat Aug 29 00:47:14 2020
mem_total:929996 mem_free:413068 mem_avail:710912 mem_drive:29916
mem_mapped:62124 mem_shmem:4044
proc_numtotal:1661 proc_running:3

New commands detected:
o cat /etc/shadow

Date: Sat Aug 29 00:47:19 2020
packet_out:102   packet_in:102
proc_numtotal:1664 proc_running:2

New commands detected:
o sudo cat /etc/shadow

Date: Sat Aug 29 00:47:39 2020
mem_total:929996 mem_free:414076 mem_avail:711948 mem_drive:29932
mem_mapped:62132 mem_shmem:4044

New commands detected:
o nano cat /etc/shadow

Date: Sat Aug 29 00:47:44 2020
mem_total:929996 mem_free:413068 mem_avail:710948 mem_drive:29940
mem_mapped:63256 mem_shmem:4044
packet_out:106 packet_in:106

Date: Sat Aug 29 00:48:54 2020
mem_total:929996 mem_free:413320 mem_avail:711240 mem_drive:29972
mem_mapped:62144 mem_shmem:4044

New commands detected:
o sudo nano /etc/shadow

Date: Sat Aug 29 00:49:04 2020
mem_total:929996 mem_free:413572 mem_avail:711500 mem_drive:29980
mem_mapped:62144 mem_shmem:4044

New commands detected:
o uname -a

Date: Sat Aug 29 00:49:09 2020

New commands detected:
o lsb_release -d

Date: Sat Aug 29 00:49:14 2020

New commands detected:
o cat /etc/os-release
o cat /etc/issue

Date: Sat Aug 29 00:49:24 2020

New commands detected:
o cat /etc/debian_release

Date: Sat Aug 29 00:49:29 2020
mem_total:929996 mem_free:413572 mem_avail:711532 mem_drive:30012
mem_mapped:62144 mem_shmem:4044

New commands detected:
ob

Date: Sat Aug 29 00:50:24 2020

New commands detected:
o cat /etc/crontab

Date: Sat Aug 29 00:50:34 2020
packet_out:110   packet_in:110

New commands detected:
o sudo -l

Date: Sat Aug 29 00:50:59 2020
packet_out:114   packet_in:114

New commands detected:
o hostname -f

Date: Sat Aug 29 00:51:09 2020

New commands detected:
o ifconfig -a

Date: Sat Aug 29 00:51:24 2020

New commands detected:
o cat /etc/network/interfaces

Date: Sat Aug 29 00:51:34 2020

New commands detected:
o netstat -r

Date: Sat Aug 29 00:51:49 2020

New commands detected:
o cat /proc/net/*

Date: Sat Aug 29 00:52:09 2020
mem_total:929996 mem_free:416156 mem_avail:714492 mem_drive:30112
mem_mapped:61000 mem_shmem:4036

Date: Sat Aug 29 00:52:19 2020
mem_total:929996 mem_free:415328 mem_avail:713688 mem_drive:30120
mem_mapped:61192 mem_shmem:4036

Date: Sat Aug 29 00:52:24 2020
mem_total:929996 mem_free:413580 mem_avail:711964 mem_drive:30120
mem_mapped:62192 mem_shmem:4044
proc_numtotal:1768 proc_running:3

Date: Sat Aug 29 00:52:44 2020
mem_total:929996 mem_free:412220 mem_avail:711008 mem_drive:30692
mem_mapped:62292 mem_shmem:4044

Date: Sat Aug 29 00:52:49 2020

mem_total:929996 mem_free:412192 mem_avail:711324 mem_drive:30700
mem_mapped:62196 mem_shmem:4044

Date: Sat Aug 29 00:53:04 2020
mem_total:929996 mem_free:412444 mem_avail:711600 mem_drive:30716
mem_mapped:62196 mem_shmem:4044

New commands detected:

Date: Sat Aug 29 00:53:09 2020

New commands detected:
o touch ~/.bash_history

Date: Sat Aug 29 00:53:39 2020
mem_total:929996 mem_free:415280 mem_avail:714468 mem_drive:30732
mem_mapped:61036 mem_shmem:4036

Date: Sat Aug 29 00:53:49 2020
mem_total:929996 mem_free:414664 mem_avail:713868 mem_drive:30748
mem_mapped:61292 mem_shmem:4036

Date: Sat Aug 29 00:53:54 2020
mem_total:929996 mem_free:410888 mem_avail:710080 mem_drive:30748
mem_mapped:61312 mem_shmem:4048
proc_numtotal:1806 proc_running:2

Date: Sat Aug 29 00:53:59 2020
mem_total:929996 mem_free:414420 mem_avail:713636 mem_drive:30756
mem_mapped:61168 mem_shmem:4056
proc_numtotal:1849 proc_running:2

Date: Sat Aug 29 00:54:09 2020
mem_total:929996 mem_free:411648 mem_avail:710824 mem_drive:30772
mem_mapped:61188 mem_shmem:4064
proc_numtotal:1925 proc_running:1

Date: Sat Aug 29 00:54:14 2020
mem_total:929996 mem_free:411380 mem_avail:711076 mem_drive:30780
mem_mapped:61188 mem_shmem:4064

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Kulmbach, den 31.08.2020

Christian, Roth