University of Piraeus

School of Information and Communication Technologies

Department of Digital Systems

Postgraduate Program of Studies

MSc CyberSecurity and AI Technologies

Master Thesis

YARA rules generator for Linux binaries and memory detection with Volatility 3

Supervisor Professor: Konstantinos Lambrinoudakis

Name-Surname	E-mail	Student ID.
Kyriakos Atalialis	kyriakos_atalialis@ssl-unipi.gr	MTE24007

Piraeus

21/02/2026

Acknowledgements

I would like to express my sincere gratitude to Professors Evangelos Dragonas and Konstantinos Lamprinoudakis for granting me the opportunity to deepen my knowledge of memory forensics, to explore its functionalities, and to examine the associated security threats. Their guidance and mentorship were instrumental in shaping this research. I am also profoundly thankful to my father, whose unwavering support and encouragement made this challenging endeavor possible.

Abstract

In recent years, cybersecurity attacks have become more sophisticated and complex, and even advanced researchers have struggled to implement measures that can mitigate them. Malware programs now adopt a stealthier and more careful approach than in the past.

Ransomware, rootkits, and file-less malware can execute their malicious activities directly in an operating system's memory. This makes them extremely difficult to detect, because traditional antivirus tools are fundamentally inadequate, meaning that they focus mainly on scanning files on the disk, while advanced threats bypass the file system entirely. Even with memory monitoring, antivirus software often falls short on modern attack techniques, leaving systems vulnerable to stealthy, fileless infections.

Historically fileless malware has been most prevalent on Windows systems, where attackers exploit native tools such as PowerShell and WMI to execute malicious code directly in memory. However, more recently Linux environments have also become a target, with adversaries leveraging legitimate binary commands to establish a stealthier and difficult to detect attack approaches. This evolution highlights the growing complexity of fileless techniques across multiple operating systems.

Researchers specializing in memory forensics have developed a range of tools to detect such attacks in a computer's memory. Volatility and Rekall are prominent examples. They also created pattern-matching tools called YARA rules, that can be applied to both file system and memory. YARA rules detect malware signatures from strings, hex patterns, and code snippets, helping trace malware in the wild.

This research project will cover LOLBins (Living off the Land binaries) in Linux systems. LOLBins are commonly exploited tools that enable fileless attacks. By applying YARA rules, the presence of known LOLBins can be identified in memory, which may indicate binaries that are frequently abused by attackers. These rules can assist memory forensic analysts in identifying potentially suspicious binaries in memory and prioritizing further investigation.

The YARA rules will be generated automatically by a Python-based YARA Generator program, a useful tool that creates YARA rules faster and without manual effort. Finally, the research thesis will also use Volatility 3 YARA plugin to take the rules for LOLBins detections in a Linux memory dump.

Contents

Abstract 4
Chapter 1 11
Introduction 11
1.1 Background and motivation 11
1.2 Problem statement 12
1.3 Research objectives and contributions 12
1.4 Scope and limitations 13
1.5 Thesis structure 14
Chapter 2 15
Background and Related Work 15
2.1 Linux basic architecture overview 15
2.2 Linux RAM forensics 16
2.3 LOLBins and Fileless malware 16
2.3.1 Introduction to Living-off-the-Land Binaries 16
2.3.2 Fileless malware 17
2.3 Volatility 3 and YARA fundamentals 18
2.3.1 Volatility 3 18
2.3.2 YARA rules 18
2.4 Identified research gap 19
Chapter 3 20
The Linux memory architecture overview 20
3.1 Linux Memory 20
3.2 Virtual memory 20
3.3 Paging 21
3.4 Swap space 22
3.5 Memory allocation 22
3.6 Linux commands for memory information 23

Chapter 4 24
Fileless techniques with the help of LOLBins on Linux systems 24
4.1 Common techniques of fileless malware that can be used in Linux machines ... 24
4.2 How LOLBins can be used in Fileless attacks 25
4.3 Examples of LOLBins attacks 25
4.3.1 Fileless attacks on Linux-based HoneyCloud (Fan Dang, 2019) 25
4.3.2 Poisoned Word document on Mac and Windows systems (Stokes, 2019) .. 26
4.4 The GTFOBINS project 26
4.5 Using the MITRE ATT&CK framework on Linux LOLBins 28
4.5.1 Execution 29
4.5.2 Persistence 30
4.5.3 Privilege Escalation 30
4.5.4 Defense Evasion 31
4.5.5 Credential Access 32
4.5.6 Discovery 32
4.5.7 Exfiltration 33
4.6 How can fileless attacks that use LOLBins be detected? 33
Chapter 5 35
Methodology and tools installation 35
5.1 The steps of this research thesis 35
5.2 Resources for this methodology 35
5.3 Volatility 3 installation 35
5.4 Volatility 3 plugins and their functionalities 38
5.5 Tools for YARA rules detection and basic YARA syntax 40
Chapter 6 42
Implementing the YARA Rules Generator (YRG) for ELF 42
6.1 Introduction to YARA Rules Generator (YRG) for ELF 42
6.2 YARA Rules Generator for ELF description 43

6.3 YARA Rules Generator for ELF results 45
6.4 The YARA Rules example binaries 47
Chapter 7 48
Evaluation of YRG 48
7.1 Testing rules on Ubuntu’s memory dump 48
7.2 Results 49
7.3 Findings Analysis 63
Chapter 8 77
Discussion 77
8.1 Key findings 77
8.2 Limitations 78
Chapter 9 80
Conclusion 80
9.1 Overview of this research thesis 80
9.2 Core contributions 80
9.3 Future work 81
9.4 Final remarks 81
Bibliography 83

List of Figures

Figure 1: The new version of Volatility 3 in REMnux Linux. 36
Figure 2: lime-5.15.0-139-generic.ko file was created after the make command in LiME/src directory. 37
Figure 3: Memory dump's banners 37
Figure 4: Volatility linux.pslist. 39
Figure 5: Volatility linux.psscan. 39
Figure 6: Volatility linux.psaux. 40
Figure 7: Volatility linux.bash. 40
Figure 8: Command of the YARA ELF generator 43
Figure 9: The YARA conditions based on the binary's contents (bin_token variable contains the sudo command of the binary). 45
Figure 10: The yara-rules-elf-generator.py execution. 46
Figure 11: The YARA rule for curl. 46
Figure 12: Part of Volatility 3 YARA scanning that has been printed inside a file. 48
Figure 13: YARA detection for wget. 49
Figure 14: YARA detection for umount. 50
Figure 15: YARA detection for tee. 50
Figure 16: YARA detection for sshpass. 51
Figure 17: YARA detection for ssh. 51
Figure 18: YARA detection for socat. 52
Figure 19: YARA detection for sftp. 52
Figure 20: YARA detection for scp. 53
Figure 21: YARA detection for rsync. 53
Figure 22: YARA detection for pkexec. 54
Figure 23: YARA detection for nmap. 54
Figure 24: YARA detection for nc. 55
Figure 25:YARA detection for mv. 55
Figure 26:YARA detection for mount. 56
Figure 27: YARA detection for mawk. 56
Figure 28: YARA detection for ksh. 57

Figure 29: YARA detection for git. 57
Figure 30: YARA detection for ed.. 58
Figure 31: YARA detection for diff.. 58
Figure 32: YARA detection for dd. 59
Figure 33: YARA detection for dash. 59
Figure 34: YARA detection for curl. 60
Figure 35: YARA detection for cp.. 60
Figure 36: YARA detection for busybox. 61
Figure 37: YARA detection for base64. 61
Figure 38: YARA detection for zsh. 62
Figure 39: YARA detection overall on all 30 binaries. 62
Figure 40: The nc binary exists in a running state and might be maliciously working because it opens a connection to the internet. Its existence and behavior had been found using psslist,psscan and psaux plugins of Volatility 3... 63
Figure 41: The bash plugin Volatility 3 shows that nc has been present and in a possible bash command. 64
Figure 42: The nc binary is present and in virtual memory using YARA detection (linux.vmayarascan plugin)... 64

List of Tables

Table 1: Memory allocation functions from C library 22

Table 2: Fileless malware techniques 24

Table 3: Example binaries and their functions 28

Table 4:Command and Scripting Interpreter 29

Table 5: Scheduled Task/Job 30

Table 6: Create or Modify System Process 30

Table 7: Abuse Elevation Control Mechanism 30

Table 8: Indicator Removal of Host 31

Table 9: OS Credential Dumping 32

Table 10: System Network Configuration Discovery 32

Table 11: Exfiltration Over Alternative Protocol 33

Table 12: The YARA binaries that have been created by the program. 47

Chapter 1

Introduction

1.1 Background and motivation

In the new era of digital information, devices can connect to the Internet and communicate with users via interfaces such as phones, tablets or computers. These devices make a person's everyday life easier and more productive. However, there is a risk that these smart, efficient machines might become tools for criminal activities. Amateur and professional hackers, as well as cybercriminals from rogue countries, might exploit these devices for various purposes such as leaking data, taking control or causing serious damage.

People can adopt simple proactive measures to protect themselves from these threats. Anti-virus and firewall protections, anti-spam email mechanisms and careful checking, when downloading files from malicious websites. These are common measures that users rely on, in their day-to-day lives in the new digital world. Unfortunately, these measures are not enough, as threat adversaries can bypass them in many different and clever ways. Therefore, users need cybersecurity researchers who can stop these kinds of attacks. Digital forensic investigators, incident responders and threat hunters are teams of cybersecurity experts who can mitigate and stop the adversaries' dangerous plans.

Yet the rise in IoT-enabled devices and cloud-connected technologies has driven digital investigations to grow at an unprecedented pace (David Lillis, 2016). The proliferation of devices in smart homes, connected vehicles, digital healthcare, industrial systems, and cloud-based platforms such as Google Drive, Dropbox, and AWS has created an environment conducive to a rapid escalation in illicit activities. Digital forensic investigators use two methods to extract information from criminal activities. The first method involves locating artifacts on hard drives or the latest SSDs. The second method involves analyzing volatile memory RAM (Ishrag Hamid, 2024; Andrew Case, 2017), which is a key issue in this research. In memory investigations, cybersecurity experts can find crucial information about many criminal attacks. One example is malware attacks, malicious programs that can damage and control a system. Every year, malware becomes more dangerous and stealthier in computer memory. Cybersecurity researchers should have

the necessary tools to battle these malicious programs, as they become increasingly sophisticated within the volatile memory.

1.2 Problem statement

Despite numerous recent advances in volatile-memory malware detection, the challenge remains unresolved. Fileless malware, a family of malware that exploits existing legitimate processes, which resides within the operating system (all these group of processes are called Living Off the Land Binaries or LOLBins and for scripts, Living Off the Land Scripting or LOLScripts) (Sudhakar, 2020). Fileless malwares are living inside RAM. That's why they are difficult for cybersecurity experts to uncover it using only disk-based forensic techniques. Even though there are many tools to analyze malware in memory, new or modified variants continually discover innovative ways to bypass most security programs, making them untraceable within the system. Consequently, detection should rely on behavioral characteristics rather than structural ones (Sudhakar, 2020; SentinelOne, 2025). By identifying which LOLBin has been activated for malicious purposes is one way to reveal such behavior.

1.3 Research objectives and contributions

Fortunately, there are various tools that search for malicious usage of LOLBins with the help of various methods, such as machine learning algorithms (Ângello Cássio Vasconcelos Oliveira, 2025). However, this research thesis has discovered another simple method for detecting LOLBin malicious behavior in the volatile memory, using two cybersecurity tools: Volatility 3 and YARA.

Living off the Land binaries can be found on many operating systems. This research thesis focuses on the Linux operating system because most devices, large or small, statistically run the Linux kernel (Oishi, 2023). In the next chapters of this thesis will describe the use of a new tool combined with the use of the Volatility and Yara rules. This new tool looks like it is promising to be employed by cybersecurity experts and make the search and detect malware process faster and more accurate. No prior research has investigated this type of experiment. Cybersecurity practitioners typically employ Volatility 3 and YARA for direct, signature-based malware identification, whereas LOLBins are frequently leveraged indirectly as part of malicious tradecraft. This thesis aims to support the investigation of

potentially illicit LOLBin-driven activity by identifying the presence of known binaries in volatile memory and correlating them with execution context using memory forensic tools.

This thesis research objectives are as follows:

- Find and validate a simple method for detecting malicious LOLBins.
- Generate YARA rules for both LOLBins and non LOLBins.
- Test and analyze the YARA rules using Volatility 3 to determine whether they detect malicious behavior.
- Use all available data to assess whether this methodology can serve as an additional tool for memory investigation and threat hunting.

It is important to clarify that this research does not aim to directly detect fileless malware, nor does it attempt to definitively classify a binary execution as malicious. Instead, the proposed methodology (which combined YARA-based rule generation with Volatility memory analysis) focuses on identifying the presence and execution context of Linux LOLBINs that may indicate potentially risky activity. These findings are intended to support memory forensics and threat hunting workflows and should be correlated with additional forensic artifacts and expert analyst judgment to assess whether they form part of a broader fileless intrusion.

1.4 Scope and limitations

Scope of the research thesis:

- **Virtualized infrastructure:** The experimental environment consists of two Linux-based virtual machines, namely Ubuntu and REMnux.
- **Data acquisition process:** A volatile-memory image is collected, and YARA rules are generated for a set of 30 sample binaries originating from the Ubuntu system, using a purpose-built Python tool for automated rule generation.
- **Forensic analysis methodology:** The generated YARA rules and the acquired memory dump are subjected to systematic examination using Volatility 3, executed within the REMnux environment, to assess detection capabilities and analytical effectiveness.

The experimental procedure will be limited to the use of two tools Volatility 3 and YARA, along with the previously described virtual machines. The Python-based YARA rule-generation utility will be configured to produce exactly 30 rules targeting binary executables, excluding any rules for scripting languages. The majority of the selected binaries will be drawn from the LOLBins corpus. Each binary will be evaluated to assess its detectability through YARA-based analysis. The procedure will help the forensic investigator strengthen his cybersecurity arsenal and does not replace any human judgment or behavioral analysis.

1.5 Thesis structure

The thesis structure is as follows:

- **Chapter 2: Background and related work**
- **Chapter 3: The Linux memory architecture overview**
- **Chapter 4: Fileless techniques with the help of LOLBins on Linux systems**
- **Chapter 5: Methodology and tools installation**
- **Chapter 6: Implementing the YARA Rules Generator (YRG) for ELF**
- **Chapter 7: Evaluation of YRG**
- **Chapter 8: Discussion**
- **Chapter 9: Conclusion**

Chapter 2

Background and Related Work

2.1 Linux basic architecture overview

Linux is an open-source operating system (OS) created by Linus Torvalds at the University of Helsinki, Finland, in 1991 (Oishi, 2023). Its architecture was based on the UNIX operating system, which was originally developed in 1970 at AT&T Bell Labs. The reason for creating Linux was that Linus wanted an OS that could be free and open-source to the world, the core principles of the GNU General Public License (GPL).
The basic architecture of a Linux system has the following layers (Oishi, 2023):

- **Hardware –** The hardware contains all the physical components required for a Linux system to function correctly. The main components are the CPU (Central Processing Unit), RAM (Random Access Memory), input devices (e.g. keyboards, mice) and output devices (e.g. monitors, printers).
- **Kernel –** The kernel is the core of Linux; it hides the complexity of the hardware layer and provides a uniform interface for all software. Every program a user sees on a monitor is built on top of the kernel. The kernel manages devices (e.g. device drivers for USB or other peripherals), resources (e.g. CPU processes), memory, system calls, and optimize performance.
- **Shell –** The shell layer contains command line shells such as Bash and ZSH, as well as graphical shells like GNOME (GNU Network Object Model Environment). This layer provides an interface that interprets user commands and forwards them to the kernel.
- **Applications –** This layer hosts all the programs that users run on top of the shell, kernel, and hardware. Applications include databases, media players, web browsers, text editors, and many other utilities. The application layer also contains system utilities (programs that perform tasks requested by users or manage system functions) and system libraries, which

supply functions that allow programs to interact with the kernel for tasks that do not require kernel level privileges.

2.2 Linux RAM forensics

Linux's multi-user security model and its open-source nature, it has historically been less targeted by malware than Windows. However, in recent years hackers have begun to gain momentum in creating Linux malwares, and many of those go undetected by most antivirus programs, even when they perform simple tasks. The main reason is that many security vendors do not create, and users do not update their detection signatures regularly. As a result, the cybersecurity professionals must be always ready to develop tools that strengthen their systems defenses. They must be able to create programs capable of investigate and threat-hunting, against both known and unknown malicious actors.
In memory-forensics, the Linux-friendly tools that investigators often use to hunt for and analyze malware are (ForensicTools.dev, 2024):

- **Volatility**
- **YARA**
- **Rekall**
- **Redline**
- **LiME**
- **AVML**

Each tool has its own strengths and weaknesses and many memory forensics researchers have employed them to detect a wide range of threats, from ransomware to file-less malwares.

2.3 LOLBins and Fileless malware

2.3.1 Introduction to Living-off-the-Land Binaries

LOLBins (Living off the Land binaries) (SentinelOne, 2024) are executable programs that exist on the operating system, usually these files are targets and can be abused by adversaries, for gaining access, escalating privileges, connecting to a command and control (C2) server for stealing useful information. LOLBins are considered legitimately

executables, as they are used both by the user or by a third-party software. Because of their legitimacy some security tools consider them as benign programs that won't cause harm to the operating system.

However, programs that are preinstalled inside the OS can be, also, weapons for attackers. Malicious actors can integrate LOLBins commands to create simple malicious malware programs. Malware can call LOLBins efficiently for connecting to a remote server, compile code, achieve persistence or dump processes.

Usually, attackers prefer to call binaries, which are common or have little capabilities to the users and third-party software, but are essential for the attackers. One common LOLBin that exists in most Linux distributions is the bash script engine, which can be exploited for executing malicious scripts.

2.3.2 Fileless malware

Fileless malware is a type of malware that doesn't need to exist on the disk file system but is injected inside a legitimate executable process (SentinelOne, 2025; Sudhakar, 2020). It exists in memory leveraging already existing processes. This way traditional cybersecurity tools such as antivirus software or other endpoint systems will have difficult time detecting its footprint inside the OS.

To run the fileless malicious code, attackers are using one of the following processes (SentinelOne, 2025):

- **VBScript (*Windows OS*)**
- **Jscript (*Windows OS*)**
- **Bash shell (*Linux OS*)**
- **PowerShell (*Windows OS*)**
- **Batch files (*Windows OS*)**

Fileless malware can be also integrated inside hidden documents (SentinelOne, 2025). Although these file types, normally, you cannot execute them, what do they do? They use exploits in Microsoft Office, LibreOffice and PDF readers and allow attackers to execute malicious code. One example is that an infected document can trigger a PowerShell command or commands and execute a malicious batch process on Windows (the same thing can happen on Linux, with the help of the Bash shell commands).

2.3 Volatility 3 and YARA fundamentals

2.3.1 Volatility 3

Volatility is a memory-forensics framework that extracts digital artifacts from volatile RAM (Volatility Foundation, 2025). Volatile system memory can be captured as a memory dump, which can then be analyzed by the Volatility.

In 2019, the Volatility Foundation released Volatility 3, the successor to Volatility 2, to address the shortcomings of its predecessor.

2.3.2 YARA rules

YARA rules (VirusTotal Revision, 2022)**,** (Hacıoğlu, 2025) are pattern-matching definitions that detect, identify, and classify programs on a file system and in memory. The rules locate programs using strings, hexadecimal sequences, or regular expressions, and they contain Boolean logic that determines whether a match is valid enough to be shown to the user. YARA rules can cover a wide variety of patterns inside a computer system, providing a powerful alternative to hash-based detection tools—which can fail if the malware is altered by even a single byte.

The simplest YARA rule that can be created, according to two sources (Hacıoğlu, 2025; Cohen, 2017) consists of the following elements:

- **Rule name –** The identifier of the signature you want to detect.
- **Tags –** Optional labels for grouping and filtering.
- **Meta –** Optional contextual fields (name, author, description, etc.).
- **Strings –** Simple strings, hexadecimal sequences, or regex patterns that the rule must detect.
- **Condition –** Boolean logic that determines when the rule matches.
- **Import statements –** Optional modules for additional analysis (e.g., PE, ELF, hash modules).

2.4 Identified research gap

As mentioned before, many researchers have covered memory forensics, fileless malware, LOLBins and detection tools, such as Volatility 3 and YARA rules. However, most of them focus on fileless malware on Windows machines. There is also a paper by (Ângello Cássio Vasconcelos Oliveira, 2025) that focuses on LOLBin detection on Linux machines, but as noted in Chapter 1, it used machine-learning algorithms for detection (This falls too far outside the intended focus of this thesis). The research paper from (Cohen, 2017) presents work on memory scanning using YARA tools, while the paper from (Myra Khalid, 2020) discusses the automatic generation of YARA rules for known malwares. The first study concentrates on memory scanning of malware signatures on Windows (not for Linux), while the second addresses automated creation of YARA rules for known malwares (not for LOLBins).

This thesis highlights a gap in existing research. No prior work has investigated memory scanning of LOLBins with YARA rules, nor has any study addressed the automated generation of YARA rules for these legitimate binaries on Linux systems. Consequently, addressing this gap is necessary to evaluate its importance and to judge whether it warrants further attention from cybersecurity professionals, particularly memory analysts and threat hunters.

Chapter 3

The Linux memory architecture overview

3.1 Linux Memory

The operating system is designed to balance performance, secure the system, and efficiently utilize resources (Lohot, 2023). To achieve this, its memory-management system includes allocating memory, isolating processes, and optimizing overall system performance.

The structure of Linux memory architecture is divided into two basic segments (Lohot, 2023):

- **Kernel region –** the region reserved for the operating system, containing the kernel core, system libraries, and device drivers.
- **User region –** the area where user applications run. They are isolated from the kernel to prevent unauthorized access and protect the system from crashes.

3.2 Virtual memory

In memory management, virtual memory is a crucial part of the OS, because it allows processes to use more memory than is physically available by extending RAM with disk storage (Lohot, 2023). This feature helps the Linux OS to do efficient multitasking, isolate processes, and maintain system stability. The kernel dynamically translates the virtual addresses used by applications to physical addresses. The component that the kernel uses for this procedure is called **Memory Management Unit (MMU)**.

Every process reserves its own virtual address space, independent of other processes and the hardware. The address space consists of the following regions (Lohot, 2023):

- **Code section / Text segment (.text) –** stores the compiled machine code of a program. This code is read-only to prevent alterations.
- **Data segment (.data) –** contains all global and static variables, including initialized ones that can be read and modified.

- **Block Started by Symbol (.bss) –** contains uninitialized global and static variables that can be allocated at runtime. Like the .data segment, .bss is a static section of memory.
- **Read-only data segment (.rodata) –** holds constant string literals, global constant variables, and static constant data. The contents of .rodata are read-only and cannot be modified.
- **Heap –** dynamically allocates memory when a program calls malloc() in C or new in C++.
- **Stack –** contains function calls, local variables, and control-flow information. The stack grows downwards in memory space, and its size is managed by the kernel.

3.3 Paging

Paging is one of the memory-management mechanisms that divides physical memory into fixed-size blocks called pages (normally 4 KB) (Lohot, 2023). In the OS, the page table maps virtual addresses to physical addresses. The page table in modern 64-bit Linux systems is multi-level, which enables efficient management of large amounts of memory (Lohot, 2023).

The levels are:

- **Page Global Directory (PGD)**
- **Page Upper Directory (PUD)**
- **Page Middle Directory (PMD)**
- **Page Table Entry (PTE)**

Paging operates as follows (Lohot, 2023):

- When a process accesses a virtual address, the CPU consults the page table to translate it to a physical address.
- If the page is present in RAM, the MMU translates the virtual address into the corresponding physical address.

- If the page is not present in RAM (a page fault occurs), the kernel retrieves the page from disk and loads it into memory.

3.4 Swap space

Swap space (Lohot, 2023) is a mechanism the OS uses to move inactive pages from RAM to disk, freeing memory (RAM) for active processes. Excessive swapping can degrade performance.

3.5 Memory allocation

Table 1: Memory allocation functions from C library

Name	Memory region	Description
malloc()/free() (Lohot, 2023; cppreference.com, 2025; cppreference.com, 2025)	User	Standard C library functions for dynamic memory allocation. The malloc() function dynamically allocates a new block of memory, while free() deallocates a block that was previously allocated by malloc().
mmap() (Lohot, 2023), (Kerrisk, mmap(2) - Linux manual page)	User	Another standard library function, mmap() maps or unmapped files or devices into the process's address space.
brk() / sbrk() (Lohot, 2023), (Kerrisk, brk(2) - Linux manual page)	User	Standards C library functions can modify the size of data segments.
kmalloc() (Lohot, 2023)	Kernel	In the Linux kernel, kmalloc() provides the same basic functionality as malloc(), but it is used for allocating memory that will be accessed by the kernel code.
vmalloc() (Lohot, 2023)	Kernel	Assigns large, sequential memory areas at the virtual level.
Slab allocator (Lohot, 2023; Gorman, 2003)	Kernel	The slab allocator is a caching scheme that keeps frequently used objects in an initialized state and ready for reuse by the kernel. Without the slab allocator, the kernel would have to allocate, initialize, and deallocate the same object each time.

3.6 Linux commands for memory information

The Linux commands that provide information about RAM and virtual memory are the following (Boelen, 2025):

- **dmesg:** Shows the amount of RAM in the system.
- **free:** Displays information about used and swap memory. The data are read from /proc/meminfo.
- **cat /proc/meminfo:** Displays all the memory information available in the proc filesystem.
- **vmstat:** Displays statistics for virtual memory.
- **hwinfo –memory** (hwinfo man | Linux Command Library, 2024)**:** Displays detailed hardware information about installed memory modules.
- **ps -e -orss=,args= | sort -nr | head:** Shows memory usage sorted in reverse numerical order.
- **dmidecode --type 17:** Shows more detailed information about the RAM modules.

Chapter 4

Fileless techniques with the help of LOLBins on Linux systems

4.1 Common techniques of fileless malware that can be used in Linux machines

There are three techniques for fileless malware that can leverage fileless capabilities to improve the ability to avoid detection (Experts, 2023). The techniques are focused on Linux environment, but they could work the same way on the Windows environment:

Table 2: Fileless malware techniques

Technique	Description
Memory-only malware	This is a common type of fileless malware. It resides in memory and is difficult to detect by antivirus scanners because of its fileless aspect. Example of memory-only malware is the Duqu worm that resides in memory to remain undetected.
Fileless Ransomware	Ransomware is a type of malicious software that typically encrypts a victim's files, after which the attackers, who hold the decryption key, demand payment in cryptocurrency. Fileless ransomware poses an even greater threat, as it can operate without leaving traditional files on disk, instead abusing native scripting languages such as macros or injecting malicious code directly into memory through exploits. For instance, fileless malware can leverage the Linux Bash shell to encrypt all victim's files. In such cases, the victim is left with no viable options other than complying with the attackers' demands.
Exploits kits	Exploit kits typically comprise structured sequences of commands or bundled exploit code designed to weaponize specific software vulnerabilities. Threat actors leverage these kits to facilitate fileless intrusion workflows, often achieving code execution directly in memory without writing artifacts to disk. Delivery vectors frequently rely on social engineering techniques that coerce users into executing malicious payloads.

4.2 How LOLBins can be used in Fileless attacks

Scripting engines such as Bash on Linux or PowerShell on Windows serve as the tools to execute these binaries. When attackers use a fileless attack and execute a LOLBin, they can carry out a range of malicious activities (SentinelOne, 2024):

- Open communication with a remote server
- Compile and execute code
- Persistence
- Dumping processes
- Memory manipulation of processes

The existence of LOLBins in the OS is always a good reason for third-party adversaries implementing a fileless attack. The main idea of a fileless attack is that the code executed in memory, rather being written and executed from a file. This allows adversaries to erase their tracks; once the memory is cleared by a shutdown or reboot, it becomes difficult for incident-response teams and threat hunters to locate them.

A common scenario for a fileless attack might begin with a phishing attempt. The user receives an email from an unknown sender and clicks a malicious link or attachment. The link or attachment may execute a LOLBin, to download and execute malicious code in memory. This code can then call other LOLBins, such as "nc" or "msfconsole", to establish persistence, open a backdoor, or exfiltrate data by contacting a C2 server.

Additionally, fileless attacks can be combined with other types of malwares, such as ransomware or keyloggers.

4.3 Examples of LOLBins attacks

4.3.1 Fileless attacks on Linux-based HoneyCloud (Fan Dang, 2019)

In a research paper made by cybersecurity experts from various American Universities in collaboration from researchers from Alibaba Group, Tsinghua University and Tencent Anti-Virus Lab, created a honeypot system called "HoneyCloud" to study malware-based and fileless attacks on Linux-based IoT devices. They have captured many fileless techniques and have discovered that most of the attacks they traced involved generally gaining access, using shell commands for unknown reasons, modifying or removing

filesystems, stopping some certain processes/services, stealing and retrieving information and establishing network attacks and communications. The researchers concluded that Linux-based IoT devices, even though they are guarded with some security measures against malware, are not adequately protected against fileless attacks.

4.3.2 Poisoned Word document on Mac and Windows systems (Stokes, 2019)

The article analyzes a Lazarus APT campaign that delivers a malicious Microsoft Word document containing VBA macros capable of targeting both Windows and macOS systems. Although the report does not mention Linux, the cross-platform logic observed in the macro (including OS detection and conditional execution paths) suggests that similar techniques could be adapted for Linux environments. The campaign specifically targets individuals in the financial and cryptocurrency sectors, consistent with Lazarus' historical focus on financially motivated operations.

The disassembled VBA script demonstrates that the macro first checks the version of the VBA environment and then executes commands tailored to the detected operating system. For example, for macOS the script invokes functions such as **popen()** and **system()** to run shell commands, while on Windows it uses alternative command execution methods. In both cases, the macro retrieves additional payloads from command-and-control (C2) servers using legitimate system utilities like **curl**.

Although the article does not explicitly classify the attack as fileless or reference the use of LOLBins, several behaviors align with these concepts. The reliance on built-in system tools to execute commands and download payloads reduces the initial forensic footprint and mirrors techniques commonly seen in fileless attacks. Likewise, the abuse of legitimate binaries such as **curl** reflects patterns associated with LOLBin-based intrusion methods, where trusted system components are repurposed for malicious activity. These characteristics highlight how such techniques can be effective across multiple operating systems and underscore their relevance when assessing potential threats to Linux systems.

4.4 The GTFOBINS project

Researchers can't remember which binaries are LOLBins and which are not. The reason for that is that there a lot of LOLBins, many of them, having a couple of ways that can be abused by an attacker. For this reason, there is GitHub repository project called

GTFOBins (GTFOBins, 2025) which has a curated list of Unix binaries that can be used for bypassing local security restrictions in misconfigured systems.

The project contains legitimate functions that adversaries use to break out from restricted shells, elevate or maintain user's privileges, create a bind or reverse shell, transfer files and promote other post-exploitation activities.

Note that in the GTFOBins repository, all these binaries are not vulnerable and don't contain any exploits. It is a comprehensive collection of binaries that can be used in case where a user is in a Living off the Land environment and needs to live with the binaries that he has in our disposal.

The GTFOBins list has the following functions that might be used for malicious purposes (GTFOBins, 2025):

- **Shell -** is an interactive system shell that can be used to break out from restricted environments.
- **Command** – can run non-interactive commands and can be used to break out from restricted environments.
- **Reverse Shell** – a listening attacker accepts a shell connection to open a remote network communication.
- **Bind Shell** – Using a local port the attacker can bind a shell and open a remote network communication.
- **Non-interactive reverse Shell** – a reverse shell without interactive access.
- **Non-interactive bind Shell** – a bind shell without interactive access.
- **File Upload** – can be used for file exfiltration over the network.
- **File Download** – can be used to download remote files.
- **File Write** – modifies data in files and could be exploited to perform privileged writes or to write files outside a confined filesystem.
- **File Read** – accesses data in files and could be exploited to perform privileged reads or expose files outside a confined filesystem.
- **Library Load** – loads shared libraries that can be used for code execution.

- **SUID (Set-User-ID)** – set a special permission which allows a binary to run as root. If a binary has the SUID bit set, the user can abuse it to gain access to files or maintain/escalate privileged access to the system.
- **Sudo** – a command that allows a user to run binaries as the superuser (root).
- **Capabilities** – some Linux binaries have the CAP_SETUID capability, which allows a process to change its UID (User ID). Attackers can use this as a backdoor for maintaining or escalating privileges.
- **Limited SUID** – the Limited-SUID mechanism works on Debian distributions and can serve as a SUID backdoor with the default /bin/sh shell.

Until now, there are 390 binaries that have been confirmed to be usable for malicious behaviors. Below are some examples of LOLBins and their functions, according to the GTFOBins project (GTFOBins, 2025):

Table 3: Example binaries and their functions

Binary	Functions
nc	Reverse shell, bind shell, file upload, file download, sudo, limited SUID
socat	Shell, reverse shell, bind shell, file upload, file download, file write, file read, sudo, limited SUID
scp	Shell, file upload, file download, sudo, limited SUID
sftp	Shell, file upload, file download, sudo
rsync	Shell, SUID, sudo
curl	File upload, file download, file write, file read, SUID, sudo
wget	Shel, file upload, file download, file write, file read, SUID, sudo
nmap	Shell, non-interactive reverse shell, non-interactive bind shell, file upload, file download, file write, file read, SUID, sudo, Limited SUID
ssh	Shell, file upload, file download, file read, sudo

4.5 Using the MITRE ATT&CK framework on Linux LOLBins

MITRE ATT&CK (MITRE ATT&CK, 2025) is a cybersecurity framework that provides a comprehensive knowledge base of adversarial tactics, techniques, and procedures. It helps organizations understand how attackers operate — from gaining access to systems, executing malicious actions, and exfiltrate data. By mapping these behaviors, cybersecurity

teams can improve detection, strengthen defenses, and develop effective mitigation strategies.

The structure of MITRE ATT&CK contains (MITRE ATT&CK, 2025):

- **Tactics:** explains the "why" of the attacks, like gaining privilege escalation or persistence.
- **Techniques:** specific methods attackers use for their attacks and explains the "how".
- **Sub-techniques:** It gives more details about the techniques.
- **Procedures:** It gives real-world cases of attacks techniques

Attackers use various techniques in their attacks. Using LOLBins they can implement more stealthy and dangerous attacks. Some of the tactics, techniques and their sub-techniques and their LOLBins that attackers might be use for their attacks are mentioned below (MITRE ATT&CK, 2025).

4.5.1 Execution

Table 4:Command and Scripting Interpreter

Sub-technique	Description	LOLBins examples
Unix Shell	Attackers are using a Unix shell for commands and script execution.	sh, ash, bash, zsh, ksh
Python	Attackers are using a python interpreter for executing python commands and for .py files execution.	python, python3
Network Device CLI	Attackers can use a Command Line Interpreter (CLI) to connect to a network device. The CLI will give the attacker the privilege to execute commands.	ssh, telnet

System Binary Proxy Execution

The System Binary Proxy Execution is a technique that adversaries use if they want to bypass a process and signature-based defenses by proxying malicious executions using signed and legitimate binaries. On Linux systems they can use signed and trusted binaries like "split" to do the job.

4.5.2 Persistence

Table 5: Scheduled Task/Job

Sub-technique	**Description**	**LOLBins examples**
Cron	Attackers abuse the cron utility to perform task scheduling for initial or repeated execution of a malware.	crontab
Systemd Timers	Attackers abuse the systemd timers to perform task scheduling for initial or repeated execution of a malware.	systemctl, systemd-run
At	Attackers abuse the "at" utility to perform task scheduling for initial or repeated execution of a malware.	at

Table 6: Create or Modify System Process

Sub-technique	**Description**	**LOLBins examples**
Systemd Service	Attackers can create or modify a systemd service to frequently execute a malicious payload for persistence tactics.	systemctl

4.5.3 Privilege Escalation

Exploitation for Privilege Escalation

Attackers can use exploits on various software inside a Linux system to elevate their current privileges. These exploits can be a bug in a program or a service in the operating system user or kernel modes. When the adversary begins gaining access to a system usually he has the privilege of a simple user, meaning that he has a restricted access to files or directories. By finding vulnerabilities in the operating system and software he can exploit and gain higher privileges and eventually he gains root privileges which are the highest privileges a Linux user can take. As a root user the attacker can exfiltrate data, establish network connections or install malicious programs, without any restrictions.

Table 7: Abuse Elevation Control Mechanism

Sub-technique	**Description**	**LOLBins examples**
Setuid and Setgid	An attacker can abuse configurations where applications have set their setuid	chmod

	and setgid bits under another user's account, gaining escalated privileges. The setuid and setgid can be found with "ls -l" and "find" command. In "ls" the setuid and setgid bits can be appear in files attributes as a "s" instead of a "x".	
Sudo and Sudo Caching	Attackers can use the sudo command for sudo caching or use the sudoers file to elevate privileges. The sudo command can help execute malicious commands or create processes with higher privileges.	sudo

4.5.4 Defense Evasion

Table 8: Indicator Removal of Host

Sub-technique	**Description**	**LOLBins examples**
Clear Linux or Mac System Logs	Attackers can clear system logs to hide evidence of infiltration. Linux and MacOS have logs for systems or user activity. Most of them are in /var/log directory.	cat, rm, truncate, echo, tee, journalctl
Clear Command History	Attackers can clear the command history of a compromised account. The history log can be found in the user's home directory with the name .bash_history.	cat, rm, history
File Deletion	Attackers can delete files during intrusion or as part of post-intrusion process to cover their tracks.	rm, unlink
Timestomp	Timestomp is a technique that attackers use to modify the file's timestamps (modify, access, create, and change times) to mimic other files that are in the same folder and blend malicious files with legitimate ones.	touch
Clear Network Connection History and Configurations	Attackers can clean or remove evidence of network connection history, to cover their tracks. Configuration settings and other artifacts can be used to create a fake network activity history which is created inside the system or with the help of a software application.	ip, ifconfig
Clear Mailbox Data	Attackers can clear the data and the mail messages from a mail program to remove any evidence of their activity	rm, truncate, mail

Clear Persistence	Attackers can clear artifacts that had been used for establishing persistence on a host system. These artifacts may be executables, processes that can collect evidence of a program's persistence.	crontab, systemctl
Relocate Malware	Attackers can use malware to create copies of themselves and use them to remove any evidence of their presence or move them to a new location to avoid defenses.	mv, cp

4.5.5 Credential Access

Table 9: OS Credential Dumping

Sub-technique	**Description**	**LOLBins examples**
/etc/passwd and /etc/shadow	Attackers might get access to files /etc/passwd and /etc/shadow to try and crack their contents to get users' and root passwords	cat, unshadow
Proc filesystem	Attackers can get credentials from the proc filesystem. The proc filesystem is used as an interface to kernel data structures for managing Linux virtual memory.	cut, grep

4.5.6 Discovery

Table 10: System Network Configuration Discovery

Sub-technique	**Description**	**LOLBins examples**
Internet Connection Discovery	Attackers can check for internet connectivity on a compromised system. The Internet Connection Discovery can be performed using various ways like using tools such as "ping" and "tracert". Their results are essential if the attackers are determined to make a connection with their Command and Control – C2 server. Also, the technique is useful for finding routes, redirectors, and proxy servers.	ping, tracert
Wi-Fi Discovery	Attackers can search for information's about the Wi-Fi connectivity like names and passwords on a compromised system. On Linux systems Wi-Fi information can	nmcli, iwlist,iw

	be found under /etc/NetworkManager/system-connections/ directory.	

4.5.7 Exfiltration

Table 11: Exfiltration Over Alternative Protocol

Sub-technique	Description	LOLBins examples
Exfiltration Over Symmetric Encrypted Non-C2 Protocol	Adversaries can extract sensitive information by transmitting it through a symmetrically encrypted network protocol that differs from the one used for the primary command-and-control (C2) channel. They may also redirect the stolen data to a secondary network destination instead of the main C2 server.	scp, sftp, rsync, openssl
Exfiltration Over Asymmetric Encrypted Non-C2 Protocol	Adversaries can extract sensitive information by transmitting it through an asymmetrically encrypted network protocol that differs from the one used for the primary C2 channel. They may also redirect the stolen data to a secondary network destination instead of the main C2 server.	gpg, openssl, scp,sftp
Exfiltration Over Unencrypted Non-C2 Protocol	Adversaries can extract sensitive information by transmitting it through a symmetrically unencrypted network protocol that differs from the one used for the primary C2 channel. They may also redirect the stolen data to a secondary network destination instead of the main C2 server. Also, attackers can use encoding and compression methods or embedding data within alternative protocol's headers and fields.	ftp, telnet, nc, sendmail, base64

4.6 How can fileless attacks that use LOLBins be detected?

As mentioned before, fileless techniques can be harder to detect. Signature-based tools and legacy cybersecurity products continue to have a challenging time detecting suspicious behavior from fileless malware. Furthermore, LOLBins cannot be discarded from users' or

organizations' systems because these are essential tools for system management, development and maintenance.

The solution (SentinelOne, 2024) to the fileless problem and LOLBins malicious behavior, lies in threat-hunting teams and the adversaries' techniques they use to detect indicators of attacks within systems. Most indicators can be traced with tools that detect process behavior, such as opening a network connection or modifying a file in a directory. Behavior-based analysis is important, as many forms of malware exhibit recurring execution patterns. Moreover, threat-hunting teams do not need to search for file signatures, which can be altered in many ways.

Chapter 5

Methodology and tools installation

5.1 The steps of this research thesis

The first step of the research project is to collect YARA rules for Linux binaries on Ubuntu. Searching for unique strings, hex patterns and binary code within an ELF program can be an exhaustive task. This is where this thesis comes to solve the problem by creating a Python-based YARA rule generator for ELFs that automatically collects the components of binary and create a YARA rule for the specific binary, more details about this program are discussed in the next chapter.

The research narrows the search and generation process to a set of 30 binary rules, most of which are drawn from the GTFOBins list. As an initial dataset, this provides a solid number of binaries and represents a meaningful step toward evaluating the success of the experiment. The rules are then transferred to REMnux, where they are used by Volatility 3's "yarascan.YaraScan". The results of this plugin will be verified by other Volatility 3 plugins to determine whether the YARA rules produce false positives. If the YARA rules successfully identify the binary in memory and the Volatility 3 plugins reveal any malicious activity associated with it, the experiment will be considered successful.

5.2 Resources for this methodology

The following resources were used for this methodology:

- Ubuntu 20.04.1 64-bit Linux VMWare virtual machine with 9.9 GB memory, 4 CPU processors and 100 GB hard disk.
- REMnux Linux 64-bit VMWare virtual machine with 8 GB memory, 4 processors and 60 GB hard disk.

5.3 Volatility 3 installation

For the research project, an "infected" memory dump must be analyzed by Volatility 3. A Linux REMnux virtual machine was prepared, and Volatility was run on it. In a separate virtual machine, Ubuntu 20.04.1 was installed and several LOLBins were executed. Afterwards, that memory dump was obtained from the volatile memory of the

Ubuntu system and was transferred to the REMnux Linux virtual machine. Volatility 3 is installed on the REMnux virtual machine, but its version 2.7.0 is quite old. Thus, the tool must be updated to the latest release (2.26.0 at the time of writing) from GitHub using a few git commands.

Additionally, for the framework to function during analysis, the correct Ubuntu-based profile name (the "symbol") must be provided for Volatility 3.

```
remnux@remnux:~/volatility3$ ls -l
total 64
-rw-rw-r-- 1 remnux remnux 3593 Dec 10 10:15 API_CHANGES.md
-rw-rw-r-- 1 remnux remnux 1277 Dec 10 10:15 CITATION.cff
drwxrwxr-x 3 remnux remnux 4096 Dec 10 10:16 development
drwxrwxr-x 3 remnux remnux 4096 Dec 10 10:15 doc
-rw-rw-r-- 1 remnux remnux 3910 Dec 10 10:15 LICENSE.txt
-rw-rw-r-- 1 remnux remnux  195 Dec 10 10:15 MANIFEST.in
-rw-rw-r-- 1 remnux remnux 2289 Dec 10 10:16 pyproject.toml
-rw-rw-r-- 1 remnux remnux 5492 Dec 10 10:16 README.md
drwxrwxr-x 3 remnux remnux 4096 Dec 10 10:16 test
drwxrwxr-x 8 remnux remnux 4096 Dec 10 10:28 volatility3
-rwxrwxr-x 1 remnux remnux  314 Dec 10 10:15 vol.py
-rwxrwxr-x 1 remnux remnux  321 Dec 10 10:15 volshell.py
-rw-rw-r-- 1 remnux remnux 2963 Dec 10 10:15 volshell.spec
-rw-rw-r-- 1 remnux remnux 5450 Dec 10 10:15 vol.spec
```

Figure 1: The new version of Volatility 3 in REMnux Linux.

First, the memory dump must be created by Ubuntu 20.04.1 virtual machine. To be implemented it needed a must-install tool called LiME:

- **sudo apt update**
- **sudo apt install build-essential linux-headers-$(uname -r)**
- **git clone https://github.com/504ensicsLabs/LiME.git (the git command must be installed inside Ubuntu)**
- **cd LiME/src**
- **make**

```
user@ubuntu:~/LiME/src$ ls
deflate.c  hash.o                        lime.mod.o  Makefile.sample
deflate.o  lime-5.15.0-139-generic.ko    lime.o      modules.order
disk.c     lime.h                        main.c      Module.symvers
disk.o     lime.mod                      main.o      tcp.c
hash.c     lime.mod.c                    Makefile    tcp.o
```

Figure 2: lime-5.15.0-139-generic.ko file was created after the make command in LiME/src directory.

After the make command, the LiME tool was ready for use. The next steps were to:

- **sudo insmod lime-5.15.0-139-generic.ko "path=/home/user/memdump.lime format=lime" (creates the memory dump)**
- **sudo rmmod lime (to close lime in the kernel after successfully create the memory dump)**

Afterwards, in the ***home/user directory*** there is a new file called ***memdump.lime,*** which is memory dump file. The memory file had to be transferred to REMnux, where the Volatility was waiting.

The second stage is to take Ubuntu 20.04.1 symbols. For someone that needs to find the symbols version from the memory dump using the Volatility 3, he can do the following steps:

- **cd volatility3/**
- **python3 vol.py -r pretty -f /home/remnux/memdump.lime banners**

```
remnux@remnux:~$ cd volatility3/
remnux@remnux:~/volatility3$ python3 vol.py -r pretty -f /home/remnux/memdump.lime banners
Volatility 3 Framework 2.26.0
Formatting...0.00               PDB scanning finished
  |      Offset |
                                                          Banner
* | 0x1197ef528 |   Linux version 5.15.0-139-generic (buildd@lcy02-amd64-067) (gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #149~20.04.1-Ub
untu SMP Wed Apr 16 08:29:56 UTC 2025 (Ubuntu 5.15.0-139.149~20.04.1-generic 5.15.178)
* | 0x1251a6c88 |   Linux version 5.15.0-139-generic (buildd@lcy02-amd64-067) (gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #149~20.04.1-Ub
untu SMP Wed Apr 16 08:29:56 UTC 2025 (Ubuntu 5.15.0-139.149~20.04.1-generic 5.15.178)
* | 0x24cdddda8 |   Linux version 5.15.0-139-generic (buildd@lcy02-amd64-067) (gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #149~20.04.1-Ub
untu SMP Wed Apr 16 08:29:56 UTC 2025 (Ubuntu 5.15.0-139.149~20.04.1-generic 5.15.178)
* | 0x258a0b508 |   Linux version 5.15.0-139-generic (buildd@lcy02-amd64-067) (gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #149~20.04.1-Ub
untu SMP Wed Apr 16 08:29:56 UTC 2025 (Ubuntu 5.15.0-139.149~20.04.1-generic 5.15.178)
* | 0x281c00200 |   Linux version 5.15.0-139-generic (buildd@lcy02-amd64-067) (gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #149~20.04.1-Ub
untu SMP Wed Apr 16 08:29:56 UTC 2025 (Ubuntu 5.15.0-139.149~20.04.1-generic 5.15.178)
* | 0x283c39778 | Linux version 5.15.0-139-generic (buildd@lcy02-amd64-067) (gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #149~20.04.1-Ubun
tu SMP Wed Apr 16 08:29:56 UTC 2025 (Ubuntu 5.15.0-139.149~20.04.1-generic 5.15.178)8)
* | 0x2b66c7d48 |   Linux version 5.15.0-139-generic (buildd@lcy02-amd64-067) (gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #149~20.04.1-Ub
untu SMP Wed Apr 16 08:29:56 UTC 2025 (Ubuntu 5.15.0-139.149~20.04.1-generic 5.15.178)
```

Figure 3: Memory dump's banners

To extract the symbols, there were two ways. The first way is to take the symbols from Ubuntu itself (a devious process, because a user needs to execute many commands on the Ubuntu machine). The other way, however, is easier and follows the instructions in the GitHub repository https://github.com/Abyss-W4tcher/volatility3-symbols (Abyss-W4tcher, 2025), which contains most Ubuntu banner versions. The banners are stored in plain JSON format. From the home directory, the user can run the following commands (Abyss-W4tcher, 2025):

- **wget https://raw.githubusercontent.com/Abyss-W4tcher/volatility3-symbols/master/banners/banners_plain.json**
- **grep -A 2 'Linux version 5.15.0-139-generic (buildd@lcy02-amd64-067) (gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34)' banners_plain.json**

The compressed .xz file has been downloaded by the wget tool and transfered to Volatility 3 Linux symbols directory:

- **wget https://github.com/Abyss-W4tcher/volatility3-symbols/raw/master/Ubuntu/amd64/5.15.0/139/generic/Ubuntu_5.15.0-139-generic_5.15.0-139.149~20.04.1_amd64.json.xz**
- **mv Ubuntu_5.15.0-139-generic_5.15.0-139.149~20.04.1_amd64.json.xz ~/volatility3/volatility3/symbols/linux**

5.4 Volatility 3 plugins and their functionalities

Volatility 3 has some plugins which are useful for tracing running and non-running processes. In our research project these plugins are important for detecting LOLBins. These are:

linux.pslist

Shows a list of processes that are present in the Linux memory image.

```
remnux@remnux:~/volatility3$ python3 vol.py -f /home/remnux/memdump.lime linux.pslist
Volatility 3 Framework 2.26.0
Progress:  100.00               Stacking attempts finished
OFFSET (V)      PID     TID     PPID    COMM    UID     GID     EUID    EGID    CREATION TIME   File output

0x9df000283f80  1       1       0       systemd 0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df000281fc0  2       2       0       kthreadd        0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df000280000  3       3       2       rcu_gp  0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df000285f40  4       4       2       rcu_par_gp      0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df000299fc0  5       5       2       slub_flushwq    0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df000298000  6       6       2       netns   0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df00029df40  7       7       2       kworker/0:0     0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df00029bf80  8       8       2       kworker/0:0H    0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df0002a3f80  9       9       2       kworker/u256:0  0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df0002a1fc0  10      10      2       mm_percpu_wq    0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df0002a0000  11      11      2       rcu_tasks_rude_ 0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
0x9df0002a5f40  12      12      2       rcu_tasks_trace 0       0       0       0       2025-10-18 14:21:59.391729 UTC  Disabled
```

Figure 4: Volatility linux.pslist.

linux.psscan

Shows processes found by scanning the Linux memory. These processes may have been running when the memory was captured, may have already terminated, become zombies, or been in a traceable exit state.

```
remnux@remnux:~/volatility3$ python3 vol.py -f /home/remnux/memdump.lime linux.psscan
Volatility 3 Framework 2.26.0
Progress:  100.00               Stacking attempts finished
OFFSET (P)      PID     TID     PPID    COMM    EXIT_STATE

0x100280000     3       3       2       rcu_gp  TASK_RUNNING
0x100281fc0     2       2       0       kthreadd        TASK_RUNNING
0x100283f80     1       1       0       systemd TASK_RUNNING
0x100285f40     4       4       2       rcu_par_gp      TASK_RUNNING
0x100298000     6       6       2       netns   TASK_RUNNING
0x100299fc0     5       5       2       slub_flushwq    TASK_RUNNING
0x10029bf80     8       8       2       kworker/0:0H    TASK_RUNNING
0x10029df40     7       7       2       kworker/0:0     TASK_RUNNING
0x1002a0000     11      11      2       rcu_tasks_rude_ TASK_RUNNING
0x1002a1fc0     10      10      2       mm_percpu_wq    TASK_RUNNING
0x1002a3f80     9       9       2       kworker/u256:0  TASK_RUNNING
0x1002a5f40     12      12      2       rcu_tasks_trace TASK_RUNNING
0x1003a8000     15      15      2       migration/0     TASK_RUNNING
```

Figure 5: Volatility linux.psscan.

linux.psaux

Shows a list of processes along with their command-line arguments.

```
remnux@remnux:~/volatility3$ python3 vol.py -f /home/remnux/memdump.lime linux.psaux
Volatility 3 Framework 2.26.0
Progress:  100.00               Stacking attempts finished
PID     PPID    COMM    ARGS

1       0       systemd /sbin/init auto noprompt
2       0       kthreadd        [kthreadd]
3       2       rcu_gp  [rcu_gp]
4       2       rcu_par_gp      [rcu_par_gp]
5       2       slub_flushwq    [slub_flushwq]
6       2       netns   [netns]
7       2       kworker/0:0     [kworker/0:0]
8       2       kworker/0:0H    [kworker/0:0H]
9       2       kworker/u256:0  [kworker/u256:0]
10      2       mm_percpu_wq    [mm_percpu_wq]
```

Figure 6: Volatility linux.psaux.

linux.bash

Shows the recovered Bash command history found in memory.

```
remnux@remnux:~/volatility3$ python3 vol.py -f /home/remnux/memdump.lime linux.bash
Volatility 3 Framework 2.26.0
Progress:  100.00               Stacking attempts finished
PID     Process CommandTime     Command

1984    bash    2025-10-18 14:22:35.000000 UTC  ps aux
```

Figure 7: Volatility linux.bash.

Also, **“yarascan.YaraScan”** and **linux.vmayarascan** are essential plugins for the research experiment, which they will be described and used later to detect LOLBins inside the memory dump.

5.5 Tools for YARA rules detection and basic YARA syntax

YARA rules are used by many forensic, threat-hunting, and incident-response tools. Two basic tools that researchers typically employ are:

- **YARA CLI** – pre-installed by default in REMnux Linux. Can be used to check for syntax errors and if the legitimate binary covers all its components.

- **Volatility 3 –** the YaraScan plugin and the linux.vmayarascan plugin which are the basic plugins the research thesis will be using.

In YARA documentation the simplest rule a cybersecurity expert can create for a YARA detection is the following (VirusTotal Revision, 2022):

```
rule dummy
{
   condition:
      false
}
```

Chapter 6

Implementing the YARA Rules Generator (YRG) for ELF

6.1 Introduction to YARA Rules Generator (YRG) for ELF

In the previous chapters it was explained that the only way to find LOLBins was through behavior-based tools. YARA rules are described as signature-based tools that use strings, hex and regex patterns for detection. Can YARA rules be used for detecting LOLBins that might be used in fileless attacks?

Even though YARA rules are not using behavior-based techniques, a memory researcher can use them to detect the presence of LOLBins in memory. The successful creation of a YARA rule needs unique signatures, like strings or hex patterns, in such a way that can detect the presence of specific binaries in memory. Also, the rule needs well known command strings that might indicate malicious behavior.

However, creating a rule is not a very easy task; there are many things to consider and many problems to solve. One of the problems is that there are a lot of binaries for examination in a Linux operating system. To be more specific, until now on the GTFOBins website storage, there are at least 390 binaries. The number of binaries and the amount of time a cybersecurity expert need to spend, to find the unique characteristics of each one and create a unique YARA rule, that sufficiently work with each of these binaries, is a lot. Finally, there is a prediction, a hypothetical case, for binaries that don't belong yet to the known LOLBin category. These binaries may have some command parameters that may cause malicious behaviors, under specific circumstances in the operating system.

In solving these challenges, the research thesis has introduced a Python program that can generate YARA rules for ELFs binaries, with the focus on the executable's files and not on the script files. Using this script cybersecurity researchers can save time locating all the relevant YARA components and creating multiple YARA rules that are stored in one file. Furthermore, with the help of the YARA Rules Generator, 30 YARA rules were created for testing purposes, most of them were LOLBins, and used them in Volatility 3. The **"yarascan.YaraScan"** and **linux.vmayarascan** plugins were the two tools used to test the rules. In addition, there is a captured an Ubuntu's memory dump that is used for searching

the components of these 30 binaries. The test worked fine, but in some cases resulted in false positives, which unfortunately are common in many YARA tools.

YARA Rules Generator (YRG) for ELFs and its 30 YARA rules examples can be downloaded from GitHub using the link: https://github.com/githubUserKA/YARA-Rules-memory-binary-generator-for-Linux (githubUserKA, 2025)

6.2 YARA Rules Generator for ELF description

The YRG for ELFs is a Python script program that automates the creation of YARA rules for ELF binaries. The script can generate:

- Rule name of the binary.
- The meta component which contains:
  1. The binary description.
  2. GTFOBins' website as a reference.
  3. The binary name.
  4. MD5, SHA1, SHA256 hashes that identify the binary file.
- The string component which has:
  1. .rodata strings.
  2. .text hex opcodes.
  3. GTFOBins binaries commands if the ELF binary is part of the GTFOBin database.
- Logic condition that must be met to detect the ELF binary.

The script has the following command usage:

```
python3 yara-rules-elf-generator.py \
    --binaries /path/to/binaries \
    --gtfobins /path/to/gtfobins/repository \
    --output /path/to/output.yar
```

Figure 8: Command of the YARA ELF generator

The program can take the following arguments (all arguments are required):

- **--binaries:** The path directory that has ELF binaries or an ELF file.
- **--gtfobins:** The path of the gtfobins directory that has all the descriptions of all the binaries commands (that might maliciously be used by attackers).
- **--output:** The path where the YARA rules file will be created.

To execute the program the user must run it as a root. The program uses functions that require root privileges to do the job. The program completes its job by following the six stages that I describe below:

1. Extract the commands of GTFOBins binaries, these commands will be used for the creation of the YARA rules.
2. Read the binary file or all the binaries in the directory. If a binary is not pure ELF file the program skips it.
3. If the binaries in the directory contain duplicates the program deduplicates them.
4. The –output argument will guide the program to create a new file (where all the Yara rules will be save) in the path the user ask for.
5. The program follows the YARA binary rule format; adds description, reference and the MD5, SHA1, SHA256 hashes to the meta section. In the string section the program adds the unique strings from .rodata sections and one 32-bytes hex opcode taken from the .text section. The GTFOBins commands are added in the string component if the binary belongs to the list of GTFOBins (including the sudo command of the binary). Finally, the program adds conditions that are created based on the types of string component that exist. For example, if the binary belongs to the GTFOBins the script adds strings, hex, commands and the sudo. If the binary is not in the GTFOBins list but has strings and hex then it builds a condition based on these strings' components.

```
if gtfobin_tokens and rodata_strings and code_patterns and not no_cmd:
   condition = f"(3 of ($str*)) or ($code and 3 of ($str*)) or {bin_token} or any of ($cmd*)"
   meta_note = "GTFOBins integrated"
elif gtfobin_tokens and code_patterns and not no_cmd:
   condition = f"($code) or {bin_token} or any of ($cmd*)"
   meta_note = "GTFOBins integrated"
elif gtfobin_tokens and rodata_strings and code_patterns and no_cmd:
   condition = f"3 of ($str*) or ($code and 3 of ($str*)) or {bin_token}"
   meta_note = "GTFOBins integrated"
elif gtfobin_tokens and code_patterns and no_cmd:
   condition = f"($code) or {bin_token}"
   meta_note = "GTFOBins integrated"
elif rodata_strings and code_patterns:
   condition = "3 of ($str*) or ($code and 3 of ($str*))"
   meta_note = "Non-GTFOBin binary"
elif code_patterns:
   condition = "$code"
   meta_note = "Non-GTFOBin binary"
else:
   return ""
```

Figure 9: The YARA conditions based on the binary's contents (bin_token variable contains the sudo command of the binary).

6. In the end the program shows a small report of the binaries that have been collected and the output file path.

6.3 YARA Rules Generator for ELF results

The Python script was executed on a Linux Ubuntu system, the same system on which the memory dump was created. The directory **bin_collection**, containing 30 binaries (28 of them belong to the GTFOBins list) was supplied as input. The final output was named **rules.yar**. The picture below displays the program's output, illustrating the stages it passes through.

```
user@ubuntu:~$ sudo python3 yara-rules-elf-generator.py   --binaries bin_collect
ion --gtfobins gtfobins_repo/_gtfobins --output rules.yar
[sudo] password for user:
🔍 Loading GTFOBins dataset...
✅ Loaded GTFOBins entries.
🔍 Searching for ELFs binaries
✅ Found 30 ELFs binaries
Processing binaries: 100%|██████████████████████| 30/30 [00:04<00:00,  6.72file/s]
Overall Report:
[*]ELF binaries: 30
[*]Number of non ELF binaries: 0
[*]Number of symlinks: 0
[*]Number of hard links: 0
[*]Number of binaries belonged to GTFOBins: 28
✅ YARA rules written to rules.yar succesfully
user@ubuntu:~$
```

Figure 10: The yara-rules-elf-generator.py execution.

Looking at one of the rules the rules.yar file contains, should be like this:

```
rule curl_
{
    meta:
        description = "YARA rule for curl (GTFOBins integrated)"
        reference = "https://gtfobins.github.io/"
        binary = "curl"
        md5 = "bc725e8ef3cfd441a9c7324fad271333"
        sha1 = "a3f491bb727f997f7b7331f495a1841167ef0dad"
        sha256 = "444ca1a335b652d4c3e6e8684dc9200984795f1d8c858aac16d24cf38bf63c0b"

    strings:
        $code = {00 BF 16 27 00 00 E8 95 8A 00 00 84 C0 0F 84 3D 1E 00 00 BF 3D 00 00 00 E8 83 8A 00 00 84 C0 75}
        $str1 = "%02d:%02d:%02d.%06ld" ascii nocase
        $str2 = "Failed to create/open output" ascii nocase
        $str3 = "%s%s, %zu bytes (0x%zx)" ascii nocase
        $str4 = "Content-disposition:" ascii nocase
        $str5 = "Received header etag is missing double quote/s" ascii nocase
        $str6 = "Y@Remote filename has no length!" ascii nocase
        $str7 = "Failed to create the file %s: %s" ascii nocase
        $str8 = "Binary output can mess up your terminal. Use \"--output -\" to tell curl to output it to your terminal anyway, or consider \"--outp" ascii nocase
        $str9 = "Refusing to overwrite %s: %s" ascii nocase
        $str10 = "Error creating directory %s." ascii nocase
        $sudo = "sudo curl" ascii
        $cmd1 = /curl\s+-X\S*/ ascii

    condition:
        (3 of ($str*)) or ($code and 3 of ($str*)) or $sudo or any of ($cmd*)
}
```

Figure 11: The YARA rule for curl.

The results show that the curl binary has a YARA rule that contains 10 strings, hex opcode, one sudo and one GTFOBin command.

6.4 The YARA Rules example binaries

Table 12: The YARA binaries that have been created by the program.

Binary name	GTFOBins integrated	Hex opcode	Number of strings	Sudo	GTFOBin command
curl	Yes	Yes	10	Yes	Yes
gdb	Yes	Yes	10	Yes	Yes
nc	Yes	Yes	10	Yes	Yes
mv	Yes	Yes	10	Yes	No flags
mawk	Yes	Yes	10	Yes	Yes
sshpass	Yes	Yes	10	Yes	Yes
tee	Yes	Yes	10	Yes	Yes
cp	Yes	Yes	10	Yes	Yes
telnet	Yes	Yes	10	Yes	No flags
busybox	Yes	Yes	10	Yes	Yes
dash	Yes	Yes	10	Yes	Yes
sftp	Yes	Yes	10	Yes	No flags
zsh	Yes	Yes	10	Yes	Yes
base64	Yes	Yes	10	Yes	No
ssh	Yes	Yes	10	Yes	Yes
openssl	Yes	Yes	10	Yes	Yes
nmap	Yes	Yes	10	Yes	Yes
ed	Yes	Yes	10	Yes	No flags
gpg	No	Yes	10	No	No
socat	Yes	Yes	10	Yes	Yes
ksh	Yes	Yes	10	Yes	Yes
mount	Yes	Yes	10	Yes	Yes
scp	Yes	Yes	10	Yes	Yes
rsync	Yes	Yes	10	Yes	Yes
wget	Yes	Yes	10	Yes	Yes
git	Yes	Yes	10	Yes	Yes
diff	Yes	Yes	10	Yes	Yes
dd	Yes	Yes	10	Yes	Yes
umount	No	Yes	10	No	No
pkexec	Yes	Yes	10	Yes	No flags

Chapter 7

Evaluation of YRG

7.1 Testing rules on Ubuntu's memory dump

The YARA scanner plugin in volatility 3, which can be used for testing, is in different distribution, REMnux Linux. The "yarascan.YaraScan" plugin scans the kernel memory using YARA rules. There is also the linux.vmayarascan plugin which can scan the virtual memory using the YARA rules too. For this research the first plugin is the most critical as it can cover all the YARA rules that our YRG created and scans all the system memory. If a memory researcher uses this YARA rules file to Volatility 3 he will have the following results:

Command: ***python3 vol.py -f /home/remnux/memdump.lime yarascan.YaraScan --yara-file /home/remnux/test.yar > /home/remnux/results.txt***

```
 1 Volatility 3 Framework 2.26.0
 2
 3 Offset  Rule    Component       Value
 4
 5 0x9df01225da31  ksh_    $cmd1
 6 6b 73 68 20 2d 63                               ksh -c
 7 0x9df015ab36c0  mv_     $code
 8 89 fb 4c 89 e2 48 89 f5 e8 c3 f2 ff ff 48 85 c0 ..L..H.......H..
 9 74 2e 45 31 c0 4d 85 ed 74 04 49 89 45 00 48 8b t.E1.M..t.I.E.H.
10 0x9df015b428c7  mv_     $str2
11 4d 75 6c 74 69 2d 63 61 6c 6c 20 69 6e 76 6f 63 Multi-call invoc
12 61 74 69 6f 6e                                  ation
13 0x9df015b42915  mv_     $str3
14 25 73 20 6f 6e 6c 69 6e 65 20 68 65 6c 70 3a 20 %s online help:
15 3c 25 73 3e                                     <%s>
16 0x9df015b42dd8  mv_     $str8
17 54 72 79 20 27 25 73 20 2d 2d 68 65 6c 70 27 20 Try '%s --help'
18 66 6f 72 20 6d 6f 72 65 20 69 6e 66 6f 72 6d 61 for more informa
19 74 69 6f 6e 2e                                  tion.
20 0x9df0159213f0  tee_    $code
21 04 22 84 c0 75 ea e9 ac f1 ff ff 48 89 ef 49 89 ."..u......H..I.
22 db 0f b6 54 24 10 48 8b 5c 24 40 44 0f b6 84 24 ...T$.H.\$@D...$
23 0x9df015b428c7  tee_    $str1
24 4d 75 6c 74 69 2d 63 61 6c 6c 20 69 6e 76 6f 63 Multi-call invoc
25 61 74 69 6f 6e                                  ation
26 0x9df015b42915  tee_    $str2
27 25 73 20 6f 6e 6c 69 6e 65 20 68 65 6c 70 3a 20 %s online help:
28 3c 25 73 3e                                     <%s>
29 0x9df015b42dd8  tee_    $str3
30 54 72 79 20 27 25 73 20 2d 2d 68 65 6c 70 27 20 Try '%s --help'
31 66 6f 72 20 6d 6f 72 65 20 69 6e 66 6f 72 6d 61 for more informa
32 74 69 6f 6e 2e                                  tion.
33 0x9df015b42e00  tee_    $str4
```

Figure 12: Part of Volatility 3 YARA scanning that has been printed inside a file.

The results cover 1731 YARA detection results overall.

7.2 Results

Volatility 3 prints YARA hit results that include the memory offset, rule name, component name, and component value (shown as hexadecimal and ASCII strings). Because so many binaries are reported and a large number of components are triggered, the raw output can be difficult to interpret. Visualizations help clarify which components were detected and which were not. In such diagrams the X-axis lists the names of components that were detected, while the Y-axis shows how many times each component was detected.

Important note: YARA reports a binary as having been detected simply because its components were found in memory. A YARA hit does not imply that the binary is malicious. A binary's malicious behavior requires additional correlation with Volatility plugins

The rules and their components were the following:

Rule for wget: all of it 10 strings components were detected twice in different offsets.

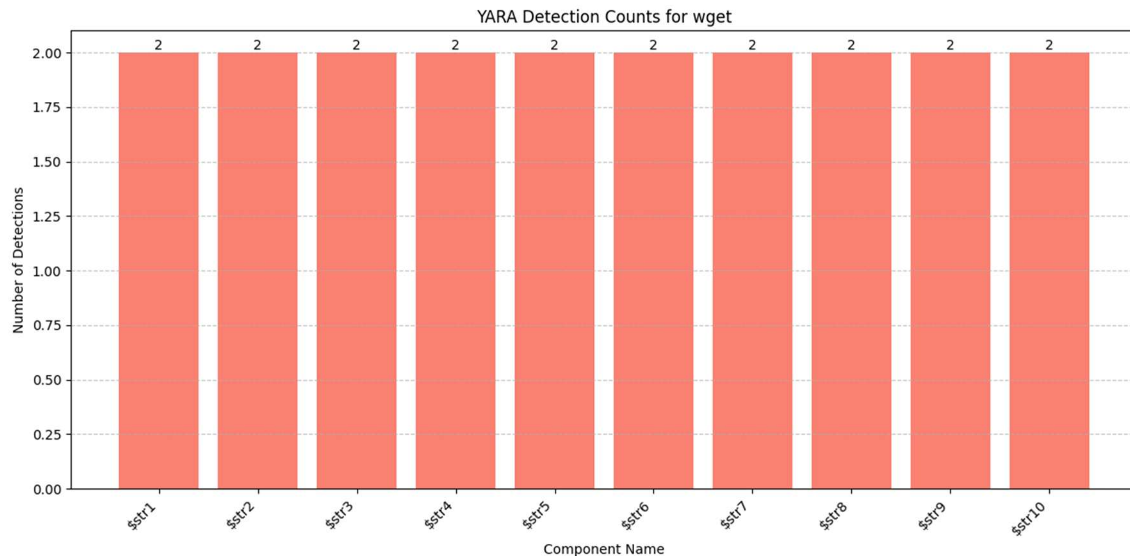

Figure 13: YARA detection for wget.

Rule for umount: Even though this binary does not belong to GTFOBins its presence is strong. All its strings were detected more than once.

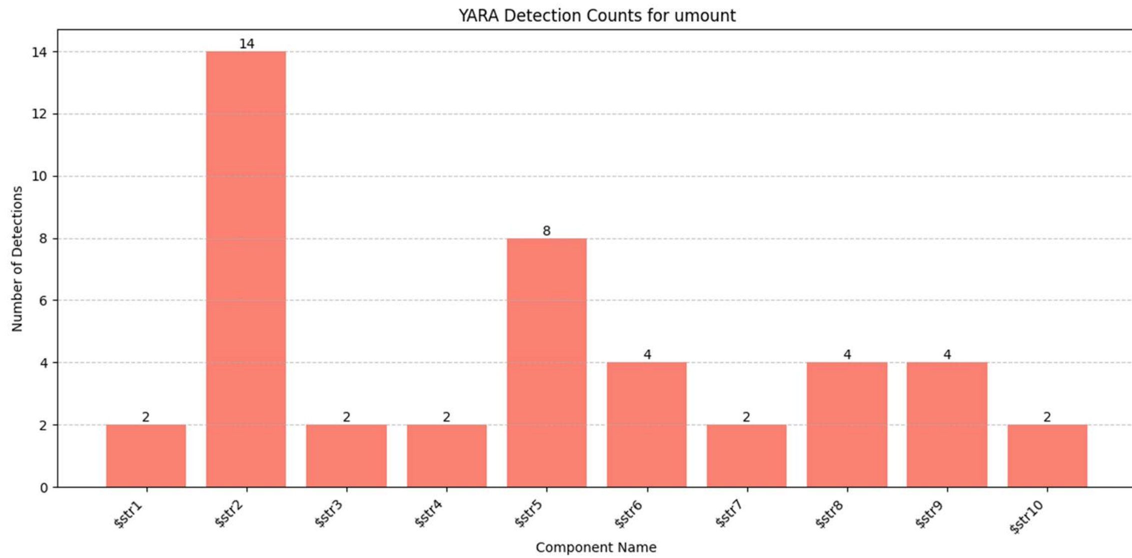

Figure 14: YARA detection for umount.

Rule for tee: The tee binary has been detected by its code and its strings str1, str2, str3, str4, str7, str8 and str10.

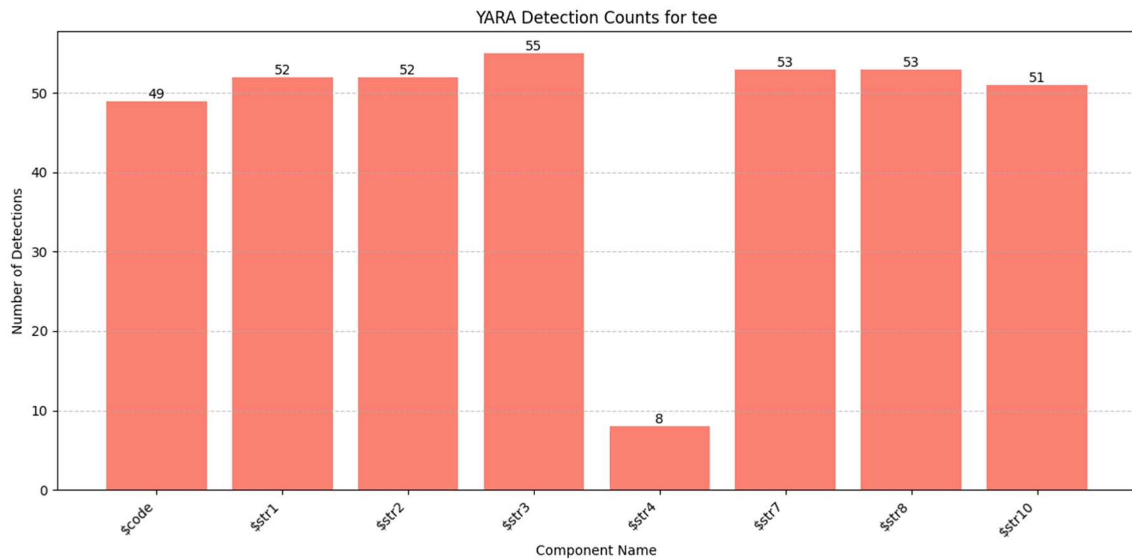

Figure 15: YARA detection for tee.

Rule for sshpass: The binary's presence had been detected by its code and all its strings twice.

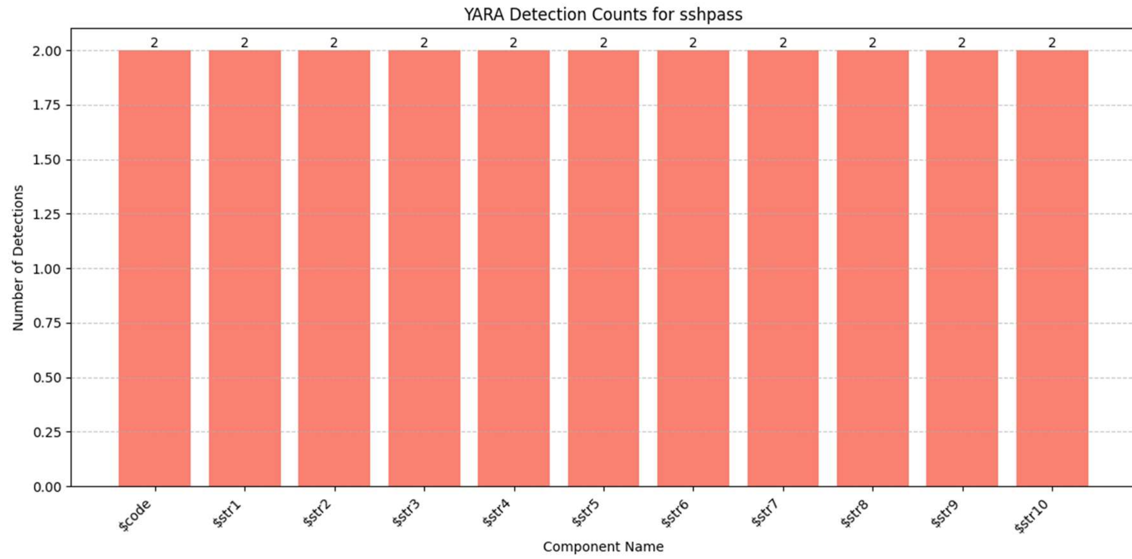

Figure 16: YARA detection for sshpass.

Rule for ssh: The binary's presence had been detected by its code and all its strings twice.

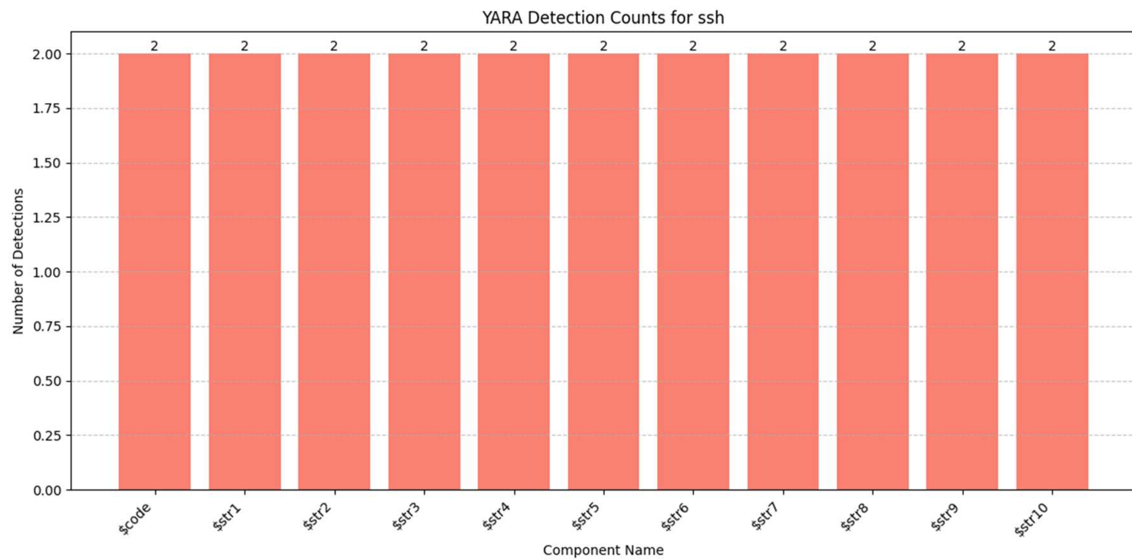

Figure 17: YARA detection for ssh.

Rule for socat: The binary's presence had been detected by its code once and all its strings twice.

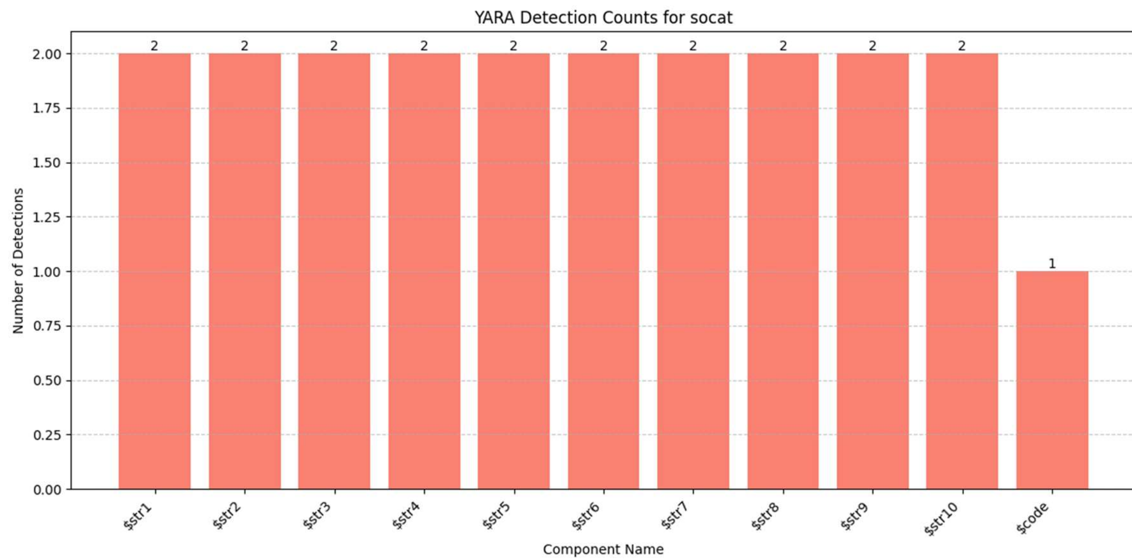

Figure 18: YARA detection for socat.

Rule for sftp: The binary's presence had been detected by all its strings. Some of its strings (str4, str6 and str10) more than four times.

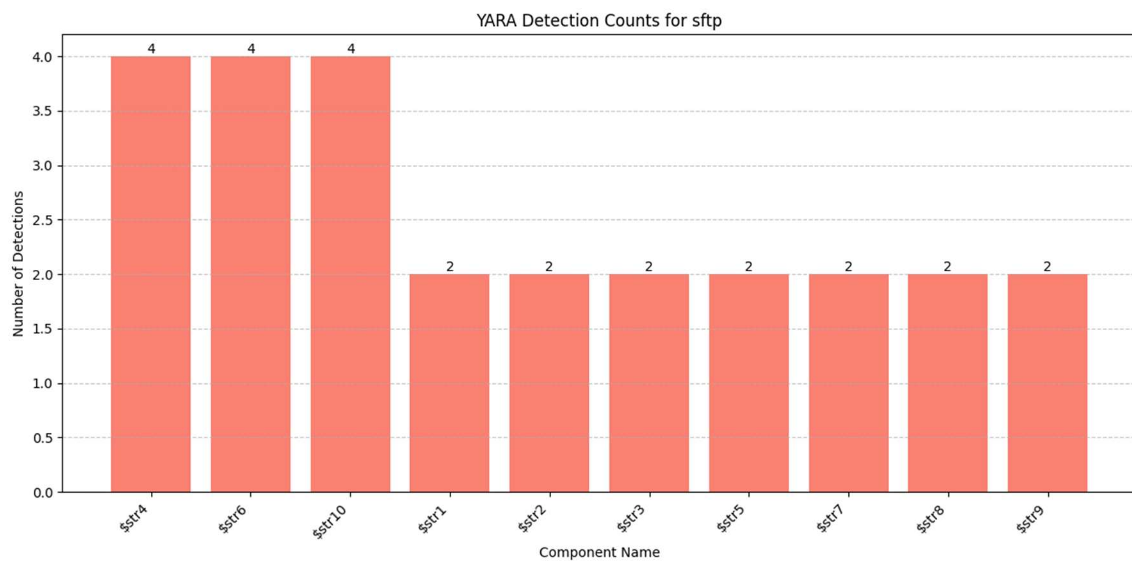

Figure 19: YARA detection for sftp.

Rule for scp: The binary's presence had been detected by all of it strings twice.

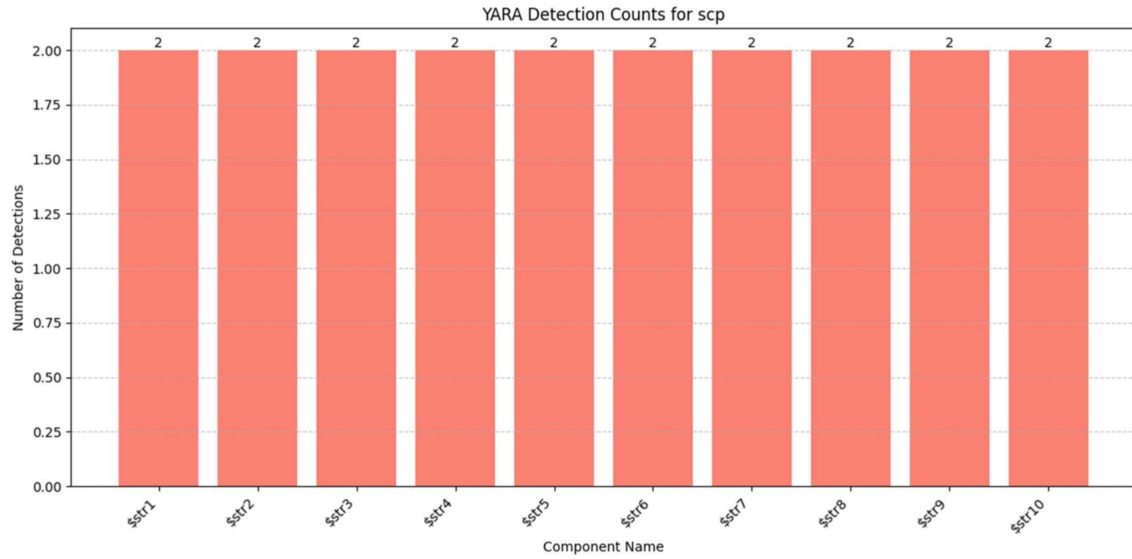

Figure 20: YARA detection for scp.

Rule for rsync: The binary's presence had been detected by its code and all its strings twice.

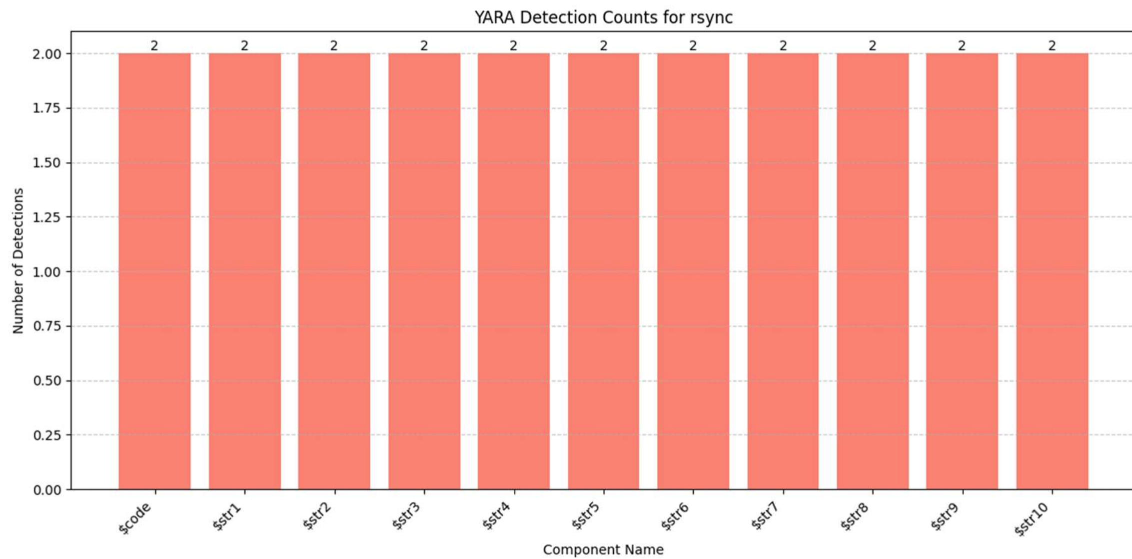

Figure 21: YARA detection for rsync.

Rule for pkexec: The binary's presence had been detected by all its strings twice and str5 eight times.

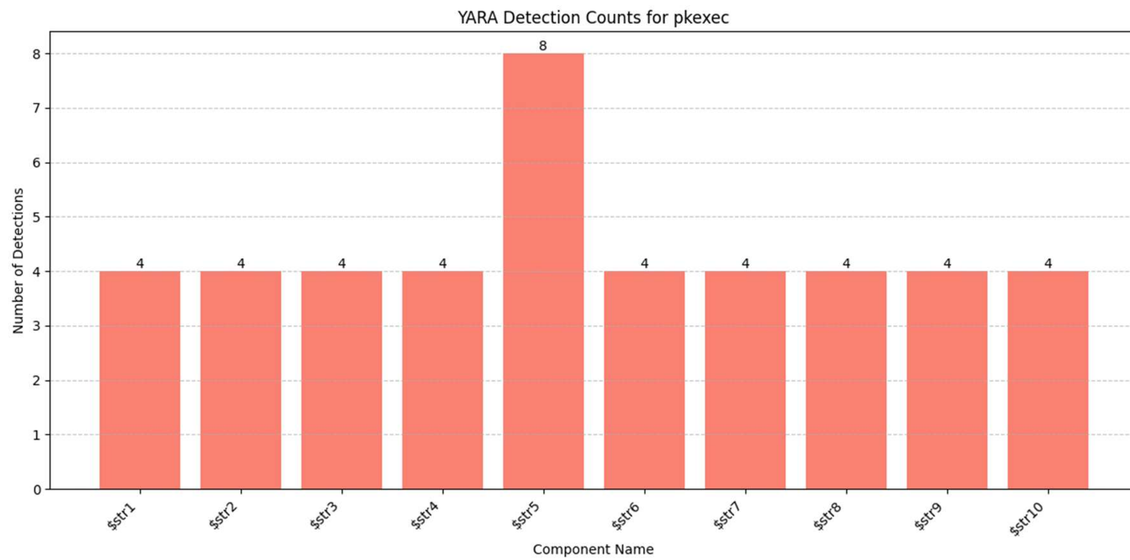

Figure 22: YARA detection for pkexec.

Rule for nmap: The binary's presence had been detected by all its strings twice.

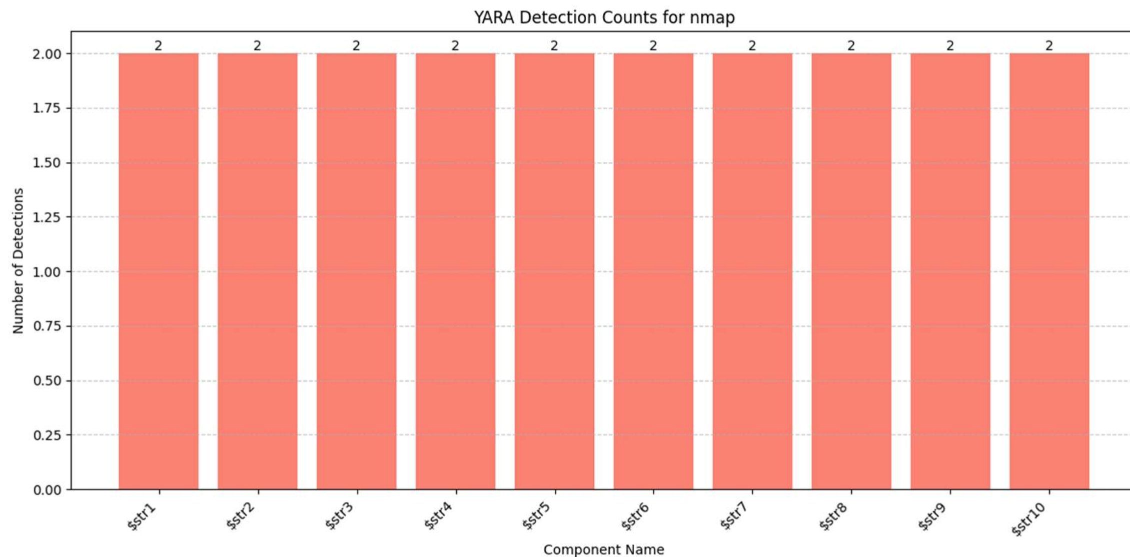

Figure 23: YARA detection for nmap.

Rule for nc: The binary's presence had been detected by its code and all its strings twice. Also, YARA detects cmd2 eight times in memory.

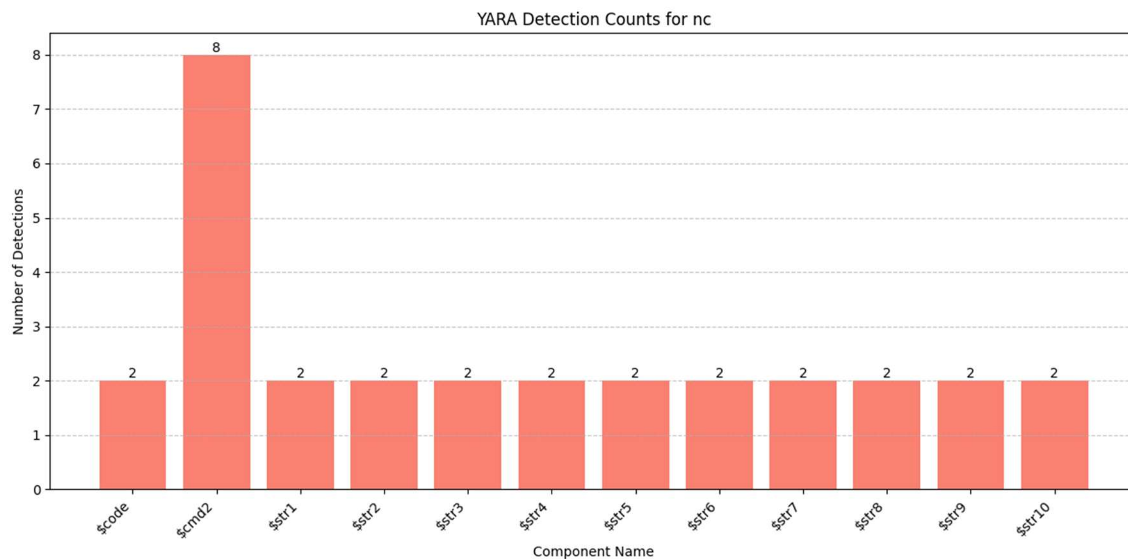

Figure 24: YARA detection for nc.

Rule for mv: The binary's presence had been detected by its code and some of it strings more than once.

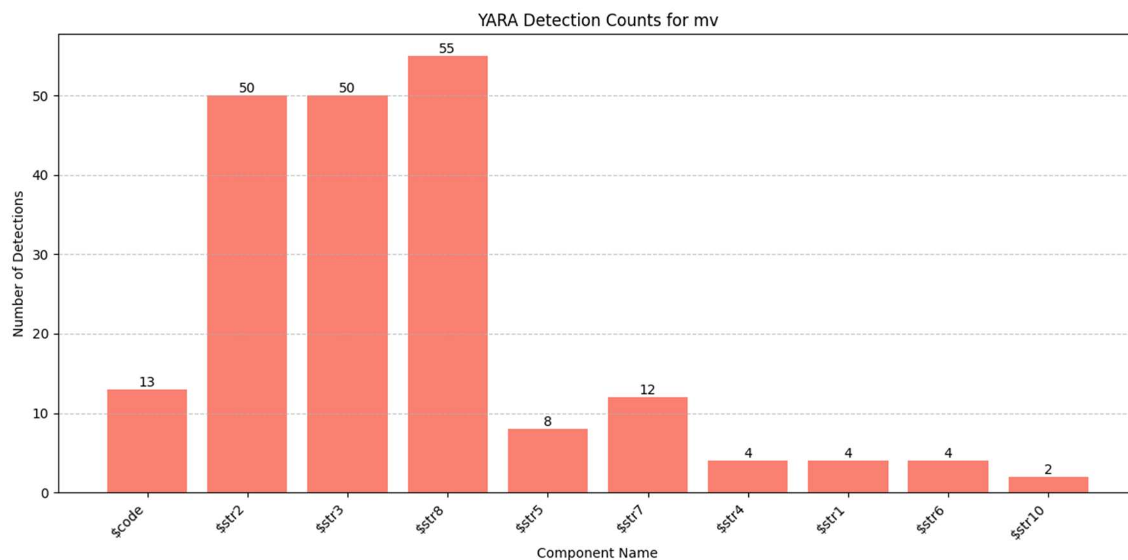

Figure 25:YARA detection for mv.

Rule for mount: The binary's presence had been detected by all its strings. The strings str5, str7 and str8 had been detected four times while str2 eight. The others were detected twice.

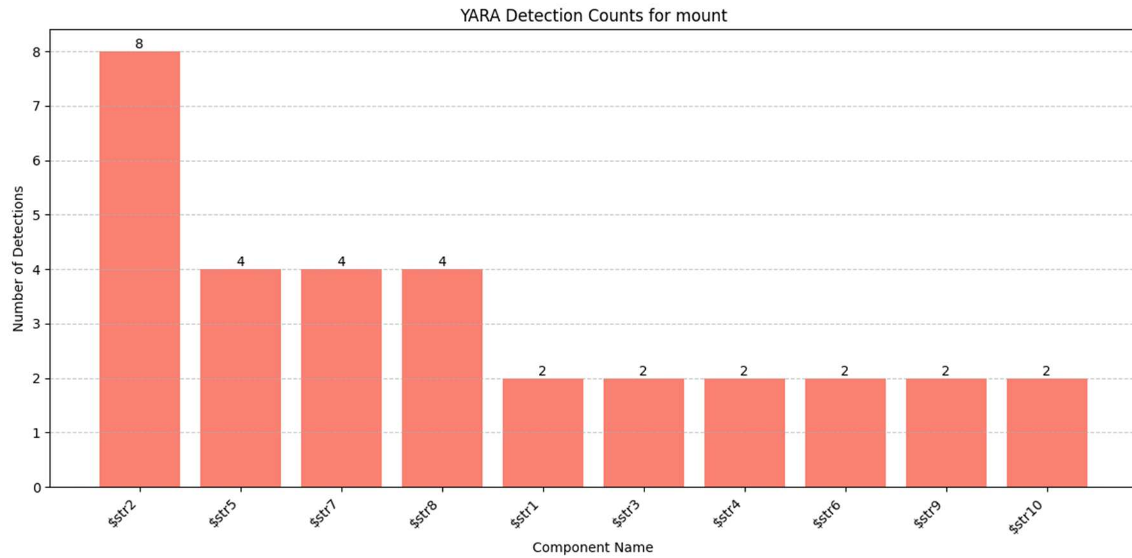

Figure 26:YARA detection for mount.

Rule for mawk: The binary's presence had been detected by all its strings twice, in exception to str4 which had been detected four times.

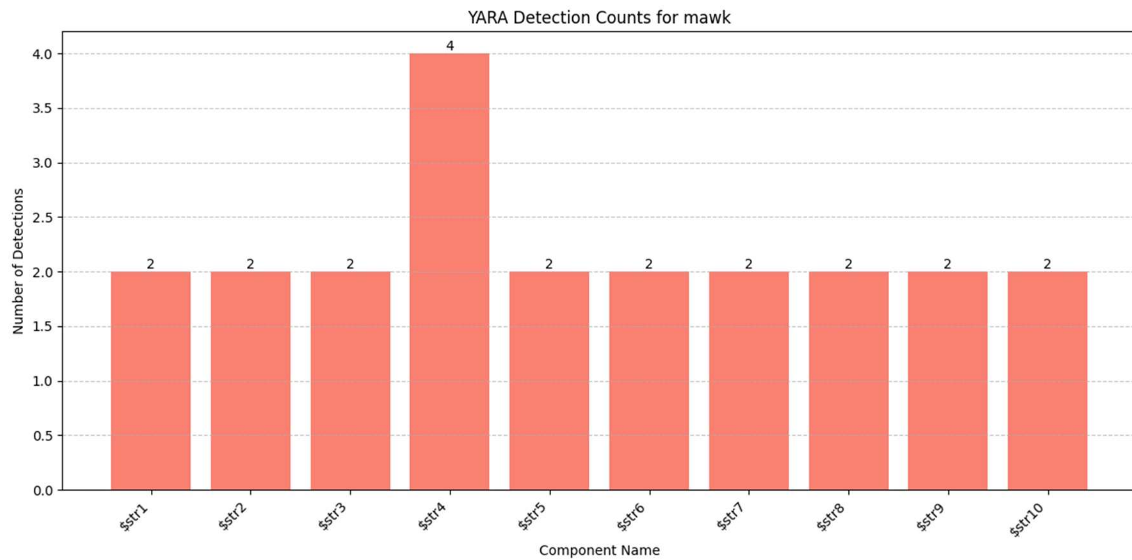

Figure 27: YARA detection for mawk.

Rule for ksh: The binary's presence had been detected by all its strings twice and the command cmd1 seven times.

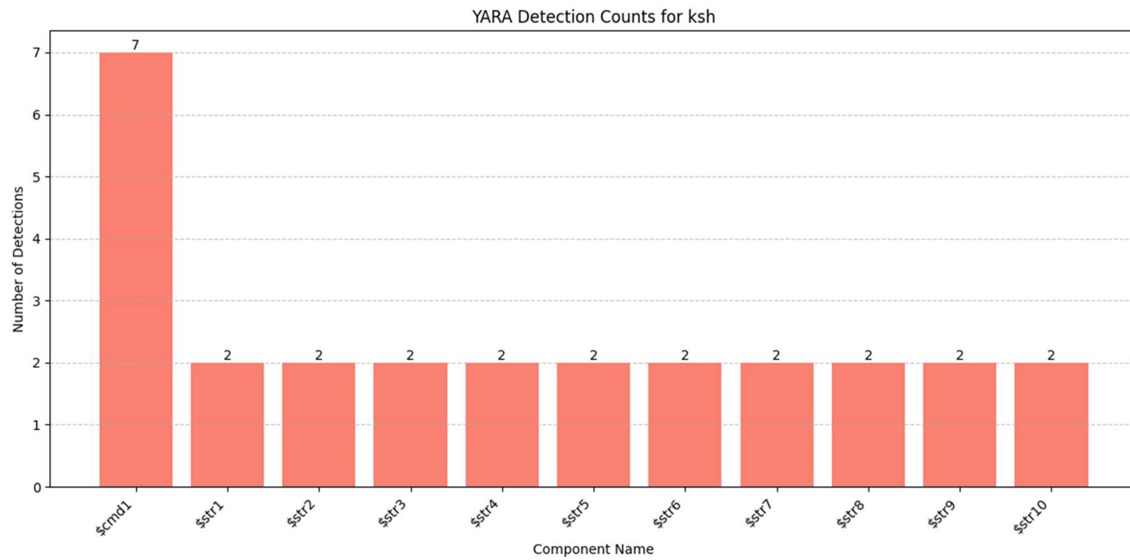

Figure 28: YARA detection for ksh.

Rule for git: The binary's presence had been detected by its commands cmd5 and cmd6.

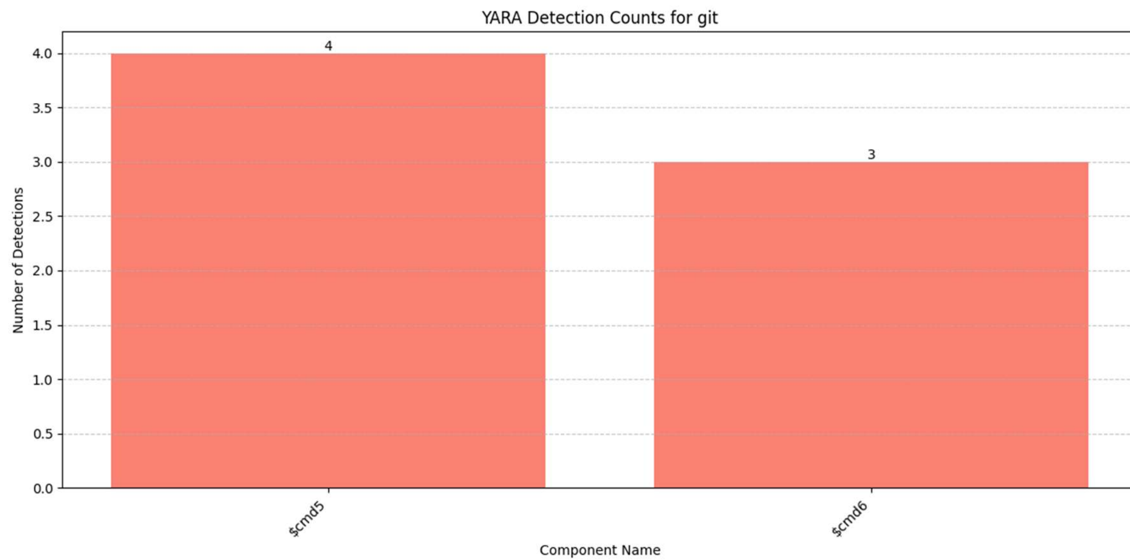

Figure 29: YARA detection for git.

Rule for ed: The binary's presence had been detected by its strings str6, str7 and str9 ten times while str8 had been detected eleven times.

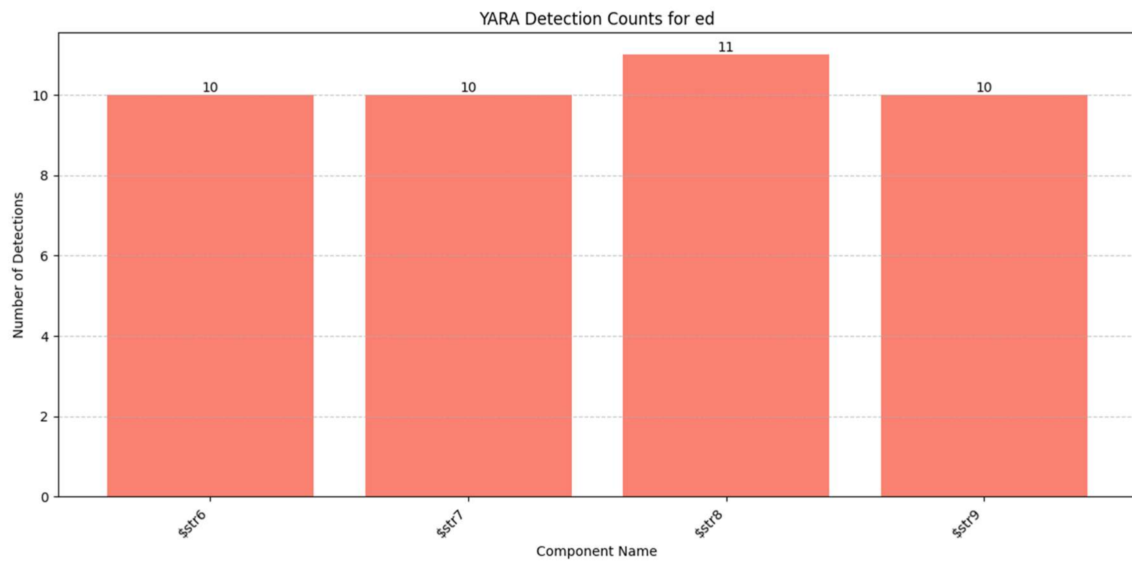

Figure 30: YARA detection for ed.

Rule for diff: The binary's presence had been detected once by its strings str1, str2 and str5.

Figure 31: YARA detection for diff.

Rule for dd: The binary's presence had been detected by its code, its strings str9 and str10 and its commands cmd1 and cmd2.

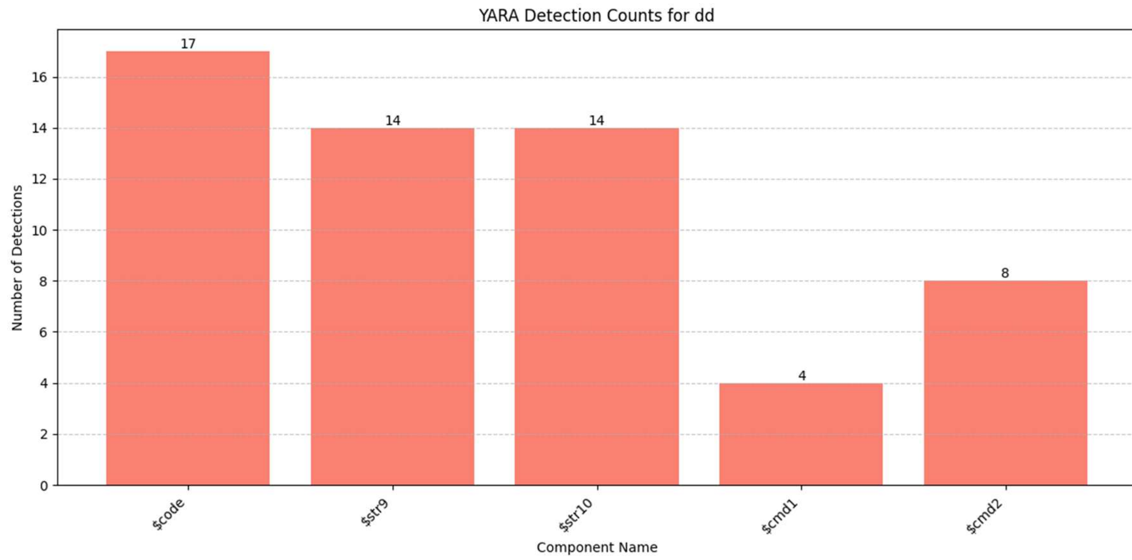

Figure 32: YARA detection for dd.

Rule for dash: The binary's presence had been detected by its code, all its strings and its command cmd1.

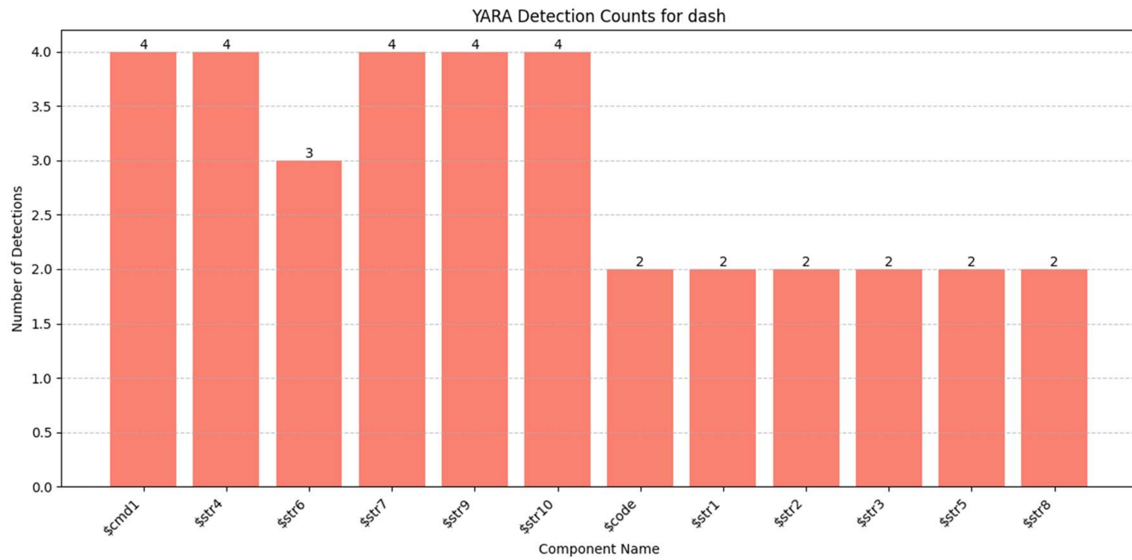

Figure 33: YARA detection for dash.

Rule for curl: The binary's presence had been detected by its code and all of it strings twice.

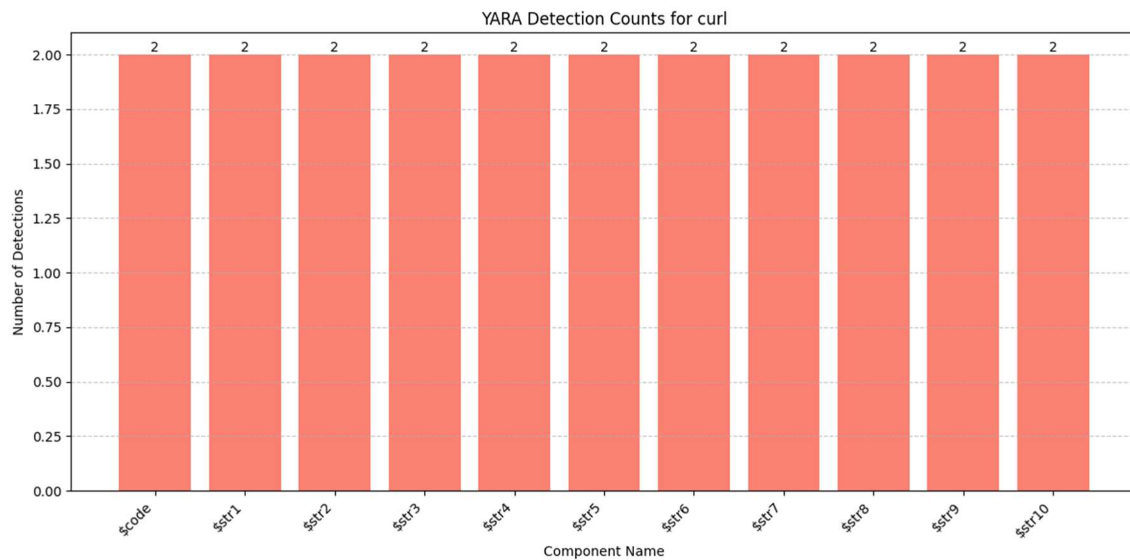

Figure 34: YARA detection for curl.

Rule for cp: The binary's presence had been detected by some of its strings at least twice and the sudo command.

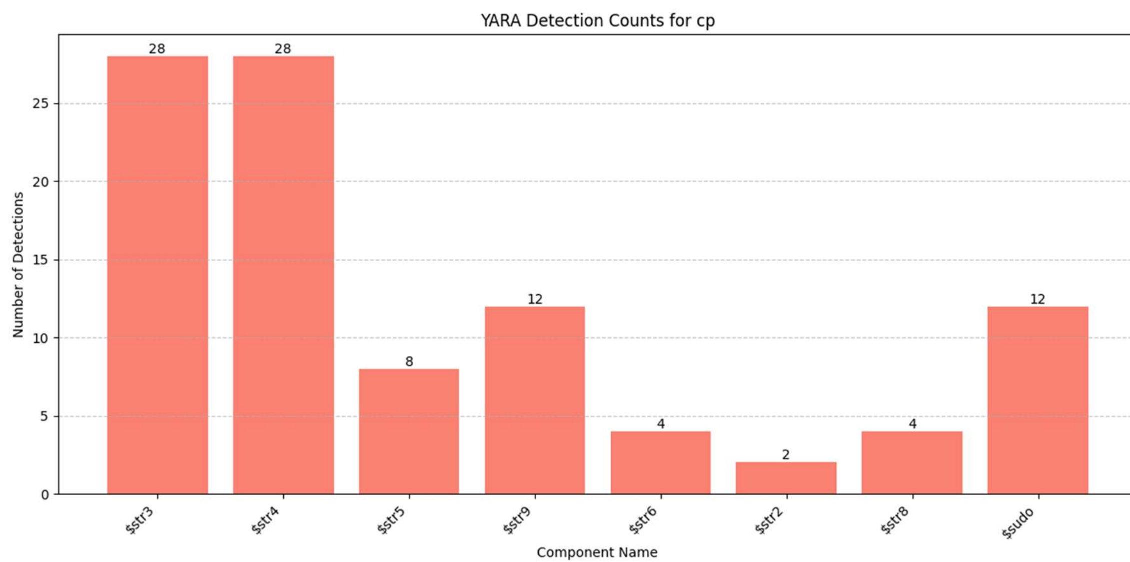

Figure 35: YARA detection for cp.

Rule for busybox: The binary's presence had been detected by all its strings twice and commands cmd2 and cmd4.

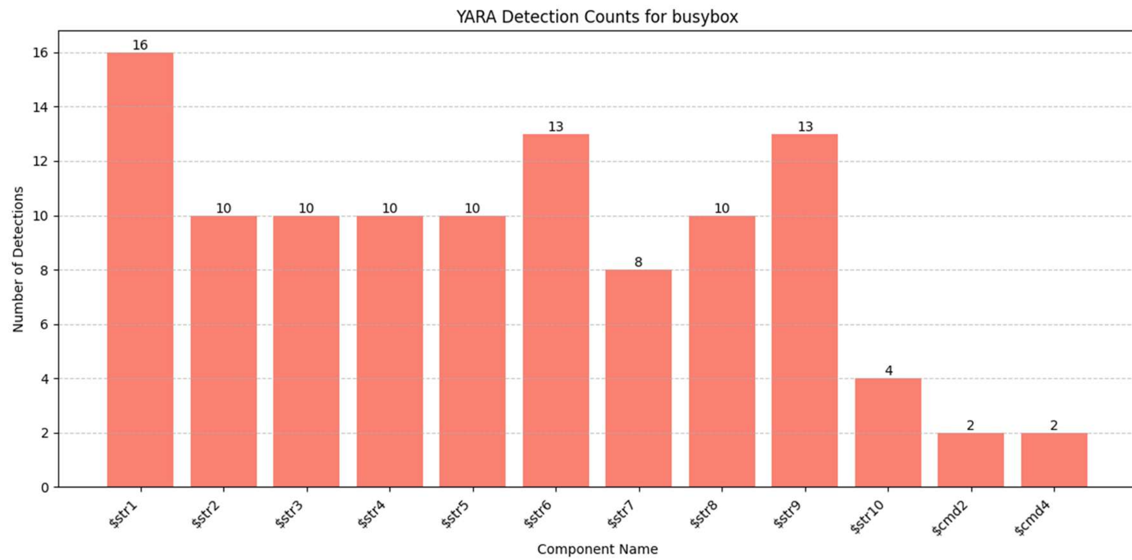

Figure 36: YARA detection for busybox.

Rule for base64: The binary's presence had been detected by its code and some of its strings (str5 and str8 hadn't been detected).

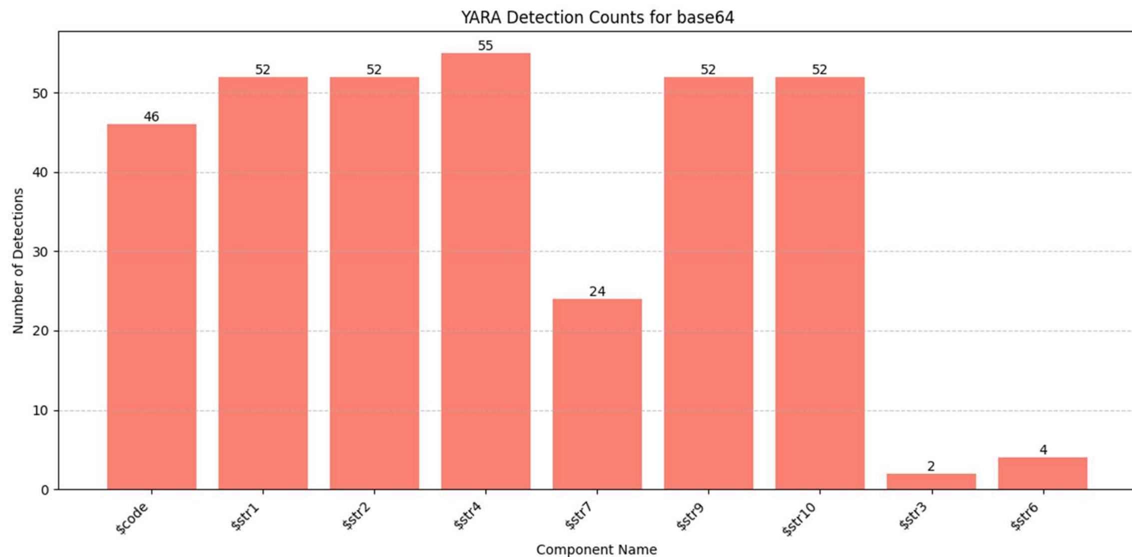

Figure 37: YARA detection for base64.

Rule for rsync: The binary's presence had been detected by all its strings twice.

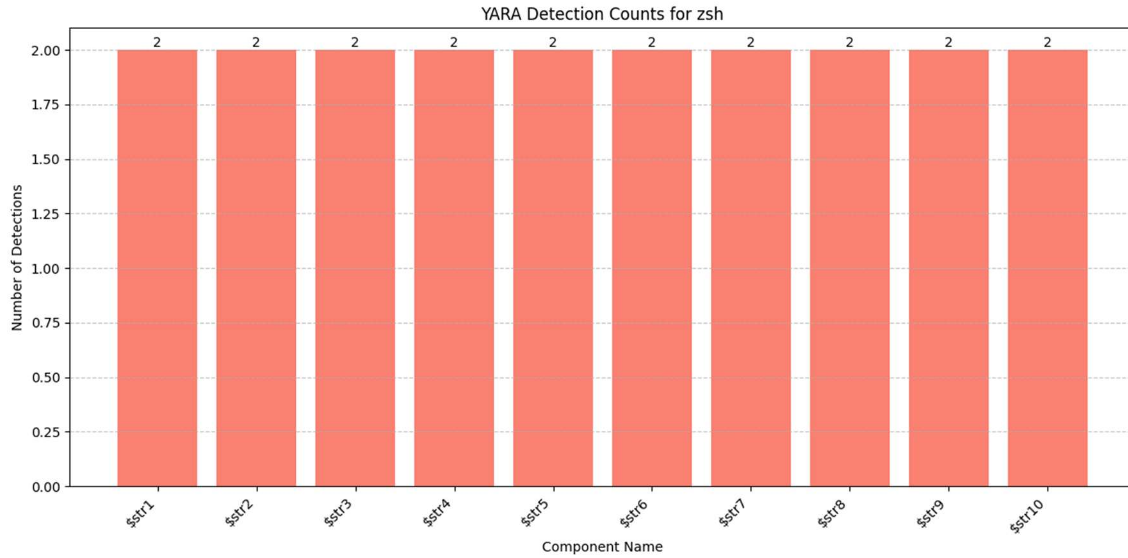

Figure 38: YARA detection for zsh.

The following figure shows the aggregated count of YARA hits per binary. Of the 30 YARA rules applied, none matched components belonging to the binaries **openssl**, **gpg**, **gdb**, or **telnet**. The horizontal (X-axis) lists the names of binaries that were flagged, while the vertical (Y-axis) shows how many times each binary's components were detected in memory.

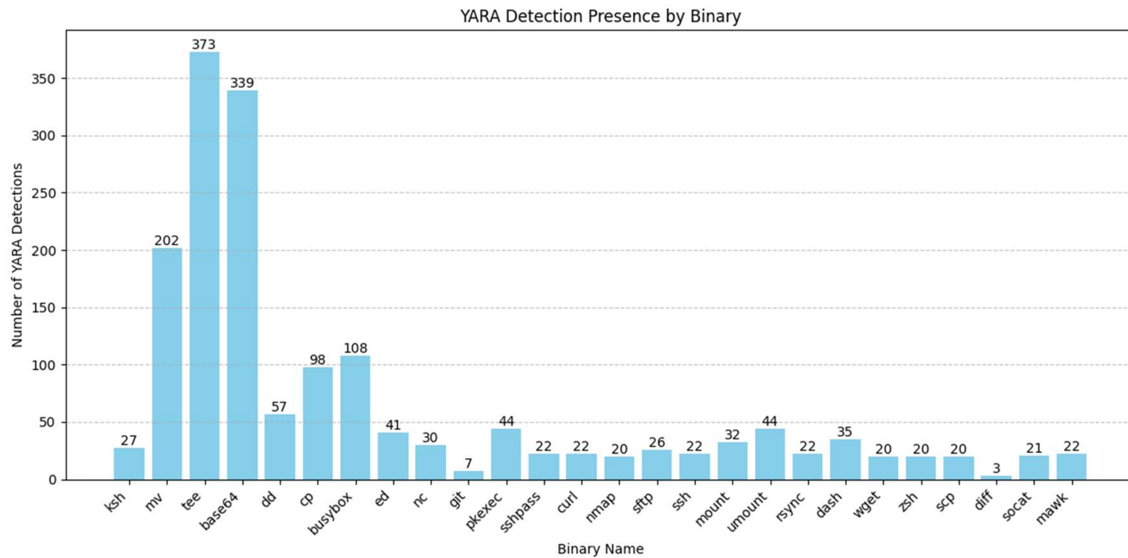

Figure 39: YARA detection overall on all 30 binaries.

7.3 Findings Analysis

Now the results need to be further analyzed. However, YARA flagged the binary presence based on strings, hex and commands. The presence does not mean that the binary has been used for malicious purposes or even executed.

The YARA plugin tool for Volatility is a pattern-based detection tool, this means that it may flag a binary or binaries using a specific rule based on strings, commands, ELF headers or libraries that may passed undetected. This is a serious issue for memory forensics investigators, because they need to be absolutely certain that the behavior of a specific binary or binaries, is or is not malicious. How can a YARA hit detect a binary's malicious behavior?

In Volatility 3 YARA plugin, some binaries like nc, nmap, wget, curl, ksh were shown. Whether they had any possible malicious behavior or not, they needed further investigation by other Volatility plugins. The YARA hits will be analyzed as the following:

- **Benign binaries:** These were identified by YARA, were legitimately executed, and are present in memory.
- **Potentially malicious binaries:** YARA flagged these files; they may exhibit malicious behavior (note that any behaviors cited are simulated, not actual).
- **False-positive binaries:** YARA marked these files, but other Volatility 3 plugins fail to detect them.
- **False-negative binaries:** These binaries exist in memory yet were not flagged by YARA.

From the investigation for the nc presence, it has presented the following results, as shown in the figures below:

```
remnux@remnux:~/volatility3$ python3 vol.py -f /home/remnux/memdump.lime linux.pslist | grep "nc"
0x9df00f0a0000.0837     837     1tackingirqbalance      0       0       0       0       2025-10-18 14:22:09.222107 UTC      Disabled
0x9df053aedf40  2036    2036    2034    nc      0       0       0       0       2025-10-18 14:23:06.469222 UTC      Disabled
remnux@remnux:~/volatility3$ python3 vol.py -f /home/remnux/memdump.lime linux.psscan | grep "nc"
0x10f0a0000100.0837     837     1tackingirqbalance      TASK_RUNNING
0x153aedf40     2036    2036    2034    nc      TASK_RUNNING
0x1605c0000     2040    2040    2034    rsync   EXIT_DEAD
0x24228e880     837     837     1       irqbalance      TASK_RUNNING
0x28c7527c0     2036    2036    2034    nc      TASK_RUNNING
0x299624880     2040    2040    2034    rsync   EXIT_DEAD
remnux@remnux:~/volatility3$ python3 vol.py -f /home/remnux/memdump.lime linux.psaux | grep "nc"
763gress1  100.0systemd-timesyn /lib/systemd/systemd-timesyncd
837     1       irqbalance      /usr/sbin/irqbalance --foreground
1622    1555    ssh-agent       /usr/bin/ssh-agent /usr/bin/im-launch env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu
1639    1444    at-spi-bus-laun /usr/libexec/at-spi-bus-launcher
2036    2034    nc      nc -l -p 5555
```

Figure 40: The nc binary exists in a running state and might be maliciously working because it opens a connection to the internet. Its existence and behavior had been found using psslist,psscan and psaux plugins of Volatility 3.

```
2058    bash    2025-10-18 14:23:11.000000 UTC  # - Requires: iproute2 (ip), gdb (for gcore), socat, netcat (nc) packages to be installed for full effect.
2058    bash    2025-10-18 14:23:11.000000 UTC  for cmd in ip gdb socat nc gcore; do   if ! command -v "$cmd" >/dev/null 2>&1; then     echo "WARNING: $cmd not found in PATH. I
nstall it if you want full functionality (apt install <$cmd>).";   fi; done
2058    bash    2025-10-18 14:23:11.000000 UTC  which nc
```

Figure 41: The bash plugin Volatility 3 shows that nc has been present and in a possible bash command.

```
remnux@remnux:~/volatility3$ python3 vol.py -f /home/remnux/memdump.lime linux.vmayarascan --yara-file /home/remnux/test1.yar
Volatility 3 Framework 2.26.0
Progress:  100.00               Stacking attempts finished
Offset  PID     Rule    Component       Value

0x55c45287e4e7  1976    nc_     $cmd2
6e 63 20 2d 6c                                  nc -l
0x55c452b0475b  1976    nc_     $cmd2
6e 63 20 2d 6c                                  nc -l
0x558234a8d420  2034    nc_     $cmd2
6e 63 20 2d 6c                                  nc -l
0x55d302a930ef  2036    nc_     $str1
77 72 69 74 65 20 66 61 69 6c 65 64 20 28 25 7a write failed (%z
75 2f 32 29                                     u/2)
0x55d302a93132  2036    nc_     $str2
73 65 72 76 69 63 65 20 22 25 73 22 20 75 6e 6b service "%s" unk
6e 6f 77 6e                                     nown
0x55d302a9315b  2036    nc_     $str3
73 65 74 20 49 50 76 36 20 74 72 61 66 66 69 63 set IPv6 traffic
20 63 6c 61 73 73                                class
0x55d302a93172  2036    nc_     $str4
73 65 74 20 54 43 50 20 72 65 63 65 69 76 65 20 set TCP receive
62 75 66 66 65 72 20 73 69 7a 65                buffer size
0x55d302a9318e  2036    nc_     $str5
73 65 74 20 54 43 50 20 73 65 6e 64 20 62 75 66 set TCP send buf
66 65 72 20 73 69 7a 65                         fer size
0x55d302a931b2  2036    nc_     $str6
73 65 74 20 49 50 76 36 20 75 6e 69 63 61 73 74 set IPv6 unicast
20 68 6f 70 73                                   hops
0x55d302a931d7  2036    nc_     $str7
73 65 74 20 49 50 76 36 20 6d 69 6e 20 68 6f 70 set IPv6 min hop
20 63 6f 75 6e 74                                count
0x55d302a9320a  2036    nc_     $str8
67 65 74 73 6f 63 6b 6f 70 74 20 65 72 72 6f 72 getsockopt error
3a 20 25 73                                     : %s
0x55d302a93256  2036    nc_     $str9
63 72 65 61 74 65 20 75 6e 69 78 20 73 6f 63 6b create unix sock
65 74 20 66 61 69 6c 65 64                      et failed
```

Figure 42: The nc binary is present and in virtual memory using YARA detection (linux.vmayarascan plugin).

From the use of the plugins, as shown in the figures above, the nc binary was indeed executed and was running until the time the memory was dumped. This information is the result of the use of LiME program. Next the psaux plugin shows that a possible reverse shell connection had been opened by an attacker to enter the system.

Using the same methodology is being investigated for the other 29 binaries:

Binary curl

- YARA detection: The rule was detected
- linux.pslist: No output was shown.
- linux.psscan: Displayed the exit state.
- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands.

- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary executed normally and showed no malicious behavior.

Binary wget

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: Showed in an exit state
- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary executed normally and showed no malicious behavior.

Binary scp

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: Displayed the exit state.
- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary executed normally and showed no malicious behavior.

Binary sftp

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: Displayed the exit state.

- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary executed normally and showed no malicious behavior.

Binary ssh

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown. that relates to ssh only (it showed ssh-agent).
- linux.psscan: Displayed the exit state.
- linux.psaux: No output was shown. that relates to ssh only (it showed ssh-agent).
- linux.bash: Displayed legitimate commands.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary executed normally and showed no malicious behavior.

Binary sshpass

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: Displayed the exit state.
- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary executed normally and showed no malicious behavior.

Binary nmap

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary may not appear in the psscan output, but bash history shows it was used with the 'which' and 'apt install' commands. Despite this, there is no indication of suspicious activity. Therefore, the investigator cannot conclude that the YARA alert was a false positive, since the binary was indeed present in memory.

Binary busybox

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: Displayed the exit state.
- linux.psaux: No output was shown.
- linux.bash: Displayed a legitimate command.
- linux.vmayarascan: YARA detects its rule in virtual memory (strings only)
- Summary: The binary executed normally and showed no malicious behavior.

Binary ksh

- YARA detection: The rule was detected by YARA.
- linux.pslist: The process was presented
- linux.psscan: Displayed the running state.

- linux.psaux: Displayed a suspicious command (ksh -c sleep 300).
- linux.bash: Displayed legitimate command.
- linux.vmayarascan: The rule was detected by YARA, in virtual memory (commands and strings)
- Summary: While running in memory, the binary suspiciously spawned a sleep process. Such behavior can be indicative of malware, as sleep routines are often used to delay execution or facilitate stealthy techniques like process injection.

Binary zsh

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary may not appear in the psscan output, but bash history shows it was used with the 'which' and 'apt install' commands. Despite this, there is no indication of suspicious activity. Therefore, the investigator cannot conclude that the YARA alert was a false positive, since the binary was indeed present in memory.

Binary dash

- YARA detection: The rule was detected by YARA.
- linux.pslist: The process was presented
- linux.psscan: Displayed the running state.
- linux.psaux: Displayed a suspicious command (dash -c sleep 300)
- linux.bash: Displayed legitimate commands.

- linux.vmayarascan: The rule was detected by YARA, in virtual memory (commands and strings)
- Summary: While running in memory, the binary suspiciously spawned a sleep process. Such behavior can be indicative of malware, as sleep routines are often used to delay execution or facilitate stealthy techniques like process injection.

Binary mv

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: The rule was detected by YARA, in virtual memory (strings only)
- Summary: The binary itself did not manifest malicious activity. It may therefore represent a false positive YARA hit or simply be invoked by another legitimate program in the operating system.

Binary cp

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown. that relates to cp only (it showed cpuhp and acpid)
- linux.psscan: No output was shown. that relates to cp only
- linux.psaux: No output was shown. that relates to cp only
- linux.bash: Displayed cp -r and sudo cp -r commands (legitimate)
- linux.vmayarascan: The rule was detected by YARA, in virtual memory (sudo only)

- Summary: The binary may not appear in the psscan output, but bash history shows it was executed using the ‘cp -r’ and ‘sudo cp -r’ commands, which were also detected by vmayarascan. However, no suspicious behavior was observed. Therefore, the investigator cannot conclude that the YARA alert was a false positive, since the binary was present in memory.

Binary base64

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: The rule was detected by YARA, in virtual memory (strings only)
- Summary: The binary itself did not manifest malicious activity. It may therefore represent a false positive YARA hit or simply be invoked by another legitimate program in the operating system.

Binary ed

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: The rule was detected by YARA, in virtual memory (strings only)
- Summary: The binary itself did not manifest malicious activity. It may therefore represent a false positive YARA hit or simply be invoked by another legitimate program in the operating system.

Binary tee

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: The rule was detected by YARA, in virtual memory (strings only)
- Summary: The binary itself did not manifest malicious activity. It may therefore represent a false positive YARA hit or simply be invoked by another legitimate program in the operating system.

Binary telnet

- YARA detection: The rule wasn't detected by YARA
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary was not present in the operating system's memory, so YARA correctly reported no detection (not a false negative).

Binary gpg

- YARA detection: The rule wasn't detected by YARA
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.

- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary was not present in the operating system's memory, so YARA correctly reported no detection (not a false negative).

Binary gdb

- YARA detection: The rule wasn't detected by YARA
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: Displayed gdb inside a loop condition of a bash command (did not indicate any execution)
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary was not present in the operating system's memory, so YARA correctly reported no detection (not a false negative).

Binary openssl

- YARA detection: The rule wasn't detected by YARA
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary was not present in the operating system's memory, so YARA correctly reported no detection (not a false negative).

Binary pkexec

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.

- linux.psscan: Showed in an exit state
- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary executed normally and showed no malicious behavior.

Binary rsync

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: Showed in an exit state
- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary executed normally and showed no malicious behavior.

Binary mount

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary itself did not manifest malicious activity. It may therefore represent a false positive YARA hit or simply be invoked by another legitimate program in the operating system.

Binary umount

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary itself did not manifest malicious activity. It may therefore represent a false positive YARA hit or simply be invoked by another legitimate program in the operating system.

Binary dd

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: : The rule was detected by YARA, in virtual memory (commands only)
- Summary: The binary itself did not manifest malicious activity. It may therefore represent a false positive YARA hit or simply be invoked by another legitimate program in the operating system.

Binary diff

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory

- Summary: The binary itself did not manifest malicious activity. It may therefore represent a false positive YARA hit or simply be invoked by another legitimate program in the operating system.

Binary git

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands ("git clone" command).
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary may not appear in the psscan output, but bash history shows it was executed using the "git clone" command, which is not included in the YARA rule's command list. Despite this, no suspicious behavior was observed. Moreover, the YARA detection was not a false positive, as the binary was present in memory even though it was not identified through the expected commands.

Binary socat

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: Displayed legitimate commands ("which socat" command).
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary may not appear in the psscan output, but bash history shows it was executed with the 'which socat' command. However, no suspicious behavior was identified. Therefore, the investigator cannot

conclude that the YARA alert was a false positive, since the binary was present in memory.

Binary mawk

- YARA detection: The rule was detected by YARA.
- linux.pslist: No output was shown.
- linux.psscan: No output was shown.
- linux.psaux: No output was shown.
- linux.bash: No output was shown.
- linux.vmayarascan: The rule wasn't detected by YARA, in virtual memory
- Summary: The binary itself did not manifest malicious activity. It may therefore represent a false positive YARA hit or simply be invoked by another legitimate program in the operating system.

Chapter 8

Discussion

8.1 Key findings

Employing the 30 YARA rules generated by the YRG, the study executed a systematic, rule driven inspection of each binary within the designated evaluation test set. The resulting observations provide empirical evidence regarding both the operational robustness of the rule set and the specific limits in which further refinement may be warranted. Among the 30 binaries analyzed, 26 elicited at least one signature match, yielding a hit rate of 86.7 % and indicating substantial coverage of the binary artifacts represented in the test set. The remaining four binaries produced no matches, a result that supports the specificity of the signatures and suggests that the rule set does not exhibit undue permissiveness or false positive bias.

However, nine binaries produced matches that could represent either false positives or normal behavior triggered by another application, highlighting the importance of contextual evaluation rather than relying solely on static signature detection. Furthermore, although the git binary had indeed existed and was correctly flagged by the YARA rule, its observed command output deviated from what would typically be expected, indicating either an issue with the rule's criteria or a misunderstanding of the binary's condition during analysis.

In addition, three of the binaries contained command strings that could be interpreted as indicative of malicious intent, warranting deeper investigation, whereas the cross-verification of each binary with Volatility 3 revealed that 14 of the 30 binaries, which had been flagged by YARA, showed that they exist and have no signs of malicious activity in memory, thereby reinforcing the reliability of the rule set for those samples.

Overall, this experiment shows that the YRG's YARA rules achieve a high detection rate while maintaining a relatively low false positive profile on binaries' existent. Although there are some binaries, especially ones that might indirectly be used by another application, that needed deeper analysis to prove their existent in memory.

8.2 Limitations

Several important limitations need to be mentioned, as they influence both the scope and the reliability of the thesis' findings.

Firstly, YARA performs purely static pattern matching: it scans the raw byte stream that is present in a file at the moment the scan is run. Because of this, any code that is generated, modified, or obfuscated at run-time will not appear in the binary image and therefore cannot be matched by a YARA rule. Some examples can be JIT-compiled sections, self-decrypting payloads, or code that is assembled from a small template at load time. In effect, the rule set can only capture artifacts that are guaranteed to be present in every execution instance but limits coverage against sophisticated evasive techniques.

Secondly, the potential that false positives might be non-trivial. There is a big probability that any of the legitimate binaries might share common strings or sections that can trigger alerts, necessitating a subsequent layer of validations to distinguish genuine threats from benign artifacts.

Thirdly, the current rule generator is tailored exclusively to Linux ELF binaries. As a result, this excludes other widely used executable formats such as scripts files, like Python files (.py), Perl files (.pl) and of course Bash shells files (.sh).

In addition, the YRG for ELFs requires further refinement, specifically through the incorporation of a more extensive set of unique strings, hexadecimal opcodes and GTFOBins command references. Without refining the generator will likely enhance the discriminative power of the "yarascan.YaraScan" plugin and reduce both missed detections and false alarms.

What is more, the generated YARA rules are derived from binaries compiled and packaged for Ubuntu 20.04. As a result, detection effectiveness may vary across Linux distributions or different package versions due to differences in compiler options, binary stripping and embedded strings.

Noted that the experimental sample size is relatively modest. Only 30 YARA rules were evaluated against a similarly small set of binaries. If the experiment has expanded the study to encompass 100, 500, or even thousands of binaries, it would provide a richer dataset, uncover additional patterns, and allow for a more robust assessment of rule accuracy and resilience

Taken as a whole, these constraints emphasize the necessity for continued methodological refinement, expanded support for additional binary formats, and the use of larger, more representative validation datasets in order to fully realize the capabilities of YARA-based binary analysis.

Chapter 9

Conclusion

9.1 Overview of this research thesis

Fileless attacks have become a common weapon to modern adversaries. On Linux the lack of readily available detection mechanisms for native Living-off-the-Land Binaries (LOLBins) has left a noticeable gap in threat-hunting workflows. This project addressed that gap by creating a lightweight, on-the-fly YARA rule generator for ELF binaries and integrating the generated signatures with the Volatility 3 memory-forensic framework.

9.2 Core contributions

The core contributions of this research project are the following:

1. **Automated YARA Rule Generation** – The Python 3 script inspects ELF files, extracts structural markers (.text hex opcodes, GTFOBins commands, .rodata strings) and produces a YARA rule that reflects those characteristics. The generator runs locally, enabling analysts to generate up-to-date signatures whenever a new LOLBin is encountered.
2. **Seamless Volatility 3 Integration** – Generated rules are fed directly into Volatility 3, allowing a single workflow to identify and contextualize executions of target binaries in both live memory and offline dumps.
3. **Three-Tier Detection Framework** – Analysis of the memory dump yielded three distinct outcomes for each matched binary:
    - ***Benign*****:** The binary is part of the expected software stack and shows no anomalous activity.
    - ***False-Positive / Legitimate Use*****:** The binary is flagged but is known to be invoked by non-malicious processes.
    - ***Suspicious / Potentially Malicious*****:** The execution pattern correlates with other indicators of compromise (e.g., abnormal parent processes, unexpected network activity).

These findings demonstrate that YARA signatures for LOLBins provide a valuable first-line filter that reduces the analyst's workload and surfaces potential threats for deeper investigation.

9.3 Future work

Despite the limitations of YARA rule binaries, the research can still be expanded. Some of the main future work items that need to be implemented are:

- **Adding scripting binaries:** GTFOBins' scripts such as msfconsole can also be incorporated into the YARA rule collection by examining their code, unique strings, and commands.
- **Testing more YARA rule binaries:** Testing additional binaries to determine which yield more results and prove useful for memory-forensic investigations.
- **Testing YARA rule binaries on different distributions:** Linux distributions such as Debian or Arch may contain the same binaries but behave differently and may have distinct strings, hex values, and commands. The Python YRG for ELF should generate different YARA rules for these binaries.
- **Updating LOLBins on the GTFOBins list:** In the future, new binaries will be added to GTFOBins; the Python program should therefore download the new list, compare it with the old one and if the list has changed, then it will discard the old one and keep the new ones. Therefore, the generator will always create YARA rules with their updated commands. Also, in the future upgrades, the program generator should implement a more sophisticated and improved method of extracting the GTFOBins commands.

9.4 Final remarks

By automating YARA rule creation and integrating it with a memory forensic framework, this research provides a practical and repeatable methodology for locating potentially high-

risk Linux binaries in memory. While static signatures alone are insufficient to conclusively detect fileless malware, the approach offers value as an initial filtering and investigative aid within a layered memory forensics workflow. Furthermore, even though static signatures cannot eliminate all adversarial techniques, they are a critical component of a layered defense strategy, enabling analysts to quickly flag, triage, and investigate the illicit activity of LOLBins before it can inflict lasting damage.

The human role in this research is simple: choosing which binaries to scan, inspecting YARA rule outputs, validating YARA hits against memory-forensic evidence and performing additional behavioral analysis or contextual triage. However, this thesis suggests that YARA rule generation and validation are supplementary. It speeds up repetitive parts of the workflow but does not replace the cybersecurity analyst's judgment or behavioral analysis after a YARA match is flagged.

Bibliography

Abyss-W4tcher. (2025). *Collection of Linux and macOS Volatility3 Intermediate Symbol Files (ISF)*. Retrieved from Github: https://github.com/Abyss-W4tcher/volatility3-symbols

Andrew Case, G. G. (2017). Memory forensics: The path forward. *Elsevier*, 23-33.

Ângello Cássio Vasconcelos Oliveira, D. C. (2025). *Supervised Learning Algorithm used for LOLBins detection in Linux Machines.* Brasília: Universidade de Brasília (UnB).

Boelen, M. (2025, March 12). *Understanding memory information on Linux systems*. (Linux Audit) Retrieved from Linux Audit: https://linux-audit.com/understanding-memory-information-on-linux-systems/

Cohen, M. (2017). Scanning memory with YARA. *Elsevier*, 34-43.

cppreference.com. (2025). *std::free.* Retrieved from https://en.cppreference.com/w/c/memory/free

cppreference.com. (2025). *std::malloc.* Retrieved from https://en.cppreference.com/w/c/memory/malloc

David Lillis, B. A. (2016). Current Challenges and future research areas for digital forensic investigation. *arXiv*, 11.

Experts, T. C. (2023, February 14). *aquasec*. Retrieved from How Fileless Attacks Work and How to Detect and Prevent Them: https://www.aquasec.com/cloud-native-academy/application-security/fileless-attacks/

Fan Dang, Z. L. (2019). Understanding Fileless Attacks on Linux-based IoT Devices with. *MobiSys*, (p. 13). Seoul.

ForensicTools.dev. (2024). *Understanding the Legal Considerations in Digital Forensics: What Every Investigator Should Know*. Retrieved from forensictools.dev: https://forensictools.dev/2024/09/29/top-5-memory-forensics-tools-for-cyber-investigations/

githubUserKA. (2025). *YARA Rules memory binary generator for Linux*. Retrieved from Github: https://github.com/githubUserKA/YARA-Rules-memory-binary-generator-for-Linux

Gorman, M. (2003). *Slab Allocator*. Retrieved from kernel.org: https://www.kernel.org/doc/gorman/html/understand/understand011.html

GTFOBins. (2025). Retrieved from https://gtfobins.github.io/

Hacıoğlu, S. Ö. (2025, December 4). *What Are YARA Rules? A Complete 2025 Guide with Examples*. Retrieved from www.picussecurity.com: https://yara.readthedocs.io/en/stable/getting-started/what-is-yara.html

hwinfo man | Linux Command Library. (2024). Retrieved from https://linuxcommandlibrary.com/man/hwinfo

Ishrag Hamid, M. M. (2024). A Comprehensive Literature Review on Volatile Memory Forensics. *Electronics*, 24.

Kerrisk, M. (n.d.). brk(2) - Linux manual page. Linux Documentation Project (man7.org). Retrieved from https://www.man7.org/linux/man-pages/man2/brk.2.html

Kerrisk, M. (n.d.). mmap(2) - Linux manual page. Linux Documentation Project (man7.org). Retrieved from https://www.man7.org/linux/man-pages/man2/mmap.2.html

Lohot, N. (2023, April 20). *Linux Internals: Memory Management Explained*. (InfosecBytes) Retrieved from https://infosecbytes.io/linux-internals-a-deep-dive-into-memory-management/

MITRE ATT&CK. (2025). *ATT&CK*. (MITRE ATT&CK) Retrieved from MITRE ATT&CK: https://attack.mitre.org/#

MITRE ATT&CK. (2025). *Enterprise Techniques*. (MITRE ATT&CK) Retrieved from MITRE ATT&CK: https://attack.mitre.org/techniques/enterprise/

Myra Khalid, M. I. (2020). Automatic YARA Rule Generation. *IEEE* (p. 5). IEEE.

Oishi, A. Z. (2023, November 20). *Architecture of Linux Operating System*. (LinuxSimply) Retrieved from https://linuxsimply.com/linux-basics/introduction/architecture-of-linux-operating-system/

Oishi, A. Z. (2023, April 29). *History of Linux* . (LinuxSimply) Retrieved from https://linuxsimply.com/linux-basics/introduction/history-of-linux/

Ryan Robinson, N. F. (2024, April 23). *Memory Analysis 101: Memory Threats and Forensic Tools*. (Intezer) Retrieved from https://intezer.com/blog/memory-analysis-forensic-tools/

SentinelOne. (2024, July 8). *What are LOLBins? | How Attackers Use LOLBins?* (SentinelOne) Retrieved from https://www.sentinelone.com/blog/how-do-attackers-use-lolbins-in-fileless-attacks/

SentinelOne. (2025, July 29). *What is Fileless Malware? How to Detect and Prevent Them?* (SentinelOne) Retrieved from https://www.sentinelone.com/cybersecurity-101/threat-intelligence/fileless-malware/#common-fileless-malware-techniques

Stokes, P. (2019, April 25). *Lazarus APT Targets Mac Users with Poisoned Word Document*. (Sentinel Labs) Retrieved from https://www.sentinelone.com/labs/lazarus-apt-targets-mac-users-with-poisoned-word-document/

Sudhakar, S. K. (2020). An emerging threat Fileless malware: a survey and research challenges. *SpringerOpen*, 12.

VirusTotal Revision. (2022). *Writing YARA rules*. Retrieved from yara.readthedocs.io: https://yara.readthedocs.io/en/stable/writingrules.html

Volatility Foundation. (2025). *Volatility 3: The volatile memory extraction framework*. Retrieved from Github: https://github.com/volatilityfoundation/volatility3