ShiftyLoader: Syscall-free Reflective Code Injection in the Linux Operating System

Michele Salvatori[1,*], Giorgio Bernardinetti[1,2,*], Francesco Quaglia[1,2] and Giuseppe Bianchi[1,2]

[1]CNIT National Network Assurance and Monitoring (NAM) Lab, Rome, IT
[2]University of Rome "Tor Vergata", Rome, IT

Abstract

Reflective code injection is a technique frequently employed to elude detection mechanisms in cybersecurity. The majority of Antivirus (AV) and Endpoint Detection and Response (EDR) systems detect such threat by monitoring system calls during program execution. In this paper, we introduce ShiftyLoader, a tool specifically developed for Linux systems and ELF binaries which employs a straightforward, but highly effective, strategy to circumvent standard AV/EDR defenses: deliberately refraining from making system calls during the binary loading process. While the concept is simple, such a strategy appears to be neither adopted in common existing loaders nor accounted for in commercial AV/EDR behavioral detection strategies, as experimentally confirmed by our evaluation. Interestingly, the actual implementation of such an approach does not present any significant hurdle, except for the necessary detailed low-level knowledge of the target operating system. To assess the effectiveness of the proposed approach, after incorporating known encryption and payload obfuscation techniques to thwart baseline signature matching defenses, we conducted an experimental evaluation. This involved testing 162 Linux-specific malware samples across various online sandboxes and 11 commercial AV/EDR solutions. Our results show that, at present, none of the examined defensive solutions have the ability to detect the malicious activity. This highlights the limitation of relying solely on system call tracing, even when performed at the exceptionally fine granularity achievable through eBPF-based kernel defenses.

Keywords

Malware, Evasion strategies, Antivirus/EDR, Linux/eBPF

1. Introduction

Malware has evolved over the years, becoming more sophisticated and harder to detect using traditional methods. While static analysis remains a crucial tool in the cybersecurity arsenal, it has its limitations in identifying complex malware that exhibits dynamic and polymorphic behavior. To address these challenges, behavioral analysis [1, 2, 3] has emerged as a powerful approach, enabling security professionals to monitor a system's behavior and identify potentially malicious activities in real-time. This shift from solely relying on static analysis to a more dynamic approach has proven instrumental in staying one step ahead of cyber threats.

ITASEC 2024: The Italian Conference on CyberSecurity, April 08–11, 2024, Salerno, Italy
*Corresponding authors.
michele.salvatori@cnit.it (M. Salvatori); giorgio.bernardinetti@cnit.it (G. Bernardinetti); francesco.quaglia@uniroma2.it (F. Quaglia); giuseppe.bianchi@uniroma2.it (G. Bianchi)
0009-0005-5510-8769 (M. Salvatori); 0000-0001-6222-0365 (G. Bernardinetti)

Along this path, Linux systems, the actual specific focus of this article, provide particularly effective tools for controlling and monitoring the system-level behavior of applications. A relatively recent trend consists in exploiting kernel-level tools and solutions, such as the kernel framework provided by the extended Berkeley Packet Filter (eBPF), to capillary track system calls and network activities comprehensively [4, 5]. The granularity and extensibility of eBPF [6], in particular, offers a unique advantage in scrutinizing and controlling software behavior at the system call level, while providing an extremely robust defensive environment.

With the above context in mind, this article aims to address the following research questions:

- **RQ1**: can we devise a reflective code injection approach, to be exploited by malware, that completely avoids any system call invocation during the binary code loading process?
- **RQ2**: Are there technical hurdles to overcome in order to implement such a strategy?
- **RQ3**: Is this strategy capable of outsmarting the current defensive strategies implemented in commercial Antivirus (AV) and Endpoint Detection and Response (EDR) systems?

Not surprisingly, practitioners with sufficient in-depth knowledge of low-level Operating System (OS) mechanisms will find the answers to both **RQ1** and **RQ2** to be straightforward. Specifically, the solution to **RQ1** revolves around the recognition that the necessity for invoking system calls can be bypassed by just refactoring *one class* of functions in the system standard: `execve`-related ones. In particular, `execve` is a widely used Posix/Linux system call that, when invoked by a program, requests the OS to load a specified executable file into the current process's address space, thereby discarding the existing program and its data.

As demonstrated in this article through the actual implementation of ShiftyLoader for Linux, and thus addressing **RQ2**, the operations delegated to the OS via the `execve` system call can not only be reprogrammed by ShiftyLoader without the need to interact with the OS through an explicit system call, but such refactoring is relatively easy. There are no specific technical hurdles to overcome, and its development only requires a somewhat in-depth knowledge of certain low-level mechanisms embedded in the Linux OS. Given the simplicity of the proposed approach, it therefore comes to us as a surprise that our article appears to be among the first to explicitly propose and demonstrate a syscall-free reflective loading strategy for bypassing malware detectors, an approach which, at least based on our necessarily partial review of the related work, has apparently been so far neglected in tools documented in the literature or in relevant blog posts [7, 8, 9, 10, 11, 12, 13, 14].

More concerning is the answer we experimentally provide to **RQ3**: our evaluation conducted assessing 162 Linux-specific malware samples over 11 commercial AV/EDR including Tracee [1]—a recent kernel/eBPF-based defensive technique for intercepting system calls—shows that none of the examined defensive solutions could detect the malicious activity. We are not able to judge whether our findings imply that the currently deployed behavioral detection strategies have so far overlooked the potential of syscall-free reflective loading strategies, or have deliberately decided not to cope with such threats because of the extra monitoring cost in front of their apparently very limited deployment. But we believe our work underscores the severe limitation of relying solely on system call tracing, even when executed at the fine granularity achievable through eBPF-based kernel defenses. In essence, while system call

[1] https://github.com/aquasecurity/tracee

tracing, when properly designed [15, 16, 17] is an extremely valuable and effective approach, it is not a Panacea, prompting the necessity for a comprehensive exploration of complementary defensive strategies, which we briefly discuss in our conclusive section.

The remainder of this article is organized as follows. In Section 2, we discuss the background and related work. In Section 3, we explore the malware packing technique employed in our evasion mechanism. In Section 4, we explain the technical details of the refactoring of the "suspicious" `execve` system call, as carried out by ShiftyLoader. In Section 5, we present and analyze the results of our evaluation. Conclusions are discussed in Section 6.

2. Background and Related Work

ELF (Executable and Linkable Format) loading is the process by which executable files and shared libraries in the ELF format are loaded into memory by the OS for execution. During loading, the system parses the ELF file headers, allocates memory for code and data sections, maps segments into the appropriate memory regions, resolves symbols for dynamic linking, and manages the program's address space. Although ELF loading is typically handled by the OS, application-level software may initiate this process upon request. However, the applications themselves do not directly execute the loading operation. Moreover, this task is always handled by specialized functions in the system standard, designed specifically for ELF loading (e.g., the `exec*` functions).

The *Reflective Loading* technique [2], in contrast to traditional approaches that involve the creation of new threads or processes for running executable files stored on disk, adopts a more discreet strategy: it allocates and executes payloads directly within the memory of the host process, leaving minimal traces on the file system. This approach is commonly adopted within Windows environments for evasion purposes, as it shifts the task of loading and executing a binary from kernel to user space, thereby granting user-level code finer-grain control. Moreover, *Reflective Loading* allows to execution of file-based payloads without any disk involvement, thus presenting a significant advantage over defensive scanners that rely on disk events for their operations.

Most ELF loaders using this technique are based on system calls such as `memfd_create`, which generates an anonymous file that only lives in RAM—it has volatile backing storage—and `execveat`, which is invoked using the newly created file descriptor. The high-level steps to implement this technique are the following:

- Create an anonymous file
- Write malicious ELF payload in the anonymous file
- Execute the anonymous file

One key benefit of this technique is that the system call `memfd_create` returns a file descriptor that behaves like a regular file. However, unlike a regular file, it only resides in RAM. All backing pages of the file utilize anonymous memory. Therefore, once all references to the file are removed, the memory it uses is automatically released.

[2] https://attack.mitre.org/techniques/T1620/

However, there are notable drawbacks to this implementation. The execution of the malicious payload always involves at least one syscall [8, 9], making it susceptible to detection by a defensive system based on syscall monitoring. Furthermore, although the memory hosting the payload is anonymous and lacks a corresponding file on disk, it is still detectable through memory allocation tracing [18, 19].

Another recent technique employed by reflective ELF loaders involves the creation of a file in the *procfs* filesystem, commonly stored in RAM [10]. Nonetheless, the final step of this loader always involves an `execve` syscall. An alternative reflective loader [11] bypasses `execve` syscalls by loading an ELF file using the `dlopen` API. While the memory allocation technique remains consistent, the "execution" of the ELF is triggered via a `libc` API call without any syscall execution. However, a limitation of this method is that the target ELF to be loaded must be a shared object, which may be restrictive in certain scenarios. Furthermore, it's important to mention another category of loaders that inject code into remote processes using the `ptrace` syscall [12]. This loader inserts a shared ELF into the SSH daemon process by leveraging the capabilities of the aforementioned system call. The drawbacks include the usage of a system call, i.e. `ptrace`, the necessary privileges to read/write memory to a target process, and the limitations imposed by shared object files.

Alternatively, besides `ptrace`, there are other system calls that can manipulate the memory of a remote process—even though the same drawbacks persist, namely the usage of a system call—such as `process_vm_readv` [3] and `process_vm_writev`. These system calls facilitate data transfer between the address space of the calling process (referred to as "the local process") and the process identified by the target pid (known as "the remote process"). This data movement occurs directly between the address spaces of the two processes, bypassing kernel space. Another class of loaders operates primarily through API obfuscation. In this approach, the access to the payload source code is compiled in a way that obfuscates the APIs imported from standard libraries, along with their corresponding calls. Consequently, defensive software that relies on monitoring [2, 20], cannot determine which APIs are being invoked and what arguments are being passed. Major examples of this kind of evasion are [13] and [14]. However, a significant drawback of such techniques is their inability to create stealth payloads when the source code is inaccessible.

Table 1 summarizes the comparison of the loaders discussed above. This comparison covers memory allocation methods, utilization of suspicious APIs and/or syscalls, referenced loaders, potential limitations on the types of ELF files that can be loaded, and whether the loader requires access to the source code. It's clear that every technique has both advantages and disadvantages. However, to the best of our knowledge, there is currently no available loader that does not require access to the source code, has no limitations on the types of ELF files, and avoids using suspicious system calls for binary code execution.

[3] https://man7.org/linux/man-pages/man2/process_vm_readv.2.html

Table 1
Comparison of state-of-the-art Linux loaders

Memory	Suspicious APIs/syscalls	Ref.	ELF limits	Need source
Anonymous file	`memfd_create() + execve()`	[7, 8, 9]	N	N
`procfs` file	`execve()`	[10]	N	N
Anonymous file	`memfd_create() + dlopen()`	[11]	SO only	N
Disk	`ptrace()`	[12]	SO only	N
Disk/mmap-ed	`process_vm_{read,write}v()`	X	SO only	N
X	API Obfuscation	[13, 14]	N	Y

3. Linux Packing

The packing process that we exploit in our ShiftyLoader, is illustrated in Figure 1. It demonstrates how, starting from an ELF file—which is malicious for our intent—it generates an ELF-reflected file ready to be executed in concealed mode, meaning it is equipped with our evasion mechanism, which will be further explained later in this article. The build-chain consists of the following multiple stages:

- Encryption of the malware bytecode
- Compilation of the ELF loader using the LLVM[4]/clang toolchain
- Introduction of polymorphism through obfuscation using YansoLLVM [5]
- Linking the obfuscated bytecode with the malicious payload, embedding it into the loader

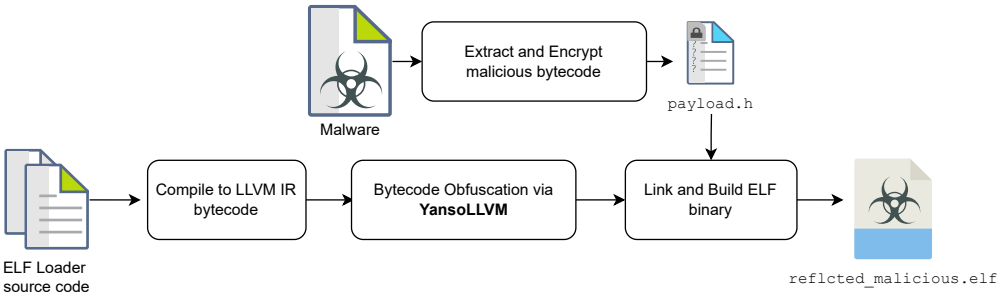


Figure 1: Packing Process

The initial step involves extracting the bytecode from the malicious ELF file and encrypting it. This process involves generating a static array that encapsulates the hexadecimal representation of the malicious payload. Subsequently, this array is append to a C header (`.h`) file. By compiling the loader with this file header, the malicious payload becomes embedded within the resultant ELF loader. This execution flow was chosen to ensure that the payload is compiled and linked with the loader only after the latter has been obfuscated.

[4]https://llvm.org
[5]https://github.com/emc2314/YANSOllvm

The encryption process employs byte-by-byte XOR encryption with a variable-size random encryption key added to the header file, enabling the subsequent run-time decryption phase. While XOR encryption is straightforward and vulnerable to brute-force attacks, the decision to use it was based on its simplicity for effectively obfuscating the payload bytecode. Indeed, the main focus of this phase is the payload concealment rather than its encryption. Choosing a more complex cipher, like those from the Advanced Encryption Standard family, would have necessitated additional libraries and functions associated with ransomware, increasing the loader's susceptibility to being flagged as potentially malicious or suspicious during analysis. Moreover, opting for more sophisticated cryptographic algorithms increases the entropy of the resulting ELF within the packing chain, which is itself an indicator of potential malicious activity.

3.1. Entropy Analysis

An IOC that malware analysts often use to determine the threat of an executable is the entropy analysis. When applied to this context, entropy measures the statistical unpredictability of a suspicious binary's data, enabling analysts to estimate the degree of encryption or obfuscation present within the file, identifying packed and encrypted regions. In our case, although XOR encryption does not significantly increase entropy, all the packed binaries exhibit a significantly high entropy value, surpassing the typical entropy of 7.2 found in packed malware [21]. Specifically, the `.data` section, where the encrypted and embedded payload is located, has a mean entropy level of 7.46, greater than that observed in the `.text` section. Reducing entropy can be done by increasing the amount of plaintext data. For this reason, a translation table consisting of 256 words randomly chosen from an English ASCII word dictionary was implemented next to the malicious payload. With this table the XORed payload can be re-encoded once again, transforming it into a list of meaningful words.

3.2. Obfuscation

Avoiding simple signature-based detection isn't just about hiding the bad stuff. When closely examining the packed malicious ELF, signatures connected to the custom ELF loader may be identified. To deal with this, a layer of obfuscation has been added to give the resultant packed file a polymorphic nature. It's relevant to state that the process of obfuscation is exclusively limited to the custom loader's source code, while the malicious payload is incorporated separately at a later stage. The obfuscation process is achieved through the LLVM/clang compile chain and leverages the LLVM's optimization component, `llvm-opt` [6]. This allows the integration of external libraries like *YansoLLVM*, a key tool for implementing polymorphism and advanced obfuscation techniques. YansoLLVM's modular approach allows developers to apply various transformations to the source code, ranging from basic conversions of logical blocks into functions to establishing inter-dependencies among different code sections through false branches. Additionally, it includes constant obfuscation using Mixed Boolean-Arithmetic (MBA).

[6] https://llvm.org/docs/CommandGuide/opt.html

4. Deployment Process: execve Refactoring

Figure 2 illustrates the deployment process of the malware using the revisited Reflective Loading technique, offered by ShiftyLoader.

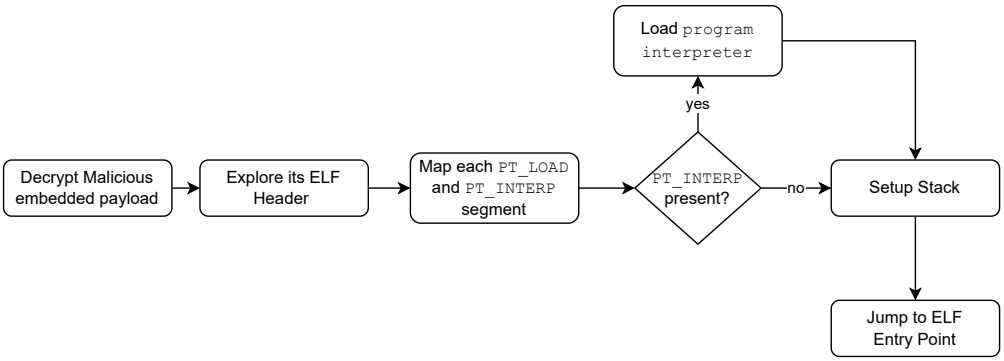


Figure 2: "Invisible" Deployment Process

The reflected ELF file, as a result of the packing process, triggers the in-memory mapping procedure for the malicious payload during execution. Following the decryption phase of the payload, only the run-time relevant content, i.e. all the loadable segments (PT_LOAD), is mapped into memory. Furthermore, this process is optimized by pre-calculating the required memory space, thereby minimizing the number of system calls to just one `mmap()` invocation. This strategy reduces the likelihood of the syscall call rate being analyzed or recognized as a malicious pattern, increasing the "invisibility" capability of the evasion mechanism. The mapping process is then completed with the proper configuration of memory protection for each mapped segment, guaranteeing the proper execution of the malware by assigning the necessary permissions like `read`, `write`, or `execute`.
To ensure a loader's versatility and compatibility also with dynamically linked ELFs, it is imperative to map in memory the *dynamic linker*, which has a dual role: it prepares the execution environment for the target ELF, and performs **relocations** for symbol references from external libraries. The loader's adaptability would be constrained if solely depended on manual relocation, potentially resulting in a loader that lacks universal effectiveness across all ELF files.

The control transfer from the ELF loader to the target malicious ELF file is the last step performed by our implementation of the Reflective Loading technique. For the execution within memory, an essential component is the entry point address of the file or the linker. This entry point typically points to the `_start` routine, which resides at the beginning of the `.text` section. However, a simple jump to the entry point is not sufficient to execute the target ELF correctly because this approach lacks the necessary parameters that the `_start` routine expects to find in the stack like `argc`, `argv` and `envp`. To achieve this, it's necessary to replicate the stack setup process by placing the necessary elements in their proper positions based on the calling convention.
Another crucial aspect in accurately reproducing the execution of the `execve` system call is

the Auxiliary Vector[7]. This vector consists of meticulously crafted key-value pairs generated by the ELF system binary loader when introducing a new executable image into a process. The information carried by this vector, such as references to the ELF header of the linker/malicious-ELF, reference to the first program header of the target executable, and the entry point of the target malicious file, regulate the interaction among the loader, the ELF file, and the underlying OS.

Once the stack and auxiliary vector setup are completed, the control flow is passed to the target ELF file or its interpreter using machine code embedded within the loader program, using the `asm`. In summary, the machine code snippet first relocates the stack pointer to the recently configured stack and then, after clearing some registers, executes a **jump** to the designated entry point using the `jmp` instruction. After the jump is performed, the malicious file is running in the address space of the loader itself, without relying on none of the `exec*` functions in the system standard.

5. Evasion Results

As previously mentioned, the evasion mechanism proposed in this paper has been tested with a dataset of 162 packed malware samples, including notable types such as Mirai, Gafgyt, and various coinminers. These samples, once packed, were deployed against a custom cloud EDR platform [22], incorporating 8 AV solutions for both Linux and Windows environments, and against solutions with advanced behavior analysis mechanisms such as eBPF-based monitoring. Table 2 and Table 3 summarize the tested defensive solutions.

Table 2
AVs in Phoenix EDR

Phoenix AVs
WithSecure Endpoint (win)
Kaspersky Anti-Virsu
ESET Endpoint Security
DrWeb Antivirus
Windows Defender
Comodo Antivirus
Clam Antivirus Scanner
AVG

Table 3
Local/Online AVs

AV/EDR
Hybrid Analysis[a]
ESET Endpoint for Linux
Kaspersky Endpoint Security for Linux
Tracee

[a]https://www.hybrid-analysis.com/

An expected outcome of our evaluation is the test against basic file scanners like ClamAV [8]. Concealing the malicious payload and introducing polymorphism into the packed version naturally eliminates matches in the signature database. For this reason, local file scanners are not the primary line of defense at the OS level but they serve as supplementary tools.

Regarding dynamic analysis, Windows-based antivirus solutions have demonstrated remarkable effectiveness despite Windows' inherent limitation of not supporting direct execution of

[7]https://www.gnu.org/software/libc/manual/html_node/Auxiliary-Vector.html
[8]https://www.clamav.net

ELF files. This exceptional performance is credited to the robust internal execution sandboxes within these antivirus solutions, enabling detailed emulation and execution of input ELF binaries for thorough behavioral analysis. However, although all 162 malware samples in their original not-packed form were successfully detected by all AV, their version packed with the evasion mechanism successfully bypassed all antivirus programs, allowing malware deployment and execution within the hosted environments on the platform.

The final testing phase evaluates defense products from Aquasec, Kaspersky, and ESET within three distinct execution environments. Tracee, which utilizes eBPF technology to monitor system calls and network events in real-time, can monitor the "Fileless Execution Detected" event, which precisely detects the Reflective Loading technique utilized in our evasion mechanism. However, it misses our in-memory execution due to its reliance on tracepoints left by the kernel at various points during the execution of the `execve` system call. Tracee and other eBPF-based tools can only detect our packed malware execution by monitoring specific memory management/allocation system calls like `mprotect`, but the extensive usage of these calls in everyday Linux operations limits the possibility of immediate/simple use of this approach.

"Kaspersky Endpoint Security for Linux" [9] and "ESET Endpoint for Linux" [10], renowned as robust solutions tailored for safeguarding Linux-based systems, *promise* comprehensive protection against diverse cyber threats, including *real-time behavioral* analysis capabilities. However, these solutions rely on specific Kernel Probes installed exclusively for particular system calls like `execve`. Consequently, similar to our findings in the prior analysis with Tracee, our evasion mechanism implemented in ShiftyLoader effectively bypasses detection by Kaspersky and ESET by avoiding the use of these specific system calls.

6. Conclusions and Limitations

In this paper, we have proposed a novel pipeline to develop reflected versions of malware that come equipped with an evasive deployment method. Through polymorphism, payload concealment, and entropy reduction, our mechanism successfully evades static and dynamic analysis by AVs. The results underscore the critical need for the development of sophisticated security solutions, emphasizing the necessity for more advanced and adaptive measures to counter evolving threats, particularly within the Linux environment. Of particular concern is the observation that with only a basic refactoring of the syscall `execve` it's possible to bypass most behavioral analysis tools, as they rely heavily on monitoring this system-call and the family of functions in the system standard which use it. Indeed, as mentioned earlier in this article, these tools cannot implement a more exhaustive monitoring approach, such as tracking `mmap` and `mprotect` syscalls, because these system calls are fundamental to the everyday operations of any Linux application and thus do not exhibit any discernible patterns of behavior that could be simply interpreted as indicative of suspicious activities.

In this regard, to the best of our knowledge, detecting evasion comprehensively requires the utilization of defense mechanisms operating at a kernel level different from system call interception. One category of analysis tools capable of achieving this is that of *memory scanners*,

[9] https://www.kaspersky.it/small-to-medium-business-security/endpoint-linux
[10] https://www.eset.com/it/aziende/endpoint-protection

with JITScanner [23] standing out. JITScanner uses YARA rules to identify in-memory malicious code. Once the content of an executable page is materialized in memory and accessed to perform an instruction fetch, a scan is triggered on the page content. If any previously defined YARA rules are matched during this scan, it signals the presence of a malicious payload in memory that could potentially execute. Consequently, a defensive system utilizing this technology could successfully identify the runtime malware decryption performed by our Reflective Loader. Since it operates at the kernel level, the only way to bypass JITScanner detection is to elude the YARA rules used. One simple approach to achieve this is flooding the malicious payload with superfluous bytes, such as `0x90` representing the NOP instruction. However, fully obfuscating a malicious file poses a not insignificant challenge due to tasks like address recomputation, that had to be performed, and the extensive set of instructions that need to be managed.

Considering the future course of this research, a potential avenue for further exploration could involve the replication of additional system calls: by replicating a broader spectrum of system calls in user-level code, our methodology could continue to evolve, enhancing its ability to circumvent detection techniques. Moreover, besides only "encrypting" the original input payload, it is of high interest the research on metamorphic engines capable of replacing the machine code of the input with another version that is semantically equivalent, although it generates a completely different signature. This mechanism has mainly three benefits: i) the encryption stage of the packer can be removed, ii) the entropy of the morphed input does not increase, and finally iii) evasion of signature-based defensive software, even JITScanner-like ones.

Acknowledgments

This work was partially supported by the project **I-NEST**, "Italian National hub Enabling and Enhancing networked applications & Services for digitally Transforming SMEs and Public Administrations" G.A. 101083398 - CUP F63C22000980006.

References

[1] C. Li, Z. Cheng, H. Zhu, L. Wang, Q. Lv, Y. Wang, N. Li, D. Sun, Dmalnet: Dynamic malware analysis based on api feature engineering and graph learning, Computers & Security 122 (2022) 102872. URL: https://www.sciencedirect.com/science/article/pii/S0167404822002668. doi:https://doi.org/10.1016/j.cose.2022.102872.

[2] D. C. D'Elia, S. Nicchi, M. Mariani, M. Marini, F. Palmaro, Designing robust api monitoring solutions, IEEE Transactions on Dependable and Secure Computing 20 (2023) 392–406. doi:10.1109/TDSC.2021.3133729.

[3] Ö. A. Aslan, R. Samet, A comprehensive review on malware detection approaches, IEEE Access 8 (2020) 6249–6271. doi:10.1109/ACCESS.2019.2963724.

[4] M. Abranches, O. Michel, E. Keller, S. Schmid, Efficient network monitoring applications in the kernel with ebpf and xdp, in: 2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 2021, pp. 28–34. doi:10.1109/NFV-SDN53031.2021.9665095.

[5] M. Bachl, J. Fabini, T. Zseby, A flow-based ids using machine learning in ebpf, 2022. arXiv:2102.09980.

[6] Z. Zhou, Y. Bi, J. Wan, Y. Zhou, Z. Li, Userspace bypass: Accelerating syscall-intensive applications, in: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), USENIX Association, Boston, MA, 2023, pp. 33–49. URL: https://www.usenix.org/conference/osdi23/presentation/zhou-zhe.

[7] R. Guo, Reflective code loading in linux — a new defense evasion technique in mitre att&ck v10, https://medium.com/confluera-engineering/reflective-code-loading-in-linux-a-new-defense-evasion-technique-in-mitre-att-ck-v10-da7da34ed301, Last Update: 2021.

[8] Stuart, In-memory-only elf execution (without tmpfs), https://magisterquis.github.io/2018/03/31/in-memory-only-elf-execution.html, Last Update: 2018.

[9] G. T. Bonicontro, Running elf executables from memory, https://www.guitmz.com/running-elf-from-memory/, Last Update: 2019.

[10] EntySecBlog, Remote in-memory elf loader, https://blog.entysec.com/2023-04-02-remote-elf-loading/, Last Update: 2023.

[11] J. M. Fernández, Loading fileless shared objects (memfd_create + dlopen), https://x-c3ll.github.io/posts/fileless-memfd_create/, Last Update: 2018.

[12] A. Chester, Linux ptrace introduction aka injecting into sshd for fun, https://blog.xpnsec.com/linux-process-injection-aka-injecting-into-sshd-for-fun/, Last Update: 2017.

[13] Y. Li, F. Kang, H. Shu, X. Xiong, Y. Zhao, R. Sun, Apiaso: A novel api call obfuscation technique based on address space obscurity, Applied Sciences 13 (2023). URL: https://www.mdpi.com/2076-3417/13/16/9056. doi:10.3390/app13169056.

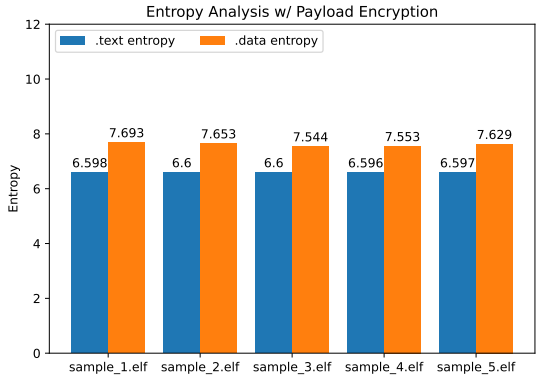
[14] Y. Kawakoya, E. Shioji, Y. Otsuki, M. Iwamura, T. Yada, Stealth loader: Trace-free program loading for api obfuscation, in: M. Dacier, M. Bailey, M. Polychronakis, M. Antonakakis (Eds.), Research in Attacks, Intrusions, and Defenses, Springer International Publishing, Cham, 2017, pp. 217–237.

[15] Y. Agman, D. Hendler, Bpfroid: Robust real time android malware detection framework, 2021. arXiv:2105.14344.
[16] R. Guo, J. Zeng, Trace me if you can: Bypassing linux syscall tracing, 2022. DEFCON30.
[17] R. Guo, J. Zeng, Phantom attack: Evading system call monitoring, 2021. DEFCON29.
[18] R. Guo, Detection and response for linux reflective code loading malware— this is how, https://medium.com/confluera-engineering/detection-and-response-for-linux-reflective-code-loading-malware-this-is-how-21f9c7d8a014, Last Update: 2021.
[19] C. Rowland, Detecting linux memfd_create() fileless malware with command line forensics, https://www.linkedin.com/pulse/detecting-linux-memfdcreate-fileless-malware-command-line-rowland, Last Update: 2020.
[20] D. C. D'Elia, E. Coppa, F. Palmaro, L. Cavallaro, On the dissection of evasive malware, Trans. Info. For. Sec. 15 (2020) 2750–2765. URL: https://doi.org/10.1109/TIFS.2020.2976559. doi:10.1109/TIFS.2020.2976559.
[21] P. S. A. LLC, Threat hunting with file entropy, https://practicalsecurityanalytics.com/file-entropy, Last Update on Oct 12, 2019.
[22] G. Bernardinetti, P. Caporaso, D. Di Cristofaro, F. Quaglia, G. Bianchi, Phoenix: A cloud-based framework for ensemble malware detection, in: 2023 21st Mediterranean Communication and Computer Networking Conference (MedComNet), 2023, pp. 11–14. doi:10.1109/MedComNet58619.2023.10168868.
[23] P. Caporaso, G. Bianchi, F. Quaglia, Jitscanner: Just-in-time executable page check in the linux operating system, in: Proceedings of the 18th International Conference on Availability, Reliability and Security, ARES 2023, Benevento, Italy, 29 August 2023- 1 September 2023, ACM, 2023, pp. 78:1–78:8. URL: https://doi.org/10.1145/3600160.3605035. doi:10.1145/3600160.3605035.

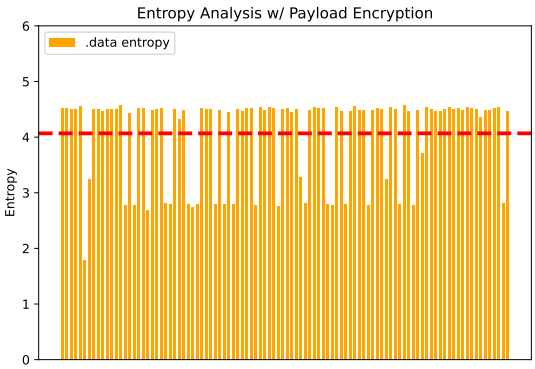
Appendix

Figure 3a shows a subset of the entropy values calculated on both the `.data` and `.text` sections of our malware dataset. Even the `.text` section alone exhibits relatively high entropy, likely attributed to the introduced obfuscation techniques.
On the other hand, Figure 3b displays the mean entropy value reduced to 4.6 after implementing the mechanism mentioned earlier in this document.

Figure 3: Entropy Analysis results on `.data` section



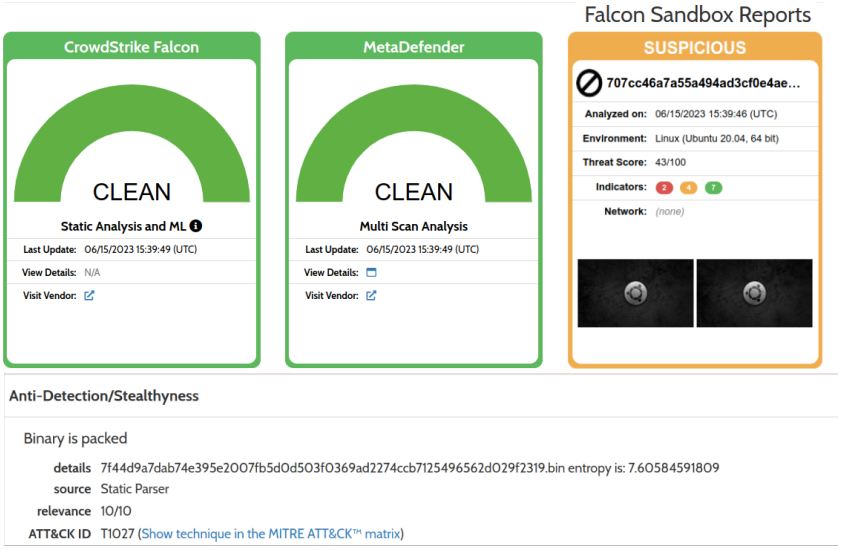
(a) Entropy of `.data` and `.text` sections



(b) Reduced entropy of `.data` section

Figure 4 shows evidence of the use of entropy analysis by various EDR/AVs. Specifically, displays the results obtained by Hybrid Analysis on a Linux coinminer malware, where Falcon Sandbox flags the **packed version** as suspicious due to its high entropy level.

Figure 4: Analysis results of a Coinminer malware deployed through the evasive pipeline

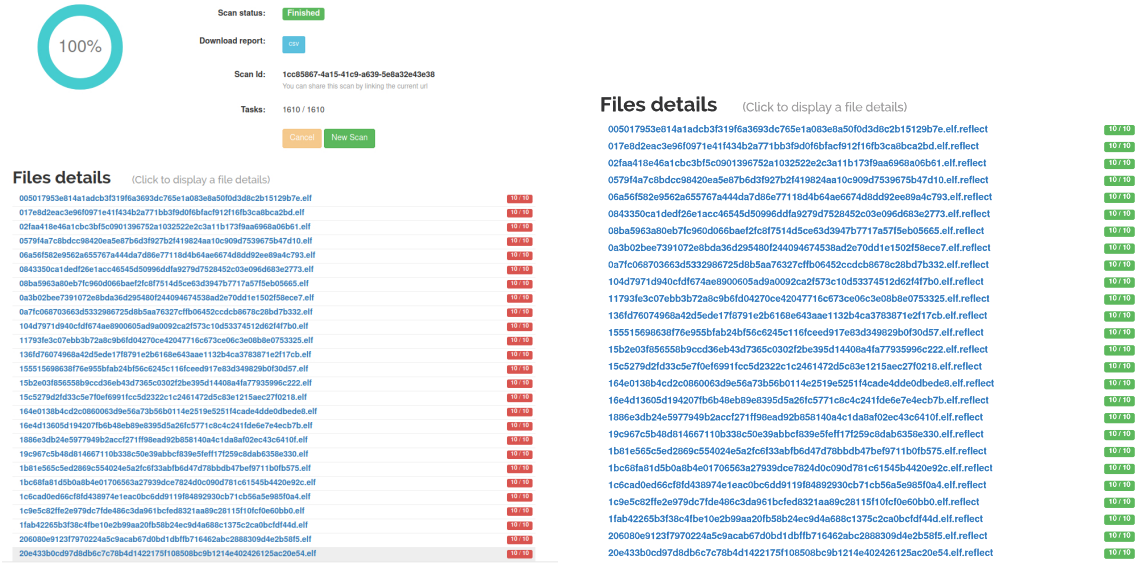


The analysis has been performed also on some well-known malware samples, of known behavior when executed, like Meterpreter. Figure 5 shows the results of Hybrid Analysis, highlighting how the reflected version is not identified.

Figure 5: Results from Hybrid Analysis on Meterpreter Shell: original vs. evasive versions

Figure 6 presents a subset of results extracted from the final report of Phoenix. Here, it is evident, indicated by the red highlights, that none of the antivirus programs hosted by the EDR are able to detect the evasive version of the 162 malware.

Figure 6: Phoenix Results



(a) Detected Malware

(b) Bypassed Malware