Malicious Linux Binaries: A Landscape

Lucas Galante¹, Marcus Botacin², André Grégio², Paulo Lício de Geus¹

¹ University of Campinas (Unicamp) {galante, paulo}@lasca.ic.unicamp.br

²Federal University of Paraná (UFPR) {mfbotacin, gregio}@inf.ufpr.br

Abstract. Linux applications are finding their role on important computer systems. As their use grow, they become target for malware. Therefore, understanding the security impacts of malware infections on them is essential to allow system hardening and countermeasures development. In this paper, we evaluate malicious ELF binaries to present a landscape of current threats. We discuss the challenges and pitfalls of analyzing samples on this platform and compare the identified behaviors to the ones presented by other platforms' samples.

1. Introduction

Fighting malware is currently a major security task for incident response teams, as this kind of threat is responsible for a myriad of damages, from privacy leaks to financial losses [TrendMicro 2017]. To provide proper countermeasures, understanding samples behavior is essential.

Recently, Linux systems have grown their market share [Itsfoss 2017], being present as back-end of many services. As it brings new, benign opportunities, it makes this environment target for malicious actors. Therefore, understanding the impact of Linux malware is essential to protect modern computer systems.

Previous work on Linux malware was guided by sandbox development [A 2015, 0x71 2016], thus not presenting a panorama of existing threats. Existing landscapes are limited to the Android ecosystem [Lindorfer et al. 2014], thus leaving other contexts underexplored. In this work, we evaluate Linux malware samples to present a panorama of their behaviors. Our goal is to understand their impact over systems as a whole, thus allowing more precise and effective incident response.

This work is organized as follows: in section 2, we present related work; in section 3, we present our assumptions and methods; in section 4, we present the threat landscape; in section 5, we discuss the impact of our findings; finally, we draw our conclusions in Section 6.

2. Related Work

The first step for analyzing Linux malware is to adopt a sandbox solution. In the literature, many solutions were proposed, such as a Linux version of Cuckoo Sandbox [0x71 2016]. In this work, we developed our own solution based on the use of Linux built-in tracing tools, such as strace, an approach also adopted by other sandbox solutions, such as Limon [A 2015].

A drawback of most solutions is to rely only on generic characteristics, such as the performed API calls. Few solutions consider OS particularities, such as the ELF (Executable and Linkable Format) format and Linux internal structures [Damri and Vidyarthi 2016, Shahzad et al. 2011]. In this work we considered these in our analysis, covering for instance, the passwd and shadow files, structures not present in other OS.

Based on the sandbox results, most solutions adopt classification approaches [Asmitha and Vinod 2014, KA and P 2014] to distinguish malicious from benign applications. Although important for individual sample analysis, these do not provide insights

regarding the whole malware scenario. In this sense, our work contributes for better understanding the whole context.

Previous work addressed the malware landscape issue on other platforms. Lindorfer et al. [Lindorfer et al. 2014] surveyed the Android ecosystem. Bayer et al. [Bayer et al. 2009] surveyed the Windows one. This work extends these for the Linux scenario.

During the development of this work, we were noticed of the publication of a Linux malware survey [Cozzi et al. 2018], thus being this the closest work to ours so-far. As a significant distinction, our work digs into more details about x86 samples' behavior during dynamic analysis, thus complementing it.

3. Methodology

3.1. Dataset Description

To provide a comprehensive evaluation of Linux binaries, we collected samples from distinct sources. In total, this study considers 5,680 unique ELF binaries—identified by their MD5—crawled from MalShare¹, VirusTotal² and VirusShare³.

A noticeable Linux characteristic is its multi-platform support: the collected ELF samples cover 8 distinct architectures, as shown in Figure 1.

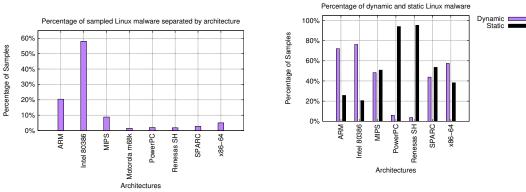
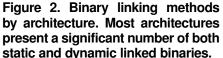


Figure 1. ELF binaries by architectures. x86 and ARM are the most prevalent architectures.



We observe that the most prevalent architectures are Intel x86, found in most desktop computers and servers, and ARM, often found in mobile phones and tablets. Besides that, we observe a diversity on the remaining platforms, thus showing the heterogeneity of the Linux ecosystem, which covers a myriad of embedded systems, from co-processors to IoT devices.

The ELF heterogeneity is observed not only in the target platform but also in the binaries themselves, Figure 2 presents how samples of each architecture are linked⁴—statically or dynamically. Whereas some architectures present a higher rate of statically linked samples, other presents higher rates of dynamically linked ones. The linking project decision is not only tied to environment characteristics but also to evasion attempts, as statically linked libraries cannot be traced by some analysis solutions (ltrace).

In addition to linking methods, malware creators also adopt distinct project decisions regarding the distributed object file, as shown in Figure 3. Whereas executables are prevalent in most

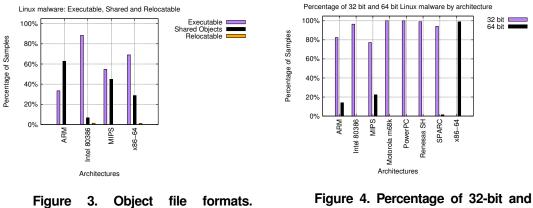


Figure 3. Object file formats. Samples are distributed both as executables and libraries.

Figure 4. Percentage of 32-bit and 64-bit binaries by architecture. 32-bit binaries are prevalent.

platforms, shared objects (libraries) also appear in significant rates. Leveraging executables is interesting for malware creators as it enables users infecting themselves by directly running the objects. Shared objects, in turn, allow attackers to inject their payloads in any other binary in the form of a library. Finally, shared objects are also employed to allow code modularization, a strategy employed in malware to bypass detection methods.

The most homogenous characteristic in our ELF dataset are binaries word size (32 or 64 bits). As presented in Figure 4, almost all architectures present higher rates of 32 bit samples, as was the standard until few years ago. Modern samples, however, are already compiled as 64 bits.

3.2. Analysis Methods

First, all samples were submitted to VirusTotal to retrieve anti-virus detection rates and label information; secondly, static analysis was performed, by disassembling (using objdump) all files and retrieving header information; finally, dynamic analysis was performed to evaluate samples' behavior and capture network traffic. As samples may be equipped with anti-analysis techniques, the strategy presented in Table 1 was employed.

Technique	Tool	Evasion	Countermeasure
Static analysis	objdump file	obfuscation	Dynamic analysis
	strings		
	ltrace	Static compilation	ptrace step-by-step
Dumannia analysia	ptrace	ptrace check	binary patching
Dynamic analysis	strace	Long sleep	LD_PRELOAD
	LD PRELOAD	Injection blocking	Kernel hooks

Table 1. Analysis techniques. Adopted strategy to handle evasive samples.

As static analysis procedures may be defeated by obfuscation, we also submitted samples to dynamic analysis procedures. Dynamic analysis may be performed in a series of ways [Gebai and Dagenais 2018]. In our evaluation, we leveraged strace for system call inspection and ltrace for function call inspection.

Dynamic analysis, however, may also be defeated in diverse ways: i) ltrace analysis may be avoided by the use of static libraries, as it handles only dynamic ones. These samples are analyzed in more details through step-by-step instruction tracing by using ptrace, which is able to dig into samples despite their linking mode; ii) Ptrace analysis in turn, may be defeated by ptrace checks. In this case, the check may be removed by using a binary patching procedure; iii) ltrace

and strace may be evaded by a long sleep, aimed to trigger a timeout on the sandbox. Such cases are handled by the injection of a library—through LD_PRELOAD—to hook the sleep function so it immediately returns; iv) the LD_PRELOAD method may be blocked by some samples. Such cases may be inspected by a kernel driver which hooks API calls to log them.

In addition to anti-analysis-armored samples, other particular behaviors were considered, as shown in Table 2.

Table 2. Handling suspicious behaviors. Adopted strategy to keep log files sa	Table 2. Handling	suspicious	behaviors.	Adopted	strategy	to keep	o log file	s safe
---	-------------------	------------	------------	---------	----------	---------	------------	--------

· ·			
Behavior	Action	Countermeasure	Method
Evidence removal	delete logs	log access	syslog/audit
Ransomware	delete files	shadow copy	inotify

Some samples present the evidence removal behavior, deleting the stored logs. For these cases, a logging mechanism was implemented to register such occurrences and thus characterize the samples as evidence removers. Ransomware samples also may damage the filesystem by encrypting all files, including the collected logs. Therefore, a shadow copy of files using inotify was implemented, thus keeping all original files safe.

All aforementioned analysis procedures were conducted on a network-isolated, virtual machine-based sandbox solutions running *Ubuntu 16*. The samples were individually analyzed by up to 3 minutes and the clean system state was restored through snapshots after each execution. 4. Linux Malware Landscape

4.1. Static Features

We initially submitted all samples through static analysis procedures to get general insights about how samples look like. We first retrieved (via objdump) the linked function calls to understand which behaviors the samples were supposed to present. For such, we classified the obtained functions in categories, according to the behaviors defined in [Grégio et al. 2015].

The Network category encompasses functions responsible for allowing samples to communicate through the Internet, thus enabling malicious content download and information exfiltration. The Evasion category encompasses functions which can be used to thwart an analysis procedure thus keeping samples undetected. It covers functions used to modularize malware code and the ones used to finish and/or block other processes execution. The Environment category encompasses functions which allows environment fingerprinting, such as retrieving username information. Such information can be used for evasion and/or for infection accountability. The Removal category encompasses functions related to anti-forensics produces, thus allowing the sample to cover its track. Finally, the Timing category encompasses functions which allows the sample to measure the spent time while processing. Such information can be used for evasion procedures, as the samples may detect the performance overhead imposed by an analysis solution. Figure 5 shows how often samples of each architecture link functions from one or more of these categories.

We notice that attempting to establish a network connection is the most prevalent suspicious behavior among all architectures, being it present in over 25% of the entire dataset samples. Attempts to evade analysis procedures are also frequent, either in the form of analysis termination or in the form of overhead measurement. Environment information was collected in fewer samples, which indicates such information is not being used for evasion in a broad way but for other purposes, such as information leaking, according to each samples specific goal.

The identified prevalent use of network capabilities is an even more significant result when we consider it is a lower bound, because objdump only identifies function entries present in the dynamic symbol table. Therefore, function calls from statically linked and obfuscated samples were not retrieved. Figure 6 shows the rate of samples whose dissasembly attempts failed. Omitted architectures are due to lack of objdump support.

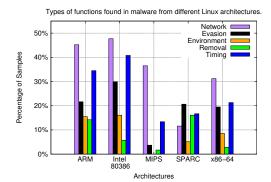


Figure 5. Malware behavior prevalence by malware architectures. We observe that network functions are prevalent.

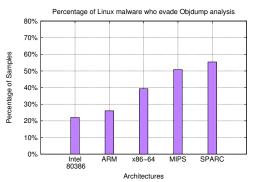


Figure 6. Percentage of malware that failed to disassembly. Some architectures aren't present because of lack of objdump support.

After identifying the high use of network functions, we queried (via strings ⁵) network-related information embedded in the binary. By matching the retrieved strings with regular-expressions patterns, we identified information about IP addresses, URLs and E-mail contacts. The ratio of samples presenting network-related strings and the fraction of distinct strings are present in Figure 7.

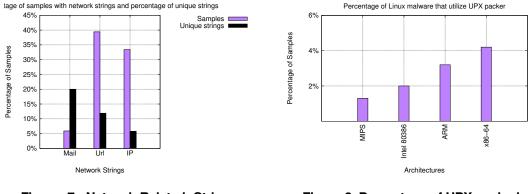
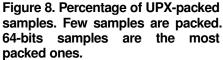


Figure 7. Network-Related Strings. Percentage of samples with network-related strings and the fraction of unique strings.



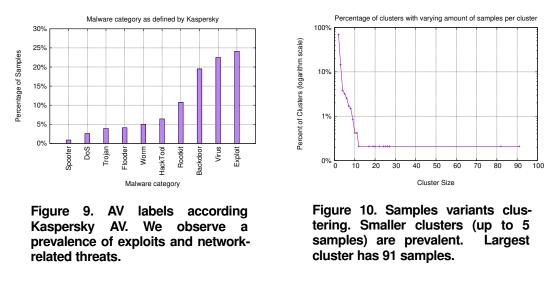
Among all identified strings, we found suspicious IP and URL addresses, including local and remote hosts, of which many are related to shell script downloads. We also identified embedded Email addresses, which are probably related to phishing campaigns. As for functions, embedded strings can also be hidden by packer-based obfuscation. Figure 8 shows the rate of samples leveraging UPX^6 , a popular open-source packing solution.

To confirm our findings about the intense network usage, we checked how AVs label the samples. Figure 9 shows labels attributed to all samples by the *Kaspersky* AV. Among all 10 attributed labels, the three more prevalent ones (*Exploits, Virus* and *Backdoor*) account for 60% of all samples.

The high presence of *Backdoor* samples explains the high linkage rate of network-related functions—presented in the Figure 5—, as *Backdoors* make use of network connection to allow the external attackers to remotely access the infected system.

⁵ man strings ⁶ upx.github.io

The prevalent labels also explain the low rate of UPX packed samples, as presented in Figure 8. *Exploits*, which represent nearly 25% of all samples tend to present low obfuscation rates due to their nature. These aren't self-contained applications which unpack themselves, but payloads which are injected into third processes to cause these to behave maliciously.



Given many samples present similar behaviors, we checked whether these samples were independently developed or were variants of the same original code. To perform such check, we computed the fuzzy hash of all samples using SSDeep⁷ with a 90% threshold. Further, all samples were matched against each other. Figure 10 shows the identified distinct clusters, their sizes and the number of samples on each. We discovered that most samples are located in the smaller clusters. On the other hand, many clusters hold at least 1 large variant family; the largest variant family presented 91 samples.

4.2. Dynamic Analysis & Behaviors

Whereas static analysis is useful to determine several features, it is subject to be defeated by obfuscation. To overcome such limitation, we submitted samples to dynamic analysis. As dynamic analysis procedures require effectively running the samples, we limited our evaluation to inspect Intel x86 and x64 ones, as they can be run in common machines without emulation. Each sample was executed up to 3 minutes, being terminated by a timeout. Their termination signals rates are presented in Figure 11.

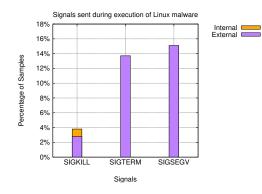
We first observe that $\approx 15\%$ of samples were terminated due to a segmentation fault error. It happens due to malware-environment incompatibilities, such as distinct library versions, nonexistent peripheral communication attempts or lack of a required resource.

Another portion of $\approx 15\%$ of samples were terminated due to timeout⁸ expiration. It happens when a sample enters on an infinite loop or keep a long time waiting a resource. Most samples were terminated by the usual SIGTERM signal. Fewer samples handled and ignored this signal, being forcibly terminated by the SIGKILL one.

We also discovered a small fraction ($\approx 3\%$) of samples making use of the SIGKILL signal to terminate their own processes. It happens mostly due to evasion attempts, as a child process may detach itself from a debugger after killing its own father.

As for static analysis, we classified system calls into behaviors. Figure 12 shows the fraction of samples presenting each one of these behaviors.

⁷ ssdeep-project.github.io ⁸ man timeout



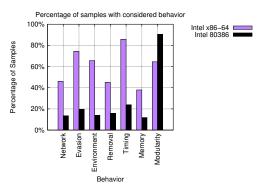


Figure 11. Observed Signals during execution. Most samples terminated prior timeout expiration. Few samples exhibited inter-process interactions.

Figure 12. Malware behavior prevalence. Modularity is the prevalent behavior. Evasion and Timing are also significant.

We observe that many samples implement some kind of anti-analysis protection, both directly and indirectly. Direct approaches make use of methods such ptrace and exit to detach from a debugger. Indirect approaches make use of methods such as time to infer the performance overhead imposed by analysis solutions.

During dynamic analysis execution, the samples presented fewer network interactions than expected given the number of function identified on static analysis. We credit this effect to samples requiring resources unavailable in our system—such as old libraries—to run. This hypothesis is corroborated by the fact that this effect is greater on 32 bit—thus, older—samples. In newer, 64-bit ones, dynamic analysis produced more network interactions than identified during static analysis. This fact is expected as some calls are runtime-generated.

Regarding construction, we observe most samples are implemented in a modular way, launching child processes, through fork and clone, and relying on third-party binaries, through execve.

To better understand how the samples internally operate, we retrieved the accessed filesystem locations, as shown in Figure 13. We discovered the most prevalent samples action is to read and write information from the /proc directory. The /proc is a filesystem-mapping for configuration and environment variables, thus allowing malware to leak process information and even tamper with their execution. The second most prevalent action is to modify the resolv.conf file, responsible for storing DNS configuration. This is typical Proxy behavior and is also related to the high rate of network use. In addition, some samples also access the shadow and passwd files, responsible for storing login credentials. Such accesses are related to privilege elevation attempts.

We observe that most interactions are performed in the form of filesystem accesses, due to the Linux paradigm of "*everything is a file*". It reflects in the number of file reads and writes, as shown in the Figure 14. It also shows few user interactions, such as stdio reads and writes, indicating most samples operates autonomously in the background.

All presented data can be considered as a lower bound for malware behavior as the samples present a significant use of evasive methods, as presented in Figure 15.

Around 10% of samples rely on the ptrace syscall for analysis evasion. By acquiring the ptrace lock, samples block inspection mechanisms, such as debuggers, from attaching to them. Samples also avoid being analyzed by preventing monitoring solutions from injecting

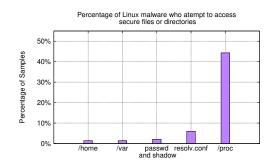


Figure 13. Accessed files and directories. Samples interfere with system configurations and steal credentials.

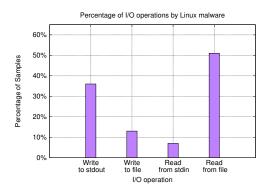


Figure 14. I/O operations. Most samples do not present direct user interaction

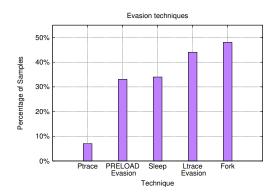


Figure 15. Evasion Techniques. Samples present diversified evasion methods.

instrumentation code within them. In this sense, 30% of samples block LD_PRELOAD injection attempts. Moreover, 30% of samples use a sleep call for analysis evasion. As sandboxes solutions often stop their execution after a timeout, a long enough delay may prevent the malicious payload from being inspected.

Some samples adopt indirect strategies to avoid analysis procedures. 40% of samples are statically-linked, thus preventing ltrace from dynamically tracing them. Other samples adopt modular constructions to obfuscate the execution flow. Given the creation of multiple (forked) malicious processes, analysts need correlated independent tasks to draw the general malicious scenario.

4.3. Network Traffic

From the firewall logs generated during dynamic execution was retrieved source and destination IP addresses. The source IP were all from the local machine, but the destination IP addresses were from attempted connections. Figure 16 shows the rate of samples which performed at least one network connection attempt.

Corroborating dynamic analysis results, we observe Intel x86-64 samples perform many more connections attempts than Intel 80386. When discarding network scanning samples, 50% of all contacted IPs, on average, were unique, indicating diversity. The scanners impact is noticeable as we have identified a sample which uniquely attempted to contact more than 75 thousand distinct IP addresses.

In addition to IP information, we performed reverse DNS queries to identify the associated domains. Given the scanners, most domains ($\approx 60\%$) are associated to domestic internet providers.

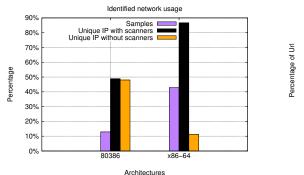


Figure 16. Identified network usage. Scanners dominate unique IP rate.

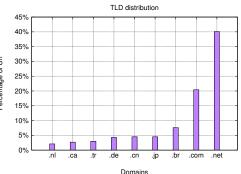


Figure 17. TLD distribution. Global domains (.net and .com) are prevalent. Local domains are present due to scanners enumeration.

This fact is also noticeable when we observe the most prevalent top level domains, presented in Figure 17. Whereas global domains (.net and .com) are prevalent, regionalized domains are well-distributed, as scanners aren't region-aware.

5. Discussion

In this section, we discuss our findings and compare the obtained results with other work to draw a landscape of Linux threats. Our first finding is that the Linux environment is very diverse, presenting samples of distinct architectures, endianesses and word sizes. Whereas this fact have already been identified by a previous study [Cozzi et al. 2018], we are the first to discuss samples implementation in details, presenting, for instance, a comprehensive analysis of linked libraries and network traffic.

Besides comparing Linux studies, we also identified differences when comparing Linux threats to Windows ones [Botacin et al. 2015], such as the use of packers. 50% of Windows malware samples are packed (24% of these are UPX) whereas our dataset presented a rate of at most 4% of packed samples. Such difference is explained by the high rate of exploit samples present in our dataset, as shown by the AV labels. In comparison, no exploit was identified in the Windows dataset.

In common, both environments present a similar rate of network traffic (\approx 50%), which indicates it is a general trend regarding malware. However, on each environment, the performed network action is distinct. The Windows dataset presents a major share of downloaders whereas the Linux one presents a significant amount of backdoors. Moreover, samples of both OS install connection proxies in the target machine. Windows samples redirect network traffic by using Proxy Auto Configuration (PAC) files whereas Linux ones modify the resolv.conf file.

Finally, we discovered that both Linux and Windows malware present comparable, significant potential to cause damage in their target machines. Nevertheless, due to environmental, internal reasons, their malicious actions are deployed by distinct methods.

6. Conclusion

In this paper, we have presented an overview of malicious Linux binaries. Through static and dynamic analysis, we discovered the most prevalent system calls (fork and execve) and their associated behaviors (evasion and modularization). We also performed network traffic analysis and discovered that \approx 50% of samples rely on the Internet to achieve their malicious goals. We compared malware samples targeting Linux and Windows and discovered that they can cause the same damage extent and present similar characteristics, including the use of anti-analysis tricks. Given OS particularities, some behaviors are more tied to OS internals, which should be understood to allow proper countermeasure development. An extended version of this paper is available at https://github.com/marcusbotacin/Linux.Malware.

Acknowledgments. This work is supported by PIBIC-CNPq 800295/2016-1 and FORTE-CAPES 23038.007604/2014-69.

References

0x71 (2016). Cuckoo for linux. https://github.com/0x71/cuckoo-linux.

- A, M. K. (2015). Automating linux malware analysis using limon sandbox. https://ubm.io/2MH5qRz.
- Asmitha, K. A. and Vinod, P. (2014). A machine learning approach for linux malware detection. In 2014 Int. Conf. on Issues and Chal. in Intel. Comp. Tech. (ICICT).
- Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., and Kruegel, C. (2009). A view on current malware behaviors. In *Proc. of the 2Nd USENIX LEET*.
- Botacin, Geus, and Grégio (2015). Uma visão geral do malware ativo no espaço nacional da internet entre 2012 e 2015. https://bit.ly/2ouVDzF.
- Cozzi, E., Graziano, M., Fratantonio, Y., and Balzarotti, D. (2018). Understanding linux malware. In 2018 IEEE Sec. & Priv.
- Damri, G. and Vidyarthi, D. (2016). Automatic dynamic malware analysis techniques for linux environment. In 2016 INDIACom.
- Gebai, M. and Dagenais, M. R. (2018). Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Comput. Surv.*, 51(2).
- Grégio, A. R. A., Afonso, V. M., Filho, D. S. F., Geus, P. L. d., and Jino, M. (2015). Toward a taxonomy of malware behaviors. *The Computer Journal*, 58(10):2758–2777.
- Itsfoss (2017). Desktop linux now has its highest market share ever. https://bit.ly/2outxo3.
- KA, A. and P, V. (2014). Linux malware detection using non-parametric statistical methods. In 2014 Int. Conf. on Adv. in Comp., Com. and Inf. (ICACCI).
- Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Veen, V. v. d., and Platzer, C. (2014). Andrubis 1,000,000 apps later: A view on current android malware behaviors. In *BADGERS '14*.
- Shahzad, F., Bhatti, S., Shahzad, M., and Farooq, M. (2011). In-execution malware detection using task structures of linux processes. In 2011 IEEE Int. Conf. on Communications (ICC).
- TrendMicro (2017). Erebus linux ransomware: Impact to servers and countermeasures. https://bit.ly/2wCyA9S.