Malicious Linux Binaries: A Landscape

Lucas Galante[1,3], Marcus Botacin[2], André Grégio[2], Paulo Lício de Geus[1]

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

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

[3]PIBIC-CNPq 800295/2016-1

Abstract. *Linux applications are finding their role on important computer systems. At the same time 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 such 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. At the same time it brings new, benign opportunities, it makes this environment target for malicious authors. Therefore, understanding the impact of `Linux` malware is essential to protect modern computer systems. Previous work on `Linux` malware was guided by sandbox development [Monnappa 2015, 0x71 2016], thus not presenting a panorama of existing threats. Existing landscapes are focused on the Android ecosystem [Lindorfer et al. 2014], thus leaving other contexts underexplored.

In this work, we propose evaluating `Linux` malware to present a panorama of their behaviors. Our goal is to understand their impact over system 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, which is based on the use of `Linux` built-in tracing tools, such as `strace`. This same approach is adopted on other sandbox solutions, such as Limon [Monnappa 2015].

A drawback of most solutions is to rely only on generic characteristics, such as the performed API calls. Few solutions consider O.S. particularities, such as the ELF binary and `Linux` internal structures [Damri and Vidyarthi 2016, Shahzad et al. 2011]. In this work, we considered these on our analysis, covering, for instance, the `passwd` and `shadow` files, structures not present in other O.Ses. From sandbox results, many 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 presents the same analysis for the Linux scenario.

During the development of this work, we noticed 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 being a complement for such work.

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 MalShare[1], VirusTotal[2] and VirusShare[3].

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

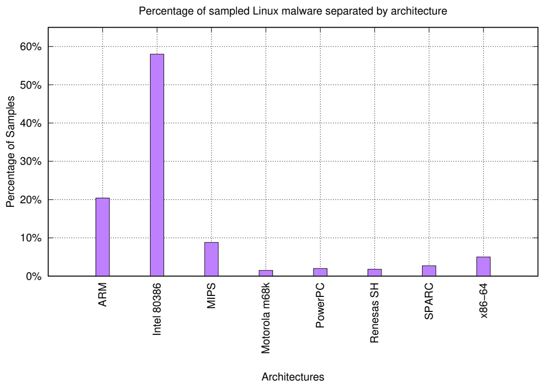

Figure 1. ELF binaries by architectures. x86 and ARM are the most prevalent architectures.

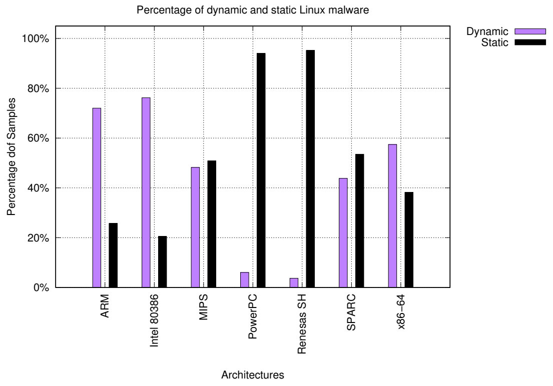

Figure 2. Binary linking methods by architecture. Most architectures present a significant number of both static and dynamic linked binaries.

We observe the most prevalent architectures are Intel x86, found in most desktop computers and servers, and ARM, often found in mobile phones and tablets. Moreover, we observe a diversity in 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 also observed not only in the target platform but in the binaries themselves, Figure 2 presents how samples of each architecture are linked[4]—statically or dynamically. Whereas some architectures present a higher rate of statically linked samples, other present 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 take distinct project decisions regarding the distributed object file, as shown in Figure 3. Whereas executables are prevalent in most platforms, shared objects (libraries) are also present in a significant rate. Executables are interesting for malware creators as they allow users infecting themselves by directly running them.

[1] malshare.com [2] virustotal.com [3] virusshare.com [4] Unavailable info for m68k.

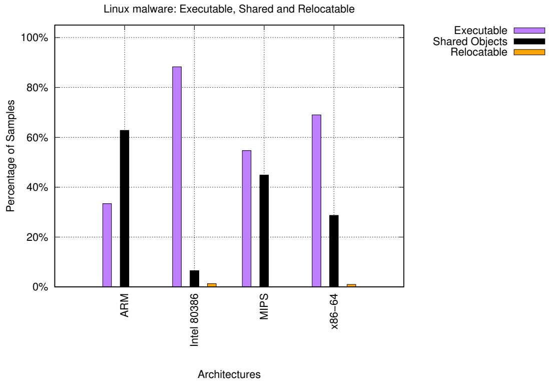

Figure 3. Object file formats. Samples are distributed both as executables and libraries.

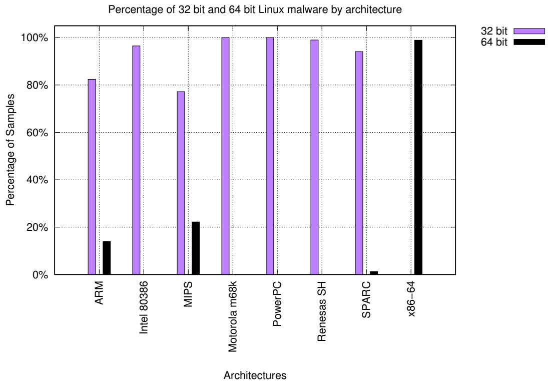

Figure 4. Word size by architecture. 32-bit binaries are prevalent.

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 by malware to bypass detection methods.

The most homogeneous 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.

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

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

Initially, basic information was retrieved using static analysis procedures. However, as they may be defeated by obfuscation, we also submitted samples through dynamic analysis procedures. Dynamic analysis may be performed in many 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 be also defeated in diverse ways: i) `ltrace` analysis may be prevented 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 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 for at most 3 minutes and the clean system state was restored through snapshots after each execution.

4. Linux Malware Landscape

4.1. Static Features

In our evaluation, we initially submitted all samples to static analysis procedures to get general insights about how samples look like. The first analysis procedure consisted on retrieving (via objdump) the linked function calls to understand which behaviors the samples were supposed to present. To do so, we classified the obtained functions in categories, according the behaviors defined in [Grégio et al. 2015].

The Network category encompasses function responsible for allowing the sample 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 executions. 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 by few samples, which indicates such information is not being used for evasion in a broad way but for other purposes, such as information leaking, according each sample specific goal.

The identified prevalent use of network resources 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, functions calls from statically linked and obfuscated samples were not retrieved. Figure 6 shows the rate of samples whose disassembly attempts failed. Omitted architectures are due to lack of objdump support.

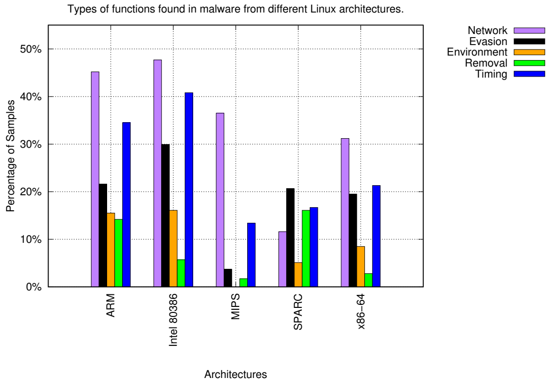

Figure 5. Malware behavior prevalence by malware architectures. We observe that network functions are prevalent.

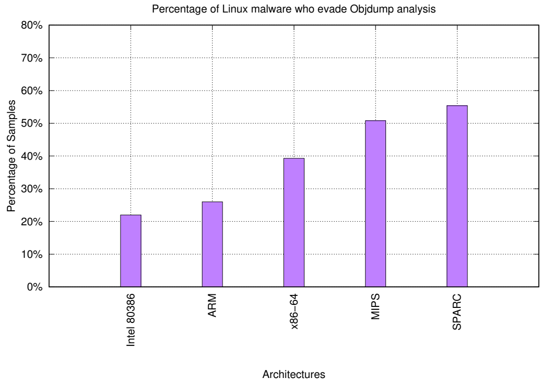

Figure 6. Percentage of malware that failed to disassembly. Some architectures aren't present because of lack of objdump support.

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

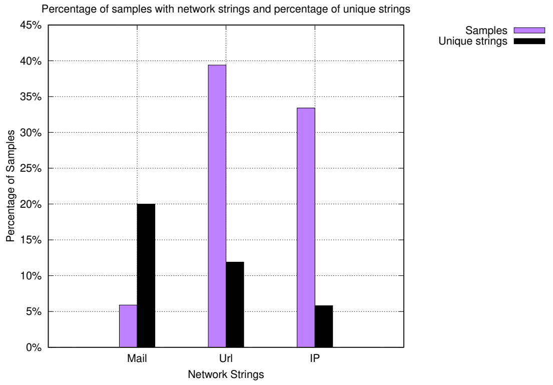

Figure 7. Network-Related Strings. Rate of samples with network-related strings and the fraction of unique strings.

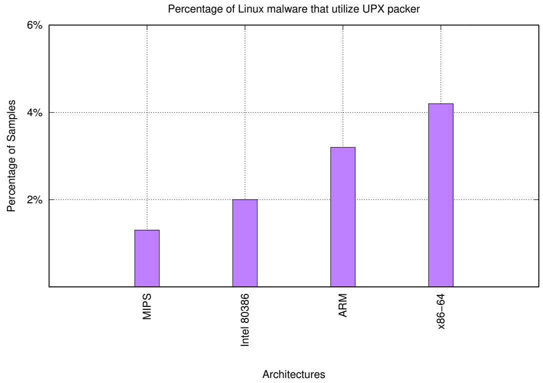

Figure 8. Rate of UPX-packed samples. Few samples are packed. 64-bits samples are the most packed ones.

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

[5] man strings [6] `upx.github.io`

functions—presented in the Figure 5—, as *Backdoors* make use of network connection to allow external attackers to remotely access the infected hosts.

The prevalent labels also explains 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 are not self-contained applications which unpack themselves, but payloads which are injected into third party processes to cause these to behave maliciously.

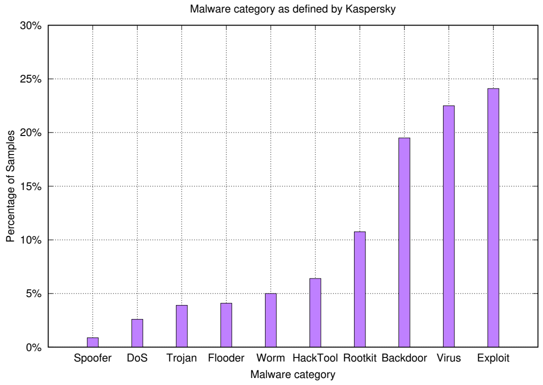

Figure 9. AV labels according Kaspersky AV. We observe a prevalence of exploits and network-related threats.

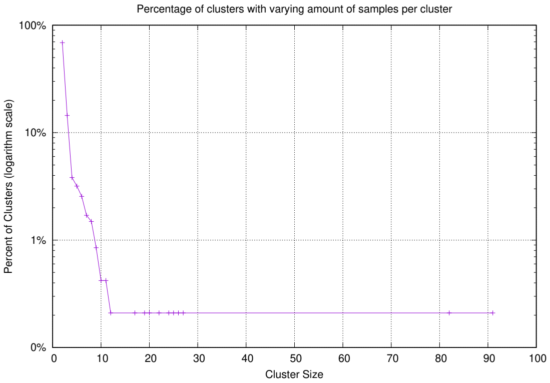

Figure 10. Samples variants clustering. Smaller clusters (uo to 5 samples) are prevalent. Largest cluster has 91 samples.

Given many samples present similar behavior, 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 `SSDeep`[7] 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 by up to 3 minutes, being terminated by a timeout. Their termination signals and rates are presented in Figure 11.

We first observe that ≈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 ≈15% of samples were terminated due to `timeout`[8] expiration. It happens when a sample enters an infinite loop or awaits a long time for 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 (≈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.

[7] `ssdeep-project.github.io` [8] man timeout

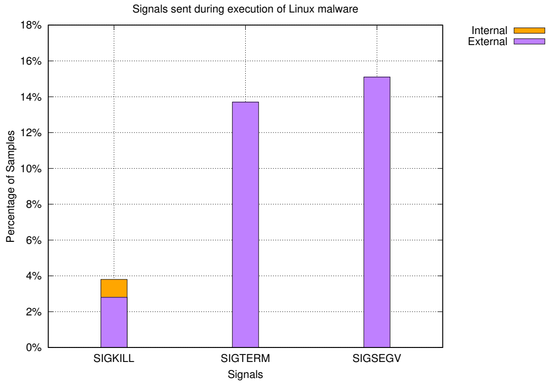

Figure 11. Observed Signals during execution. Most samples terminated prior timeout expiration. Few samples exhibited inter-process interactions.

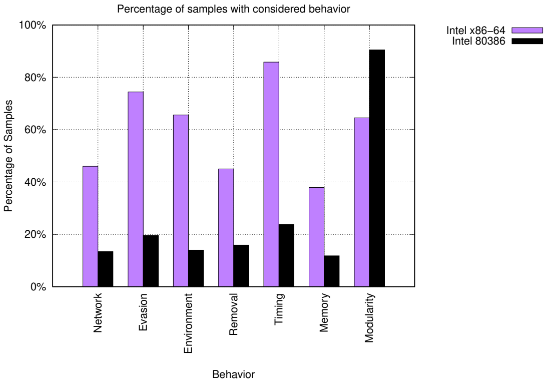

Figure 12. Malware behavior prevalence. Evasion is the prevalent behavior.

As for static analysis, we classified system calls into behaviors. Figure 12 shows the fraction of samples presenting each one of these behaviors. We observe many samples implement some kind of anti-analysis protection, either 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 measure and 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 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 operate 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

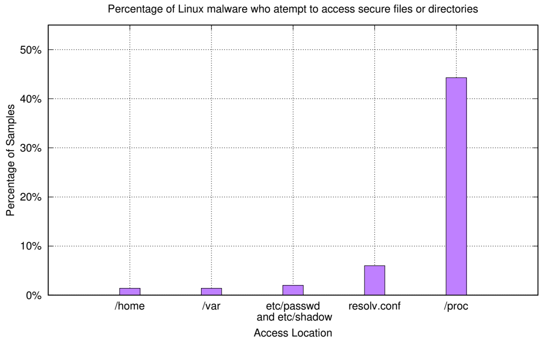

Figure 13. Accessed files and directories. Samples interfere with system configurations and steal credentials.

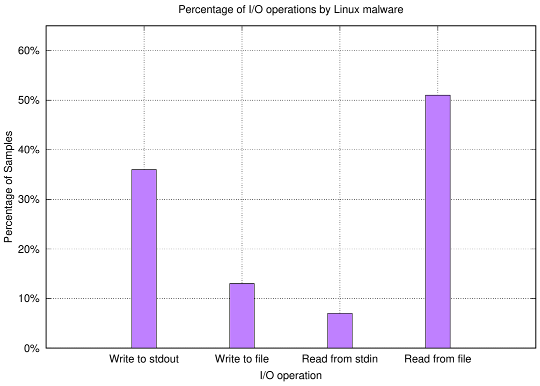

Figure 14. I/O operations. Most samples do not present direct user interaction

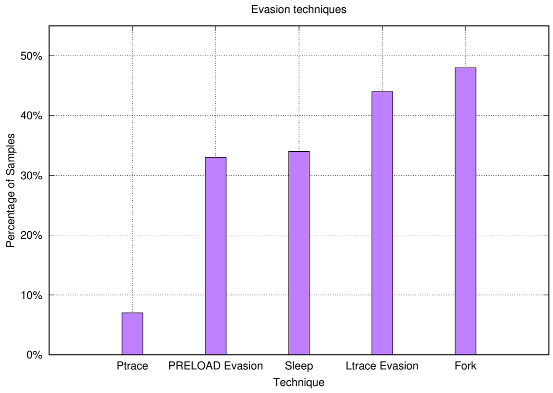

Figure 15. Evasion Techniques. Samples present diversified evasion methods.

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 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 to correlate independent tasks to draw the general malicious scenario.

4.3. Network Traffic

We retrieved source and destination IP addresses from the network traffic generated during dynamic analysis to gather more insights about how samples use network resources. 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

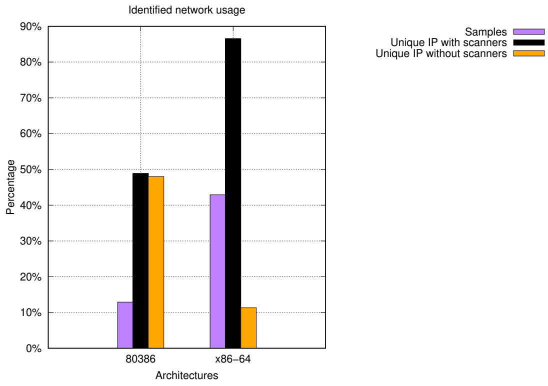

Figure 16. Identified network usage. Scanners dominate unique IP rate.

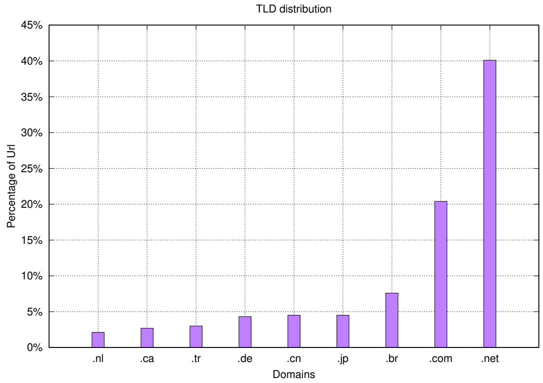

Figure 17. TLD distribution. Global domains (.net and .com) are prevalent. Local domains are present due to scanners enumeration.

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 (≈60%) are associated to domestic internet providers. This fact is also noticeable when we observe the most prevalent Top Level Domains (TLDs), presented in Figure 17. Whereas global domains (`.net` and `.com`) are prevalent, regionalized domains are well-distributed, as scanners are not 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 this environment is very diverse, presenting samples from distinct architectures, endianess and word sizes. Whereas this fact have already been identified by previous `Linux` researchers [Cozzi et al. 2018], we are the first to discuss samples implementation in depth, presenting, for instance, a comprehensive analysis of linked libraries and network traffic.

In addition to similarities and differences when comparing our results to ones from other `Linux` studies, we also identified these when comparing `Linux` threats to `Windows` ones [Botacin et al. 2015]. The first significant difference is the packer usage rate. `Windows` malware present 50% use of packers (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 the 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 (≈50%), which indicates it is a general trend regarding malware. However, on each environment, the performed network action is distinct. On `Windows`, samples present a major share of `downloaders` whereas `Linux` samples present a significant amount of `backdoors`. Moreover, both OSes install connection proxies in the target machine. `Windows` samples redirect network traffic by using Proxy Auto Configuration (PAC) files whereas `Linux` ones modifies the `resolv.conf` file.

Finally, we discovered both `Linux` and `Windows` malware present comparable, significant potential to cause damage on their target machines. Nevertheless, due to environmental, internal reasons, their malicious actions are deployed by leveraging 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 ≈50% of samples relies on the Internet to achieve their malicious goals. We compared malware samples targeting `Linux` to the ones targeting `Windows` and discovered they can cause the same damage extent and present similar characteristics, including the use of anti-analysis tricks. Given O.S. particularities, some behaviors are more tied to O.S. internals, which should be understood to allow proper countermeasure development.

References

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

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. `http://siaiap34.univali.br/sbseg2015/anais/WFC/artigoWFC02.pdf`.

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://itsfoss.com/linux-market-share/`.

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*.

Monnappa, K. A. (2015). Automating linux malware analysis using limon sandbox. `https://www.blackhat.com/docs/eu-15/materials/eu-15-KA-Automating-Linux-Malware-Analysis-Using-Limon-Sandbox-wp.pdf`.

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://www.trendmicro.com/vinfo/us/security/news/cyber-attacks/erebus-linux-ransomware-impact-to-servers-and-countermeasures`.