Malicious Linux Binaries: A Landscape

Lucas Galante[1], 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

Abstract. *Linux-based solutions are finding their role on important computer systems. At the same time they grow on usage, they become target for malicious actors. 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`-based systems have grown their market share [Itsfoss 2017], being present as back-end of most services. At the same time it brings new, benign opportunities, it makes this environment target of malicious authors. Therefore, understanding the impact of `Linux` malware is essential to protect modern computer systems.

Previous work on `Linux` malware is guided by sandbox development [A 2015, 0x71 2016], thus not presenting a panorama of existing threats. Existing landscapes are focused on the Android ecosystem [Lindorfer et al. 2014], thus leave other binary formats underexplored.

In this work, we propose evaluating ordinary `Linux` binaries to present a panorama of their behavior. Our goal is to understand their impact over system as a whole, thus allowing more precise and effective incident response.

We have collected 5,680 samples and analyzed them regarding their static features, dynamic behavior and network traffic. Our results show that the samples present a high rate of obfuscation, evasion techniques and network connectivity.

This work is organized as follows: we motivate our work in section 2; in section 3, we present related work to better position our one; in section 4, we present our assumption, data collection and analysis methods; in section 5, we present the threat landscape; in section 6, we discuss the impact of our discoveries; finally, we draw our conclusions in section 7.

2. Motivation: Are there Linux malware ?

"*Linux is not affected by malware*" is a frequent heared statement, but is far from the truth, being it referred as a myth [Sophos 2015]. Although it presents fewer samples, given its minor market share, `Linux` threats do exist.

In practice, `Linux` malware may cause the same damage as malware causes on other operating systems (OS), as the involved OS concepts are the same, as demonstrated by SANS [Institute 2015].

`Linux-based` system are important targets when considering server machines, since these empowers current Internet services [Wired 2017]. A `Linux` server malware may break with significant part of the web, thus their importance. An example of such was seem on the `Erebus` case, a `Linux` server ransomware which took over a host provider [TrendMicro 2017].

3. Related Work

The first step for analyzing `Linux` malware is to adopt a sandbox solution. In the literature, many solutions were proposed, such as `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 [A 2015].

A drawback of most solutions is to rely only on general characteristics, such as the performed API calls. Few solutions consider OS particularities, such as the Executable and Linkable Format (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 on 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 a better understanding of 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 were noticed of the publication of a `Linux` malware survey [Cozzi et al. 2018], thus being this the closest related work 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.

4. Methodology

In this section, we present the general methodology adopted on our work, such as sample collection, separation, and the performed analysis.

4.1. 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 their multi-platform support. Thus, our collected ELF samples cover 8 distinct architectures, as shown in Figure 1.

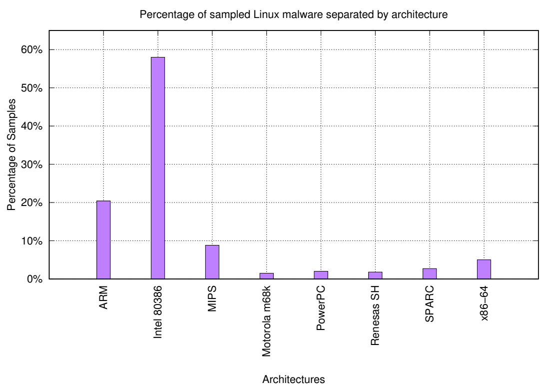

Figure 1. ELF binaries by architectures. x86 and ARM are the most prevalent architectures.

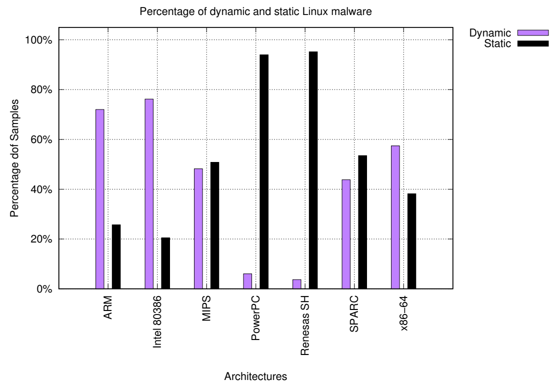

Figure 2. Binary linking methods by architecture. Most architectures present a significant number of both static and dynamic linked binaries.

We observe the most prevalent architectures—covering 60% of all samples—are Intel x86, found in most desktop computers and servers, and ARM, often found in mobile phones and tablets. Despite such fact, 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 also observed not only in the target platform but in the binaries themselves, figure 2 presents how samples on each architecture are linked—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).

We were not able to retrieve linking information for the Motorola m68k binaries.

In addition to linking method, malware creators also take distinct project decisions regarding the distributed object file, as shown in figure 3. Whereas executable are prevalent in most platform, shared objects (libraries) still present a significant rate. Executables are interesting for malware creators as they allow users infecting themselves by directly running the object. Shared objects, in addition, allows attackers to inject their payloads

[1] http://www.malshare.com/
[2] https://www.virustotal.com/
[3] https://virusshare.com/

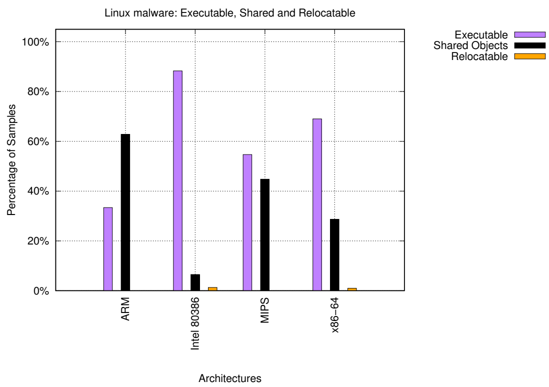

Figure 3. Object file formats. Samples are distributed both as executables and libraries.

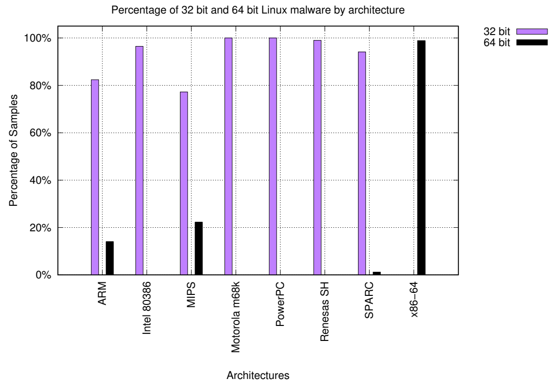

Figure 4. Percentage of 32-bit and 64-bit binaries by architecture. 32-bit binaries are prevalent.

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 is regarding binaries word size (32 or 64 bits). As presented in figure 4, almost all architectures present higher rates of 32 bit samples, as it was the standard until few years ago. Modern samples, however, are already compiled as 64 bits.

4.2. Analysis Methods

Samples analyses proceeded as follows: First, the 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* *ptrace* *strace* *LD_PRELOAD*	Static compilation *ptrace* check Long *sleep* Injection blocking	*ptrace* step-by-step binary patching *LD_PRELOAD* Kernel *hooks*

The static analysis step may be defeated by obfuscation. Such cases are naturally handled by the dynamic analysis step. 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 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.

5. Linux Malware Landscape

In this section, we present the results of evaluating ELF multiple aspects to draw a panorama of the `Linux` malware scenario. We first introduce static analysis results; secondly, we present dynamic analysis ones; third, we dig into network traffic details; finally, we present a detailed study case regarding a popular `Linux` ransomware.

5.1. 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 considered functions and their respective categories are presented in Table 3.

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 execution. The `Environment` category encompasses functions which allows

Table 3. Malware Behavior Taxonomy. Identified linked functions and their associated behavior in the malware context.

Network	Evasion	Environment	Removal	Timing
socket	fork	gettimeofday	remove	alarm
connect	kill	getlogin	rmdir	wait
poll	ptrace	getenv		sleep
select		setenv		

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.

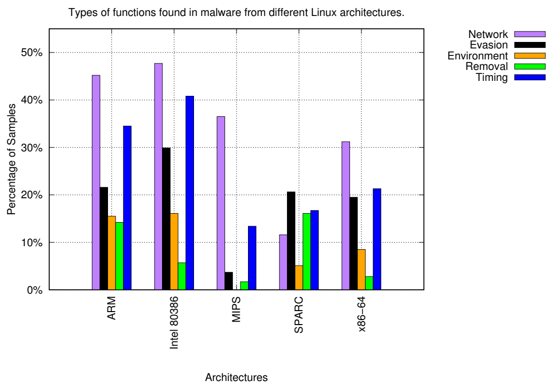

Figure 5. Malware behavior prevalence by malware architectures. We observe that network functions are prevalent.

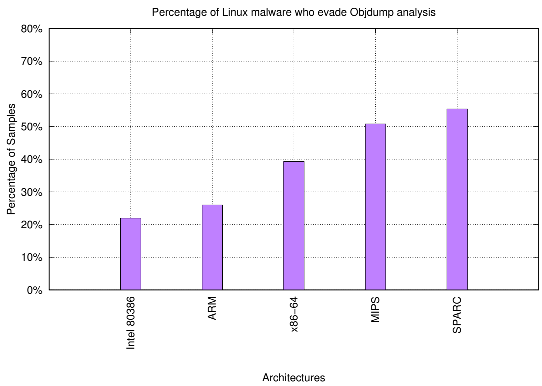

Figure 6. Percentage of malware that failed to dissasembly. Some architectures aren't present because of lack of objdump support.

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 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 functions 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 dissasembly attempts failed. Omitted architectures are due to lack of `objdump` support.

After identifying the high use of network functions, we queried (via `strings` [4]) 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.

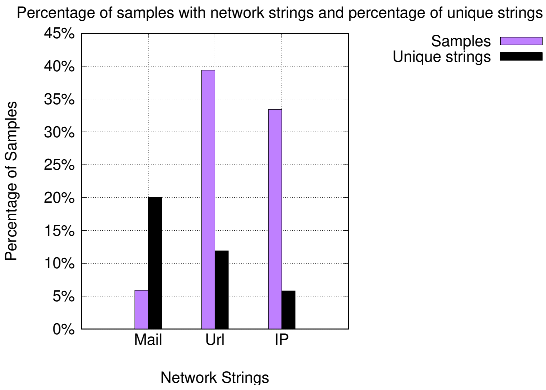

Figure 7. Network-Related Strings. Rate of samples with network-related strings and the fraction of unique strings.

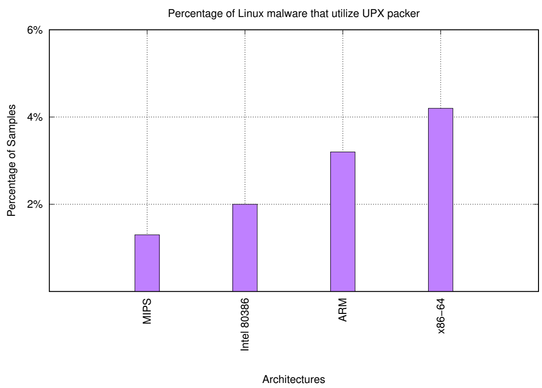

Figure 8. Rate of UPX-packed samples. Few samples are packed. 64-bits samples are the most packed ones.

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`[5], 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.

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 processes to cause these to behave maliciously.

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`[6] with a 90% threshold. Further, all samples were matched against each other. Figure 10 shows the

[4] `http://man7.org/linux/man-pages/man1/strings.1.html`
[5] `https://upx.github.io/`
[6] `https://ssdeep-project.github.io/ssdeep/index.html`

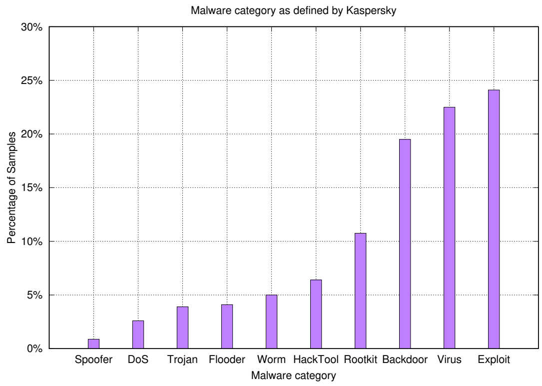

Figure 9. AV labels according Kaspersky AV. We observe a prevalence of exploits and network-related threats.

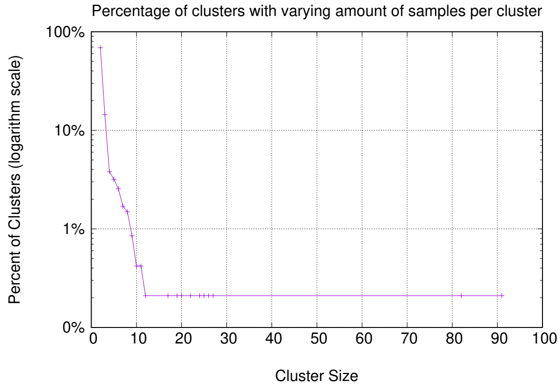

Figure 10. Samples variants clustering. Smaller clusters (up to 5 samples) are prevalent. Largest cluster has 91 samples.

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.

5.2. 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.

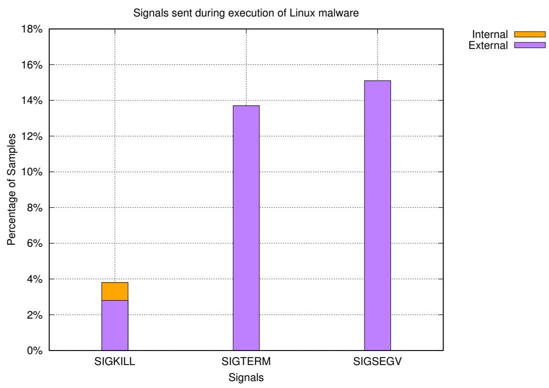

Figure 11. Observed Signals during execution. Most samples terminated prior timeout expiration. Few samples exhibited inter-process interactions.

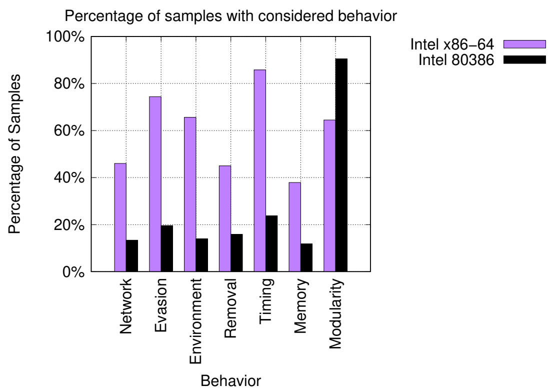

Figure 12. Malware behavior prevalence. Modularity and timing are the prevalent behavior.

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`[7] 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 (≈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, as shown in Table 4. Figure 12 shows the fraction of samples presenting each one of these behaviors.

Table 4. Malware Behavior Taxonomy. Identified invoked system calls.

Network	Evasion	Environment	Removal	Timing	Memory	Modularity
socket	fork	gettimeofday	unlink	time	mmap	execve
connect	kill	access	rmdir	wait	munmap	fork
poll	ptrace	uname	kill	nanosleep	mprotect	clone
select		ioctl				exit
getsockname						getppid

We observe 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 measure 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.

[7] `http://man7.org/linux/man-pages/man1/timeout.1.html`

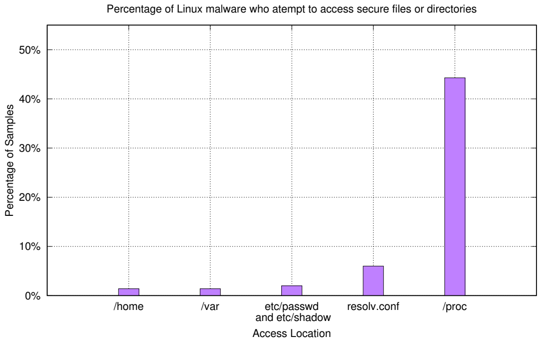

Figure 13. Accessed files and directories. Samples interfere with system configurations and steal credentials.

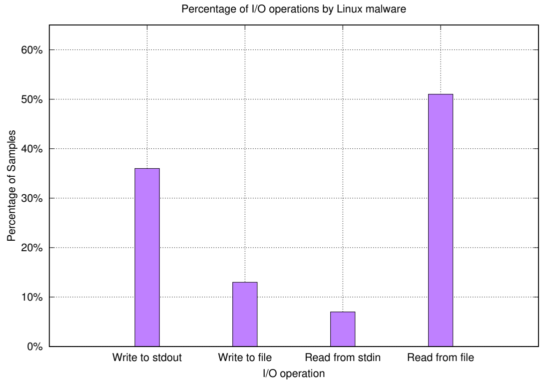

Figure 14. I/O operations. Most samples do not present direct user interaction

We observe most interactions are performed in the form of filesystem access, 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.

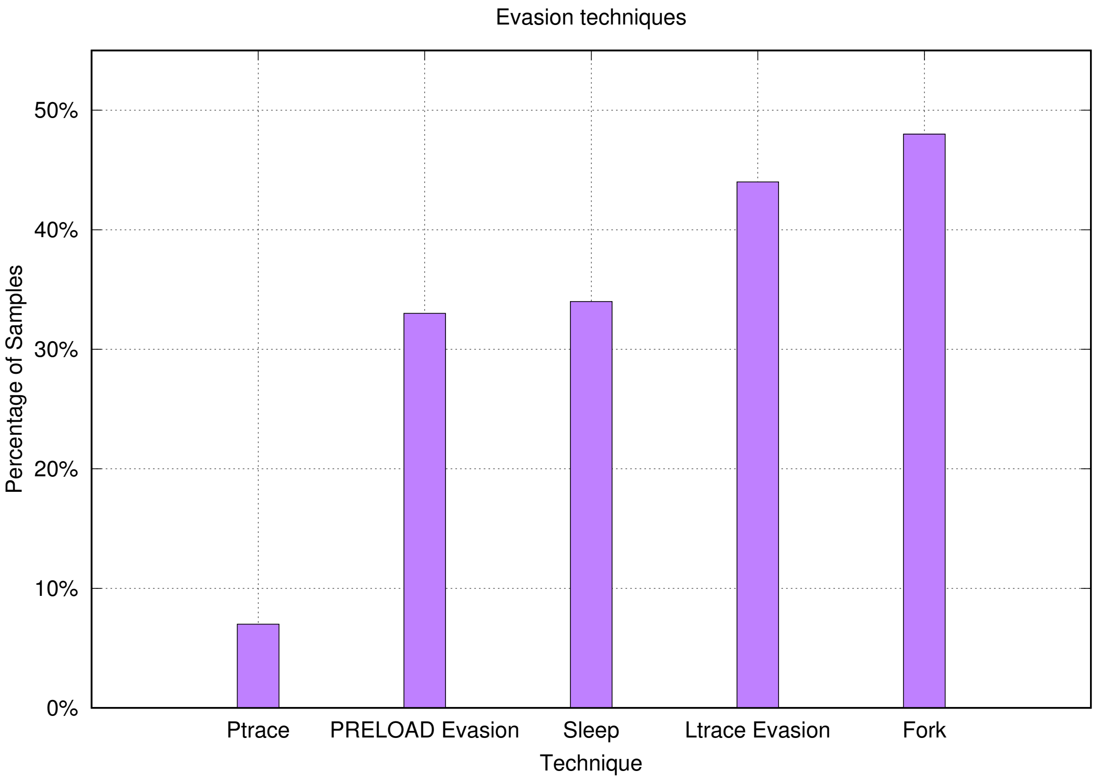

Figure 15. Evasion Techniques. Samples present diversified evasion methods.

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 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.

5.3. Network Traffic

As network use is a significant behavior in many malware samples, we inspected network traffic to better understand how samples make use of network resources.

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.

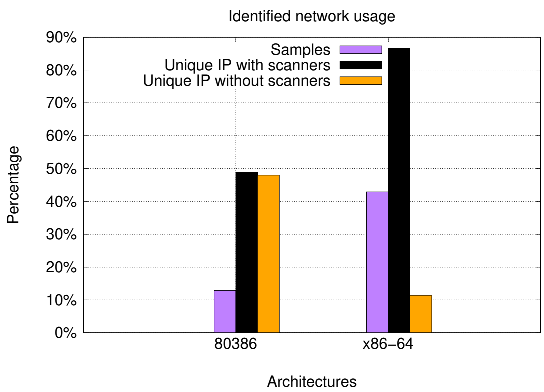

Figure 16. Identified network usage. Scanners dominate unique IP rate.

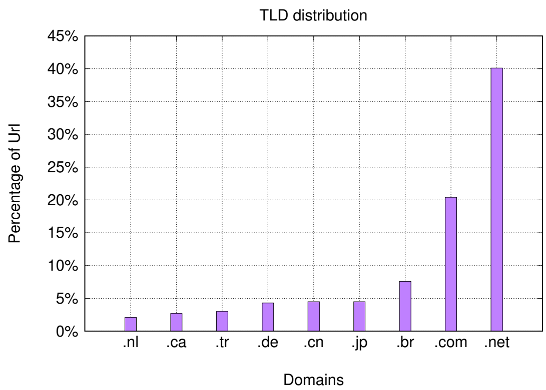

Figure 17. TLD distribution. Global domains (.net and .com) are prevalent. Local domains are present due to scanners enumeration.

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 (≈60%) are associated to domestic internet providers. 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 are not region-aware.

5.4. Study Cases

We have previously presented a landscape of Linux threats. Whereas such "big picture" is essential for understanding the overall threat impact over Linux environments, looking to individual samples may also help us on spotting localized behaviors and predicting future trends. Therefore, in this section we present a more detailed analysis of two threat classes: the SSH backdoor and the Erebus ransomware.

5.4.1. SSH Backdoor

`SSH` is a popular protocol for remote server management and secure data communication. However, it may be abused by malware samples to provide attackers with communication channels to infected machines.

A significant number of analyzed samples (**number here**) presented embedded "ssh" strings, a deemed suspicious pattern. By correlating the identified samples with AV labels, we discovered that these samples were backdoors [Duquette 2013]. Our hypothesis is that these samples are built as trojanized ssh server versions to enable remote machine access. To confirm that, we following present a more detailed analysis of one of these samples[8] while executing in our sandbox solution..

Listing 1 shows that the sample allocates enough space for an entire ssh payload and start matching payloads until finding a public key. By running an SSH port and storing attackers public key in the allowed hosts, the malware grants attackers remote access to the compromised machine.

Listing 1. Backdoor sample in action. It drops attacker key into the system, thus granting remote access.

```
malloc(381) = 0x2083c60
strlen("PPK\016QPB\003bbbba\020mYB‘\022Z@\021fbbbbgbrba"...)
strcat("", "ssh-rsa AAAAB3NzaC1yc2EAAAADAQAB"...)
```

5.4.2. Erebus

Ransomware is the latest trend in malware, being *Erebus* [TrendMicro 2017] the most proeminent sample in the `Linux` enviroment. Understanding it may allows us to foresee further ransomware developments in this platform. To do so, we performed an in-depth analysis of an erebus sample dynamic execution trace.

The sample presented noticeable more advanced behaviors when compared to generic threats. Listing 2 shows that the sample encrypted user files with a loaded public key and contacted runtime-generated Internet IP addresses to notify attacker about the succesful infection. In addition, it refers to onion domains to present user the message which asks for ransom. Whereas the use of `TOR` hidden services by malware samples is not currently widespread, it might happen in a near future given the TOR-enabled anonimous capabilities.

[8] MDR: ELF_Linux_SSHDoor_90DC9DE5F93B8CC2D70A1BE37ACEA23A

Listing 2. Erebus Execution. It connects to runtime-generated IP addresses and to TOR-based hidden services and onion domains.

```
strncmp(""-----BEGIN PUBLIC KEY-----\\nMII"..., "null", 4)
strncmp("3,"tg":"216.126.224.128\\/24","bu"..., "null", 4)
strncmp(""7fv4vg4n26cxleel.hiddenservice."..., "null", 4)
strncmp(""qzjordhlw5mqhcn7.onion.to","qzj"..., "true", 4)
```

6. 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 the 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 ($\approx$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 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.

In summary, we discovered that 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.

We acknowledge that collecting `Linux` is harder than collecting samples targetting other platforms, such as `Windows`, and, therefore, our study can be biased towards the age of the examined samples. However, we consider this limitation as acceptable because our goal is not evaluate individual samples but present a landscape of `Linux` malware capabilities as a whole.

Finally, as future work, the knowledge of the malicious behaviors presented by `Linux` malware will be used along with data from benign software for the development of a machine learning-based classification solution, thus allowing predicting malicious software intents.

7. 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.

Furthermore, we compared malware samples targeting the `Linux` OS to the ones targeting `Windows`. We discovered 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.

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://www.blackhat.com/docs/eu-15/materials/eu-15-KA-Automating-Linux-Malware-Analysis-Using-Limon-Sandbox-wp.pdf.

Asmitha, K. A. and Vinod, P. (2014). A machine learning approach for linux malware detection. In *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pages 825–830.

Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., and Kruegel, C. (2009). A view on current malware behaviors. In *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09, pages 8–8, Berkeley, CA, USA. USENIX Association.

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 Symposium on Security and Privacy (SP)*, volume 00, pages 870–884.

Damri, G. and Vidyarthi, D. (2016). Automatic dynamic malware analysis techniques for linux environment. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 825–830.

Duquette, S. (2013). Linux/sshdoor.a backdoored ssh daemon that steals passwords. https://www.welivesecurity.com/2013/01/24/linux-sshdoor-a-backdoored-ssh-daemon-that-steals-passwords/.

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):26:1–26:33.

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.

Institute, S. (2015). An introduction to linux malware. `https://www.sans.org/reading-room/whitepapers/malicious/introduction-linux-based-malware-36097`.

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 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 356–361.

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 *Proceedings of the 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, BADGERS '14, pages 3–17, Washington, DC, USA. IEEE Computer Society.

Shahzad, F., Bhatti, S., Shahzad, M., and Farooq, M. (2011). In-execution malware detection using task structures of linux processes. In *2011 IEEE International Conference on Communications (ICC)*, pages 1–6.

Sophos (2015). Don't believe these four myths about linux security. `https://news.sophos.com/en-us/2015/03/26/dont-believe-these-four-myths-about-linux-security/`.

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`.

Wired (2017). Linux took over the web. now, it's taking over the world. `https://www.wired.com/2016/08/linux-took-web-now-taking-world/`.