Executive Summary of the Thesis

The Illusion of Randomness: Demystifying the Entropy of ASLR on Common Operating Systems

Laurea Magistrale in Computer Science Engineering - Ingegneria Informatica

Author: Gregorio Barzasi

Advisor: Prof. Mario Polino

Co-advisor: Lorenzo Binosi

Academic year: 2022-2023

1. Introduction

ASLR is a crucial defense mechanism employed by modern operating systems to mitigate exploits that rely on precise object location by randomizing the memory layout of processes. Unfortunately, the performance of this system is very platform-dependent and often proven insufficient [2]. This work analyzes the performance of this mechanism by assessing the degree of Absolute Entropy, or randomness, in memory object placement, and highlighting the Correlation Entropy between objects. We identified many weaknesses in various OS mainly related to the fixed position of executable objects, the most easily exploitable.

2. Motivation

The examination of Address Space Layout Randomization (ASLR) performance is of crucial importance in the study of operating system security as it serves as a last line of defense along with NX and Stack Protection mechanism. The effectiveness of ASLR implementation can be assessed by considering three key aspects: when it randomizes memory, what components it randomizes, and how it executes randomization. The results of this behavior can be captured by estimating the entropy of each memory object. A common weakness in ASLR implementation is the presence of low Absolute Entropy; this is often attributed to fixed address bits, the existence of sections that can grow dynamically, memory fragmentation, and non-uniform probability distributions within memory sections. Another vulnerability is low Correlation Entropy between memory objects, which can potentially reduce the effort needed by an attacker to guess the position of an executable object in the presence of a memory address leak. We call these *Positive Correlation Paths*. Existing research predominantly focuses on Linux as it's the most common kernel used in enterprise systems [4]; however, this left many consumer OSes without evaluation, exposing many final users to security risks. Our study starts with Ubuntu and proceeds to consider multiple consumer operating systems such as MacOS (specifically the ARM version), Windows 11, and Android 13. To facilitate our research, we developed a novel ASLR analyzer tool. Even though it's not the first time that some of the mentioned OS have been analyzed, it must be noted that previous research has some limitations, such as insufficient sampling sizes, and a lack of Correlation Entropy estimation, moreover, they considered only a narrow scope of OS system. Addressing these limitations is vital for conducting a comprehensive and robust ASLR analysis.

3. Approach

The analysis of ASLR implementation can be approached in two ways. The first involves reading the implementation code inside the kernel of the operating system to evaluate the randomization algorithm. However, this approach is challenging due to closed-source operating systems like Windows and MacOS, making it difficult to access the source code. Even for open-source kernels like Linux, runtime interactions between memory objects complicate the analysis. The second approach, which we have chosen, involves empirical analysis, using an ad-hoc script to collect many samples and then perform statistical analysis on the data.
To perform the sampling and analysis a new ASLR analyzer tool has been developed, able to run on many of the considered OS with minor tuning and sharing most of the code; the only exception was Android which due to its peculiarity required an ad-hoc approach. The sampling focuses on collecting addresses of multiple memory objects from various allocation flows. It uses an ad-hoc C program to allocate objects and print their addresses to standard output for collection; this approach is significantly faster than using virtual memory mapping information provided by OS-embedded tools. After careful testing, we identified a suite of 6 allocation sizes able to stress each OS memory allocation system: `16B`, `512B`, `4KB`, `256KB`, `4MB`, and `128MB`.
To analyze slow entropy sources, such as system reboots, we needed to work using as less samples as possible to complete the data collection in a reasonable time. The number of samples needed is directly dependent on the entropy estimator chosen in the analysis process. Unlike many other researchers, we used the *NSB estimator* [3], an estimator specifically developed to work with under-sampled sources, able to provide robust results even with few samples compared to full source sampling. To determine the minimum sample requirements for different operating systems and scenarios we conducted multiple tests with few samples and then used the *posterior standard deviation* to quantify the uncertainty in the result. The resulting sample sizes for having bias less than 5% range from 3,8 Mln for Linux to just 10,000 for Android 13. Also, the reboot number was estimated with the same method giving results in the range of 3,000 reboots for Android 13 to 10,000 for Windows 11.

4. Results Discussion

We analyzed various operating system configurations: *Linux 5.17.15*, *Linux 6.4.9*, *MacOS M1 Ventura*, *MacOS M1 Ventura* using Rosetta, *Windows 11*, and *Android 13*. To better understand how effectively they randomized memory and protected against attacks we evaluated the memory layout, probability distributions, absolute entropy, and correlation entropy between allocated objects.
Examining the memory layout of the various OSs confirmed that all of them utilize the so-called **partial-VM** randomization, a technique that divides the memory into sections and then randomizes the objects inside them. Looking at probability distributions, Linux is the best-performing kernel, having a uniform shape across all sections. Windows 11 follows by employing a triangular shape, that hints at the use of two random variables to increase entropy. Other OSs have an unknown distribution shape, with spikes and some high probability zones that suggest a low absolute entropy. To evaluate the entropy performance we compared our results to the commonly accepted 20 bit threshold used to indicate sufficient randomness. In the analysis, Linux was revealed to be the only Kernel to randomize all executable objects with an excellent entropy of 27 bit. On the contrary, MacOS M1 Ventura and Android 13 randomized only the text object, with insufficient entropy of around 13 bit. Windows 11 was the worst of all, having all executable objects fixed. When we sampled multiple reboots of the devices the performance improved with every executable object now randomized: libraries now experience around 10 bit of entropy on Android, 13 bit on Windows, and 12 bit on MacOS. Still insufficient in providing any kind of protection as a remote attacker could still brute force the addresses in less than 10,000 tries and also exploit the **byte-for-byte** cracking technique as the address changes only

path	ent	diff	path	ent	diff
Linux 5.17.15					
`text←heap_object`	13.00	-14.41	`lib←thread_heap_object`	15.15	-12.25
`text←global_var`	0.00	-27.41	`lib←mmap_object`	0.00	-27.41
`lib_1←lib_2`	0.00	-27.41	`lib←shared`	0.00	-27.41
Linux 6.4.9					
`text←heap_object`	13.00	-14.41	`lib_2←thread_heap_object`	15.15	-12.25
`text←global_var`	0.00	-27.41	`lib_2←mmap_object`	0.00	-27.41
`lib_2←lib_1`	9.00	-18.41	`lib_2←shared`	0.00	-27.41
`lib_1←tls_var_thread`	0.00	-19.02	`lib_1←thread_heap_object`	6.38	-12.79
`lib_1←huge_mmap_object`	0.00	-19.02	`lib_1←mmap_object`	9.00	-10.02
`lib_1←stack_thread`	9.00	-10.02	`lib_1←shared`	9.00	-10.02
MacOS M1 Ventura					
`text←shared`	0.00	-11.58	`text←mmap_object_thread`	2.16	-9.42
`text←global_variable`	0.00	-11.58	`text←mmap_object`	0.01	-11.58
MacOS M1 Ventura Rosetta					
`text←shared`	0.00	-13.58	`text←mmap_object_thread`	1.19	-12.39
`text←global_var`	0.00	-13.58	`text←mmap_object`	0.00	-13.58
Android 13					
`text←global_variable`	0.00	-13.10	`text←malloc_128MB_from_main`	9.55	-3.55

Table 1: Positive Correlation Paths to executable objects

at boot time. When we compare the absolute entropy performance on various Linux versions we clearly observe a reduction in the latest updates. In fact, after Linux 5.17.15 a new memory management structure was introduced, *Linux Folios*, that improved performance at the cost of less entropy on objects greater than 2MB; this resulted in a reduction from 27.4 bit to 19 bit entropy, around 8.5 bit loss on the big code libraries like *glibc*, thus decreasing the effort needed to build a working ROP by almost **400x**. The reduction is still present in the last available version 6.4.9. and no clear way to disable this system is known. Overall, the other objects are well randomized on Linux, with the worst performance belonging to thread allocation and "second-time" allocation, enforcing the theory that allocation patterns have a significant impact on randomization capabilities on Linux. On this aspect Windows 11 was the best, appearing immune to those pattern dynamics and having a greater than 20 bit entropy almost in every non-executable object. The worst performing are MacOS and Android, which have entire memory sections practically not randomized. The randomization of non-executable objects is important mainly in the context of correlation entropy and positive correlation path, where the address leak could potentially reduce the entropy of an executable object. In Table 1 we specifically called out the most concerning positive correlation paths to executable objects. In particular, we confirmed that the leak of a heap address in Linux 6.4.9 still reduces the entropy of text to just 13 bit, as previously highlighted in other research [1], reducing the entropy by 14 bit. For what concerns Windows 11, there are no positive correlation paths as the position of all executable objects is fixed, so no leak can beat that information.

5. Attack scenario

While all unrandomized objects pose vulnerabilities, some of them, like executable ones, are higher-priority targets for attacks such as ROP. We defined profiles for local and remote attackers with or without address disclosure to evaluate the severity of the weaknesses found. All scenarios assumed the ability of the attacker to bypass stack smashing protection.
Fixed positioning of text and libraries allows immediate exploitation as local attackers could gather executable object positions from other programs. If we rule out such information disclosure, both local and remote attackers resort to brute-forcing addresses, with an expected ef-

fort based on entropy introduced in the rebooting process. They could involve also more sophisticated exploitation techniques to reduce the attack complexity such as byte-for-byte crack of the addresses. If we consider kernel with process-level randomization of executable objects, such as Linux, the best attack approach is to gather information about other allocation positions, reducing the complexity of exploitation by using *Positive Correlation Paths*. In case no information disclosure is available the best it can do is to brute-force the address of an executable object; since Linux 5.17.15, we discovered it to be a reasonable approach thanks to the reduction in entropy of big code libraries such as *glibc*.
Distributed attack scenarios were also considered. In the presence of a wide diffused vulnerability exploitable remotely, an attacker could target multiple systems at once. In this case, reboot entropy will indicate the number of expected successes in a single attack attempt. For instance, a vulnerability of this type affecting 10,000 MacOS systems with a reboot entropy of libraries of 12.3 bit, means that on one try the attacker should hit the target on around 2 remote MacOS systems.
To validate our findings and our entropy estimator choice we set up a dummy executable with a common buffer overflow vulnerability; inside the text, we placed a function able to disclose a flag, and we used it as the exploit target. The objective was to use the information gathered by leaking the address of a heap object to hit the target function inside the text. In a no-leak scenario, we will need to brute force the entire 27.4 bit entropy associated with the text object, expecting a hit every 177Mln tries. To reduce the complexity, we used the Positive Correlation path in Linux that correlates heap and text with just 12.99 bit of entropy. After 2Mln tries we counted 261 hits; that average to a hit every 7650 tries, so an empiric entropy of 12.90bit. This means that our estimated entropy of 12.99 bit was just 0.753% away from the empirical one, so our estimate was revealed to be solid.

6. Conclusion

In this document, we evaluated the effectiveness of Address Space Layout Randomization (ASLR) across major desktop and mobile platforms through a statistical analysis of memory object positions; utilizing the NSB entropy estimator permitted the analysis of slower processes like device reboots by reducing the sample requirements.
Major issues were highlighted: in some cases, problems persisted long-term, like the lack of entropy for libraries and executable code in Windows, MacOS, and Android while others started recently, such as entropy reduction in recent Linux distributions, likely introduced by the adoption of Large Folios. Generally, Linux distributions provide strong randomization, however, positive correlation paths on Linux reduced entropy, lowering it dangerously. A proof-of-concept exploit has validated our analyses. Our findings revealed opportunities to fortify implementations by resolving the correlation entropy problems, optimizing allocation patterns, and increasing overall absolute entropy.
While proposals to improve Linux ASLR exist, like ASLR-NG [2], they are currently not adopted. Moreover, the introduction of Linux Folios demonstrates how evolution occasionally exchanges security for performance. Broad adoption of new 5-level paging architectures, providing more bits to the randomization process, may resolve this compromise. Future work includes profiling real software memory mappings and allocated objects to evaluate the performance at a software level and not at the operating system level. Extending the comparisons across security-focused Linux distributions could also offer insights for enterprise operating system selection.

References

[1] Hector Marco Gisbert and Ismael Ripoll. Exploiting linux and pax aslr's weaknesses on 32-bit and 64-bit systems. March 2016. Black Hat Asia 2016 ; Conference date: 29-03-2016 Through 01-04-2016.

[2] Hector Marco Gisbert and Ismael Ripoll Ripoll. Address space layout randomization next generation. *Applied Sciences*, 9(14), 2019.

[3] Ilya Nemenman. NSB Entropy Estimator.

[4] W3Techs. Usage statistics of operating systems for websites, Sep 2023.