Exploiting Address Space Layout Randomization (ASLR)

Deni Bačić
Faculty of Computer and Information Science,
Ljubljana, Slovenia
deni@bacic.si

***Abstract*—Address Space Layout Randomization (ASLR) is a computer security technique that provides protection from buffer overflow attacks. ASLR randomizes address space positions of key data of a process, thus making it difficult for an attacker to reliably jump to a particular exploited function in memory. In this paper we study limitations of ASLR and present viable exploitation options that can circumvent kernel space ASLR on current operating systems. We also discuss mitigation strategies against some of the attacks with their strengths and weaknesses revealed.**

***Index Terms*—Address Space Layout Randomization, Side Channel, Kernel Vulnerabilities, Exploit Mitigation, Timing Attacks.**

I. Introduction

MODERN computer architectures implement a wide variety of methods to prevent memory corruption attacks and overflows. ASLR is one of the most advanced ways of protection. The basic idea is simple: randomize parts of the memory and make it more difficult for an attacker to predict target memory addresses. In most cases attackers are either trying to locate their code to be executed (return-to-libc for example) or trying to find the stack address to execute injected shell code. In both cases, the operating system obscures memory addresses and the attacker has to guess them, taking him more time and resources. Usually a mistaken guess is non-recoverable as it leads to the crash of either the application or operating system. ASLR is currently widely used and can be found in most modern computer operating systems such as Linux, Windows, macOS and even most popular mobile platforms such as iOS and Android. Adoption timeline of both user-space and kernel-space ASLR can be seen on Fig. 1.

II. Technical Background

Different kinds of buffer overflow attacks account for major part of present exploits and 0-day vulnerabilities. C programming language assumption that programer should be responsible for data integrity, allows us to write more data than allocated to the buffer [2]. Program usually crashes, but additional data is nevertheless overflown to other memory addresses. Basically the attacker wants to execute custom code that in most cases calls some kind of privilege escalation, that gives him desired rights on the system. Modern programming languages have implemented memory protection mechanisms, but almost all vital OS and kernel modules are still written in C.

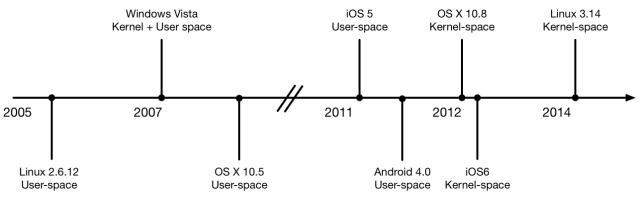

Fig. 1. Timeline with adoption status of both user and kernel-space ASLR in popular operating systems [1].

III. Attack Approaches

Brute-force and memory disclosure attacks on ASLR are the topic of several works [3] and [4], even mobile platforms could be exploited to attacks of this kind. One of the first ideas about attacking ASLR using cache-based timing side attack was by R. Hund et al. [5]. They developed three attack strategies, that were theoretically working, but in practice very difficult to implement. Additionally, they exploited a particular property of Intel CPU's to determine which memory addresses are allocated. Real world exploitation is limited by the last-level cache noise from processor and need for the prior knowledge of physical address of data. All described attacks are caused by hardware side-channel or cache analysis and are OS independent.

A. Intel specific improvements

Y. Jang et al. [1] improved the proposed attacks (methods?) by introducing a highly stable timing attack against kernel-space ASLR, called DrK. The attack can precisely derandomize address the kernel layout. Basics of the exploit are hidden in the CPU hardware feature called Intel Transactional Synchronization Extensions (TSX). As the name implies, it is present in modern Intel processing units. TSX aborts a transaction without notifying the underlying kernel even when the transaction fails due to critical error (page fault, access violation). Errors of that kind would otherwise crash the probing code or the system itself. DrK turns that property into a precise timing channel. It determines the mapping and execution status of the privileged kernel space. Noise reduction, in contrast to attacks presented before, is visible on Fig. 2. The exploit is very precise, works on all operating systems (even in virtualized environments!) and generates no visible footprint, making detection very difficult in practice.

B. Exploiting Branch Target Buffer

Article by D. Evtyushkin et al. [6] develops similar attack to derive kernel and user-level ASLR offset using a side-channel attack on the branch target buffer (BTB). It exploits the observation that an adversary can create BTB collisions between the branch instructions. This collisions influence the the timing of attacker's code, allowing him to identify the locations of known branch instructions in the address space, similarly as in DrK attack mentioned above. Work was further improved with proof of concept how attacker can exploit base OS even through virtual machine [7].

IV. Mitigation Approaches

Currently there is no viable countermeasure to avoid described attacks on ASLR, that would not produce massive overhead and hurt both usability and performance of the system.

First idea is to modify CPU to eliminate timing channels totally, but obvious problem arises, hardware is already shipped and deployed and cannot be modified only by microcode or firmware updates. Another way of making attacks more difficult is having more coarse-grained timer in system. In reality, a lot of code uses the benefit of precise timing for everyday operations. Using separate page tables for kernel and user processes could help avoid TLB exploitation, but it would have high overhead due to frequent cache flushes. We could insert fake mapped and executable pages between the real ones, but ASLR does not give enough space to insert them efficiently. Monitoring of hardware events could be implemented, but it is difficult to separate benign programs from real attacks. Live ASLR re-randomization would also improve safety, but has very high performance impact as of this date.

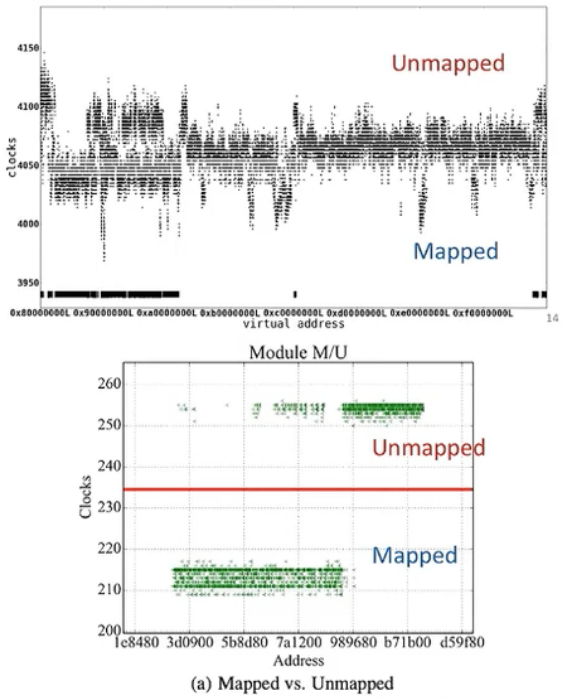

Fig. 2. Noise reduction with TSX approach (second diagram) can be clearly observed in contrast to previous attack methods above [1].

V. Concluding Remarks

In this paper we have discussed various ways of exploiting a security feature implemented in almost all modern operating systems. We concluded that even defence mechanisms can be a security threat and should not be trusted blatantly. New features in hardware and software that should improve the performance of the system, can in fact hinder the security when implemented incorrectly or without knowing the impact on other processes and approaches. Unfortunately viable countermeasures are still to be found and should encourage further analysis.

References

[1] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 380–392.

[2] J. Erickson, *Hacking: The Art of Exploitation*. No Starch Press, 2008.

[3] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[4] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib (c)," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 60–69.

[5] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 191–205.

[6] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *Proceedings of 49th International Symposium on Microarchitecture (MICRO)*, 2016.

[7] F. Wilhelm, "Poc for breaking hypervisor aslr using branch target buffer collisions." [Online]. Available: https://github.com/felixwilhelm/mario_baslr