A Study on Kernel Address Layout Randomization (KALSR)

Karthik Ravi Kumar[1], Ashish Kumar Rambhatla[2], and Swarali Chine[3]

[1]School of Computing and Augmented Intelligence, Arizona State University

December 8, 2022

Abstract

1 Introduction

It is crucial to take into account the guarantees that the security enforcement mechanisms may provide when setting up secure systems. One of the most prevalent groups of vulnerabilities used to compromise the control-flow integrity of computer systems is buffer-overflow attacks. Computer systems are now more commonplace and integrated into our daily lives than ever before. There are countless similar devices available now, including cellphones, washing machines, cars, etc. With so many interconnected and widely dispersed systems, a buffer overflow attack can harm much more than just one system. Billions of devices may be compromised worldwide by a single vulnerability. In the simplest terms, a buffer overflow is when a software tries to write anything outside of the memory it requested. Typically, a program's memory-bound checks may be used to guard against this type of attack. However, we cannot always rely on the developers to carry out the bound checks, therefore a new system-level protection technique called Address Space Layout Randomization (ASLR) is created.

To understand how ASLR protects a system from buffer overflow and other typical threats, we must first understand how a program is loaded into a process's address space. Before we get there, let's take a look at how each application's virtual address space is constructed. A CPU provides memory instructions with the instruction width, say 32bits, as shown in Figure 1. This is referred to as the logical address. This logical address is concatenated with a process id (pid), say 10 bits, before being sent to the memory controller. The concatenated address is known as the memory's virtual address, and it is 42 bits long. This results in a virtual address space of $2^{42} bytes$.

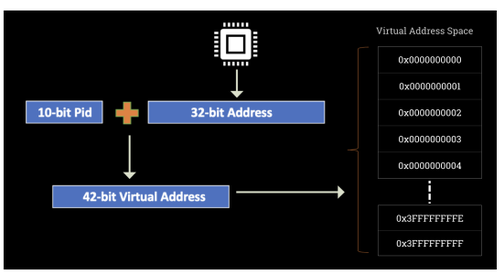

Figure 1: How a virtual address space is created for each application.

Let us now walk through the compilation process to observe how a program written in a high level language is transformed into machine code and loaded into process memory. Consider the following c code, as illustrated in 2. The compiler examines the program and turns it into assembly instructions, as illustrated in 3. These machine instructions are organized according to their nature, such as data segment or code segment. The loader then loads distinct parts of the machine code into separate locations of the process virtual memory, as shown in Figure 4.

Let us now walk through the compilation process to see how a program written in a high level language gets converted into machine code and loaded into process memory. Let us consider the following c code as shown in Figure 2. The compiler reads the program and converts it into assembly instructions as shown in Figure 3. These machine instructions are arranged based on their type such as data segment or code segment. The loaded then loads different sections of the machine code into different areas of the process virtual memory as shown in Figure 4. Before the *Address Space Layout Randomisation* (ASLR) was established, the loader used to load these pieces of code at the exact same random places for each run of the application. In the following sections, we will explain how ASLR would affect the process loading process, as well as the benefits and limits of such a method.

```c
int globalVar = 10;

int square(int num) {
    return num * num;
}

int main() {
    int result = square(2);
    return (result + globalVar);
}
```

Figure 2: Example C program.

2 What is KASLR?

KASLR stands for Kernel address space layout randomization. Before we delve into the details of the KASLR, lets discuss the basic idea of the address space layout randomization. It is a security feature used in OS systems to assist avoid vulnerabilities by randomizing the places in memory where specific data is stored. This makes it more difficult for attackers to forecast where they may locate weak sections of code or data to exploit. ASLR is employed as a defense-in-depth mechanism to make it more

```
0:  55                      push   rbp
1:  48 89 e5                mov    rbp,rsp
4:  89 7d fc                mov    DWORD PTR [rbp-0x4],edi
7:  8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
a:  0f af c0                imul   eax,eax
d:  5d                      pop    rbp
e:  c3                      ret
000000000000000f <main>:
f:  55                      push   rbp
10: 48 89 e5                mov    rbp,rsp
13: 48 83 ec 10             sub    rsp,0x10
17: bf 02 00 00 00          mov    edi,0x2
1c: e8 df ff ff ff          call   0 <_main>
21: 89 45 fc                mov    DWORD PTR [rbp-0x4],eax
24: 8b 15 00 00 00 00       mov    edx,DWORD PTR [rip+0x0]        # 2a
<main+0x1b>
2a: 8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
2d: 01 d0                   add    eax,edx
2f: c9                      leave
30: c3                      ret
```

Figure 3: Corresponding Assembly and machine code for the C program in Figure 2

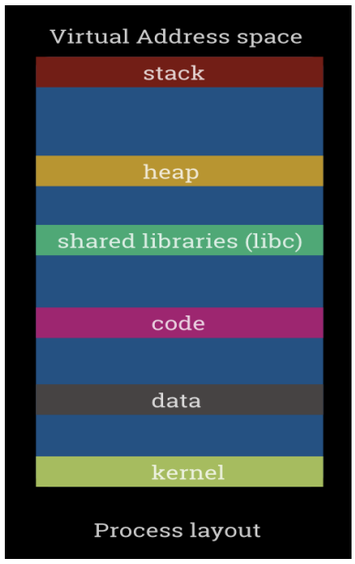

Figure 4: Process address space layout.

difficult for attackers to successfully exploit software flaws. ASLR is often implemented by the kernel of the operating system, which is the fundamental component of the system that handles memory and other resources. When a program is loaded into memory, the kernel will randomly allocate it to a position in memory rather than constantly loading it at the same address. This makes it more difficult for attackers to forecast where they may locate the data or code they wish to target, and thus improves overall system security. The Figure 5 shows how ASLR randomly re-assigns the placement of data, stack, and other portions of the program in the process address layout in subsequent application invocations.

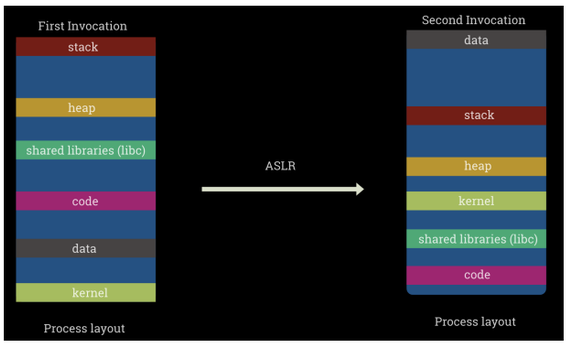

Figure 5: Address Space Layout Randomization.

When the same technique is used in the Kernel virtual address space, it is referred to as Kernel Address space layout randomization. Because the lifespan of the kernel is often the same as the system bootup time, randomization happens at every bootup in the case of KASLR. Figure 6 depicts how the various regions of the Kernel virtual address space are reassigned during subsequent bootups.

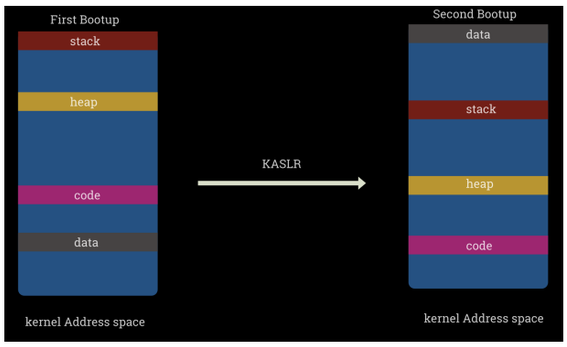

Figure 6: Kernel Address Space Layout Randomization.

The following sections will go through in depth how the KASLR mechanism will guard against, or at least make it difficult for, several common yet deadly attacks.

3 Attacks defended by KASLR

3.1 Return oriented programming attacks

Return-oriented programming (ROP) is a security exploit technique that allows an attacker to execute code on their target system. In this attack, the attacker uses control of the call stack to indirectly execute carefully chosen machine instructions or a group of machine instructions that are already present in the machine's memory, called gadgets. These gadgets usually terminate with a return instruction and is located within the existing program or shared library code.There are certain gadgets that allow the attacker to load and store data from one place to another as well. Many of these "gadgets" are chained together and used by the attacker to perform arbitrary operations of a machine. These gadgets are used to exploit different vulnerabilities which lead to different attacks such as buffer overflow, stack buffer overflow, return-to-libc attacks. By randomizing of the location of library and program code, the attacker cannot stitch gadgets as the attacker cannot predict the location of instructions which can be used to make gadgets. But if the attacker can find the location of one known instruction, the position of all the other instructions can be inferred and a return oriented programming attack can be developed. The randomization approach can be taken further hardened by relocating all instructions and other program state of the program separately, instead of just the library locations. This modified approach is successful at making it difficult to build and utilize gadgets, but comes with a significant overhead as it requires extensive runtime support to piece back the randomized instructions back at runtime.

3.2 Stack smashing attack

During a programs execution, when the program writes to a memory address which is beyond the bounds of the structure of the program's call stack,

it is called a buffer overflow. It usually occurs when a program tries to write more data into the buffer located on the call stack than the actual space allocated for that buffer. This might cause the adjacent data on the stack to be corrupted, when this happens due to an error/bug will usually lead to the program to crash or operate in an unexpected way. The stack buffer overflow is a type of a more general malfunction called buffer overflow. Stack buffer overflows are more likely to disrupt the program execution than a regular buffer overflow as the stack contains return addresses for all active function calls. Stack buffer overflow can be used deliberately as an attack by an attacker supplying data in the stack in such a way that he intends to inject executable code into the running program and use it to control the process. We can see in Figure 7, an empty stack with number of frames equal to 4. The given instructions are executed which forces the stack to overflow and the ending instructions return a pointer to the memory address of a shell. The attacker can use this shell to execute arbitrary exploits on the system.

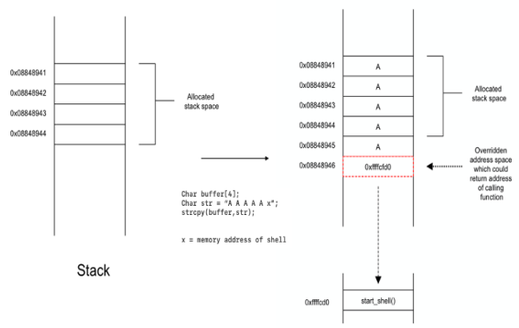

Figure 7: Stack Smashing attack.

To prevent this, instead of removing the code from the data supplied, the memory space of the executing program is randomized. As the attacker should determine where the executable code that can be used is stored, this will be difficult as the memory layout being randomized will prevent the attacker from knowing any location of the code.

3.3 Return to libc attack

This attack is used to exploit buffer overflow vulnerabilities on systems with DEP(Data Execution prevention feature) enabled. In the general buffer overflow attack the attacker writes code and injects it into the vulnerable program's stack and executes it on the stack. But on the systems with DEP enabled, the No-Execute bit is set and attackers can no longer execute their code from the vulnerable program's stack. To bypass this NX (No-Execute bit) protection and subvert the program's execution flow, the attacker re-uses executable code which is already present in the process executable memory. This is usually the standard C library shared object, which is already mapped and loaded into the program's virtual memory space. Most of the time, this exploit is used to detect the memory address of system() and is used to spawn a shell.

In this exploit, attacker bypasses NX by finding already loaded executables (libc) on the memory, KASLR prevents this by randomizing the address of these executables and thus reducing the probability of finding the memory address of system().

4 Side Channel Attacks

4.1 Breaking KASLR Using Memory Deduplication in Virtualized Environments

[1] A memory deduplication attack is a type of memory disclosure attack that allows an attacker to infer the content in the victim's memory. This attack exploits a Copy-on-Write (CoW) mechanism employed in hypervisors, such as KVM and VMWare ESXi, to enable memory deduplication Our attack attempts to break the KASLR of other VMs by reconstructing a memory deduplication attack in a virtualization environment.In our attack model, we suppose that the spy and victim VM are co-located on the same host. Both VMs run the latest version of Linux, where the KPTI is enabled by default. We also suppose that the host runs a hypervisor that provides a memory deduplication technique.As Linux applies KASLR to its kernel base and kernel modules independently, we demonstrate two attacks: one for breaking the KASLR in the kernel base and the other for breaking the KASLR in the kernel module.

4.2 Practical Timing Side Channel Attacks Against Kernel Space ASLR

[2] First Attack: Cache Probing - Our first method is based on the fact that multiple memory addresses have to be mapped into the same cache set and, thus, compete for available slots. This can be utilized to infer (parts of) virtual or physical addresses indirectly by trying to evict them from the caches in a controlled manner. More specifically, our method is based on the following steps: first, the searched code or data is loaded into the cache indirectly (e.g., by issuing an interrupt or calling sysenter). Then certain parts of the cache are consecutively replaced by accessing corresponding addresses from a user-controlled eviction buffer, for which the addresses are known. After each replacement, the access time to the searched kernel address is measured, for example by issuing the system call again. Once the measured time is significantly higher, one can be sure that the previously accessed eviction addresses were mapped into the same cache set. Since the addresses of these colliding locations are known, the corresponding cache index can be obtained and obviously this is also a part of the searched address. For successful cache probing attacks, an adversary needs to know the physical addresses of the eviction buffer, at least those bits that specify the cache set. Furthermore, she somehow has to find out the corresponding virtual address of the kernel module from its physical one. This problem is currently solved by using large pages for the buffer, since under Windows those always have the lowest bits set to 0.

Second Attack: Double Page Fault - The second attack allows us to reconstruct the allocation of the entire kernel space from user mode. To achieve this goal, we take advantage of the behavior of the TLB cache. When we refer to an allocated page, we mean a page that can be accessed without producing an

address translation failure in the MMU; this also implies that the page must not be paged-out. Although the double page fault measurements only reveal which pages are allocated and which are not, this still can be used to derive precise base addresses as we have shown by using the memory allocation signature matching. Furthermore, the method can be used to find large page regions.

Third Attack: Address Translation Cache Preloading - While it is often possible to infer the location of certain drivers from that, without driver signatures it only offers information about the fact that there is something located at a certain memory address and not what. However, if we want to locate a certain driver (i.e., obtain the virtual address of some piece of code or data from its loaded image), we can achieve this with our third implementation approach: first we flush all caches (i.e., address translation and instruction/data caches) to start with a clean state. After that, we preload the address translation caches by indirectly calling into kernel code, for example by issuing a sysenter operation. Finally, we intentionally generate a page fault by jumping to some kernel space address and measure the time that elapses between the jump and the return of the page fault handler. If the faulting address lies in the same memory range as the preloaded kernel memory, a shorter time will elapse due to the already cached address translation information.

4.3 Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR

Cache attacks are side-channel attacks exploiting timing differences introduced by CPU caches. Cache attacks have first been studied theoretically, but practical attacks on cryptographic algorithms followed since 2002. In the last ten years, fine-grained cache attacks have been proposed, targeting single cache sets. In an Evict+Time attack, the attacker measures the average execution time of a victim process, e.g., running an encryption. The attacker then measures how the average execution time changes when evicting one specific cache set before the victim starts its computation. If the average execution time is higher, then this cache set is probably accessed by the victim.

4.4 Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs

References

[1] https://www.mdpi.com/2079-9292/10/17/2174.

[2] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6547110

[3] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, Daniel Gruss. KASLR: Break It, Fix It, Repeat.

[4] Yeongjin Jang, Sangho Lee, and Taesoo Kim Georgia Institute of Technology, Breaking Kernel Address Space Layout Randomization with Intel TSX.

[5] Return Oriented Programming (ROP) attacks https://resources.infosecinstitute.com/topic/return-oriented-programming-rop-attacks/.

[6] Micro architecture attacks on KASLR https://cyber.wtf/2016/10/25/micro-architecture-attacks-on-kasrl/.

[7] Lecture 5 - Code reuse attacks, advanced exploits, discussion https://www.youtube.com/watch?v=jDp8nm1Wo1g.

[8] W1_8d : Demonstration of Shell Creation using a Buffer Overflow on the Stack. https://www.youtube.com/watch?v=fqzqdcFwbv0list=PLyqSpQzTE6M-q0Xgn0icEHvUS7WQxvenv.

[9] David J Day, Zhengxu Zhao. Protecting Against Address Space Layout Randomization (ASLR) Compromises and Return-to-Libc Attacks Using Network Intrusion Detection Systems.

[10] How ASLR protects Linux systems from buffer overflow attacks https://www.networkworld.com/article/3331199/what-does-aslr-do-for-linux.html

[11] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning, On the Expressiveness of Return-into-libc Attacks.

[12] https://security.stackexchange.com/questions/18556/how-do-aslr-and-dep-work.

[13] https://github.com/bcoles/kasld