Address Space Layout Randomization

Zachary Taylor

I. INTRODUCTION

ADDRESS space layout randomization (ASLR) is a security measure that many popular operating systems use to slow down and lessen the chances of success of buffer overflow attacks. This method randomizes key parts of a programs address space to make it more difficult for an attacker to find an exploited function in memory. Because the memory locations are randomized, it makes it much more difficult for an attacker to guess where the areas are located.

In this paper, I will be analyzing the background, implementation, and effectiveness of ASLR. I would like to understand the evolution of ASLR as it progressed from an experimental idea to a method implemented in all major operating systems. My goal will be to gain a strong understanding of the concepts and methods behind a modern implementation of randomizing address space layouts. Though I will mention a few common ASLR implementations, I will focus mainly on the implementation in the Linux kernel. Once I have set up a strong basis for ASLR, I will analyze how effective this method is for thwarting attackers. In addition, I will weigh the performance costs of implementing ASLR on a system when compared to a system without ASLR, or even a partial implementation.

II. BACKGROUND

To understand why address space layout randomization is important, it is necessary to make sense of the kinds of attacks it helps mitigate. Buffer overflow attacks are the main class of attacks that ASLR tries to prevent. A buffer overflow happens when the data being entered into a buffer is larger than the allocated buffer space. The data typically then is placed in the adjacent memory locations to the buffer. It is usually unknown what is in those overwritten memory locations. An example of this bug can be seen in Fig. 1. In this example, a string of length eight is placed into a buffer of length six. Since the input data is longer than the buffer, the extra data is overflowed into the following memory locations. In this instance, the following memory locations make up another variable.

Listing 1. Code before Figure 1a.

```
char buffer[6] = { };
unsigned short other = 0x07BB;
```

var	buffer[0]	buffer[1]	buffer[2]	buffer[3]	buffer[4]	buffer[5]	other[0]	other[1]
hex	00	00	00	00	00	00	07	BB

TABLE I

FIGURE 1A. THE BUFFERS AFTER INITIALIZATION

var	buffer[0]	buffer[1]	buffer[2]	buffer[3]	buffer[4]	buffer[5]	other[0]	other[1]
hex	38	20	6C	74	74	72	73	00

TABLE II

FIGURE 1B. THE BUFFERS AFTER OVERFLOW

Listing 2. Code before Figure 1b.

```
strcpy(buffer, "8 lttrs");
```

This failure to check the bounds on the input causes a bug. A malicious attacker could potentially use bugs like these to overwrite memory in their favor. One common method to exploit these bugs is to overwrite the return address on the programs stack. This is called a stack buffer overflow. When an exploited function returns, it jumps to the address in the return address specified by the stack. The program will begin execution somewhere other than the intended place. There are many variations of the buffer overflow attack, but this demonstrates how severe these bugs are.

The idea behind ASLR was first implemented by Memco Software in 1999 [1]. ASLR first showed up in Linux as part of PaX project in 2003 [2]. Since then, several other operating systems have implemented versions of ASLR, including Windows [3] and Mac OS X [4]. This demonstrates the importance of this security feature in modern operating systems.

III. IMPLEMENTATION

To understand how address space layout randomization works in Linux, one must first know how an address space is laid out for a particular process. There are three main parts of a processs address space: text, data, and stack. The text space holds the executable code of the process. This is a static, read-only block of memory. Next, there is the data segment. This section stores the programs variables, in particular the primitives, arrays, and strings that the programmer declared. Finally, there is the stack, which is a Last-In-First-Out (LIFO) structure that contains information about the programs execution subroutines. The text segment is traditionally at the bottom of the address space, followed by the data, with the stack located at the end of the allocated address space.

While the predictability of the locations of these parts of the address space would seem convenient, it can assist serious security attacks, such as the one detailed above. ASLR randomizes key parts of the virtual address space for a process. There are differing and weaker variations to this system. Because ASLR can introduce performance penalties on systems with smaller memory sizes, the security benefits may not always outweigh the performance loss. Different operating systems will sometimes only implement ASLR for crucial security components, for example. The level of randomization that occurs can be chosen when compiling the Linux kernel using the randomize_va_space sysctl file. There are three main options: 0 for no randomization, 1 for randomization of the mmap base, stack, and VDSO page, and 2 for complete randomization, which includes the heap. [5]

In a nonrandomized process, the location of the stack is in the same place on every execution. However, in an operating system with ASLR enabled, a processs address space is randomized at runtime, and therefore the stack will be in a different place every time the process is created. The kernel acts as a middle-man and keeps track of where the randomized pieces are located. The source for the function in the Linux kernel that is used to randomize the addresses is shown below. Its main purpose is to return a start address such that a ¡range¿ with size 'len' starting at the return value is inside in the area defined by [start, end], but is otherwise randomized. [6] The result is that the stack is now in an unknown location, significantly reducing

the effectiveness of stack buffer overflow attack.

Listing 3. Source for the range randomization function

```
unsigned long randomize_range(unsigned long start, unsigned long end, unsigned long len)
{
        unsigned long range = end - len - start;

        if (end <= start + len)
                return 0;
        return PAGE_ALIGN(get_random_int() % range + start);
}
```

IV. EFFECTIVENESS

It is useful to analyze what impacts ASLR has on any given system. While ASLR has security benefits, it can have an impact on performance as well, so there are tradeoffs. In particular, its important to analyze how effective ASLR is thwarting attacks, how it can be broken or weaken, and the overhead associated with it.

The basis for the security effectiveness of ASLR depends upon the chance of an attacker guessing the randomized locations. To decrease the likelihood of a successful attack, the search space for a guess must be increased. The search space increases when there are more bits of entropy in the random number generation. If n is the number of bits of entropy available, then 2^n represents the total number of possible addresses. This number increases very quickly as entropy increases. For a 32-bit system, the number of bits of entropy is 16-20 bits. This leaves a relatively small search space that could easily be brute forced in a small amount of time with modern computing. Realistic implementations, such as PaX, have methods of preventing brute forcing, however. With the adoption of 64-bit architectures, the number of entropy bits increased to around 48 bits, making brute force infeasible.

When implemented perfectly, ASLR is a sound defense. Unfortunately if the implementation or some supporting code has a vulnerability or bug, ASLR can be bypassed. As previously stated, brute force may be possible if the conditions are correct, however this relies on a relatively small address space. The goal to getting past ASLR would be to exploit a bug in which the code leaks the starting memory address of a randomized memory segment. One common way that the address of a pointer is leaked is through a format-string attack. [7] An attacker uses the printf function or one of its variants where a input string is placed directly in the format-string parameter. The attacker can then use the %n or %d identifiers to output a random address on the stack. Using this, the attacker can sometimes find out the location of the stack or heap and use a more traditional exploit such as a buffer overflow.

Listing 4. A dangerous printf statement

```
printf(message);
```

An information disclosure like this is the main way to bypass ASLR. There are other ways, but they rely on very rare bugs.

Performance impacts differ based on the architecture and memory segment. On Linux, the performance hit can be as significant as 26% for a 32-bit machine. [8] The main cause for this is the requirement that executables must be compiled as

position-independent in order to be compatible with ASLR. A position-independent executable is essentially a binary that is modified so that it can be run in any number of places in memory. This operation is very CPU register heavy and therefore causes significant performance losses for 32-bit architectures. On the other hand, since 64-bit architectures have more registers, ASLR has an almost negligible performance impact. It is interesting to note that the Windows implementation of ASLR suffers a negligible impact due to its method of randomization, which is at link-time rather than compile time.

V. CONCLUSION

Overall, address space layout randomization is a major benefit to any operating system. ASLR is an excellent security measure that thwarts potentially crippling attacks such as stack and buffer overflow attacks. By concealing the locations of a processs key memory segments, it minimizes the probability of an attack succeeding. The probability decreases as the search space of the randomization increases, which makes 64-bit architectures perfect for this protection. In the right scenarios, ASLR has minimal performance costs.

During this study, I explored a powerful security feature that is implemented in almost all modern operating systems, even mobile ones. I had the chance to explore some common attacks on an operating system and the vectors in which they occur. Prior to my research, I was aware of what buffer overflows were, but was not familiar with how they could be used to violate a systems intended use. This was a very interesting and useful project to engage in.

REFERENCES

[1] Y. Yarom, "Method of relocating the stack in a computer system for preventing overrate by an exploit program," Sep. 7 1999, uS Patent 5,949,973. [Online]. Available: https://www.google.com/patents/US5949973

[2] P. Team, "Documentation for the pax project." [Online]. Available: http://pax.grsecurity.net/docs/

[3] Microsoft, "Exploit mitigations in windows," WWW. [Online]. Available: http://www.microsoft.com/security/sir/strategy/default.aspx!section$_{33}$

[4] Apple, "Apple - mac os x - it's built to keep your mac safe." WWW. [Online]. Available: http://www.apple.com/osx/what-is/security.html

[5] "Documentation for /proc/sys/kernel/*," WWW. [Online]. Available: https://www.kernel.org/doc/Documentation/sysctl/kernel.txt

[6] "random.c," WWW. [Online]. Available: https://github.com/torvalds/linux/blob/master/drivers/char/random.c

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proceedings of the 12th USENIX security symposium*, vol. 120. Washington, DC., 2003.

[8] M. Payer, "Too much pie is bad for performance," 2012.