Chapter 26
Address Space Layout Randomization (ASLR)

Revision 1.2, October 2025.

Objectives

- Understand the motivation for ASLR
- Learn about its basic mechanisms
- Learn about its history
- Understand its limitations

26.1 Review of Stack Smashing and Motivation

We will start our discussion by revisiting the classic stack smashing example that we presented in Chapter 3, Thinking Like an Attacker. As you might recall, this vulnerability dates back to the Robert Morris jr. Internet worm from 1988, that brought the then-nascent Internet to its knees[1].

In Figure 1, we have a simple function written in C that declares three local variables that are allocated on the stack, `buffer`, `i`, and `j`. As is common, the return address for this function is also stored on the stack. We can see that `buffer` is an array of characters (8-bit integers) with a length of 100.

The main action of the function is to input a character string from standard input using the dangerously simple `gets` library function. `gets` is dangerous because it has no way to specify the maximum length of the input, making it easy to overflow `buffer`.

```
int foo()
{
  char buffer[100];
  int i, j;
  ...
  gets(buffer);
  ...
  return(strlen(buffer));
}
```

Figure 1: Code Vulnerable to Stack Smashing Attack

Now, suppose an attacker feeds a long input string to this function. The first 100 bytes of the input will fill up "buffer", and the next bytes overflow in adjacent memory, in this case, overwriting the return address. Figure *2*, on

[1] `https://en.wikipedia.org/wiki/Morris_worm`

the lefthand side, shows the stack frame for function `foo`, where the input has overwritten `buffer` (which is what the programmer intended) and continued past the end of `buffer` to overwrite the return address.

If the attacker carefully chooses the value to overwrite the return address, when the function returns, it will go to the location of the attacker's choice, effectively changing the behavior of the return into an attacker-controlled jump.

However, for this jump to be effective in an attack, we need to know a useful address for the jump destination.

Figure 2: Overflowing a Buffer, with Expanded View of Overflow Area

Let us examine this issue in more detail. We start with the stack that has the buffer that has been filled and overflowed into the return address.

In the righthand side of Figure *2*, we zoom in on the overflowed part of the stack. In the Morris worm, the data that he used to fill the buffer was actually code, machine language instructions that did something very close to what we see here.

This code creates a call to the UNIX `execl` system call function. The `execl` system call replaces the code in the current process with a new program. In other words, it starts a new program running. In this case, the program is `/bin/sh`, a standard shell. If this attack was successful and this code was executed, then the current process would be replaced with a command shell, allowing the attacker full access to the system, presumably as the root (administrator) user.

To design an attack such as this one, you would have to determine what address value should be overwrite the return address on the stack to point to the start of buffer.

The figure out what address should be used, an attacker could try out this program on their own system, running it with the debugger, to learn at what address the stack frame would be allocated. In our example, this address was 0x05000100, circled in yellow at the bottom.. The attacker could then use that information to choose what address to overwrite.

There is another place in this attack code where we need to know the right memory address, and that is for the call to `execl`. To generate the machine language for this `callq` instruction, we have to know the memory address where the `execl` function is located. This function, like most system call functions, resides in the standard C runtime library, libc. Again, the attacker could run program to see where libc would be placed and then use that address in the attack.

The key idea with ASLR is that we can make such attacks more difficult by making these addresses hard to guess. Note that the operating system is doing this for us with no cost or effort on our part.

With ASLR, each time that the operating system runs a program, it randomly assigns an address for each of the major components of the program. Some of these components are:

Stacks: There can be more than one stack if there are multiple threads running in this program. If the stacks are at different addresses each time that the program is run, then the example that we just saw would be difficult to use.

Shared libraries, like libc in the example: Again, if these are loaded at random addresses each time that the program is run, then it would be difficult to guess addresses for functions like “execl”.

Main program, typically in an `a.out` *or* `.exe` *file:* The main program also be located at a random address. Randomization of the address of the main program is the most recent place where ASLR has been applied.

Heap, the memory that is dynamically allocated by the program: By randomizing the location of the heap data, the attacker cannot easily guess the location of this information.

Here is a simple illustration of this idea. In Figure 3, we see an example of the layout of the virtual memory of a process. This is the traditional layout for a process and how it might look if there was no address randomization. We see the code from the executable (the `a.out` or `.exe` file) at the start of the process' virtual address space, followed immediately by the statically allocated data. Next is the heap, which is the dynamically allocated data. The boundary of the heap will grow as the process allocates more memory using something like `malloc` or `new`.

Somewhere higher up in memory, the shared libraries (`.so`'s or `.dll`'s) are allocated. Further towards the end of memory will be the main thread's stack; stacks for other threads will appear elsewhere.

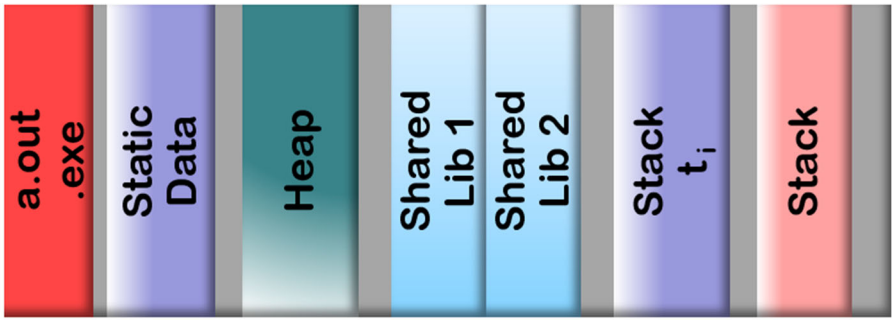

Figure 3: Typical Memory Layout of a Program without ASLR

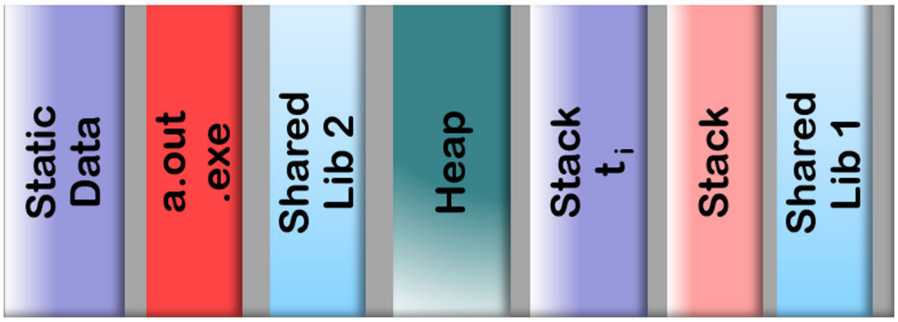

Figure 4: One Possible Memory Layout with ASLR

Figure 4 shows how that same process might look with ASLR enabled. Here we see the various sections of the process laid out randomly.

Note that most of today's computers have 64-bit addressing, which means that the 2^{64} address space is huge, so these sections of the process are really

spread out quite far. In older systems with only 32 bits of address, there is much (much) less space in which to allocate memory, creating much less randomness in address assignment.

26.2 History of ASLR

ASLR was introduced in 2001 in Linux as part of Pax operating system security project. Interestingly, the author of this first ASLR implementation has chosen to remain anonymous. Thank you, whoever you are!

ASLR spread to OpenBSD UNIX in 2003, and then makes its first appearance in Windows, in Vista, in 2007, but only for the executable and libraries, not for the stack or heap.

MacOS picks up the idea in 2007, just for system libraries, and then adopts complete coverage in 2011.

NetBSD picks it up in 2009, and Android and iOS pick it up in 2011. Solaris gets it a year later.

And, in 2014, we start seeing the idea applied to the operating system kernel, when Linux started using it there. This makes attacks on the kernel more challenging.

26.3 How Do We Measure Randomness

The goal of ASLR is to allocate each major section of a program to a random address each time that the program is run. If you do this right, it will make guessing addresses difficult to do. And part of doing this right is having enough randomness. The good news is that we can precisely describe and measure how much randomness we are using.

We often talk about this randomness as how many bits of randomness or bits of *entropy* we have. Of course, the operating system developers might tell you how many bits of randomness they used when allocating each section of a program. However, they often do not tell us, and when they do tell us, they sometimes misunderstand their own implementations and tell us inaccurately. So, alternatively, we can measure it ourselves by watching programs run.

The mathematics behind this idea are not too hard to understand. They are based on an idea introduced by Claude Shannon and Warren Weaver back in 1948 when they were developing a formal theory of digital communication[2]. These ideas are still very important today.

[2] C.E. Shannon and W. Weaver, “A Mathematical Theory of Communication”,

The goal was to try to show how unpredictable or hard-to-guess was data being sent. This was often described as how "surprising" was the data. We would like to be able to measure the randomness or entropy of a body of data. For our purposes, this data would be the starting address of a section of memory in a process.

To make such measurements, you can conduct a simple experiment: Run the program a lot of times, each time sampling the starting address value of the code or heap or stack. Then, using Shannon and Weaver's methods, you can quantify the entropy in these sampled values.

We will first take a look at the math behind Shannon and Weaver's calculation, and then we will look at how to apply it to this situation.

Assume that X is a random variable representing the address of, say, `libc.so` and the x_i are values that X can take. $H(X)$ calculates how much randomness or randomness is in X:

$$H(X) = -\sum_{i=1}^{n} P(x_i) log_2(P(x_i))$$

Basically, we're summing the product of the probability that X takes on a given value x_i times the log of that probability. In the general form of this equation, you can use any base logarithm, though in this kind of application, it is common to use the log base-2 as it produces entropy thought of in terms of the number of bits. So, security analysts often talk about a body of data having "n bits of entropy".

If we have body of data, such as a million samples of the starting address of `libc.so`, then we can use the frequency that a value appears in the data as its population probability.

To calculate the probabilities from a dataset (file), we can use the frequency of values in a file: $p_i = {}^{n_i}/_{N}$. N is the count of numbers in the dataset and n_i is the count of numbers having a specific value.

Substituting this in the equation and doing a bit of refactoring, we end up with an equation that can calculate the Shannon entropy of a body of data:

$$H = log_2 N - \frac{1}{N} \sum_{i=1}^{n} n_i \, log_2(n_i)$$

Bell System Technical Journal **27**, 3, Jul-Oct 1948, pp. 379-423. `https://web.archive.org/web/20000823215030/http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf`

26.4 ASLR in the Real World

To get an idea of how this has been implemented, we will look at a few real world numbers. In an ideal world, every operating system would document its number of bits of randomness for each section of the program for each processor type on which they run. Unfortunately, we rarely get this information. And, when we do get it, it is often inaccurate.

So, there is a great opportunity for an interesting research project to survey a variety of operating systems, using the technique that I just described.

Meanwhile, we have a few data points that we can examine in Figure 5. These numbers come from a Microsoft study a few years ago and from a study of Linux done by a few of our students at around the same time. As you can see, the number of bits varies quit a lot across systems.

	Stack	Heap	Main Code
Windows 8 (HE)	33 bits	24 bits	17 bits
Debian Linux	30 bits	29 bits	29 bits
FreeBSD	16 bits	12 bits	n/a
HardenedBSD	41 bits	21 bits	n/a

Figure 5: Measure Bits of Entropy in Popular Operating Systems

A few comments about this data. First, FreeBSD and HardenedBSD did not have PIE, position independent code for the main executable, at the time that we conducted our study, though note that all current systems now randomize the address of the main program. We see a special version of Linux, HardenedBSD, that does especially well for randomness for the aspects that they implemented.

There are some things that can affect the amount of randomness in the allocation of memory for parts of a program. First, as we mentioned earlier, older 32-bit computers provide many fewer bits to randomize than do current 64-bit computers. We could say that they provide too few bits. Second, the virtual memory architecture can affect the numbers of available bits to randomize. Since sections of a program are usually allocated on page boundaries, that gives you 10-20 bits less to randomize, depending on the page size of the system.

And practically speaking, if a program has been running for a long time, filling up its virtual memory, then future allocations have few choices where to go. There have been successful attacks in the past that have taken advantage of his behavior.

In Windows, there are some behaviors that produce a less-than-ideal amount of randomness in ASLR. The basic problem is that when a program or library is first loaded into memory, it is a assigned a virtual address. All subsequent uses of this program or library by other programs will use the same address.

As long as any program is using a library, the address of that library stays the same. So, commonly used libraries like `win32.dll` or `libc.dll` retain the same address. Such assignment of the same address gives an attacker more time and opportunity to discover a useful address.

Once way to discover such an address is to try to cause a fault in a program and trigger an exception. In the exception report, you might get the program to leak a code address in a library in which you are interested. From this address, you can calculate other code addresses relevant to your attack. We talk more about this issue in our module on exceptions.

For Windows, if we can get any program to leak an address, we might be able to use it to attack other programs.

26.5 Part of a Larger Ecosystem of Protection Techniques

ASLR is part of a bigger security picture where the systems software – the operating system and compilers – is trying to protect you from attack. The collection of techniques used by the OS and compiler include mechanisms to detect buffer overruns in both the stack and heap.

Another mechanism is to make sure that any memory that contains code is not writable by the program and memory that contains data is not executable.

Finally, there are advanced techniques, call control flow integrity checking, that will check any calculated value used in a branch or function call, to make sure that they point to a valid starting address of a function.

We will discuss these mechanisms in the following chapters.

26.6 Summary

- Understood the motivation for ASLR
- Learned about its basic mechanisms
- Learned about its history
- Understood its limitations

26.7 Exercises

1. Compute the maximum possible bits of randomness for the following two computers:
   a. A 32-bit computer with a 1KB page size. On this computer, each memory area such as code, heap, and stack start on a page boundary.
   b. A 64-bit computer with all the rest of the details the same as above.
2. Consider the following XX sections in an executable file:
   a. The code section with the machine language instructions for the program.
   b. A data section that contains a large amount of English language text.
   c. A data section that contains encrypted data.

   Place these three sections in order of increasing entropy.
3. In your favorite programming language, write a program that reads a file a byte at a time and calculates the entropy using the equation at the end of Section 26.3. Run this program on a variety of different file types and compare the results.