Address Space Layout Randomization

G. Lettieri

7 November 2023

1 Introduction

To mount a successful ROP attack, the attacker must know or guess the absolute addresses of the ROP gadgets in the memory of the victim process. One line of defense, therefore, is to make these addresses very hard to guess. This is the idea behind the Address Space Layout Randomization (ASLR for short) mitigation: load program segments at random addresses, so that an attacker cannot possibly know them without (hopefully impractical) brute-forcing.

Note that, like stack canaries and NX, this is yet another mitigation: we are not trying to eliminate bugs, just mitigate the effects of their exploitation. Like all other mitigations, ASLR has limitations and it is not the final solution to the problem.

2 Position Independent Code

In general, binary code and data cannot be loaded at random addresses without modification. For example, consider the following amd64 assembly instruction (Intel syntax):

```
        mov rax, [myvar]
```

where `myvar` is some label in the `.data` section. When the source file is assembled and linked, the `myvar` symbol will be mapped to a memory address, say 0x406020, and the instruction becomes

```
        mov rax, [0x406020]
```

which, in the final binary code, corresponds to the bytes

```
48 8b 04 25 20 60 40 00
```

We can see that the address 0x406020 is written in the instruction itself (starting from the 4th byte, in little endian). This means that `myvar` cannot be loaded at a different address, unless the instruction is patched. Patching the code, however, is undesirable, because it limits or completely prevents the sharing

of the loaded `.text` sections among the processes that are running the same program. It also slows down the loading of programs.

An alternative is to generate code that does not assume to know the address of `myvar` statically, but computes it at runtime using information provided by the loader. There are basically two models that are used, often in combination. We will explore them in turn.

2.1 Relative addresses

Continuing with our example, the linker can allocate `myvar` at a statically known offset from a "base address" chosen by the loader. The loader then communicates the base address by storing it in an agreed-upon register.

If we use this solution, the above instruction should be changed to something like

```
        mov rax, [rbx+myvar_offset]
```

where we have assumed that `rbx` is the agreed register containing the base address. This solves the problem, but at least one register, such as `rbx`, must be reserved for this purpose and cannot be used for general computation, or it must be saved and restored very often. An important special case is when `myvar` is at a known offset from the *instruction pointer*, since this register is already unavailable for general computation. In amd64 the instruction becomes

```
        mov rax [rip+myvar]
```

Note that the assembler allows us to use the label of `myvar` in this case instead of its offset: the assembler and/or the static linker will calculate the offset of `myvar` from `rip` and put it in the instruction.

The above rip-relative addressing is not available on 32 bit x86 systems. The same effect can only be achieved by first loading `eip` into a general register—a process called "thunking". Thunking can be implemented with the following two instructions:

```
        call here
here:   pop ebx
```

The only purpose of `call` is to store `eip` on top of the stack so that the next instruction can pop it into `ebx`. However, this breaks the pairing of `call`/`ret` instructions and confuses the branch predictors inside modern processors, so a slightly more expensive solution is used:

```
        call thunk
        ...

thunk:  movl ebx, [esp]
        ret
```

We call a function that immediately stores the contents of the top of the stack (i.e., the `rip` saved by the `call`) into a register and then returns.

2.2 Table of pointers

In this alternative solution, our `myvar` can be loaded at an address that is independent of all other addresses. The linker creates a table of pointers that is filled by the loader at runtime. One entry of the table contains the address of `myvar`, and the code must load this address first, whenever it wants to access `myvar`.

This solution is much more expensive than the one in Section 2.1 and it is only used when necessary. For example, if `myvar` is defined in an independent module (e.g., in a dynamic library), not even its offset is statically known. As another example, even if the offset is known, it may be too large to fit into the instruction encoding (this is a common problem on amd64). Finally, it may be the case that independently loaded modules may redefine `myvar` and all accesses should now be redirected to this new definition. This is the process of "interposing" and is a required feature of dynamic libraries. A table of pointers is used for all these cases.

Let us see how the access to `myvar` is implemented in this case. Assume that `rbx` now contains the address of the pointer table. The following instructions can be used to access `myvar`:

```
        movq rcx, [rbx+myvar_entry_offset]
        movq rax, [rcx]
```

First we to load `myvar`'s pointer from the table into a temporary register and then load `myvar` indirectly through this register. Note that we also need to know the address of the pointer table, so that we can load `rbx`. Typically, relative addresses are used for this access.

In the ELF environment, the pointer table is the GOT. It is created by the static linker at a constant offset from the code, so that `rip`-relative addressing can be used to access it. The static linker also reserves the GOT entries for any entity (such as `myvar`) that needs them, and patches the entry offsets into the binary code. Finally, the static linker prepares a set of relocations (in the `.rela.got` and `.rela.plt` sections) to let the dynamic loader fill the GOT with the final addresses.

From the above description, it should be clear that Position Independent Code does not come for free, at least on the x86/amd64 architectures. The `gcc` compiler will only generate it if explicitly requested with the `-fpic` or `-fPIC` options (both do the same thing on amd64/x86).

3 Address Space Layout Randomization

One of the goals of ASLR is that is should be implemented in a non-disruptive way: if possible, we should be able to enable it on pre-existing binaries. In all

cases, the easiest randomization to implement is simply to randomly select the base address of each segment, with no intra-segment randomization, since that would require much more disruptive changes to existing systems. For example, only the base load address of the C library is random, while all the offsets within the library are constant. Note that this compromise simplifies the implementation, but leaves the offsets known to the attacker, who can extract them from the library file.

Let us now review the segments that make up a process' virtual memory and see which one of them can probably be loaded/created at a random base address and still be expected to work. In a typical Linux (or Unix-like) environment, we have:

- segments loaded by the kernel from the executable ELF file (i.e., those containing the `.text`, `.data`, `.bss` sections and so on);
- the heap (a region of memory created by the kernel and managed by userspace libraries);
- the stack (created by the kernel);
- dynamic libraries (loaded by the dynamic loader, using memory-mapping system calls);
- other objects automatically provided by the kernel (e.g., the VDSO in Linux);

Dynamic libraries are probably the safest: they are already loaded at addresses which are unknown at compile time, and for this reason they are already compiled as PIC and should not make any assumptions about absolute addresses. The stack should also be safely created at a random address, since programs should only access it via the stack pointer. How the heap is used depends heavily on the userspace library used to manage it. Applications usually only access the heap only through a library that provides, for example, the `malloc()` and `free()` functions, or the **`new`** and **`delete`** operators. The application should not make any assumptions about the absolute values of the addresses returned by these libraries. If the heap libraries themselves make no assumption about where the heap actually is, then the heap can safely be created at a random address. The kernel objects are very system-specific. In Linux, the VDSO is implemented as a standard ELF dynamic library, so it can be safely randomized without any additional effort.

This leaves the segments of the main executable itself. These can only be randomized if the program was compiled as PIC, which is usually *not* the case. As a compromise, early implementations of ASLR skipped randomizing of the main executable, which was loaded at a known address chosen by the static linker. However, this was a serious weakness, since the main executable contains pointers to the other modules at known addresses (e.g., in the GOT itself), and thus an information leak bug in the executable can easily reveal the addresses of the dynamic libraries. From there, code reuse attacks become very much easier.

However, there are other limitations in the implementation of ASLR. Loading the segments at random addresses fragments the virtual memory, and this can cause problems for segments that can grow at runtime (this is true for the stack and the heap), or for segments that are dynamically loaded, such as the dynamic libraries. The virtual memory space may end up in a state where there is no room to load the next segment, or no room to expand the stack and/or the heap. The problem is worse in 32b systems. A number of compromises are usually implemented to overcome this problem: the available address space is *a-priori* divides into regions, and each region is allocated to a type of segment (the main executable, the heap, the libraries, the stack, the kernel objects, ...). Segments are randomly allocated only within their own region, with enough room at the end if they need to grow. Libraries are loaded in order in their region, with at most a random *offset* between them. Regions are implemented by fixing the higher part of the virtual base addresses. For example, code is always loaded starting at address 0x000055XXXXXXX000. The last 12 bits of the load address are also typically zero, to preserve intra-segment alignments and also to allow the sharing of the underlying physical pages between processes that have loaded the same segment at different (random) virtual addresses. The Xs n the example above are the remaining random bits. In 32 bit systems, there are very few random bits left, opening the door to brute-force attacks.

Another limitation, common to Unix-like systems, is that the randomization is only performed when a new program is `execve()`ed. New processes are created by `fork()`, and the semantics of this syscall require that the virtual memory of the new process be an exact copy of the virtual memory of the parent. This weakens ASLR protection for forking servers, since information obtained from one child process can be used to attack all other children.

4 Position Independent Executables

For ASLR to be fully effective, the executables themselves should be loaded at random addresses. This would require compiling all programs with `-fPIC`, but system vendors have been unwilling to do this, because of the perceived cost of PIC.

However, this cost is much higher than it needs to be. When a compilation unit (a source file that produces an object file) is compiled with `-fPIC`, all non-static references to global data will go through the GOT, and all calls to non-static functions will go through the PLT. This includes data and functions defined in compilation units that are later linked into the same executable, and even data and functions defined in the same compilation unit that contains the data access or function call. This is because PIC is intended for shared libraries, and these must allow for interposition. Using PIC for an *executable*, where interposition is not possible, is indeed overkill.

The situation changed with the introduction of Position Independent Executables (PIE for short). PIE is a new compilation option that implements position independence tailored for executables. All data and function accesses

are implemented using rip-relative addressing schemes. The GOT and the PLT are only used for data and function accesses that are truly external to the executable, such as calls to functions defined in the C library.

In the `gcc` compiler, PIE can be enabled with the `-pie` option. Conversely, if PIE is enabled by default, it can be disabled with the `-no-pie` option.

5 Implementation in Linux

ASLR in Linux is implemented in the way sketched above. The main executable (if compiled with `-pie`), the heap, the libraries, the stack and the kernel objects are loaded/created in a random location within their own region. The relative order of the regions is fixed and follows the traditional order. Some of the most significant bits of each region are fixed and the start address of each region is page-aligned (the 3 least significant hexadecimal digits are 0).

There is no randomness within a single ELF file (main executable or dynamic libraries): all the segments contained in the same ELF file are loaded "as a unit", with only the load addresses chosen at random (within the correct region). This means that offsets within the ELF file can be obtained from the binary, and a leak of any address in an ELF file will reveal all the addresses in that file. In addition, the offset between dynamic libraries is also fixed. This means that a leak of an address in one library will reveal the load addresses of *all* libraries. In fact, what is actually randomized is just the `mmap()` "base address", i.e., the first address where `mmap()` starts looking for space to fit the required mapping. Since the dynamic loader uses `mmap()` to map the dynamic libraries into the process address space, this translates into a random base address for the first library, which in turn shifts all the others. Recall that the dynamic loader itself is mapped into the process address space for the entire life of the process. The loader is typically compiled with `-pie` and loaded at a random address by the kernel.

Linux implements some intra-segment randomness for the stack segment: in the `execve()` system call, the kernel pushes a random number of zero stacklines between the argument and environment strings at the bottom of the stack and the argv/environ arrays at the top.

If the main executable is neither PIC, nor PIE, it is be loaded at the virtual addresses stored in its ELF program table. If the executable is PIC or PIE, the program table only contains offsets from the (unknown) load address.

ASLR can be enabled and disabled globally and on a per-process basis. There is a global parameter, that can be manipulated by writing to the

```
/proc/sys/kernel/randomize_va_space
```

pseudo-file. A value of 0 disables ASLR completely. A value of 1 enables it for everything except for the heap (this is needed for some legacy versions of the GNU libc that assumed the heap started right after the `.bss`). A value of 2 enables ASLR for the heap as well.

Every linux process has what is called a "personality", which is a set of constants and flags that determine how it runs. One of these flags is `ADDR_NO_RANDOMIZE`, which can be set to disable ASLR for this process (and its children). The flag can be set programmatically by using the Linux-specific `personality()` system call, or from the command line using the `setarch` utility. This utility is mainly used to select the reported "architecture" of a process (e.g., i386 or x86_64), but it can also be used to set personality flags. However, the architecture argument is mandatory. To change a flag without changing the architecture you can call it in this way:

```
setarch $(uname -m) -R some-program
```

The `$(uname -m)` command substitution returns the currents architecture, so that `setarch` leaves it at its current value. The `-R` flag disables ASLR before executing *some-program*. Since the personality is inherited through `fork()` and `execve()`, you can run a shell with `setarch` to start a session where all commands will run with ASLR disabled.

Note that you cannot disable ASLR for setuid/setgid programs this way: the kernel will reset the "dangerous" flags to their system defaults when it it finds that it needs to change the effective ids of a process.