# Address Space Layout Randomization

G. Lettieri

11 November 2022

## 1 Introduction

To mount a successful ROP attack, the attacker must know or guess the absolute addresses of the ROP gadgets in the victim process memory. 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 don't try to remove bugs, but rather mitigate the effects of their exploitation. Like all other mitigations, ASLR also has its limits 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 change. Consider, for example, 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 will become

```
        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 0x406020 address 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, since it limits or completely prevents sharing the

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

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

## 2.1 Relative addresses

Continuing with our example, the linker may 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 into an agreed 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 that contains 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 lets us use the label of `myvar` in this case, rather than the offset: the assembler and/or the static linker will compute 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` in a general register—a process called "thunking". Thunking can be implemented using the following two instructions:

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

The **call**'s only purpose is to store `eip` on top of the stack, so that the next instruction can pop it in to **ebx**. This, however, disrupts the pairing of **call**/**ret** instructions and confuses the branch predictors inside modern processors, so a slightly more costly 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 which is independent of all other addresses. The linker creates a table of pointers that will be filled by the loader at run time. 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 costly than the one in Section 2.1 and it is used only 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 issue in amd64). Finally, it may be the case that the offset or even the absolute address of `myvar` are statically known, but independently loaded modules may redefine `myvar` and all accesses should be redirected to this new definition. This is the process of "interposing" and is a required feature of dynamic libraries. For all these cases, a table of pointers is used.

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`**. For this access, relative addresses are typically used.

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 each entity (like `myvar`) that needs it, and patches the entry offsets into the binary code. Finally, it 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 in the x86/amd64 architectures. The `gcc` compiler will only generate it when explicitly requested with the `-fpic` of `-fPIC` options (both do the same thing in amd64/x86).

## 3 Address Space Layout Randomization

One of the aims 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 most easily implemented randomization consists in just randomly selecting the base address of each segment with no intra-segment randomization,

since this would require much more intrusive changes in the existing systems. So, for example, only the base load address of the C library is random, while all the offsets inside the library are constant. Note that this compromise simplifies the implementation, but leaves the offsets known to the attacker, since she can extract them from the library file.

Let us now review the segments that make up the virtual memory of a process, and see which one of them can be likely 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 libraries in userspace);
- 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 might also be safely created at a random address, since programs should only access it via the stack pointer. How the heap is used is highly dependent on the userspace library that is used to manage it. Applications usually access the heap only through a library that provides, e.g, the `malloc()` and `free()` functions, or the **`new`** and **`delete`** operators. The application should not make any assumptions on the absolute values of the addresses returned by these libraries. If the heap libraries themselves also do not make any assumption on where the heap actually is, then the heap can be safely created at a random address. The kernel objects are highly 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 had been compiled as PIC, which is *not* usually the case. As a compromise, initial implementations of ASLR skipped the randomization of the main executable, which was loaded at a known address chosen by the static linker. This was a serious weakness, however, since the main executable contains, at known addresses, pointers to the other modules (e.g., in the GOT itself), and therefore 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.

There are, however, other limitations in the implementation of ASLR. Loading the segments at random addresses fragments the virtual memory, and this may cause problems for segments that can grow at runtime (this is true for the stack and the heap) or for segments that are loaded dynamically, such as the dynamic libraries. The virtual memory space can 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 more severe in 32b systems. A set of compromises is usually implemented to overcome this problem: the available address space is *a-priori* split into regions and each region is allocated to one kind of segments (the main executable, the heap, the libraries, the stack, the kernel objects, ...). Segments are allocated randomly only within their own region, with enough room on either end when they must be allowed to grow. Libraries are loaded in-order in their region with, at most, only a random *offset* between them. Regions are implemented by fixing the higher part of the virtual base addresses. So, for example, code is always loaded starting from an 0x000055XXXXXXX000 address. 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 among processes that have loaded the same segment, but at different (random) virtual addresses. The random bits are the remaining ones, marked with Xs in the example above. In 32 bit systems there remain very few randomizable bits, thus opening the door to brute-force attacks.

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

# 4 Position Independent Executables

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

This cost, however, is much higher than it actually needs to be. When a compilation unit (one source file, producing one 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 will later be 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 when Position Independent Executables (PIE for short) where introduced. PIE is a new compilation option that implements

position-independence tailored for executables. All data and function accesses are implemented with rip-relative addressing schemes. The GOT and the PLT are only used for accesses to data and functions that are genuinely external to the executable, like 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 along the lines 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 place within their own region. The relative order of the regions is fixed and follows the traditional ordering. 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 selected at random (within the proper 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 reveals all the addresses of that file. Moreover, the offset between dynamic libraries is also fixed. This means that the leak of an address in one library reveals the load addresses of *all* libraries. Indeed, 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 translated into a random base address for the first library, that 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 random address.

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 on the top.

If the main executable is neither PIC, nor PIE, it will 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, which can be manipulated by writing into the `/proc/sys/kernel/randomize_va_space` pseudo-file. A value of 0 turns off 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 too.

Each linux process has something called a "personality", which is a set of constants and flags that determine how it is executed. 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. The architecture argument is mandatory, though. 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 return the currents architecture, so that `setarch` will leave 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 sesssion where all commands will run with ASLR disabled.

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