Non executable data

G. Lettieri

30 October 2023

1 Introduction

In code-injection attacks, the attacker's code is written into data areas in the virtual memory of the victim process. For example, in stack-based buffer overflows, the code is written to the stack, but other types of bugs can be used to inject code into other data areas, such as the heap. These types of attacks can only be successful if the processor does not distinguish between "data" and "code" sections in memory, and therefore fetches and executes the instructions that the attacker has injected into the data sections.

2 Non-Executable Data

"Non-Executable data" (or NX for short) is a mitigation that prevents data from being interpreted as code. As with all mitigations, the idea is to accept the fact that thery may be unknown bugs in the code and try to mitigate the effects of attacks that exploit those bugs. The mitigation, as usual, tries to turn a potential privilege escalation attack into a more benign denial of service.

Note that ELF files already contain flags that indicate which segments are executable. When the program is loaded, non-executable segments should be marked as such in the CPU memory management unit data structures. However, original Intel x86 processors only support this distinction through their "segmentation" feature. Unfortunately, all major operating systems essentially disable segmentation. The only protection then comes from the paging hardware, where only the following bits are assigned to each virtual page:

- The P flag, which denies all access when set to 0;
- the R/W flag, which denies write access when set to 0;
- the U/S flag, which denies userspace access when set to 0.

The paging hardware makes no distinction between memory accesses used to load/store operands and accesses used to fetch instructions. Writable data section have all three bits set to 1, so all accesses are granted.

2.1 The PaX implementation

The PaX patch for the Linux kernel is able to implement non-executable data even if the MMU has no hardware support for it. Its interest today is mostly historical, as NX support has later been added to x86 MMUs.

The oldest PaX implementation (PAGEEXEC) cleverly exploits some MMU hardware quirks. First, we need to know that x86 processors, starting with the Pentium in 1993, actually contain at least two independent TLBs:

- The Instruction TLB (ITLB), which is used when fetching instructions;
- the Data TLB (DTLB), used when accessing memory operands.

The processor implements these two TLBs to better exploit the different patterns observed when accessing data vs. code. The idea of PAGEEXEC is to create and maintain an artificial inconsistency in the two TLBs. Non-executable pages (pages marked as such in the ELF files, plus heap and stack pages) are marked as inaccessible by resetting either U/S or P. The first time that the process accesses these pages, a page fault is raised. The page fault handler of the PaX-enabled kernel determines the type of access by looking at the page-fault address and the address that was at the instruction pointer when the fault was generated (the latter address is stored on the kernel stack by the fault microcode). It can then understand whether the processor was trying to fetch an instruction or access data. If the former, it kills the process; otherwise, it temporarily sets the U/S (or P) flag and performs a load on the same address. This causes the MMU to load the translation into the DTLB. It then resets the flag and resumes execution of the process. The process retries the data access, which this second time will be granted by the DTLB. Note that if the translation is later removed from the DTLB and the process tries to access the page again, another page fault will be generated. This can cause a lot of overhead, especially for some access patterns.

PaX also implements a second method, SEGMEXEC, which uses the segmentation hardware to implement non-executable pages with lower overhead, but it must divide the available address space in half.

2.2 The AMD/Intel hardware support

AMD added support for the NX bit in page tables in its AMD64 architecture, which later became the de facto standard 64bit architecture for x86 processors and was also implemented by Intel. This is the current architecture of our PCs.

The NX bit is bit 63 in the page table entries. Fetch operations targeting pages with the NX bit set will cause a page fault.

For older 32-bit systems, the NX feature is only available if the "Physical Address Extensions" (PAE) are also used. This is an extension that allows 32 bit x86 processors to address more than 4 GiB of physical memory, while still using only 4 GiB of virtual memory for each process.

2.3 Implementation in Linux

Implementation in Linux requires cooperation between the compiler, the dynamic loader, the kernel and the C library.

The legacy compiler already marks the ELF segments as executables or not, depending on whether they contain some `.text` section or not. However, stack and heap are not mentioned in the traditional ELF files. To mark these regions as non-executable, `gcc` introduced a new ELF segment type, `GNU_STACK`, and a new section, `.note.GNU-stack`. These segments and sections use only one entry in the program table and section table of the ELF file, respectively, without any corresponding bytes in the body of the file. Their main purpose is to provide a place where the executable flag for the stack can be placed, reusing the flags fields of the program and section tables (the `GNU_STACK` segment can also be used to configure the stack size). During compilation, `gcc` determines wether the object file needs an executable stack. In the vast majority of cases, the executable stack is not needed and, therefore, `gcc` will *not* set the `x` flag in the `.note.GNU-stack` section. Currently, an executable stack is only needed when the program uses pointers to nested functions (a GNU C extension).

The linker will create the final `GNU_STACK` segment by considering the least restrictive request coming from the object files. This means that if any object asks for an executable stack, the final stack will be executable. Note that if an object file does not have any `.note.GNU-stack` section, this is taken as a request for an executable stack, for compatibility with the past. The linker's decision can be overridden by explicitly passing the `-z noexecstack` or the `-z execstack` options (the `-z` options can also be passed to `gcc` which will simply forward them to the linker). This leaves the final decision about the executable stack up to the programmer, or in the Linux world, to the Linux distribution. Legacy programs may have taken advantage of the ability to run code from data regions, so a good deal of testing must be performed before enabling `noexecstack`.

If `GNU_STACK` is present and not marked as executable, NX will be used for all the stack pages, and in addition, all memory requests that do not explicitly ask for executable memory will also have the NX bit set. This will (most likely) include the heap and all data segments from the executable and its dynamic libraries. Up to version 5.7, the Linux kernel interpreted a missing or executable `GNU_STACK` as a request for legacy behavior: the NX bit would not be used at all in these cases. In versions 5.8 and 5.9 Linux will only completely disable NX if the `GNU_STACK` segment is missing. If the segment is present and marked as executable, only the stack will have NX turned off, while other data sections will work as intended. Instead, as of version 5.10, NX is enabled by default on amd64, even for binaries that lack `GNU_STACK`.

Note that old versions of the Linux kernel also used to push code onto the process stack, to implement returns from signal handlers. The signal mechanism allows processes to attach handlers to signals. When the kernel needs to deliver a signal to a process, it changes the stored state of the process so that, the next time it is scheduled to run, it will jump to the handler. However, when

the handler terminates, the kernel should be informed, so that it can restore the previous process state so that the process can continue from the point of interruption. This work is done by the `sigreturn()` system call, but signal handlers are written like normal C functions and programmers are not expected to call `sigreturn()`. Old Linux kernels injected a call to `sigreturn()` by pushing some trampoline code onto the process' userspace stack: the code called the handler and then called `sigreturn()`. This mechanism can only work if the stack is executable, and it had to be changed in order to improve the NX support: the trampoline code is now in the C library.

A little cooperation is also needed from the userspace runtime support. The dynamic loader has to be careful to map the loaded library data sections as non-executable. However, if any of the loaded libraries asks for an executable stack, the dynamic loader must ask the kernel to turn off the NX bit for the stack pages (using the `mprotect()` system call). Finally, the dynamic memory allocator, implemented in the C library, should avoid asking for executable permission when asking the kernel for more pages. Note that this has probably always been the case already.