Protecting function pointers

G. Lettieri

20 November 2023

1 Introduction

Once an attacker has obtained an arbitrary-memory-write primitive, she usually looks for function pointers to overwrite, in order to redirect control flow to, say, a one gadget. We now examine some mitigations that have been developed to try to block the pointer-overwriting part of this type of attack.

2 Partial/Full RELRO

The first mitigation is implemented in the linker and dynamic loader tools. The idea is to mark the contents of the GOT and the ini and fini arrays ad read-only, using the protection bits available in the MMU.

Note that these arrays are created by the linker and filled by the dynamic loader, which must be able to write to them. Since the dynamic loader runs in the same process as the user program, the MMU cannot ditinguish between legitimate accesses from the loader and illegitimate accesses from the user program. The only way to implement the mitigation is to tell the dynamic loader itself to mark the pages containing the pointers as read-only after it has finished updating them. This technique is called *Relocation Read-Only*, or RELRO for short. It can be enabled in `gcc` by passing the `-z relro` option at link time. The linker will place all the relevant pointers into the same pages of memory, then create a `GNU_RELRO` program segment containing these pages. The segment's protection flags are used to require that these pages be marked read-only. However, these pages are also part of other writable segments. The `GNU_RELRO` segment is ignored at first, and only obeyed by the dynamic loader when it has finished processing all the relocations. "Obeying" the segment just means that the loader will call the `mprotect()` system call on the segment pages, asking the kernel to mark them as read-only in the MMU page tables.

This works well for the ini and fini array, which are filled before the user program starts and don't need to be updated afterwards. The lazy update of the GOT/PLT, on the other hand, creates a problem because the loader should temporarily unprotect the segment pages whenever a new function is resolved, in order to write the function pointer to the GOT entry. This is an expensive operation, since it usually involves flushing the MMU caches (TLBs), and it also

creates time windows during which the pointers are writable: attackers could exploit these windows in a multi-threaded program. The actual implementation is the result of a series of compromises: the GOT/PLT is write-protected only if *all* functions are resolved at load time, thus behaving like the ini and fini arrays. Load time resolution can be requested with a link-time option, but of course this can be very expensive, especially for a library. So RELRO comes in two forms:

- *Partial* RELRO, where the protection only applies to the ini/fini arrays and those parts of the GOT that are not involved in lazy binding (e.g., pointers to global variables instead of pointers to functions);
- *Full* RELRO, which protects everything already protected by Partial RELRO, and also asks the dynamic loader to resolve all bindings at load time and then protect the entire GOT.

In `gcc`, the `-z relro` option gives you only *Partial* RELRO. To get the full version you must also pass the `-z now` option, which tells the loader to resolve all symbols at load time. If these options are enabled by default, you can disable the first one by passing `-z norelro`, and the second one by passing `-z lazy`. Note that the `-z now` option is actually independent of RELRO, since you may want to resolve all functions at load time for other, unrelated reasons. The `-z now` option works by setting a `NOW` flag in the flags entry of the dynamic section of the executable/shared object. In addition, if RELRO is also enabled, the entire GOT is placed inside the `GNU_RELRO` segment. The `NOW` flag instructs the loader to resolve the entire PLT before starting the program, then the entire GOT is write-protected during normal processing of the `GNU_RELRO` segment.

Typically, executables are now built with Full RELRO, while dynamic libraries are only built with Partial RELRO. The idea is that a program that contains a call to a library function will most likely use it, so it is reasonable to pay the symbol resolution cost unconditionally. However, this cannot be assumed for libraries, since programs typically use only a small part of them, and resolving all library symbols at load time can end up wasting a lot of work.

Exercises

2.1. The `myheap1b` binary[1] contains a double-free bug that can be exploited to overwrite memory, but it is protected with full RELRO. How can we drop a shell from it? (Hint: `man malloc_hook`).

3 Pointer Guard

RELRO is concerned with function pointers as defined by the ELF standard, but other, equally overwritable pointers can be found in many other places. For example, the C standard library implements the `atexit()` function, which can be

[1] Available here: `https://lettieri.iet.unipi.it/hacking/heap.tar.gz`.

used to register callbacks to be called on program exit. The GNU C library implements this feature by internally allocating a list of `exit_function_list` structures. Each structure can contain 32 function pointers and several structures can be linked in a list. The first structure of the list is statically allocated in the variable `initial`, and a pointer to it can be found in the variable `__exit_funcs`. Other structures are allocated on the heap. Obviously, an attacker who has leaked the libc address, or perhaps a heap address, can potentially access these structures and overwrite their pointers.

These pointers can be updated at any time during program execution, and they are not isolated in their own pages. It is therefore impractical to use the same solution as RELRO to protect them. The GNU libc instead protects these pointers by "encrypting" them. In particular, the `PTR_MANGLE()` macro, used when saving a new pointer, encrypts a pointer by XORing it with a secret key and then rotating it. The `PTR_DEMANGLE()` macro restores the original pointer before using it. This feature is called "Pointer Guard" in the library documentation and can only be disabled by recompiling the library.

The secret key is obtained at program startup in much the same way as the secret canary. We know that the kernel stores a random number on the new process stack during `exec()`, and signals its presence and location using the `AT_RANDOM` entry of the auxiliary vector. The kernel-supplied random number is 16 bytes wide: the first 8 bytes are used to generate the canary; the other 8 bytes become the Pointer Guard secret key. The key is also stored in the same place as the canary, i.e., in the Thread Control Block, accessible via the **`fs`** segment selector register. Of course, since it is stored in several places in the process memory, leaks are always possible.

4 Removing the malloc hooks

The malloc hooks are a set of function pointers implemented by GNU malloc as part of the C library. For example, the `__malloc_hook`, if not null, is called instead of the actual `malloc`, with the same arguments, and similarly for `__free_hook`. The programmer can make these hooks point to her own functions by simply assigning a function pointer to them. The purpose is to extend the malloc functionality for debugging, accounting and so on.

Neither RELRO nor Pointer Guard can protect these hooks. RELRO is out of the question, for the same reasons as above (these pointers can be updated during program execution, and they are not segregated in memory), but Pointer Guard cannot be used either: the programmer expects to be able to directly values to these pointers directly, without going through some library function like `atexit()`. If a legacy program uses these hooks, a call to `PTR_MANGLE()` cannot be inserted by just updating the C library. Instead, the program's source, if available, must be modified and recompiled.

Note that these hooks are rarely used in normal programs. Nevertheless, every program linked with the GNU C library has them, and the GNU `malloc()` and `free()` functions will duly call them if they are not null. The safest thing

```
struct B { virtual void f() = 0; };
struct D1: B { void f() {} };
struct D2: B { void f() {} };
struct D3: B { void f() {} };

int main(int argc, char *argv[])
{
  B *b;
  if (argc > 1) b = new D1(); else b = new D2();
  b->f();
}
```

Figure 1: An example C++ program whose translation contains an indirect jump.

to do, in this case, seems to be to simply remove the hooks from the library. The hooks have been deprecated (for unrelated reasons) for many years, and were finally removed in the 2.34 glibc release. New programs will have to do without this functionality, but it will make life a little more difficult for attackers.

Exercises

4.1. The `objects1`[2] binary contains exploitable heap-related bugs, but all malloc hooks have been removed. Nevertheless, we can still drop a shell from it (hint: think of C++ vtables).

5 Control-Flow Integrity

The above mitigations attempt to protect function pointers that are known to exist in the runtime support of the C language. However, this leaves out any function pointers defined by user programs themselves, or by implementations of some features of other languages (such as the C++ vtables that can be exploited in Exercise 4.1). Control Flow Integrity refers to a class of techniques that attempt to mitigate the exploitation of *all* indirect jumps, regardless of their purpose. This includes all indirect jumps/calls through a register or memory, covering all kinds of exploitable function pointers. The definition also includes the so-called *backward* indirect jumps, as exemplified by the **`ret`** instruction in the Intel architecture, and thus CFI is also an attempt to combat ROP. The idea common to all CFI techniques is to extract a *Control Flow Graph* (CFG for short) from the program, and then check at runtime that the indirect jumps only take paths allowed by the CFG.

[2]Available here: `https://lettieri.iet.unipi.it/hacking/heap.tar.gz`.

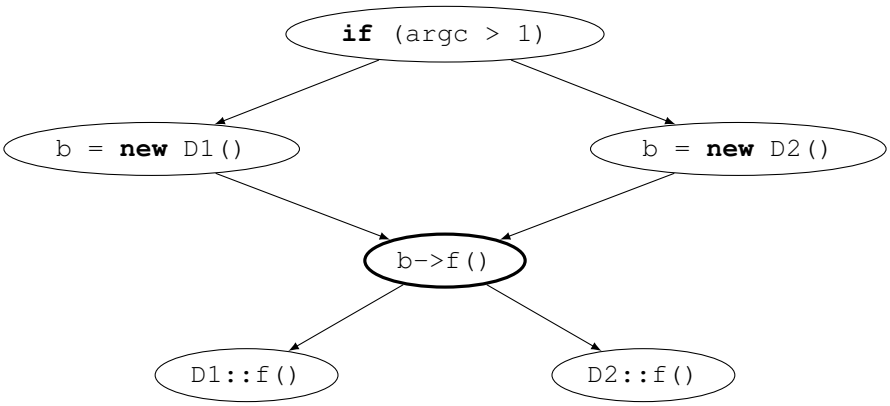

Figure 2: The CFG of the program in Figure 1.

For example, consider the C++ program in Figure 1. The corresponding CFG is in Figure 2. The thick node, containing the virtual function call, is implemented as an indirect forward jump (a **`call`** in this case). The idea is that, in all legitimate executions, this jump should always land on either the entry point of `D1::f()` or the entry point of `D2::f()`.

The set of legitimate targets of an indirect jump is called its *equivalence class*, so the equivalence class of the `b->f()` statement is {`D1::f()`, `D2::f()`}. Any jump to a target outside the equivalence class of the jump should cause the process to terminate. This can be implemented in the compiler as follows:

- the compiler assigns a unique numeric label to each equivalence class;
- it stores the label before the first instruction of each target in the equivalence class;
- each indirect jump is translated into a sequence of instructions that execute the jump only if the target contains the expected label, and abort otherwise.

The CFG is usually obtained by static analysis of the program. Typically, a static analysis will only give an over-approximation of the equivalence classes, since the exact dynamic properties of a program are either uncomputable or too expensive to compute. For example, a simple analysis that looks only at the declared types might conclude that `D3::f()` also belongs to the equivalence class of `b->f()`, even though this function can never be called in the program of Figure 1. This is unfortunate, since any additional path can be useful to an attacker, and so we would want our equivalence classes to be as precise as possible. This is especially bad for backward jumps, since the equivalence class of a **`ret`** statement at the end of a function is usually the set of all the function's callpoints. Researchers have shown that such large equivalence classes are usually sufficient for attackers to find all the ROP gadgets they need. For

this reason, CFI techniques tend to treat **`ret`** instructions specially. The most effective technique is to implement a *shadow stack*, which works like this:

- every **`call`** (either direct or indirect) pushes the return address on both the normal stack and the shadow stack;
- every **`ret`** pops the return address from both stacks and aborts if they differ;
- the shadow stack is otherwise inaccessible (but see below);

This last requirement is of course an important part of the mitigation, but needs to be relaxed a bit to allow for common constructs like exceptions and thread switching.

5.1 Intel CET

Intel has added a Control-flow Enforcement Technology (CET for short) to its processors starting from the 11th generation. CET is a type of CFI implemented in hardware. It consists of two mechanisms, that can be enabled independently of each other:

- Indirect Branch Tracking (IBT), which protects forward indirect jumps;
- a shadow stack, which protects backward indirect jumps.

With IBT enabled, all forward indirect jump instructions (i.e., indirect **`jmp`** and **`call`** instructions) cause the processor to raise an exception if the next instruction is not `endbr64`. To support this mechanism, the compiler must place an `endbr64` instruction at the beginning of each function that can be called indirectly. The `gcc` compiler will (conservatively) put it at the beginning of each function when the `-fcf-protection=branch` option is passed. The `endbr64` encoding is interpreted as **`nop`** by old processors where IBT is not implemented, or by new processors where IBT is disabled. In essence, IBT implements a single equivalence class for all forward indirect jumps. For this reason, many researchers consider it a very weak mitigation.

Much more interesting is the shadow stack mechanism, which implements the above idea in hardware. The shadow stack must be allocated by the OS kernel and marked as such in the page tables. The MMU will prevent normal write access to this page, thus protecting the shadow stack from tampering. A set of new instructions can be used to manipulate the shadow stack in special ways, to implement exceptions, thread switching and so on. Most programs can run with the shadow stack without modification.

Note that even though most Linux distributions have been shipping programs compiled with `-fcf-protection=branch` for years, the Linux kernel doesn't support the IBT part of Intel CET for userspace applications (IBT is supported for kernel code since v5.18). Starting with Windows 10 19H1, Windows has added support for the shadow stacks, as an opt-in feature for processes. Microsoft however, has decided not to support IBT, preferring its own

alternative CFI technology (Control Flow Guard). Userspace shadow stacks are supported in Linux only starting from kernel v6.6.