Offensive and Defensive CyberSecurity

–

Binary Mitigations and how to bypass them

23-24

Binary defence mechanism!

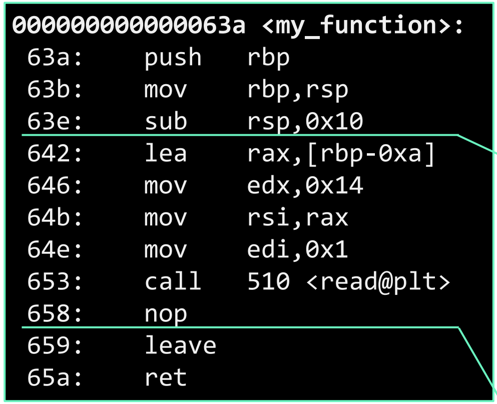
- Stack Canary
- Address Space Layout Randomization
- Position Independent Executable
- Not eXecutable bit
- RELocation Read Only

Stack Canary

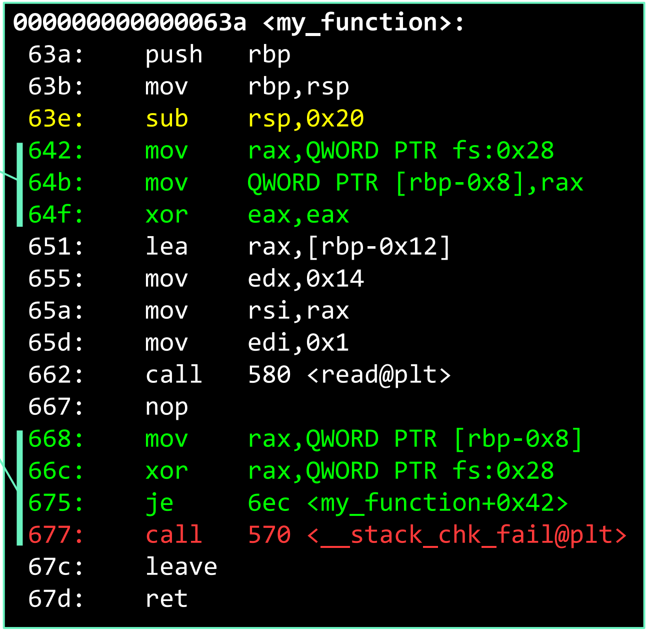
- **Compiler** insert a pseudo-random value in the stack between local variables and return address
- The value is checked at the **return**
- If the value changed something bad happened. (**Abort**)

Stack canary: code example

Without canary



```
0000000000000063a <my_function>:
 63a:     push   rbp
 63b:     mov    rbp,rsp
 63e:     sub    rsp,0x10
 642:     lea    rax,[rbp-0xa]
 646:     mov    edx,0x14
 64b:     mov    rsi,rax
 64e:     mov    edi,0x1
 653:     call   510 <read@plt>
 658:     nop
 659:     leave
 65a:     ret
```



```
0000000000000063a <my_function>:
 63a:     push   rbp
 63b:     mov    rbp,rsp
 63e:     sub    rsp,0x20
 642:     mov    rax,QWORD PTR fs:0x28
 64b:     mov    QWORD PTR [rbp-0x8],rax
 64f:     xor    eax,eax
 651:     lea    rax,[rbp-0x12]
 655:     mov    edx,0x14
 65a:     mov    rsi,rax
 65d:     mov    edi,0x1
 662:     call   580 <read@plt>
 667:     nop
 668:     mov    rax,QWORD PTR [rbp-0x8]
 66c:     xor    rax,QWORD PTR fs:0x28
 675:     je     6ec <my_function+0x42>
 677:     call   570 <__stack_chk_fail@plt>
 67c:     leave
 67d:     ret
```

More space for the canary.

Load canary value.

Check if canary is unchanged.

Abort if the canary changed.

Bypass Stack Canary

- **Not overwrite** the canary!
  - Need a vulnerability that lets you write saved EIP and not the canary.
- Overwrite the canary with the **right value**
  - Need a memory leak that lets you read the canary
- Overwrite **error handling** function.
  - <__stack_chk_fail@**plt**>

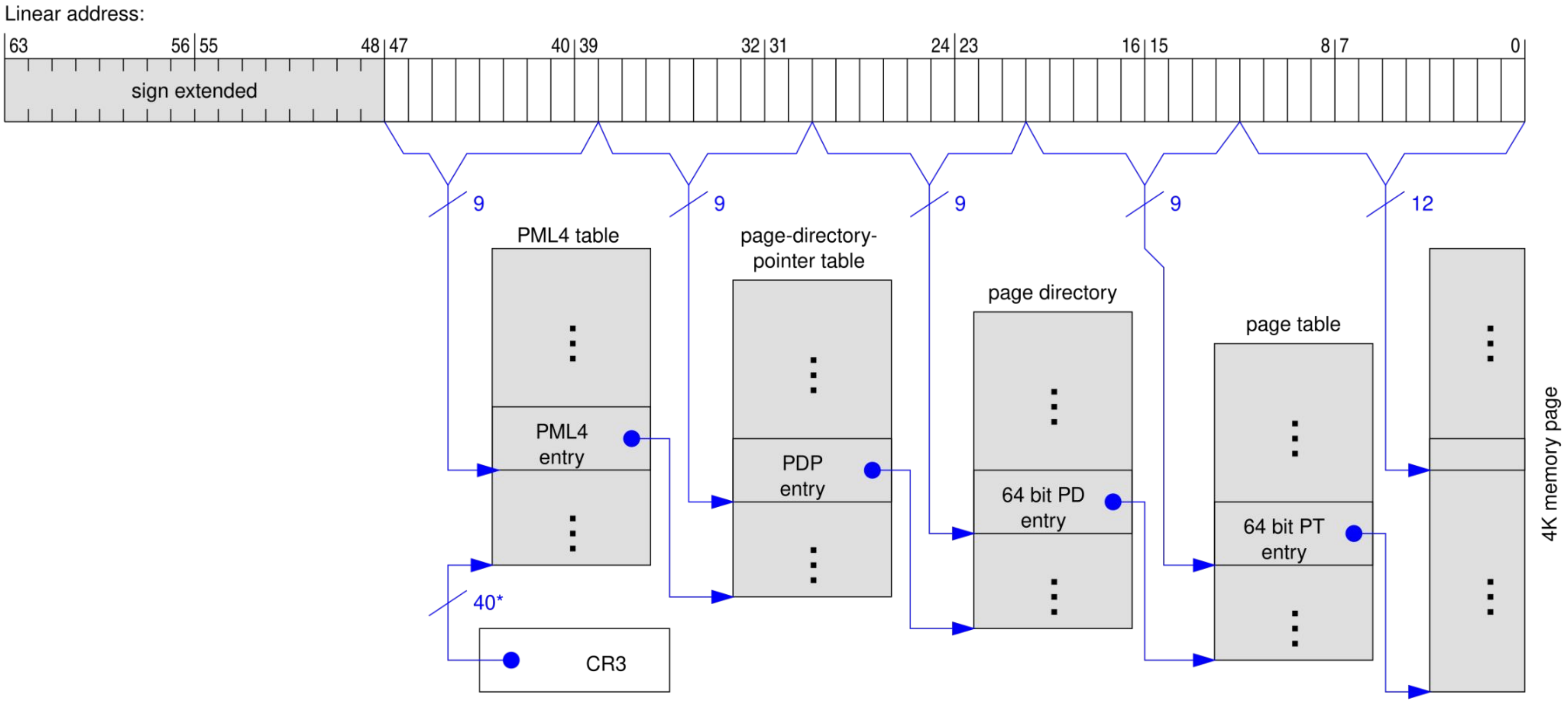
Address Space Layout Randomization

- **Randomize Base** Address of Sections
- Enabled into the **kernel**, enforced by the loader
- Stack, Libraries and Heap can always be randomized
- .text is randomized only if the binary is compiled as **Position Independent Executable** (PIE)

ASLR Attacks

- .text Section (and .got, .bss, .rodata etc.) are **not** always **randomized**.
- Randomization works per page (4k bytes):
  - **Leak** an address and you know all the addresses
  - **Contiguous pages** stay contiguous
    - Leak a .bss address and you know .text and .got
- **Probabilistic attack**: you can overwrite the first 2 bytes and get the right value with 6.25% probability (~50% 10 attempt, >95% 50 attempts)
- Side Channel Attacks: ASLR on the Line: Practical Cache Attacks on the MMU

Addresses Virtual to Phy



*) 40 bits aligned to a 4-KByte boundary

Interesting Stuff and where to Leak them

LibC:	stack (return address), got, main_arena
.**text** (when PIE):	stack, function pointers
Canary:	stack, fs segment (initialized by kernel)
Stack:	stack (stack frames)
Heap:	stack, heap (pointers)

Not eXecutable bit

- Pages have **permissions**
- If X bit is not set (or NX is set) **processor** will not execute the page
- only .text is executable (not really true)
- you do **not** have a **WX page**.

NX Bypass

- Try to execute code that's **already executable**(**Code ReUSE**)
  - **ROP / JOP**
- Write inside the **GOT**
  - you can change function pointers.
  - There always be there some function pointer in GOT
- Create a **RWX** page
  - you may try to trigger a memprotect
  - using ret to libc

Static linking

A *statically linked* executable does NOT need external symbols to work. Everything is already contained in the binary itself, which brings the advantage of not needing any other file to run (i.e. shared objects for external libraries).

Statically linking generates large files because all the needed library code has to be explicitly copied in the binary itself. It is in general not a common practice (specially for Linux).

Dynamic linking

A *dynamically linked* executable needs external functions (or symbols, in general) present in other shared objects to work.

It knows the name of such symbols, and the name of the shared object where they should be, but nothing else.

At runtime, when a symbol is needed (e.g. need to call a library function), it is the duty of the **dynamic loader** to resolve it to know where exactly it is located.

Dynamic linking: useful tools

Check if an executable is dynamically linked:

```
$ file prog
prog: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=c8a130be67a9452b6682a643cf6bb00dc35113a3, not stripped
```

Check which libraries are needed by a dynamically linked executable:

```
$ ldd prog
    linux-vdso.so.1 (0x00007ffed13b3000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f074289f000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f0742e40000)
```

Check external symbols needed by a dynamically linked executable:

```
$ objdump -TC
# Long output... check by yourself!
```

Runtime symbol resolution

In order to be able to resolve symbols at runtime (i.e find their real address), ELF programs have two auxiliary tables:

Global Offset Table (GOT): one entry per symbol, holding its real address, or a default value if not yet resolved. Acts like a cache.

Procedure Linkage Table (PLT): one entry per symbol, holding a small set of instructions to execute to correctly load and call the symbol. Calls to external library functions are made through their PLT entries.

Runtime symbol resolution: lazy loading

1. Call the PLT entry:

```
(main)  call  printf@plt
```

2. Load address from GOT and jump to it:

```
(PLT entry)   jmp   [printf@got]
```

2a. If the symbol was *already resolved* by the loader, then the GOT entry already contains the address of the function, so *this executes the function*. **Done!**

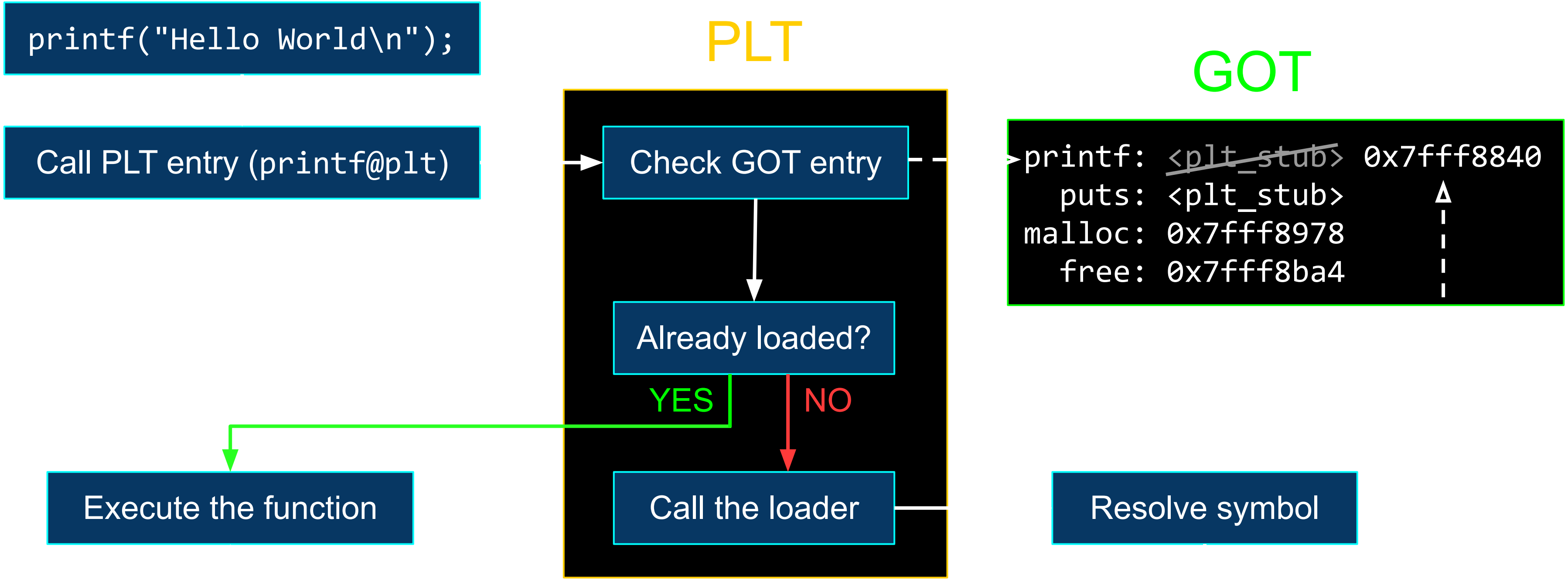
2b. **Otherwise** the GOT contains a PLT stub address and this just jumps to it.

3. The PLT stub loads the symbol offset and jumps to the loader to resolve it:

```
(PLT stub)   push  <offset>
(PLT stub)   jmp   <loader>
```

4. The loader resolves the symbol, writes the correct address in GOT, and automatically jumps to it to execute the function. **Done!**

Runtime symbol resolution: lazy loading



printf("Hello World\n");

Call PLT entry (printf@plt)

PLT

Check GOT entry

Already loaded?

YES

NO

Call the loader

Execute the function

GOT

printf: ~~<plt_stub>~~ 0x7fff8840
puts: <plt_stub>
malloc: 0x7fff8978
free: 0x7fff8ba4

Resolve symbol

Exploiting the GOT

The GOT *needs to be writable* in order for the loader to fill it with addresses of resolved symbols.

This however could be dangerous: when calling an external function the program will jump to whatever address is stored in the GOT. If we somehow manage to overwrite a GOT entry with an arbitrary address, we can make the program jump where we want when that library function is called.

GOT Protection - RELocation Read Only

- **Partial RELRO:** Do not put .got after .bss and avoid overflow from bss
- **Full RELRO**: Do not be lazy
  - Load everything at the beginning, make .got Read Only

RELRO Bypass

- Partial RELRO:
    - Any **arbitrary write**.
- Full RELRO:
    - You can still overwrite the saved EIP
        - Ret to LIBC
        - **ROP**
    - Leakless (How the ELF Ruined Christmas)
- .got it is always a good point to search for a **leak of libc** (or any other lib)

RELRO: RELocation Read Only

Security feature **provided by the compiler and the loader**.
Two types of RELRO:

Partial RELRO: special ELF sections are reordered before `.data` and `.bss` to prevent a rewrite via overflow, and some are also marked read only (`.dynamic`, `.dtors`, ...), but **NOT `.got.plt`** (the one we care about).

Full RELRO: all of the above, plus **all the GOT is read only**. The dynamic loader resolves all symbols *before* starting the program, filling the GOT and remapping it as read only. Program startup is slower for obvious reasons.

Latest `gcc/clang` versions compile using partial RELRO by default.

RELRO: bypass?

Partial RELRO is not a problem. The GOT (the `.got.plt` section) is still readable and writable, so basically for what we care: **partial RELRO ≈ no RELRO**.

Full RELRO *is* a problem. We cannot write any of the GOT entries anymore. The only thing we can use the GOT for is to leak library addresses and use them to calculate the base address of the library in memory.

RELRO: bypass?

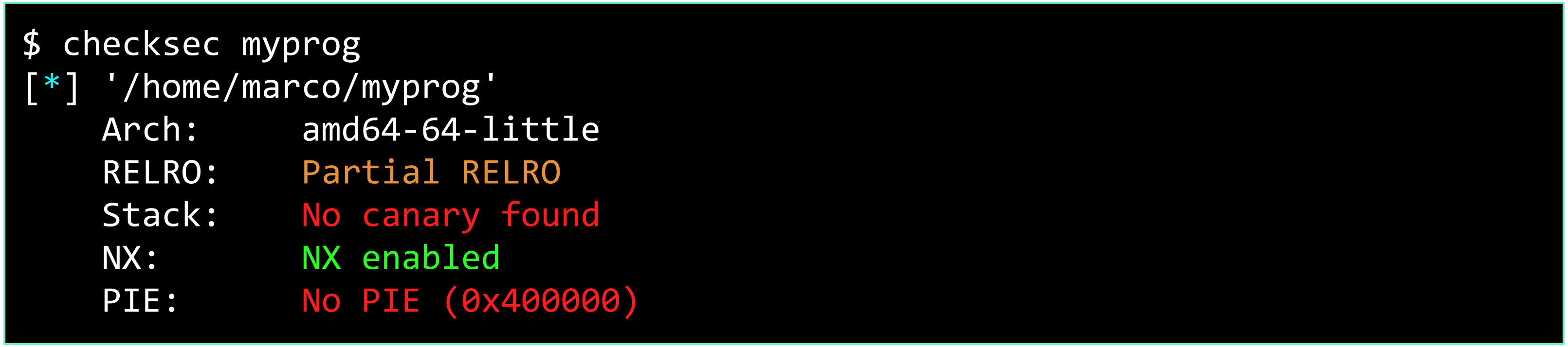
That's not 100% true though... some complex exploitation technique exists which potentially allows to completely bypass both ASLR and full RELRO *without any leak*, tricking the dynamic loader into resolving arbitrary symbols for us.

This is not in the scope of the lesson, but if you're interested go ahead and take a look at this beautiful talk & paper:

How the ELF ruined Christmas — Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna

One tool to rule them all

The `pwntools` toolset provides a quick and easy to use tool called `checkseck` to check for all these security measures in one or more ELF programs at once:



```
$ checksec myprog
[*] '/home/marco/myprog'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```