CYBER CHALLENGE
CyberChallenge.IT

The Roman Xpl0it

A gentle introduction to exploitation

Part 1 : Buffer overflow 101

by anticlockwise & malweisse

About us

Andrea Fioraldi [@andreafioraldi]
Pietro Borrello [@pietroborrello]

Cyberchallengers 2017

We founded TheRomanXpl0it and started pwning right after

Then mHACKeroni with Milan, Venice and Padua guys, to participate to DEFCON CTF

Already qualified for next DEFCON ;)

What can possibly go wrong?

what do you see?

`0x58 0x33 0x58 0x30`

What can possibly go wrong?

what do you see?

`0x58 0x33 0x58 0x30`

↓ ascii?

```
>>> '58335830'.decode('hex')
'X3X0'
```

CYBER CHALLENGE
CyberChallenge.IT

What can possibly go wrong?

The Roman Xpl0it

what do you see?

`0x58 0x33 0x58 0x30`

↓ an integer?

`0x30583358 = 811086680`

What can possibly go wrong?

what do you see?

`0x58 0x33 0x58 0x30`

an address?

`0x30583358  <main>`

What can possibly go wrong?

what do you see?

0x58 0x33 0x58 0x30

code?

```
0: 58          pop rax
1: 33 58 30    xor ebx,DWORD PTR [rax+0x30]
```

What can possibly go wrong?

what do you see?

`0x58 0x33 0x58 0x30`

adrian?

What can possibly go wrong?

what do you see?

`0x58 0x33 0x58 0x30`

↓ they are just bytes!

`0x58 0x33 0x58 0x30`

It is just the context that gives the meaning to the bytes

What can possibly go wrong?

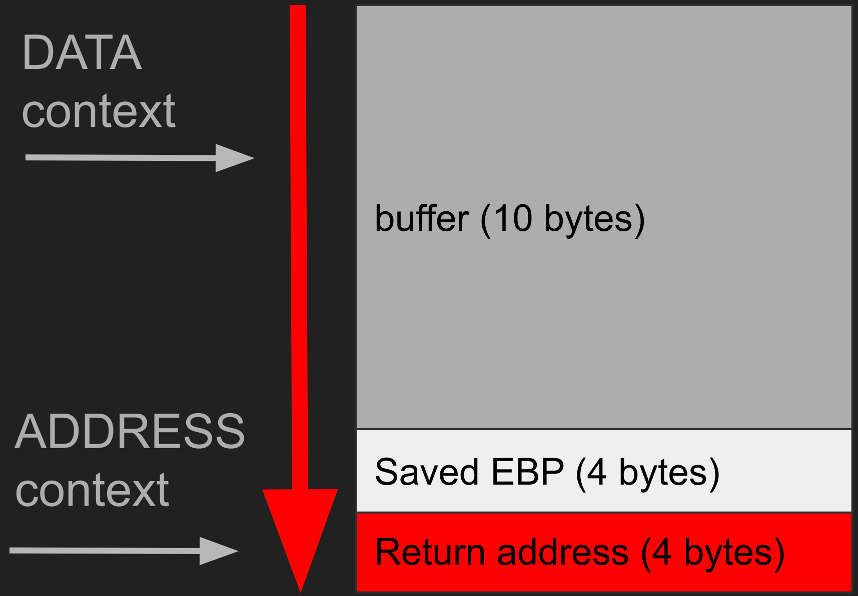
0x58 0x33 0x58 0x30

Any time you mess with the context you open the window to exploitation

A vulnerability allows us to make the CPU manage bytes in the wrong context

Buffer overflow

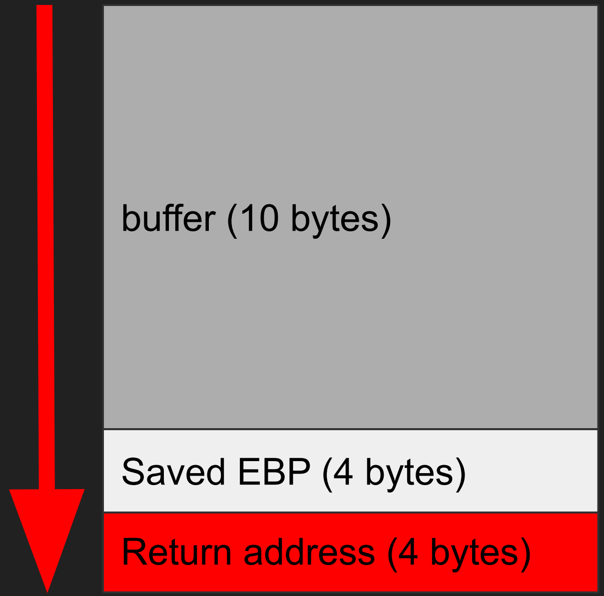
```
void foo() {
    char buffer[10];
    scanf("%s", buffer); //DOH
}
```



Buffer overflow

Let's override the return address content!

After foo() the execution will be resumed from the return address.



Buffer overflow

For example:

“aaaaaaaaaa” + “aaaa” + addr_to_return

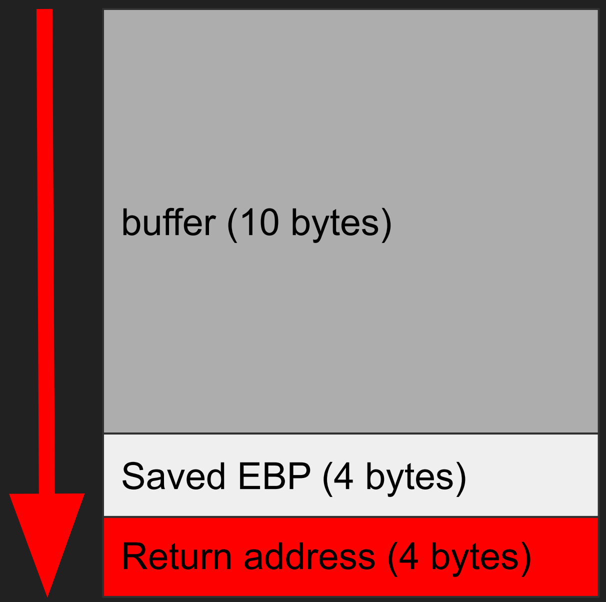
padding (10)   ebp

Buffer overflow

```
void foo() {
    char buffer[10];
    scanf("%s", buffer); //DOH
}
```

⇒ return anywhere you want!

but… Where?



Data or Code? Shellcode!

- Mix up code and data is the worst possible vulnerability!
- A shellcode is the payload you manage to execute

1. Write the shellcode in memory (as DATA context)
    - es: read(0, buffer, 0x1000)
2. Redirect the execution to the shellcode (as CODE context)
    - es: overwrite the return address with a vulnerability
3. WIN

- Assuming you can write in a RWX page... (Who said “please switch to ROP”?)

Shellcode?

- Assume:
  1. You can write to RWX memory
  2. You can override the return address to jump to RWX memory

- What do you write?

Shellcode?

- Assume:
  1. You can write to RWX memory
  2. You can override the return address to jump to RWX memory

- What do you write?

⇒ Usually the goal is to spawn a shell in the victim server

`system(“/bin/sh”)` will do the job!

But, how do you call it, if is not present in the binary?

⇒ Implement it with your shellcode!

Spawn a Shell!

CYBER CHALLENGE CyberChallenge.IT

The Roman Xpl0it

Let's dissect system!

Browse the source code of glibc/sysdeps/posix/system.c
do_system

```c
#ifdef FORK
  pid = FORK ();
#else
  pid = __fork ();
#endif
  if (pid == (pid_t) 0)
    {
      /* Child side.  */
      const char *new_argv[4];
      new_argv[0] = SHELL_NAME;
      new_argv[1] = "-c";
      new_argv[2] = line;
      new_argv[3] = NULL;

      /* Restore the signals.  */
      (void) __sigaction (SIGINT, &intr, (struct sigaction *) NULL);
      (void) __sigaction (SIGQUIT, &quit, (struct sigaction *) NULL);
      (void) __sigprocmask (SIG_SETMASK, &omask, (sigset_t *) NULL);
      INIT_LOCK ();

      /* Exec the shell.  */
      (void) __execve (SHELL_PATH, (char *const *) new_argv, __environ);
      _exit (127);
    }
```

Spawn a Shell!

`__execve` is the stub for the execve system call: the system call that transforms the calling process into a new process to execute.

It sets for `32 bits`:

```
eax -> 0x0b
ebx -> address of "/bin/sh"
ecx -> NULL
edx -> NULL
```

then executes:

```
int 0x80
```

or for `64 bits`:

```
rax -> 0x3b
rdi -> address of "/bin/sh"
rsi -> NULL
rdx -> NULL
```

then executes:

```
syscall
```

Spawn a Shell!

Therefore our shellcode will be something like:

```
push 0x68732f    <-|
push 0x6e69622f  <-| Write /bin/sh into the stack
mov     ebx, esp
mov     ecx, 0x0
mov     edx, 0x0
mov     eax ,0xb
int     0x80
```

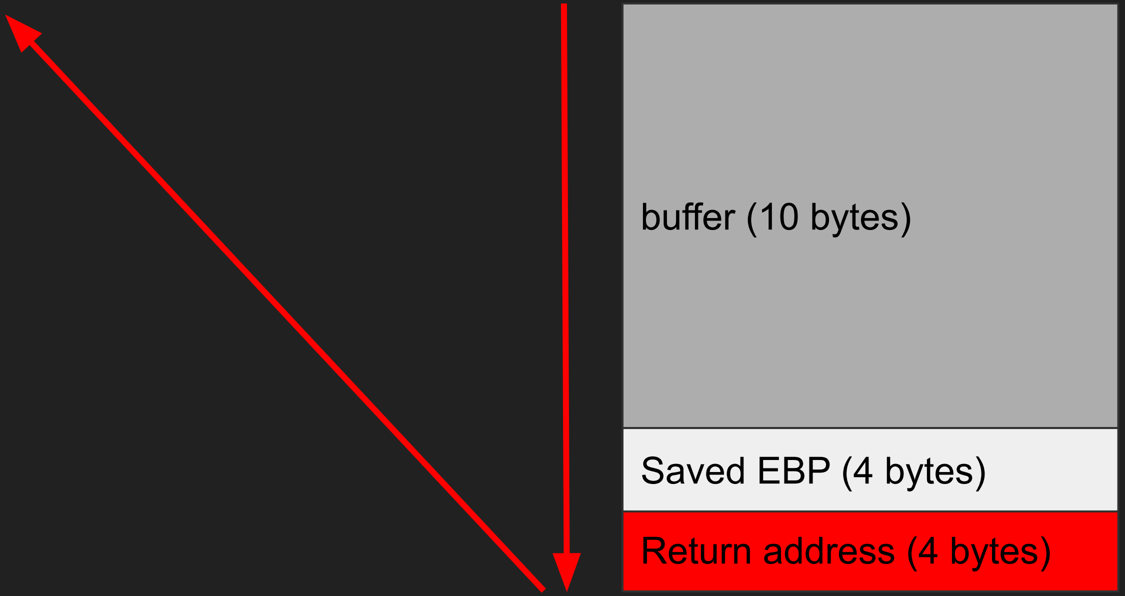
CYBER CHALLENGE
CyberChallenge.IT

Spawn a Shell!

The Roman Xpl0it

Write the shellcode in RWX memory and jump to it!

```
push 0x68732f
push 0x6e69622f
mov     ebx, esp
mov     ecx, 0x0
mov     edx, 0x0
mov     eax ,0xb
int     0x80
```



Are shellcodes dead?

- All modern OS enforce NX protection:
  - No region in the binary can be writable and executable at the same time when compiling -> no RWX regions!

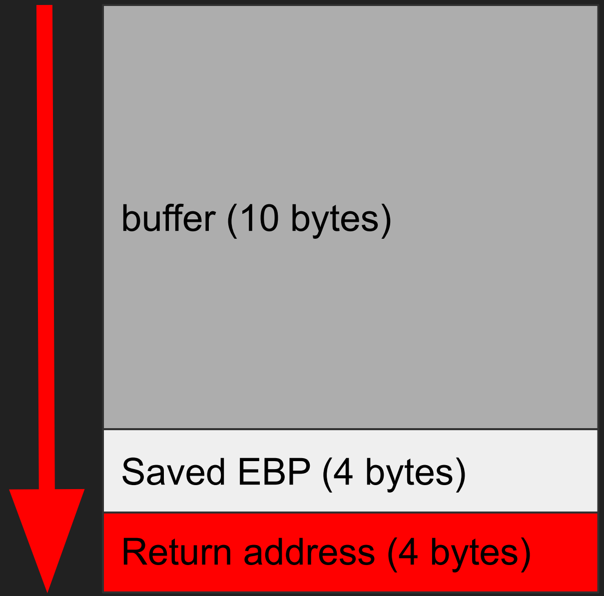
- But any self modifying program will need writable and executable pages!
  - Browser JIT compilers
  - Browser WASM compilers
  - QEMU (sometimes not enforces neither ASLR nor NX)

⇒ Still useful to know how shellcodes work

Buffer overflow returned

```
void foo() {
    char buffer[10];
    scanf(“%s”, buffer); //DOH
}
```

⇒ Now we have NX so we cannot jump to shellcode!

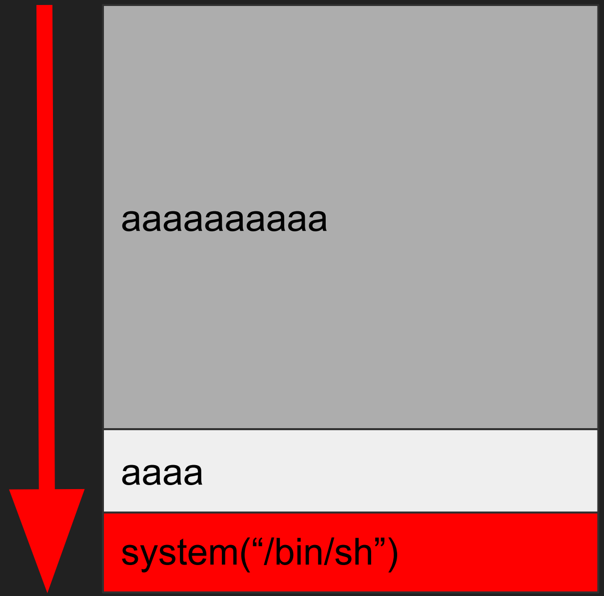


Buffer overflow

We can still put anything we want in addr_to_return!

Calling system(“/bin/sh”) will execute a shell in the current context

But what if we don’t have system available to call?



Spawn a Shell!

We need to execute something similar to:

```
mov     ebx, /bin/sh_string_address
mov     ecx, 0x0
mov     edx, 0x0
mov     eax ,0xb
int     0x80
```

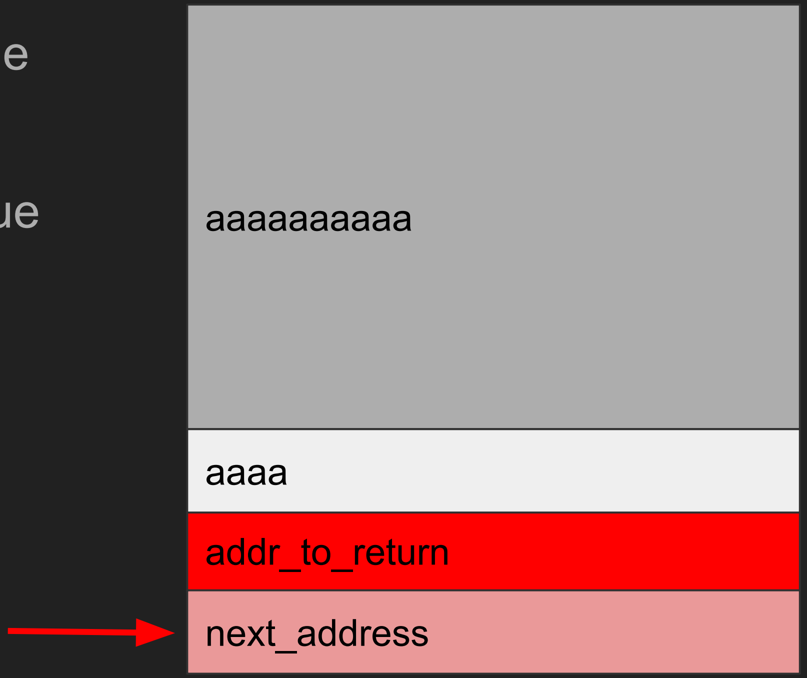
But how?

… Let’s notice something

Composing a chain

If addr_to_return point to some code that ends in “ret”

the execution will then continue from the address after



...and we can iterate again and again

Composing a chain

So why we don’t insert more than one address to return to?

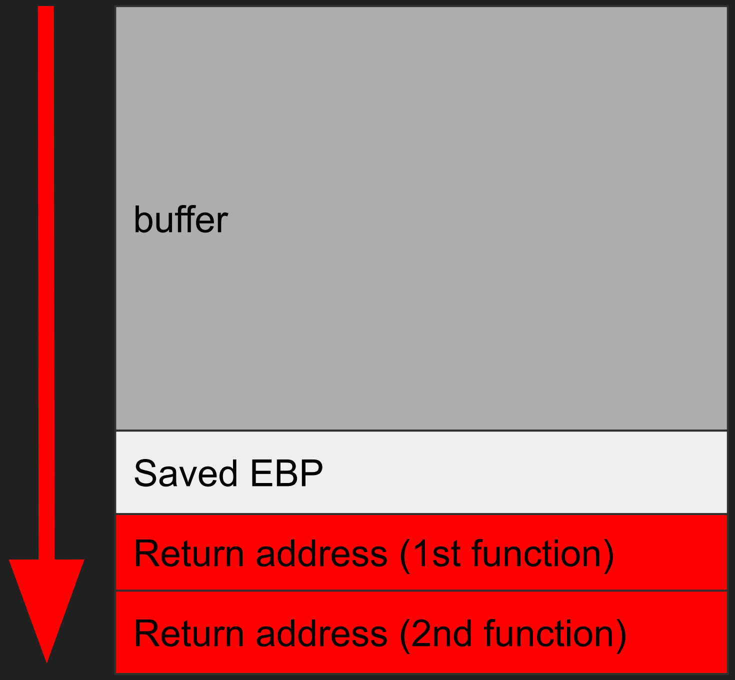
For example:

“aaaaaaaaaa” + “aaaa” + fn1_addr + fn2_addr

padding (10) ebp

Composing a chain

```
fn1:
    mov eax, 0xabadcafe
    ret
fn2:
    mov ecx, eax
    ret
```



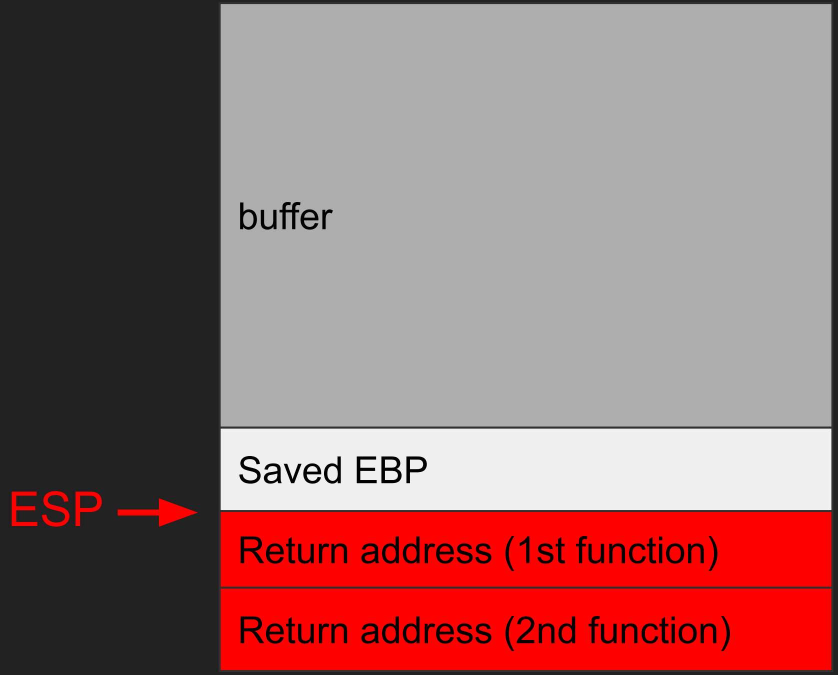
Composing a chain

```
foo:

    ...

    ret   ← EIP
```

EAX: ?
ECX: ?



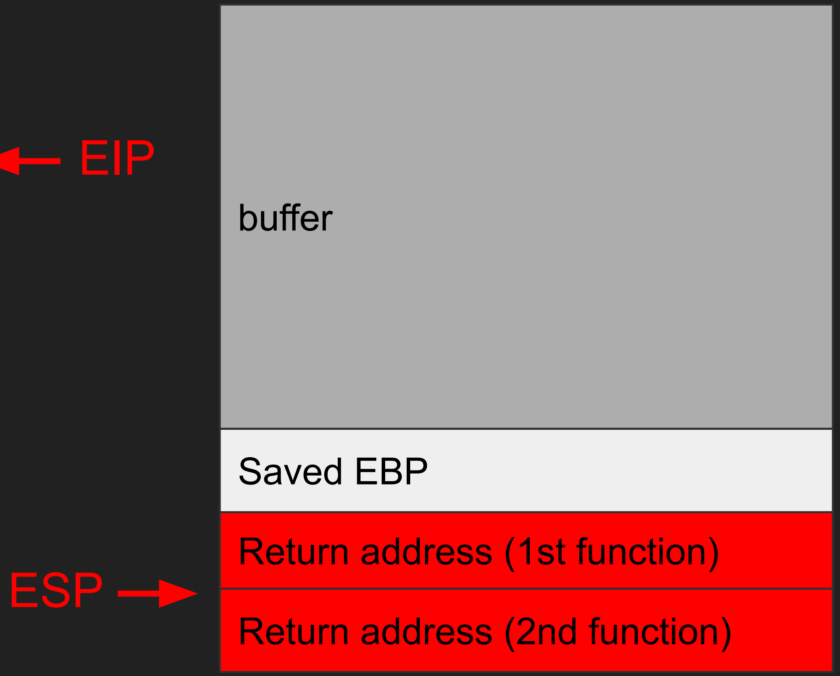
Composing a chain

fn1:

    mov eax, 0xabadcafe   ← EIP

    ret

EAX: ?
ECX: ?



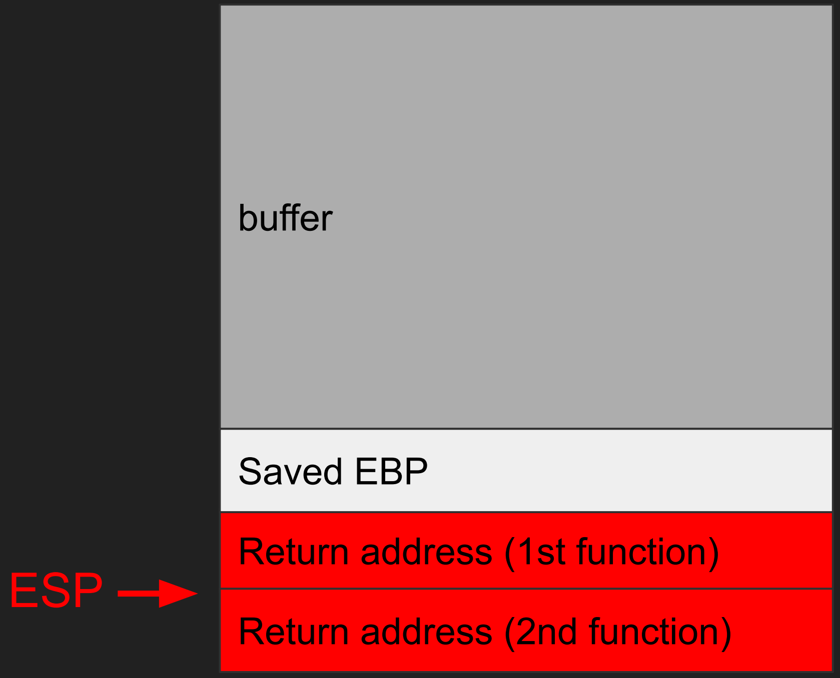
Composing a chain

```
fn1:

    mov eax, 0xabadcafe

    ret    ← EIP
```

EAX: 0xabadcafe
ECX: ?



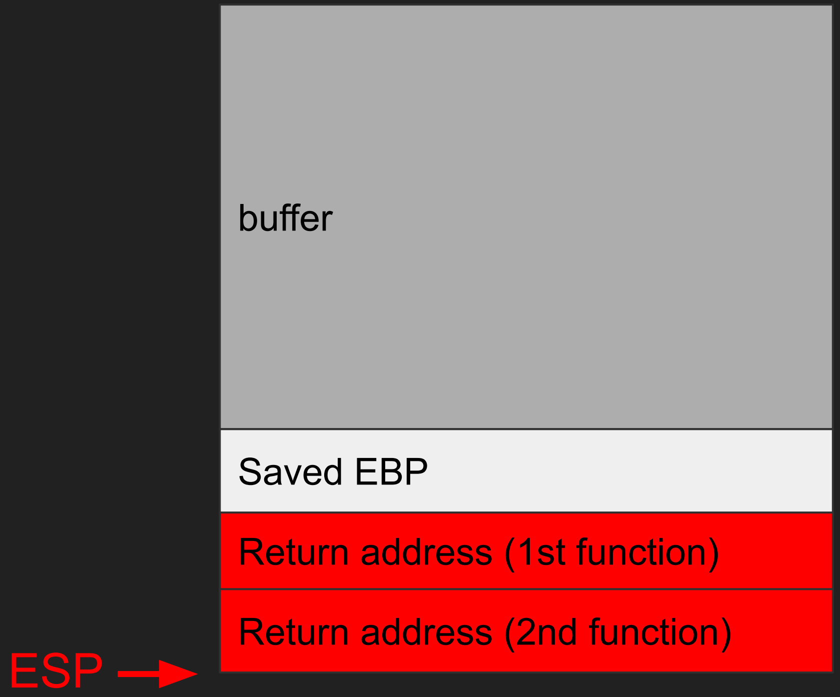
Composing a chain

fn2:

```
    mov ecx, eax    ← EIP

    ret
```

EAX: 0xabadcafe
ECX: ?



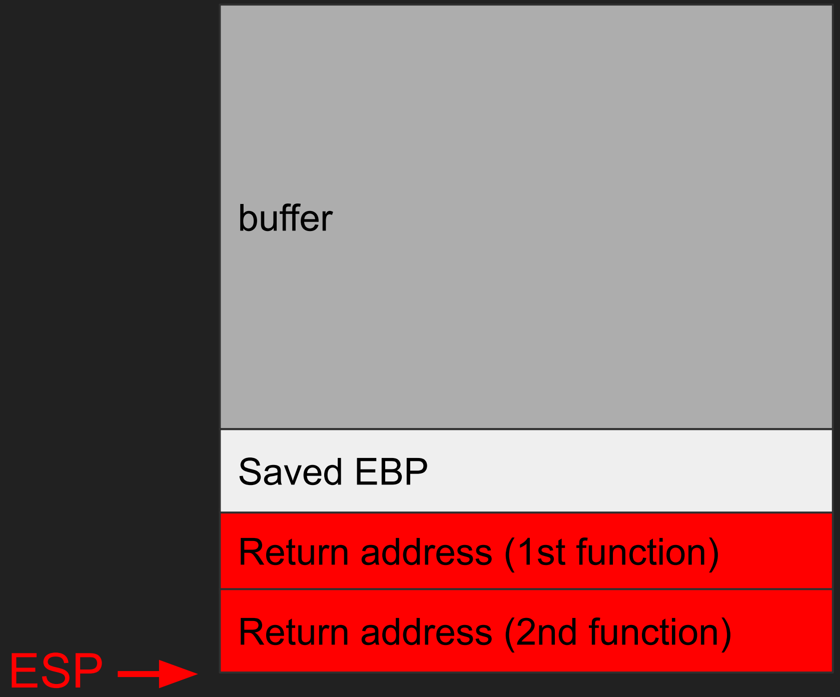
Composing a chain

fn2:

```
    mov ecx, eax

    ret   <- EIP
```

EAX: 0xabadcafe
ECX: 0xabadcafe



Composing a chain

We can insert addresses to return to execute, as long as we want!

BUT: every piece of code we execute must end with a ret instruction, to continue the chain

Return address (1st function)
Return address (2nd function)
Return address (3rd function)
Return address (4th function)
Return address (5th function)

Composing a chain

For example, let’s load ebx with the address of a /bin/sh string

mov ebx, writable_address; ret;
pop eax; ret;
“/bin”
mov [ebx], eax; ret;
pop eax; ret;
“/sh”
mov [ebx+4], eax; ret;

Now ebx points to the /bin/sh string

Gadgets

x86 instructions are not aligned:

mov ah, 90; ret ---> b4 5a c3   pop edx; ret ---> 5a c3

Every sequence of bytes endings with ret (0xc3) can be used to build a chain.

Find these sequences with tools like ROPGadget or ropper.

```
$ ROPGadget --binary <bin_to_pwn>
or
$ ropper -f <bin_to_pwn>
```

Existing Countermeasures

- ASLR
- Stack Canary
- NX
- PIE

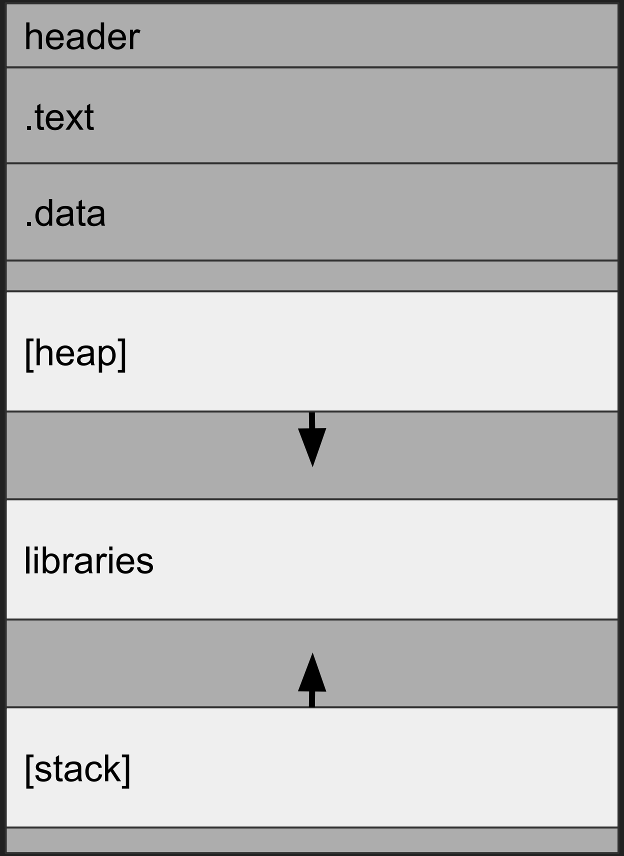
ASLR

Address Space Layout Randomization

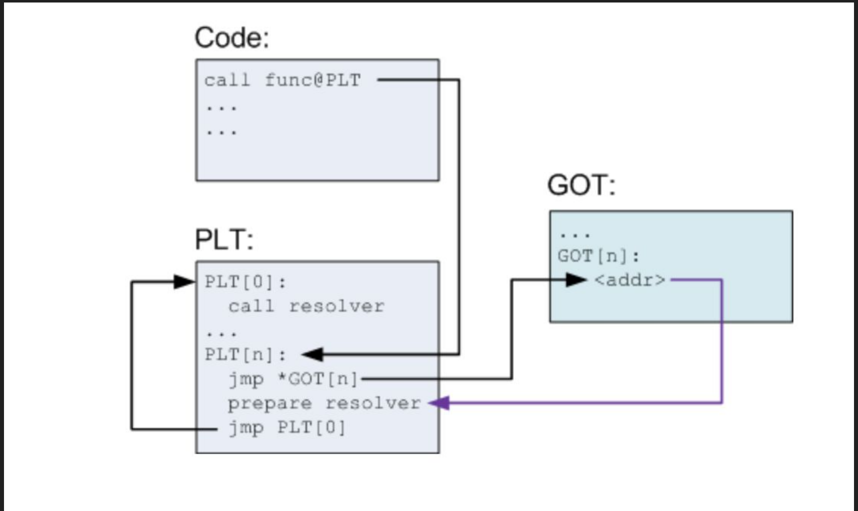
is a protection technique that randomizes

the base address of stack, heap and

library code



The GOT and the PLT

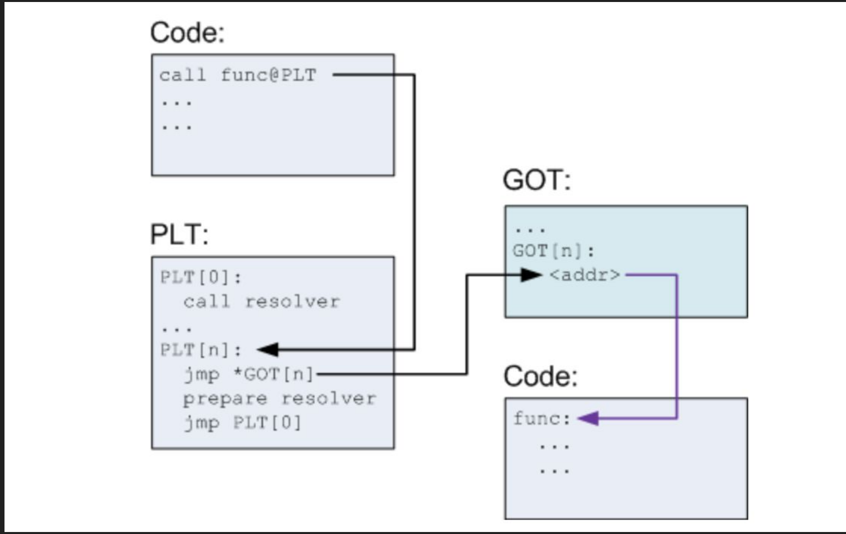


when called for the first time, func@PLT will jump to the corresponding

got_entry:<func_loader_stub>

that will call the loader for the function

The GOT and the PLT



then the loader will replace the entry with the right address, consulting at runtime the symbol map of the library, to resolve the randomized address

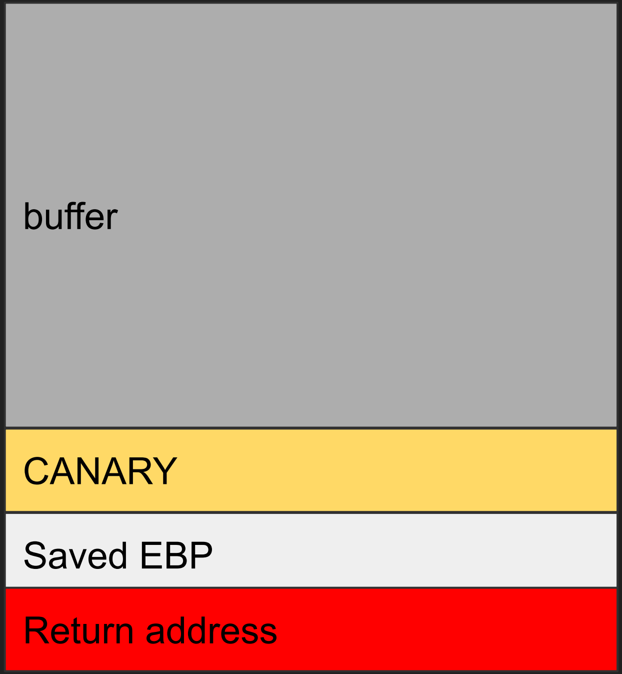
got_entry: <address of func in libc>

Stack Canary

A random value is automatically inserted before any return address and checked that remained the same before doing any ret instruction.

To modify the return address need to overwrite the canary.

The program is crashed if the canary is modified!

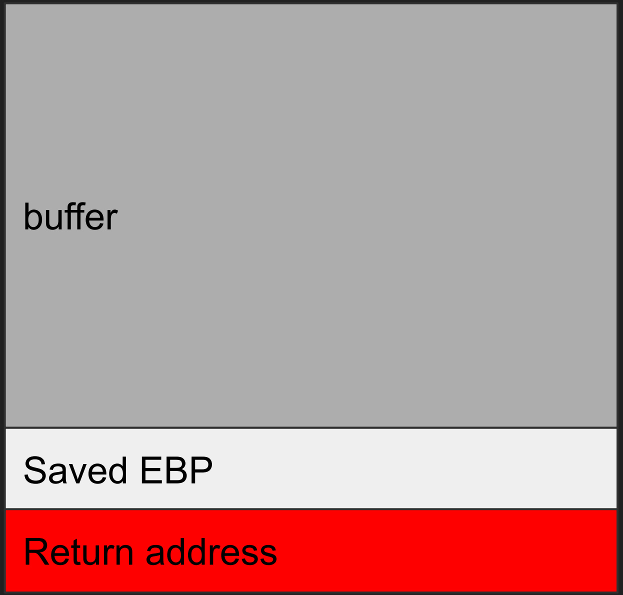


NX

No region in the binary can be writable and executable at the same time when compiling.

ROP is not affected by this protection in any way ;)

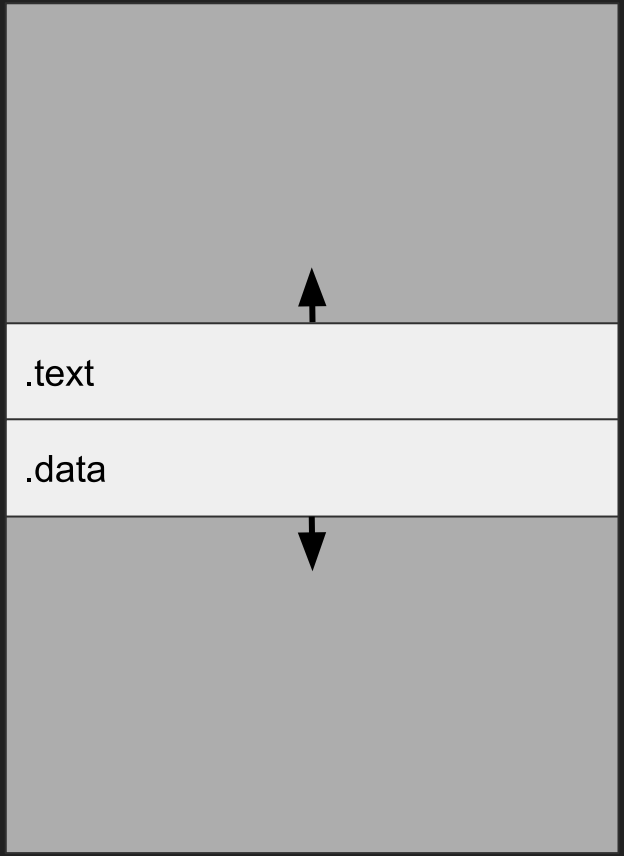
Only against classical shellcodes



PIE

Position Independent Executables (PIE) are binaries made entirely by position independent code. The base of the whole executable is loaded at random memory position.

ROP mitigation, since need to discover the base address before the exploit can begin.



Other Protections

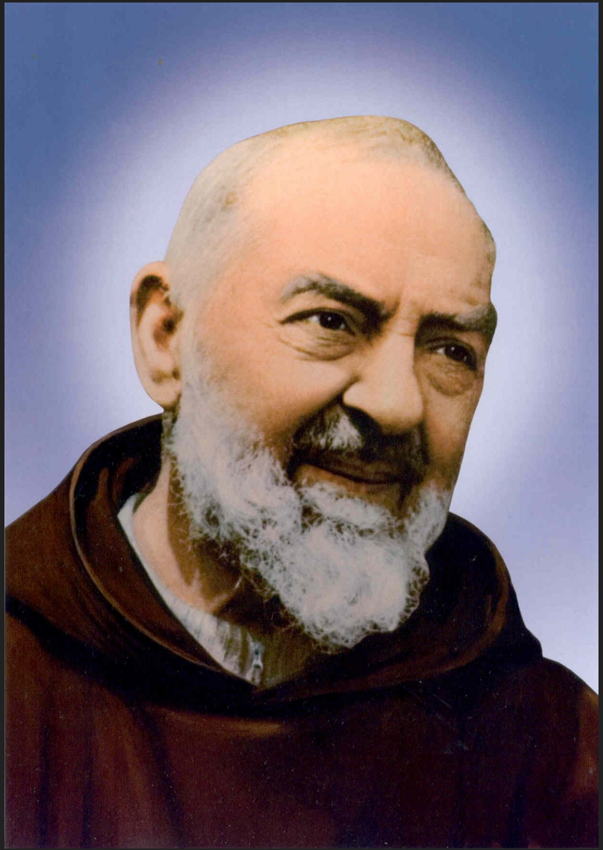
- Partial RelRO
- Full RelRO

  See https://mudongliang.github.io/2016/07/11/relro-a-not-so-well-known-memory-corruption-mitigation-technique.html

- Control flow integrity

  See https://nebelwelt.net/blog/20160913-ControlFlowIntegrity.html

Ultimate Protection



Now it’s your turn!

pwncc19.herokuapp.com