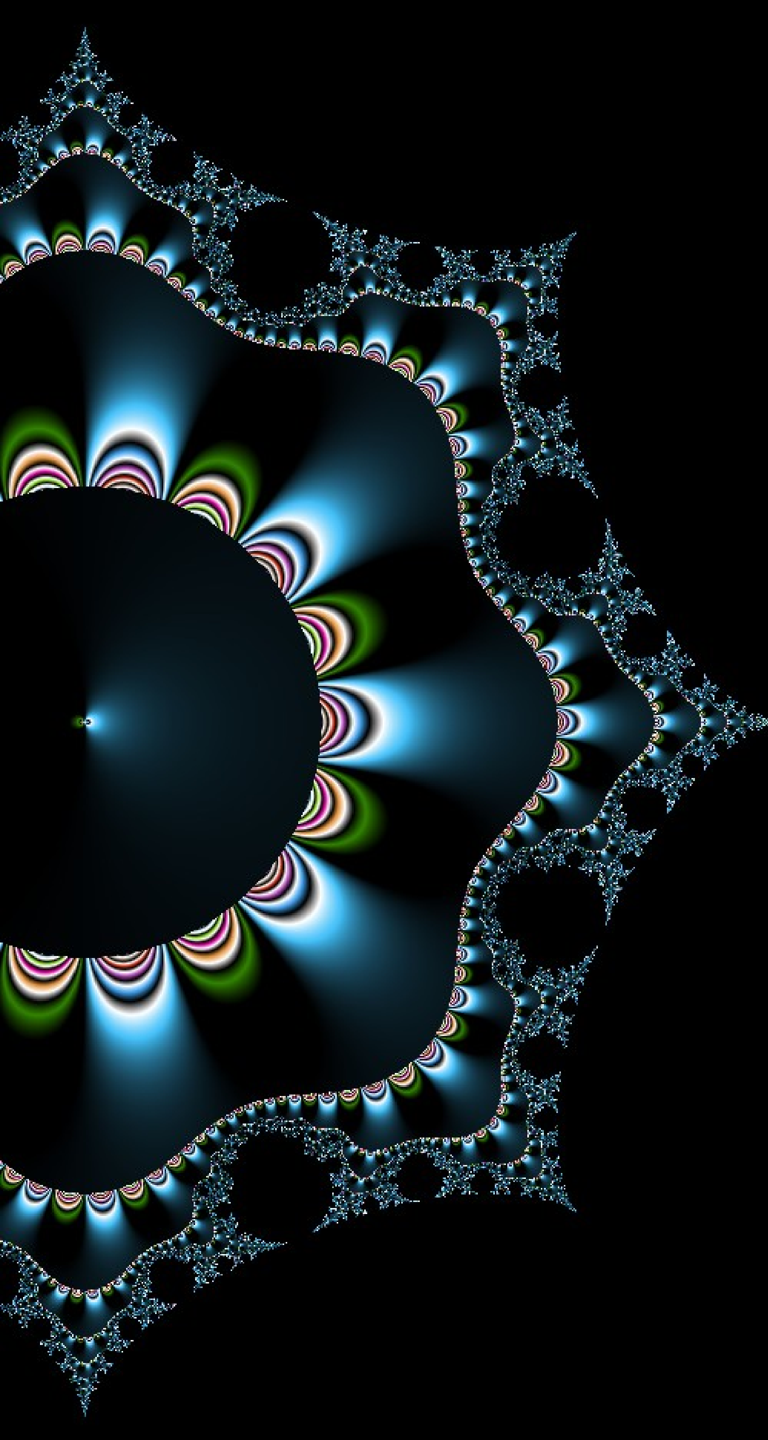


Stack based buffer Overflow

Corinne HENIN

www.arsouyes.org

What's the subject ?

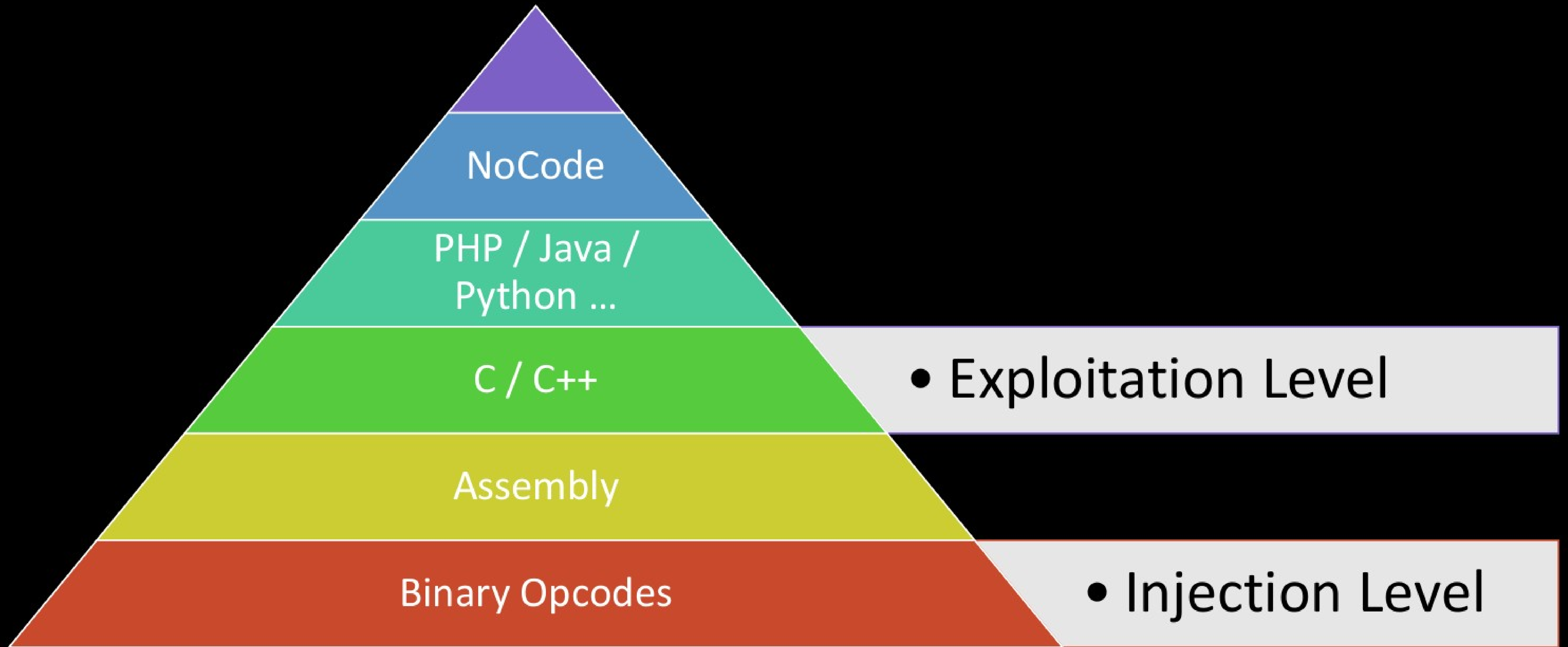
overwrite datas

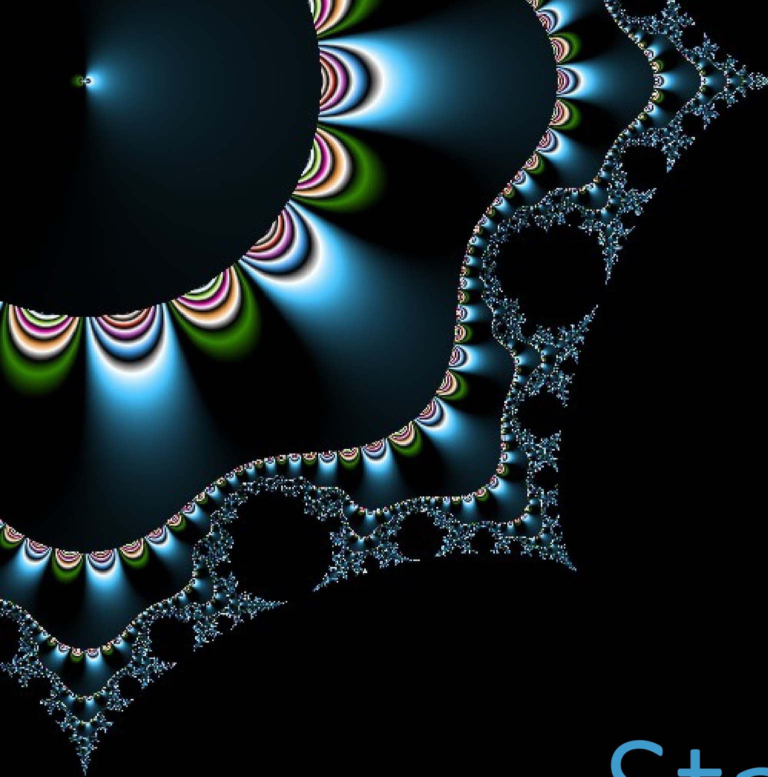
That don't belong to us

overwrite instructions

And do what we want

Which level ?





Stack based overflow

phrack 49- file 0x0e

What is a stack based overflow

Overwrite return address in the stack

And modify execution flow

It looks old

Computer Security Planning Study (1972)

First mention

Morris Worm (1988)

First attested use

Smashing the Stack for Fun and Profit (1996)

First documentation

But it’s still up to date

Local root in sudo

CVE-2019-18634

Local privileges escalation Linux kernel

CVE-2022-4378

Dos or code execution in glibc

CVE-2022-23218/23219

Local root in glibc

CVE-2023-4911

How it works

A Function call

Once upon a time

A function which does nothing

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

THE STACK

When main() is called

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

Main’s prologue

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
main :
    push %ebp
    mov %esp, %ebp
    [...]
```

esp

ebp

THE STACK

Saved ebp

@ret

Main prelude()

```c
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
[...]
push $3
push $2
push $1
[...]
```

Calling the function

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
[...]
call function
[...]
```

Function's prologue

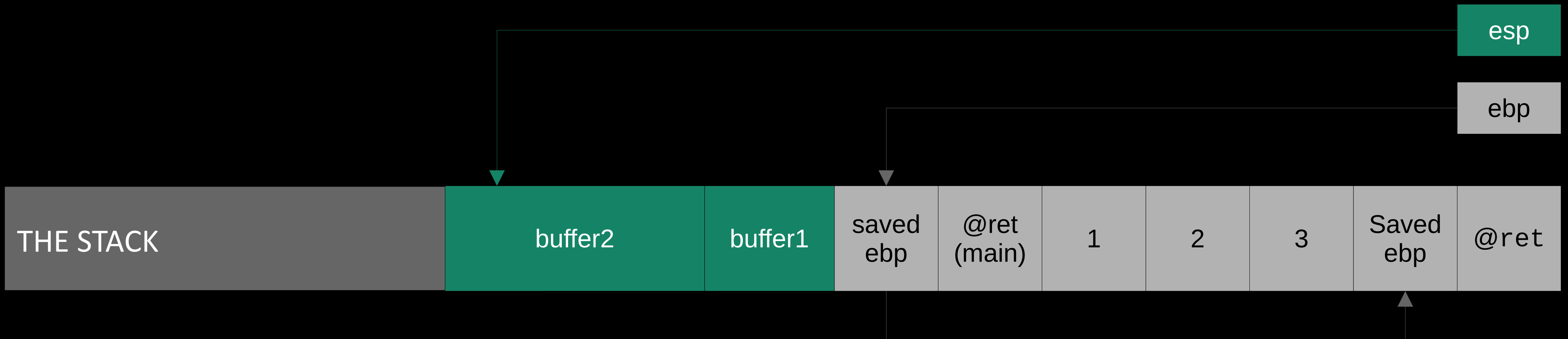
```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
function:
    push %ebp
    mov %esp,%ebp
    [...]
```

Function’s local variable

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

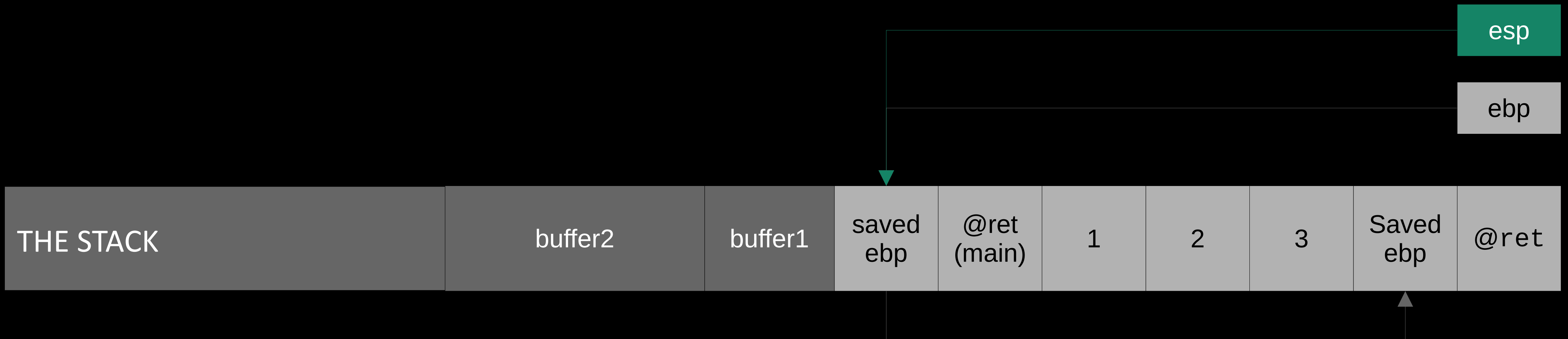
```
[...]
Sub $15, %esp
[...]
```



Function's epilogue

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
[...]
add 15, %esp
pop %ebp
ret
[...]
```



Function’s epilogue

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
[...]
add 15, %esp
pop %ebp
ret
[...]
```

Function's epilogue

```c
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
[...]
add 15, %esp
pop %ebp
ret
[...]
```

Main’s epilogue

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
[...]
add 12, %esp
pop %ebp
ret
[...]
```

Main’s epilogue

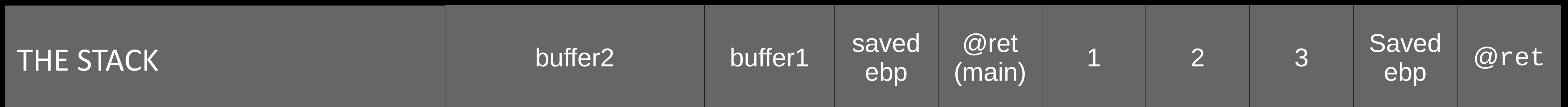
```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
[...]
add 12, %esp
pop %ebp
ret
[...]
```

Main's epilogue

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
[...]
add 12, %esp
pop %ebp
ret
[...]
```



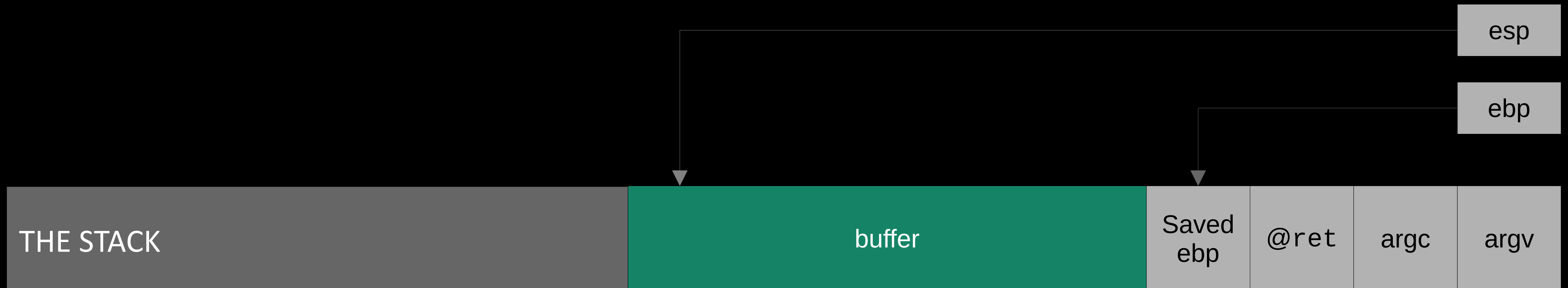
Vulnerability

where is the problem ?

Main’s epilogue

```
void main(int argc, char* argv[]) {
    char buffer[20];

    if (argc > 1)
        strcpy(buffer, argv[1]);
}
```

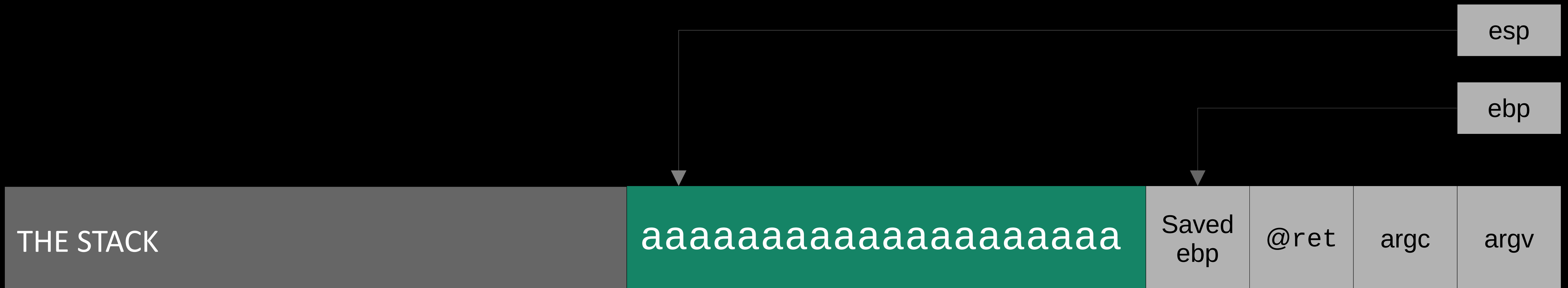


Main’s epilogue

```
void main(int argc, char* argv[]) {
    char buffer[20];

    if (argc > 1)
        strcpy(buffer, argv[1]);
}
```

```
$ ./a.out aaaaaaaaaaaaaaaaaaaa
```

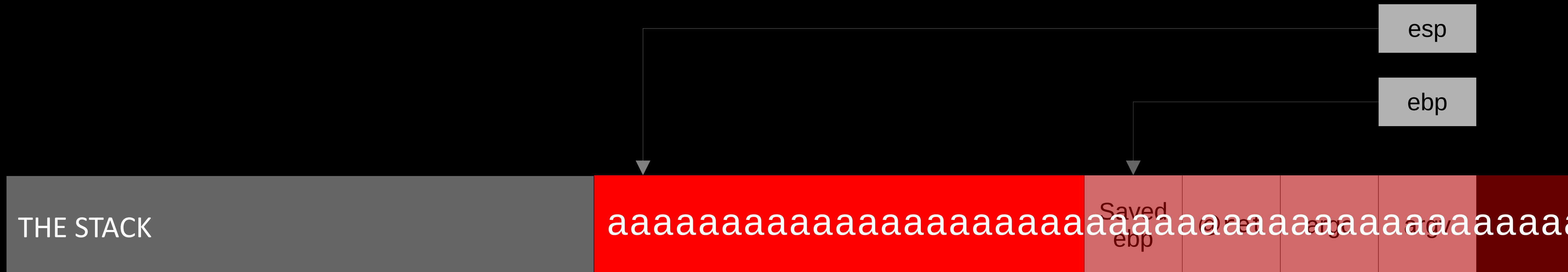


Main’s epilogue

```
void main(int argc, char* argv[]) {
    char buffer[20];

    if (argc > 1)
        strcpy(buffer, argv[1]);
}
```

```
$ ./a.out
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaa
Segmentation Fault (SIGSEGV)
```



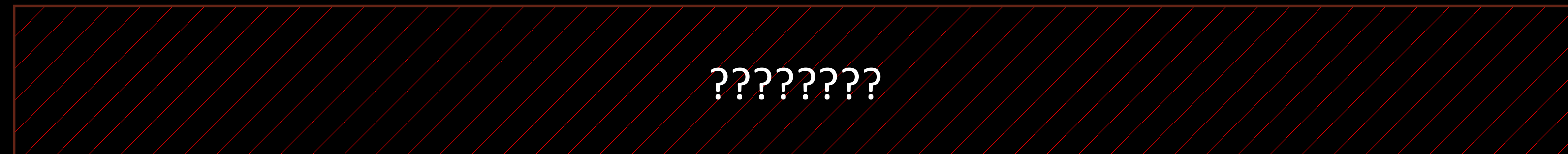
Exploit intelligently

From DOS to BOF

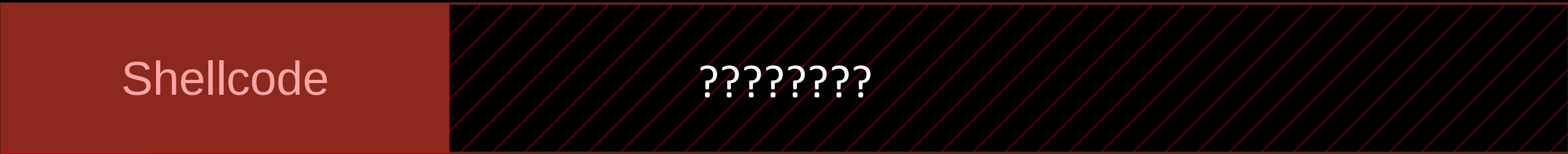
So what can we do ?

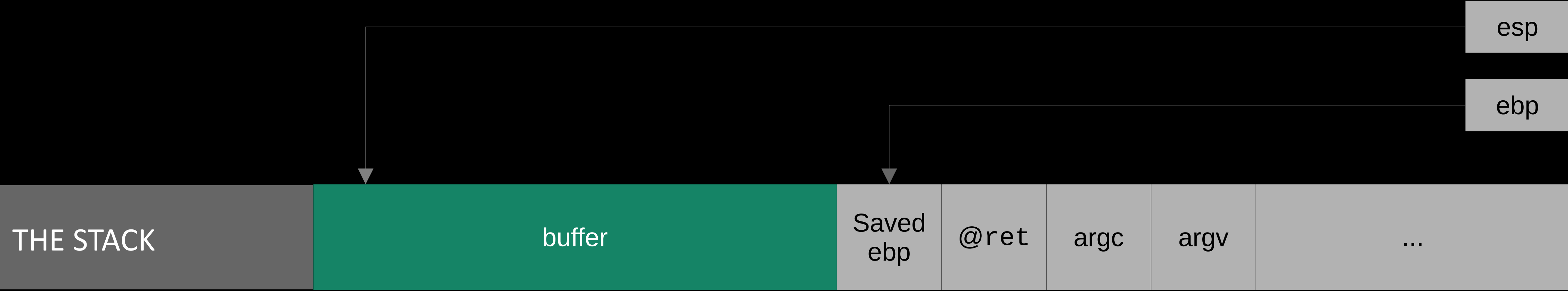
To hijack the execution flow ?

Injection :

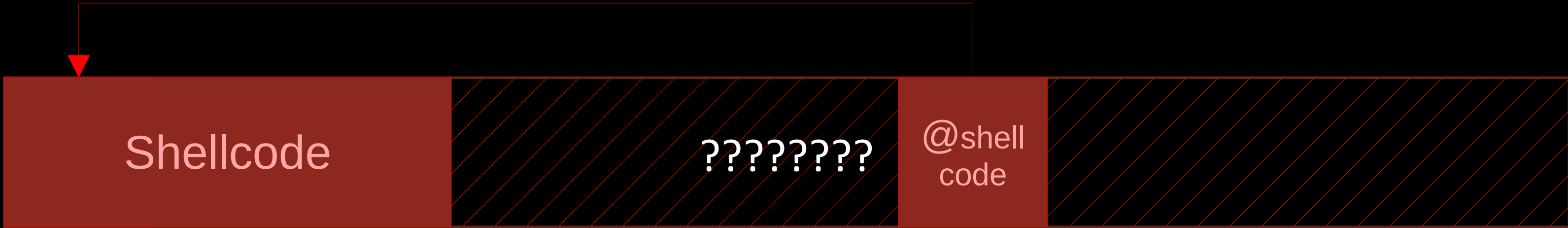


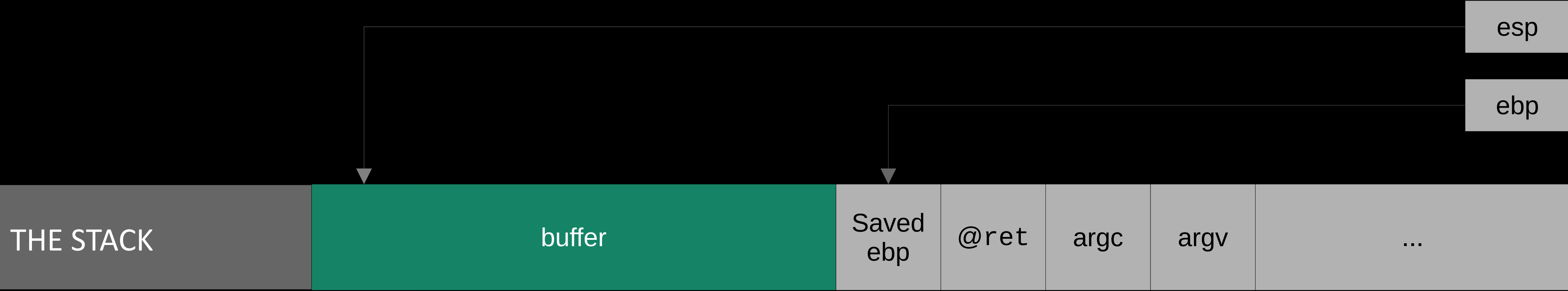
Jedi Mode
with class





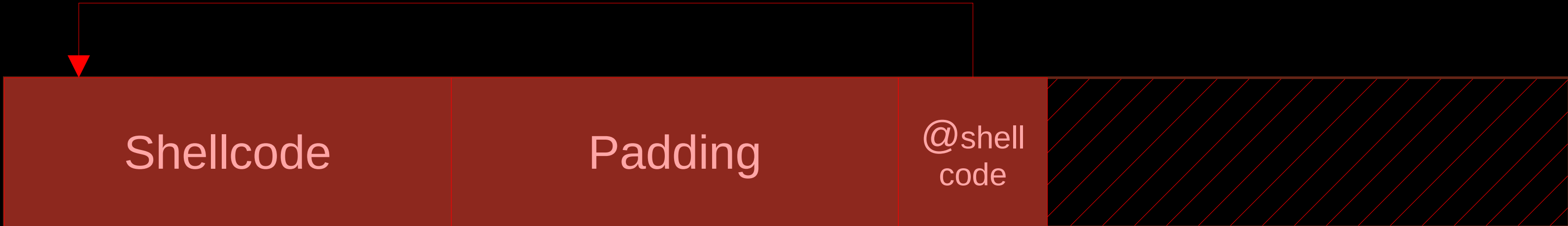
Jedi Mode
with class

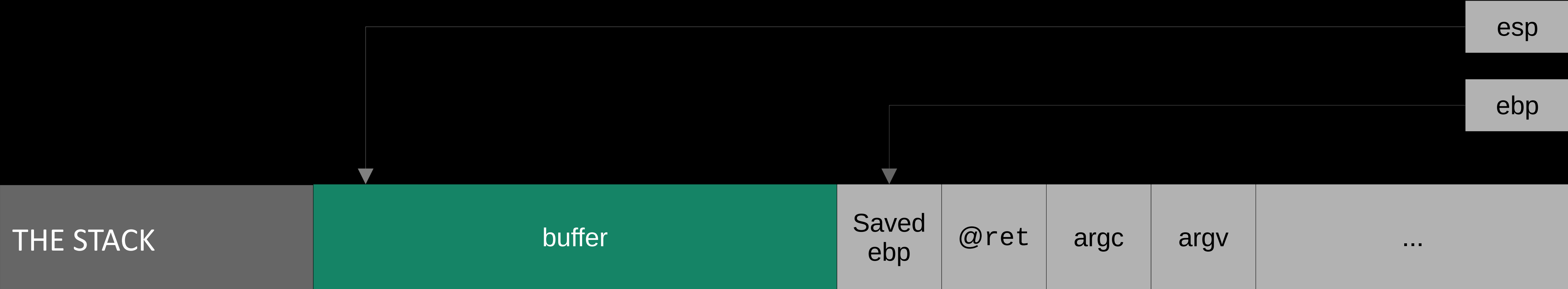




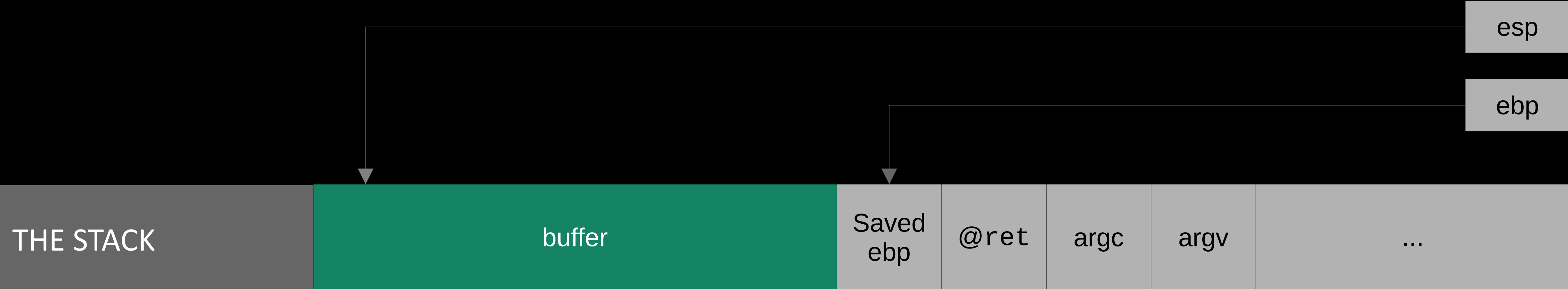
Jedi Mode
with class

Injection :





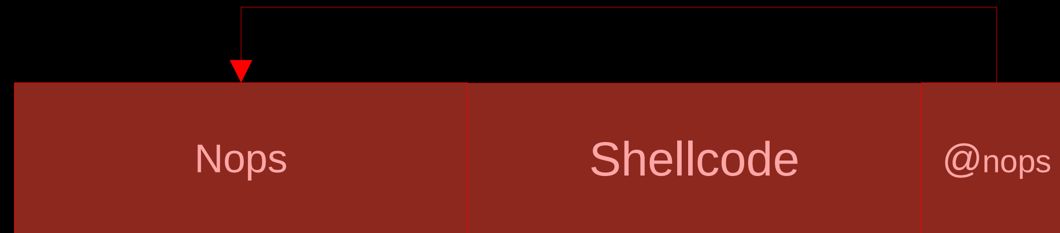
Jedi Mode
with class



Padawan Mode

Don’t be too presomptuous

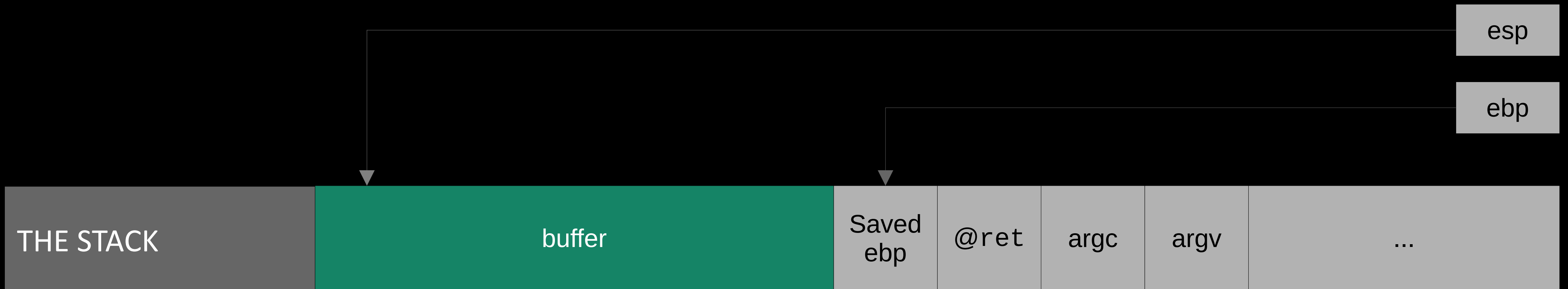
Injection :



Sith Mode

A little pushy

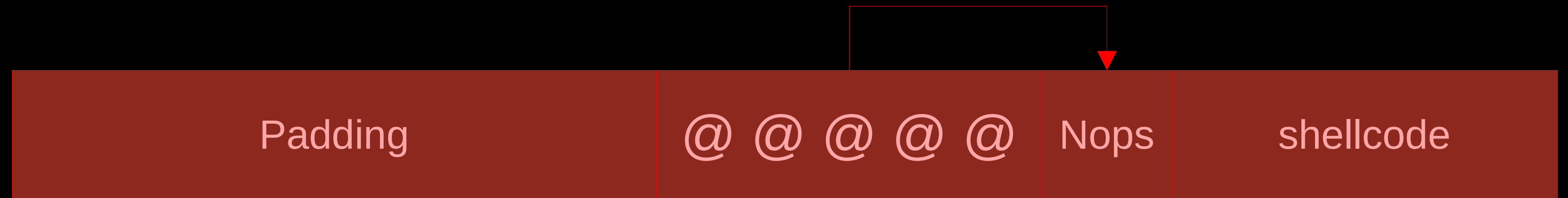
Injection :

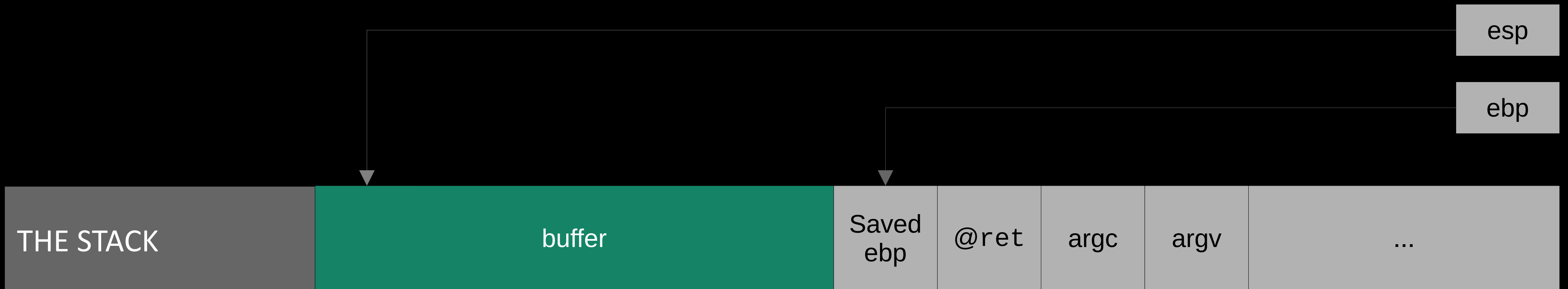


Sith Lord Mode

No subtelty at all

Injection :



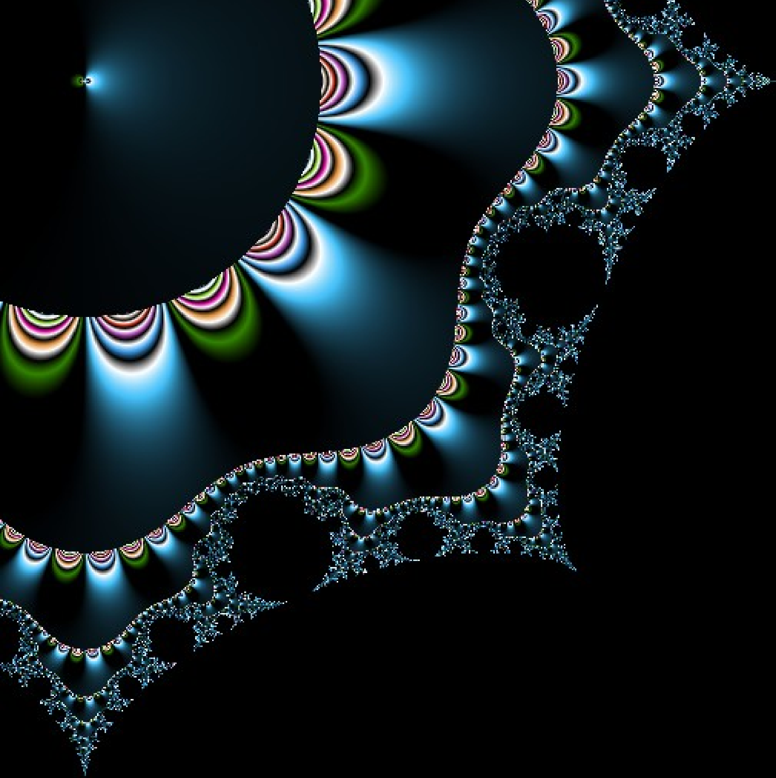


What's next ?

Environnement unfriendly

Variables, environment, whatever

Injection :



Protections

what to to against bof ?

Defense in depth

a posteriori

Compiler extension

Canari

OS configuration

Non eXecutable Stack, ASLR

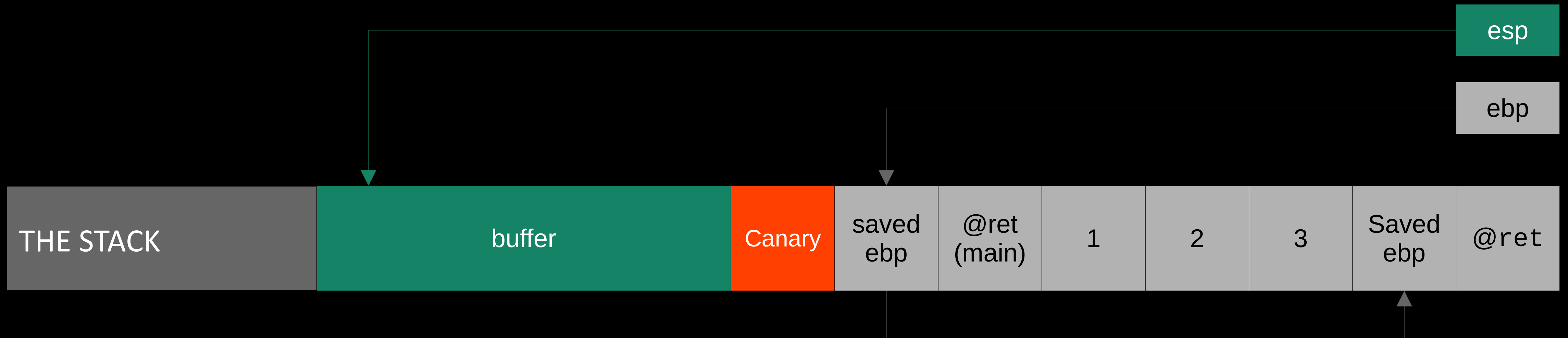
Canaries

And how to bypass

Function’s prologue
not-contractual asm

```c
void main() {
    char buffer[20]
    /* … */
}
```

```
main:
    [...]
    push <canary>
    sub $20, %esp
    [...]
```

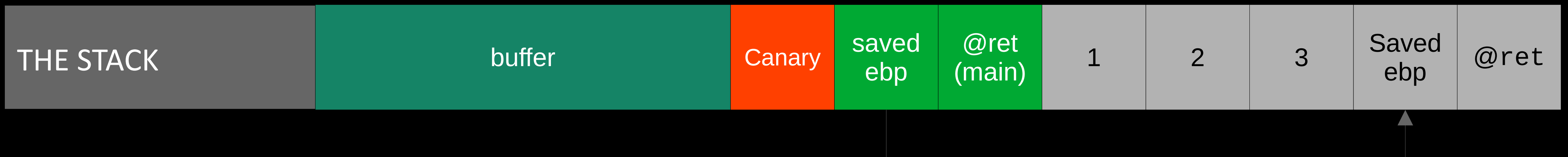


Function's epiloque

not-contractual asm

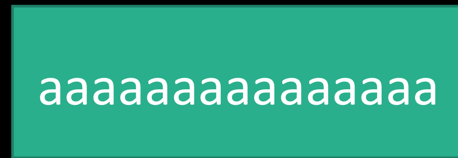
```
void main() {
    char buffer[20]
    /* … */
}
```

```
    [...]
    sub $20, %esp
    pop %eax
    cmp <canary>, %eax
    jnz ok
    call <__stack_chk_fail@plt>
ok: pop %ebp
    ret
```

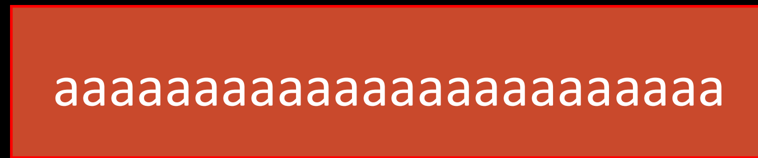


Canari

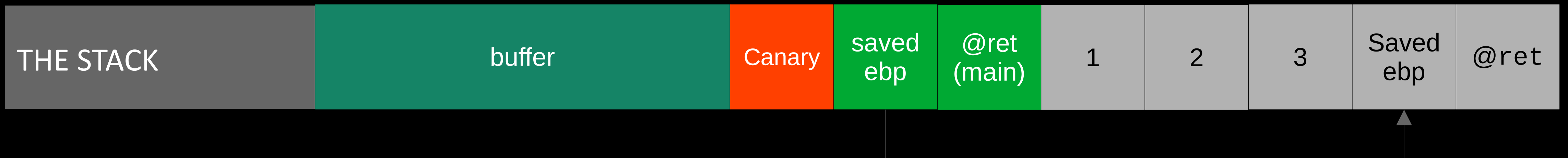
Principle



Execution continues



Execution stops



Random Values

2^{32} or 2^{64} values

Stored at some location (i.e. %gs:14)

At every launch

Follow Poisson Law

Befare of fork()

Canary duplicated in child

Value can be brute forced (enumeration)

(2^{32} steps for 32 bits)

Overflow byte by byte

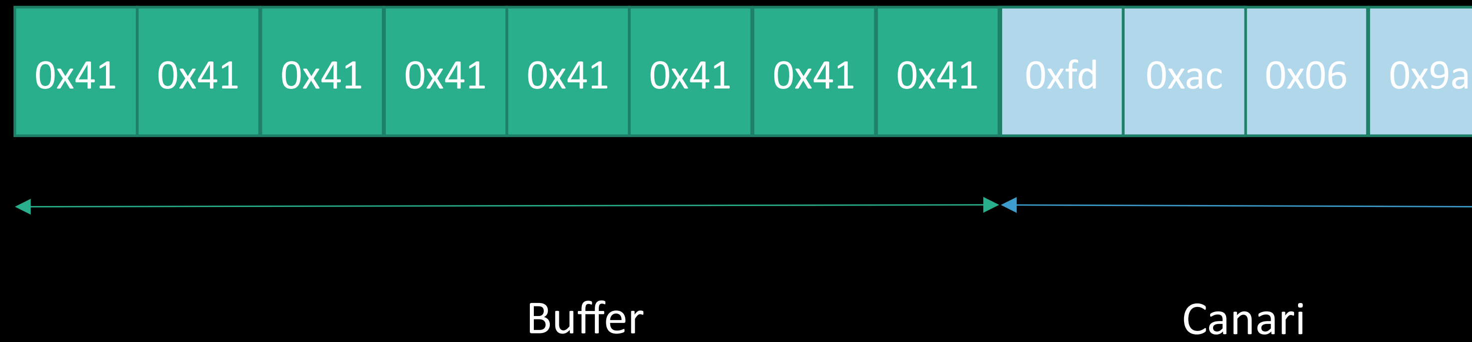
Brute-force them one at a time

(1024 steps for 32 bits)

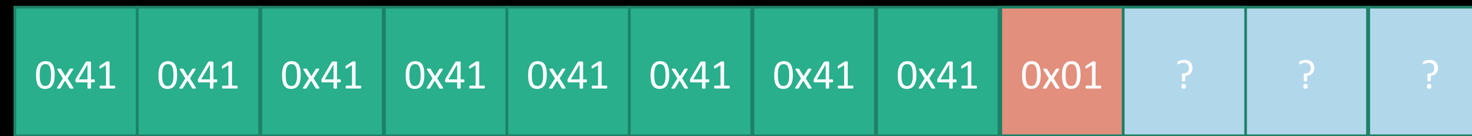
Bruteforce canari

for random canaries

Bruteforce canari



Bruteforce canari

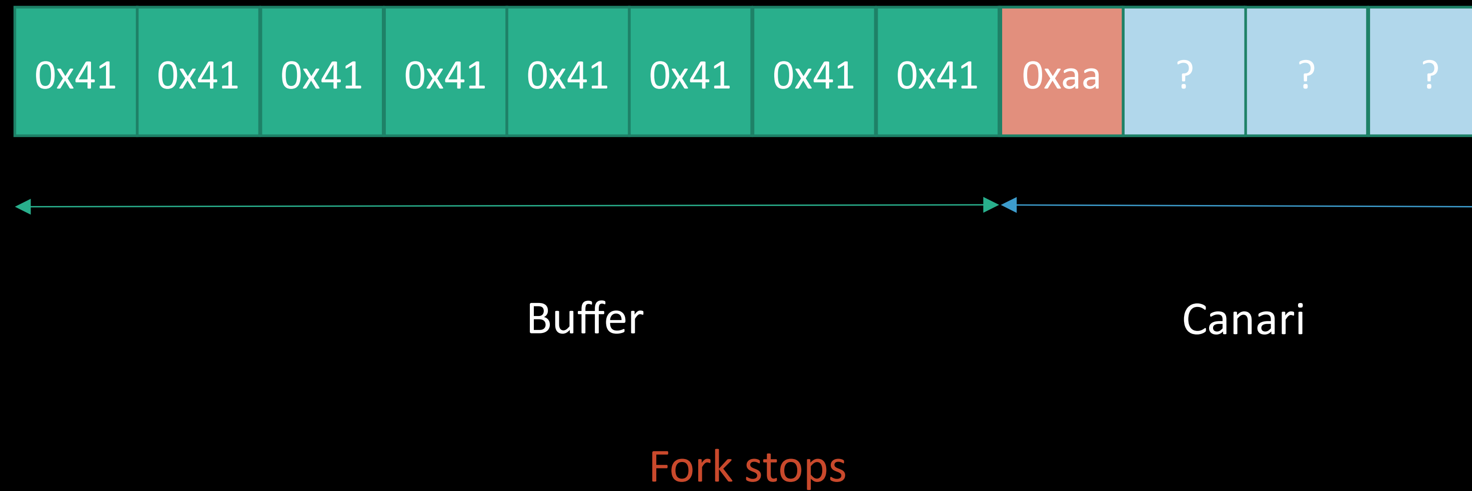


Buffer

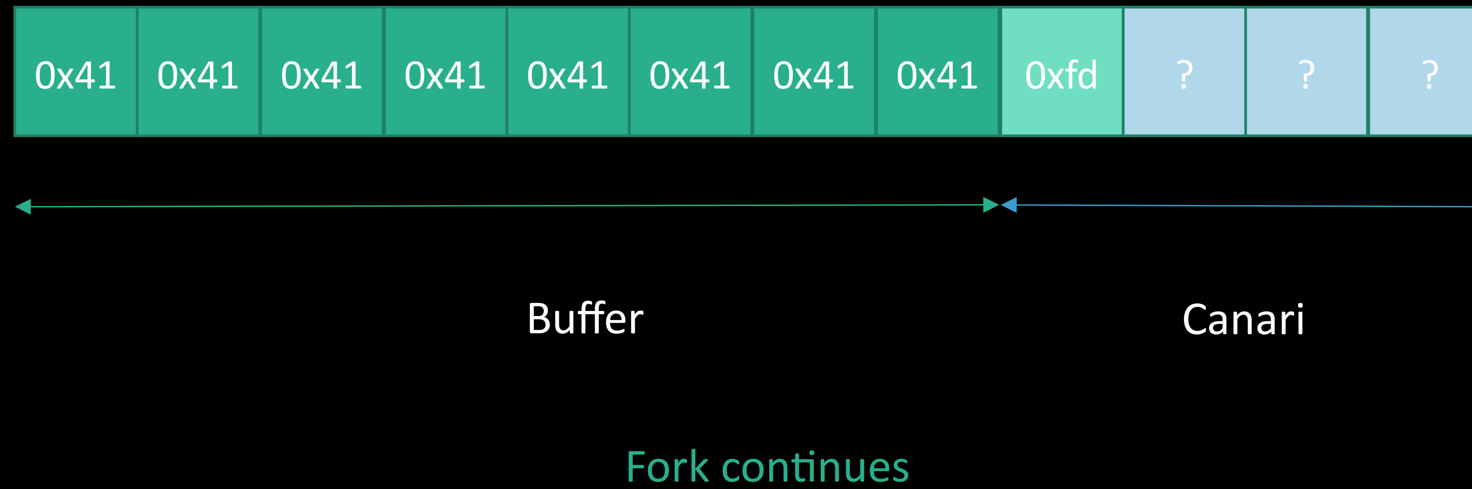
Canari

Fork stops

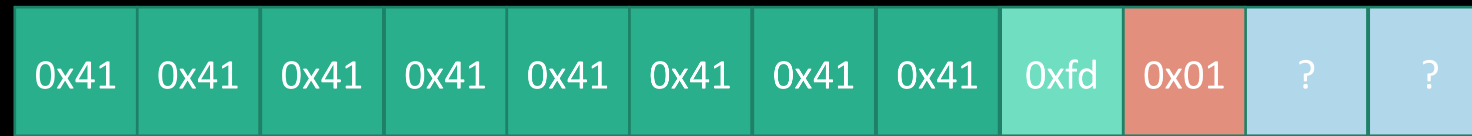
Bruteforce canari



Bruteforce canari



Bruteforce canari

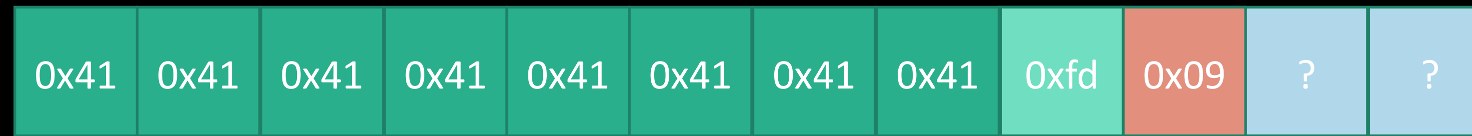


Buffer

Canari

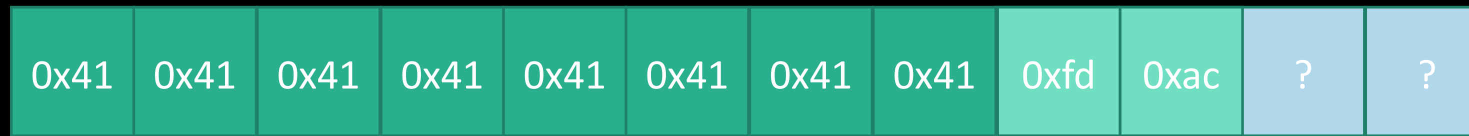
Fork stops

Bruteforce canari



Fork stops

Bruteforce canari

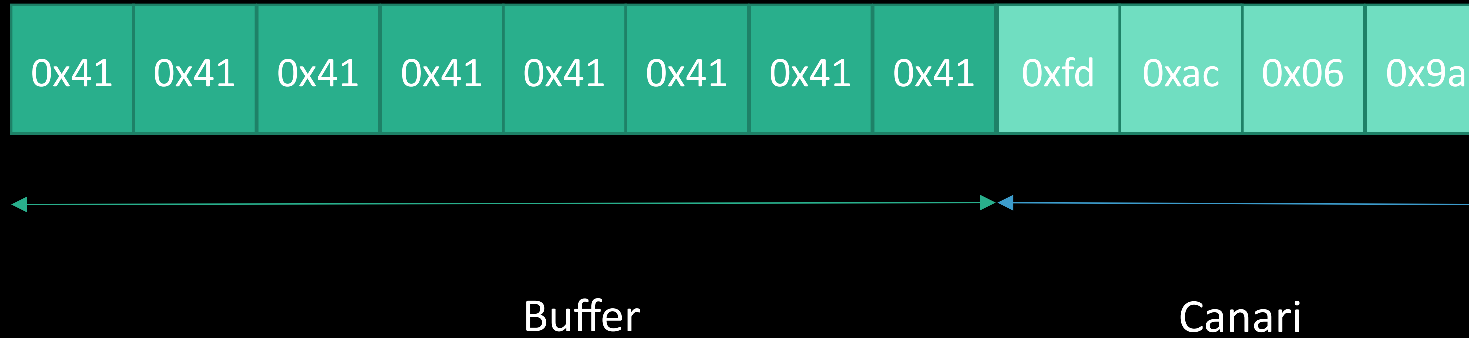


Buffer

Canari

Fork continues

Bruteforce canari



Etc…
you got the canari
Max 255 + 255 + 255 + 255 attempts

Terminator Canaries

Spaces \n \t ; ...

Terminate parse (and copy) of input string

Null bytes

Terminate copy of injected buffer

Bypass canaries

by overflowing local pointer

In a really specific configuration

The overflow overwrite a pointer address

Overwrite a pointer

Phrack 56- 5

```
void main(int argc, char **argv) {
    char * ptr ;
    char buffer[4] ;
    ptr = buffer ;
    strcpy (ptr, argv[1]) ;
    strncpy(ptr, argv[2], 4) ;
}
```

Overwrite a pointer

Phrack 56- 5

```
void main(int argc, char **argv) {
    char * ptr ;
    char buffer[4] ;
    ptr = buffer ;
    strcpy (ptr, argv[1]) ;
    strncpy(ptr, argv[2], 4) ;
}
```

```
$ ./a.out aaaa bbbb
```

THE STACK

buffer	ptr	Canary	Saved ebp	@ret

Overwrite a pointer

Phrack 56- 5

```
void main(int argc, char **argv) {
    char * ptr ;
    char buffer[4] ;
    ptr = buffer ;
    strcpy (ptr, argv[1]) ;
    strncpy(ptr, argv[2], 4) ;
}
```

```
$ ./a.out aaaa bbbb
```

THE STACK	aaaa	ptr	Canary	Saved ebp	@ret

Overwrite a pointer

Phrack 56- 5

```
void main(int argc, char **argv) {
    char * ptr ;
    char buffer[4] ;
    ptr = buffer ;
    strcpy (ptr, argv[1]) ;
    strncpy(ptr, argv[2], 4) ;
}
```

```
$ ./a.out aaaa bbbb
```

THE STACK	bbbb	ptr	Canary	Saved ebp	@ret

Overwrite a pointer

Phrack 56- 5

```
void main(int argc, char **argv) {
    char * ptr ;
    char buffer[4] ;
    ptr = buffer ;
    strcpy (ptr, argv[1]) ;
    strncpy(ptr, argv[2], 4) ;
}
```

```
$ ./a.out aaaaaaaa bbbb
```

THE STACK

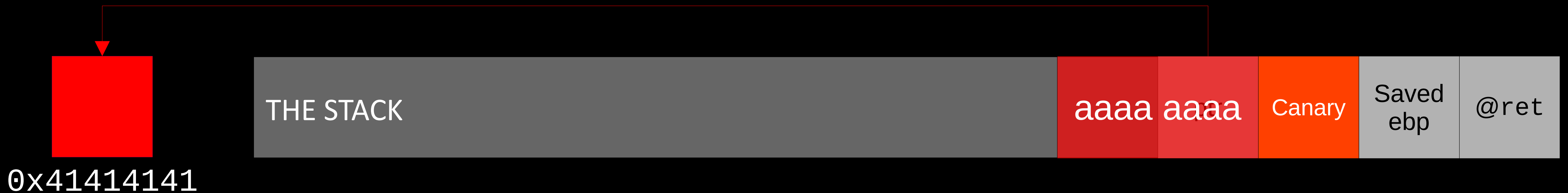
buffer	ptr	Canary	Saved ebp	@ret

Overwrite a pointer

Phrack 56- 5

```
void main(int argc, char **argv) {
    char * ptr ;
    char buffer[4] ;
    ptr = buffer ;
    strcpy (ptr, argv[1]) ;
    strncpy(ptr, argv[2], 4) ;
}
```

```
$ ./a.out aaaaaaaa bbbb
```

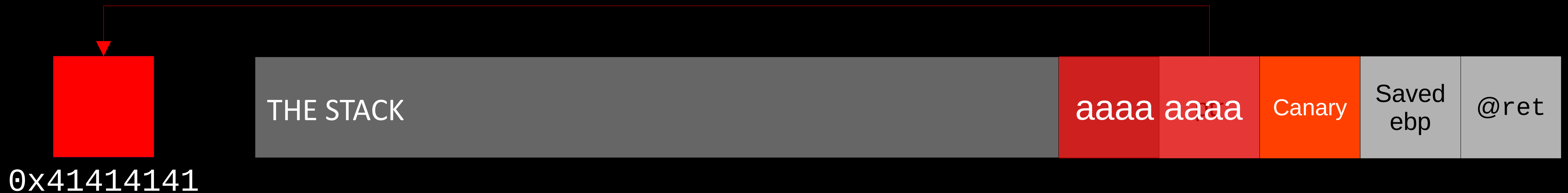


Overwrite a pointer

Phrack 56- 5

```
void main(int argc, char **argv) {
    char * ptr ;
    char buffer[4] ;
    ptr = buffer ;
    strcpy (ptr, argv[1]) ;
    strncpy(ptr, argv[2], 4) ;
}
```

```
$ ./a.out aaaaaaaa bbbb
Segmentation Fault (core dumped)
```



Overwrite a pointer

Phrack 56- 5

We can write 4 bytes anywhere

@return, GOT, PLT, ...

Canari is untouched

Because we don’t overflow after local variables

Overwrite a pointer

Phrack 56- 5

```
void main(int argc, char **argv) {
    char * ptr ;
    char buffer[4] ;
    ptr = buffer ;
    strcpy (ptr, argv[1]) ;
    strncpy(ptr, argv[2], 4) ;
}
```

```
$ ./a.out aaaa<@@ret> <@shellcode>
```

THE STACK

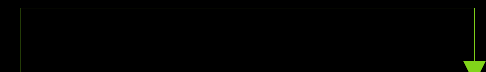
buffer	ptr	Canary	Saved ebp	@ret

Overwrite a pointer

Phrack 56- 5

```c
void main(int argc, char **argv) {
    char * ptr ;
    char buffer[4] ;
    ptr = buffer ;
    strcpy (ptr, argv[1]) ;
    strncpy(ptr, argv[2], 4) ;
}
```

```
$ ./a.out aaaa<@@ret> <@shellcode>
```



THE STACK	aaaa	@@ret	Canary	Saved ebp	@ret

Overwrite a pointer

Phrack 56- 5

```c
void main(int argc, char **argv) {
    char * ptr ;
    char buffer[4] ;
    ptr = buffer ;
    strcpy (ptr, argv[1]) ;
    strncpy(ptr, argv[2], 4) ;
}
```

```
$ ./a.out aaaa<@@ret> <@shellcode>
```

THE STACK

aaaa <@ret>

Canary

Saved ebp

NX

And how to bypass

Principle

Not eXecutable

Access right on memory pages

Read / Write / Execute

Segregate address space

Data spaces – Not executable

Instructions spaces – Not writable

Bypass

Return into an executable place

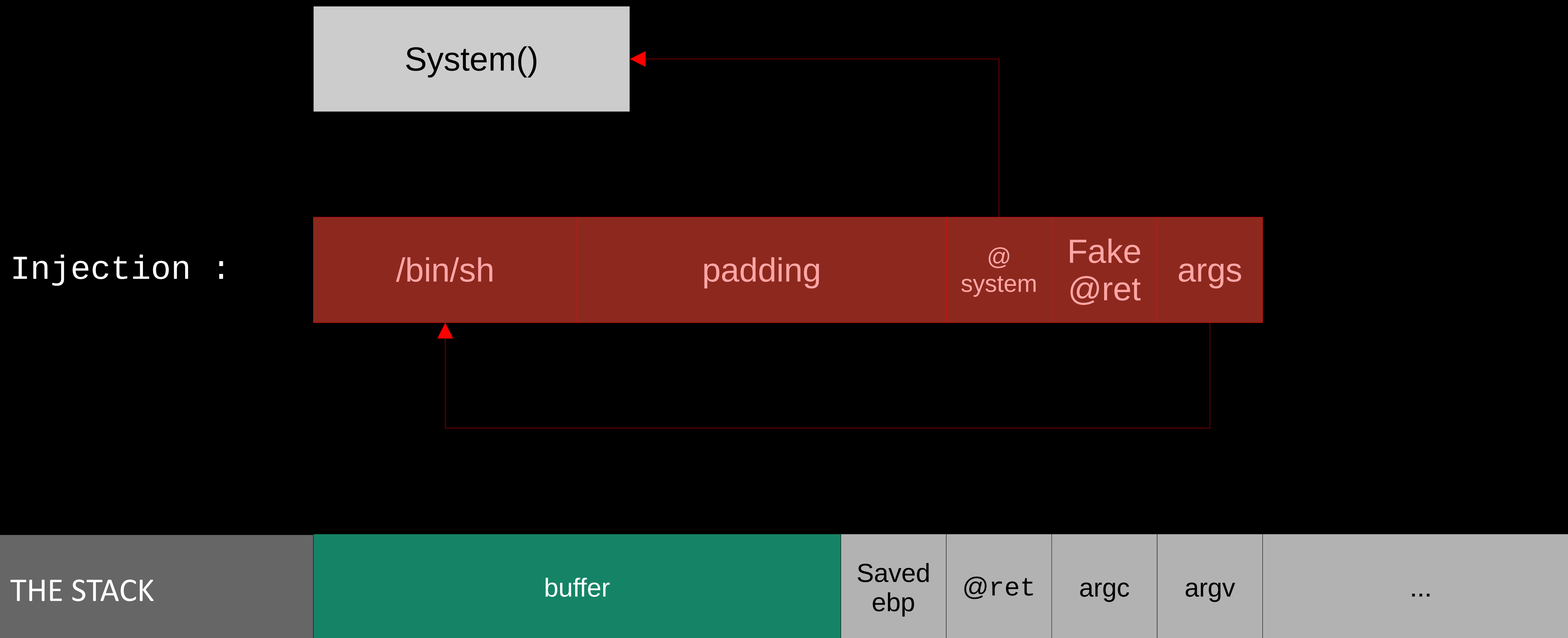
Ret2Libc

Call a function with arguments

ROP : Return Oriented Programming

Call sequence of gadgets

Ret2Libc
Principle



ROP

Example with « exit(42) ; »

Gadgets :

Mov $1,%eax ret	Mov $42,%ebx ret	syscall

Injection :

padding	@1	@2	@3

THE STACK	buffer	Saved ebp	@ret	argc	argv	...

ASLR

And how to bypass

Address Space Layout Randomization

Principle

Stack & Heap

Cannot predict @ shellcode

Executable & Libraries

Cannot predict @ libc and cie.

Bypass
Not explained here

Not so random address

i.e. 256 values for lib @

Not random at all

GOT (Global Offset Table), PLT (Process Linking Table), ...

Effectives protections

So what to ?

Defense in depth

a posteriori

Compiler extension + OS configuration

Make exploitation lot more difficult

Clean code

Avoid the problem

Check array size

Particulary in case of user inputs

Use secure functions

CERT code guidelines

Use an object oriented language

Java, C#, …

Bof Demonstration

`narnia2`