

Introduction to low-level software vulnerabilities

Lecture

Alexandre Thiroux¹

¹A company

January & February 2022



lib/logos/.png



lib/logos/.png

Table of content

lib/logos/.png

1. Introduction

2. Reverse engineering

- Use cases
- Glossary
- Compilation
- ASM x32 in a nutshell
- Methods
- Tools introduction
 - Static analysis
- Dynamic analysis
- workshop

3. Binary exploitation

- Use cases
- ELF binaries
- The memory
- Registers
- The stack and the heap
- The stack frame
- Software vulnerabilities
- Buffer overflow introduction
- Buffer overflow: exploitation
- Workshop

4. Conclusion and Q&A

Whoami

ib/logos/.png

Alexandre Thiroux

- CSIRT Manager at **Sectigo**
- Fields of interest : Forensics, Internet of Things, low level exploitation.
- **Defcon 11333** Organizer (<https://dc11333.org>)
- GCIH (SANS), OSCP & OSCE (Offensive Security) certified

Submodule introduction

This 8H submodule is composed with two parts:

- Reverse engineering (4H including a 15min break)
 - Lecture (45min)
 - Workshop (60min)
- Binary exploitation (4H including a 15min break)
 - Lecture (60min)
 - Workshop (45min)



Submodule requirements

To follow this lesson you will need the following pre-requisites:

- The Kali VM installed on your machine **4GB RAM (min)**
- Understanding C simple source code
- Python programming skills
- Basic knowledge in x32 ASM (reminders are planned)
- Basic knowledge in system architecture
- A notepad



Introduction to software exploitation

lib/logos/.png

A software is potentially subjects to vulnerabilities or flaws introduced in the source code.
In this lesson you will discover the two main axis of the software exploitation:

- The reverse engineering
- The binary exploitation

Reverse engineering allows to understand the software logic and discover hidden functionalities after the program has been compiled. Depending of the program size and complexity, the process can be quite difficult.

Binary exploitation will allow you to find and exploit a vulnerability. This sometimes leads a full control of a host.



Reverse engineering



Reverse engineering: introduction

Reverse engineering use cases

Reversing a program is used for many purposes. Here are some common use cases:

- Malware analysis
- Closed source program understanding
- Discover patent violations
- Discovering vulnerabilities



Figure 1: Reverse engineering example:
ReactOs a Windows clone

Reverse glossary

Some technical terms you have to know about:

- Disassembly
Get the assembly code of a compiled program
- Decompile
From assembly code, try to reproduce the original source code of the program
- Static analysis
Analyse a software without executing it
- Dynamic analysis
Analyse software interactions while running it

Compiling a code

lib/logos/.png

```
main:
    lea    ecx, [esp+4]
    and    esp, -16
    push   DWORD PTR [ecx-4]
    push   ebp
    mov    ebp, esp
    push   ebx
    push   ecx
    sub    esp, 64
    mov    ebx, ecx
    sub    esp, 12
    push   OFFSET FLAT:.LC0
    call   puts
    add    esp, 16
    cmp    DWORD PTR [ebx], 1
    jg     .L2
    sub    esp, 12
    push   OFFSET FLAT:.LC1
    call   puts
    add    esp, 16
    mov    eax, 1
    jmp   .L4
.L2:
    mov    eax, DWORD PTR [ebx+4]
    add    eax, 4
    mov    eax, DWORD PTR [eax]
    sub    esp, 8
    push   eax
    lea    eax, [ebp-58]
```

Figure 2: C code to x32 ASM

ASM x32 in a nutshell : basic instructions

There are two syntaxes for ASM instructions: Intel or AT&T.

The Intel one is `add eax, ebx ;` whereas the AT&T is `addq %ebx, %eax ;`

In this course, we will only use the Intel syntax.

- `MOV eax, 0x1` : move a value to a register
- `PUSH 0x1` : push a value on the stack
- `POP eax` : remove a value from the stack and put it inside a register
- `CALL eax` : call a function with arguments
- `SUB eax, 0x1` : subtract a value from a register
- `ADD eax, 0x1` : add value to a register
- `JMP eax` : jump to an instruction
- `INT3` : place a breakpoint in the code
- `INT80` : System call
- `RET` : return to previous function (= `POP RIP + JMP RIP`)

Break: 5 minutes

lib/logos/.png

Break time. If you have any question, don't hesitate to ask it now.
Otherwise, go to this website and try to play with its functionalities:

- <https://godbolt.org/>

Don't forget the -m32 parameter.

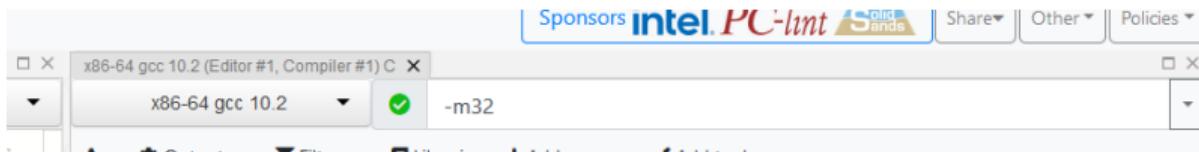


Figure 3: Compiler parameters

Different methods

lib/logos/.png

There are different methods of reverse engineering:

- **Static analysis** is a method of computer program debugging, done by examining the code **without executing** the program.
- **Dynamic analysis** is the testing and evaluation of a **program by executing data in real-time**. The objective is to find errors in a program while it is running, rather than examining the code offline.



Reverse engineering: Tools and methods

Static analysis tools: Strings

lib/logos/.png



```
QQSV
u)9U
9PEI u
SERVt
<$XZ
QYY
QYY
f;Ls
$`<t
_`][YY
:J:s:
:);?;j;
;6<`<a<
B^Z=
J?4]<l
LIP
7Yh;
d66]
"Or{[
(\v')
d=6
.c:)
MSU1
&(zL
```

Figure 4: String command results

Strings command on linux will extract text strings from binary files. It can be useful to spot passwords or configuration parameters.

Static analysis tools: Ghidra

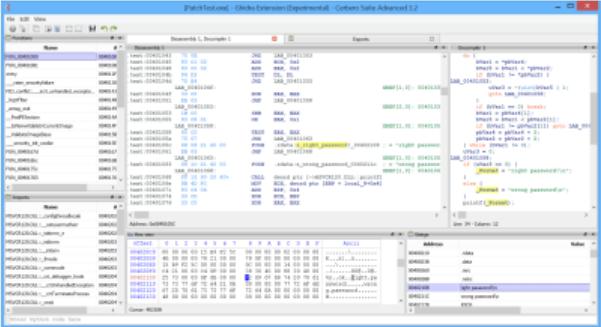


Figure 5: Ghidra's main interface

Ghidra is an all-in-one disassembly and decompiler software. It provides useful functionnalities like functions call tree.

Remark

Similar tools exists such as IDA (Paid) or Radare2.

Dynamic analysis tools: (S|L)trace

lib/logos/.png

- Strace is a diagnostic, debugging and instructional userspace utility. It is used to monitor and tamper with interactions between processes and the Linux.
- Ltrace is a debugging utility, used to display the calls in a userspace application makes to shared libraries.

Dynamic analysis tools: Wireshark

- Wireshark is packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development.
- Often, programs are communicating with their environment. Using a packet analyser allows to analyse and track those communications.

Dynamic analysis tools: GDB



Figure 6: GDB's main interface

GDB is a debugger. A debugger is a software which helps developpers/or reversers to understand the program working. By setting breakpoints you can interrupt the execution to verify the environement at a given time.

Remark

On Windows you can use OllyDBG or WinDBG



Reverse engineering: Workshop

Workshop

- You will find exercises https://github.com/Hexliath/VA_RE
- I'm here to answer your questions

lib/logos/.png

Information

Next lesson will be on Binary exploitation.

On your kali VM, please install gdb-peda (<https://github.com/longld/peda>) and have a look on how the stack is working.



Binary exploitation



Binary exploitation: introduction

Binary exploitation use cases

The principle of binary exploitation is to take advantage of bugs located inside a software in order to cause unintended or unanticipated behavior.

Here are some common use cases :

- Gaining privilege escalation
- Exploiting private devices (such as game consoles)
- Getting an unauthorized access on distant machine

Finding and exploiting such bugs is a long process. Thus, the researching part is almost never done during a pentest where already made exploits are preferred.

Exploit.db

The screenshot shows the Exploit.db website interface. On the left is a vertical sidebar with icons for Home, About, Contact, and Help. The main area has a dark header with the Exploit Database logo and navigation links for Home, About, Contact, Help, and Log In. Below the header is a search bar and a filter section with 'Filters' and 'Reset All' buttons. The main content area displays a table of vulnerabilities with columns for Date, D, A, V, Title, Type, Platform, and Author. The table lists 15 entries from January 2021, including various web application vulnerabilities like SmartAgent 3.1.0 - Privilege Escalation, Cermeti Mapping and Information System 1.0 - Multiple SQL Injections, and WordPress Plugin Custom Global Variables 1.0.5 - 'name' Stored Cross-Site Scripting (XSS). The table includes a search bar at the top right and a footer with page navigation buttons.

Date	D	A	V	Title	Type	Platform	Author
2021-01-12	+			SmartAgent 3.1.0 - Privilege Escalation	WebApps	Multiple	Orion Hisry
2021-01-12	+			Cermeti Mapping and Information System 1.0 - Multiple SQL Injections	WebApps	PHP	Mesut Cetin
2021-01-12	+			Gila CMS 2.0.0 - Remote Code Execution (Unauthenticated)	WebApps	PHP	Erendier
2021-01-11	+			Prestashop 1.7.7.0 - Id_product Time Based Blind SQL Injection	WebApps	PHP	Jairim Gondalya
2021-01-11	+			Portablebanian 4.3.6579.38136 - Encrypted Password Retrieval	Local	Windows	rootabeta
2021-01-11	+			OpenCart 3.0.36 - ATO via Cross Site Request Forgery	WebApps	PHP	Mahendra Purbia
2021-01-11	+			WordPress Plugin Custom Global Variables 1.0.5 - 'name' Stored Cross-Site Scripting (XSS)	WebApps	PHP	Swarnil Subhadev Bodkar
2021-01-11	+			Cermeti Mapping and Information System 1.0 - Multiple Stored Cross-Site Scripting	WebApps	PHP	Mesut Cetin
2021-01-11	+			EyesOfNetwork 5.3 - LFI	WebApps	Multiple	Audencia Business SCHOOL Red Team
2021-01-11	+			Anchor CMS 0.12.7 - 'markdown' Stored Cross-Site Scripting	WebApps	Multiple	Ramszan Mert GÖKTEN
2021-01-11	+			EyesOfNetwork 5.3 - RCE & PrivEsc	WebApps	Multiple	Audencia Business SCHOOL Red Team
2021-01-08	+			Wordpress Plugin wpDiscuz 7.0.4 - Unauthenticated Arbitrary File Upload (Metasploit)	WebApps	PHP	SunCSR Team
2021-01-08	+			Wordpress Plugin Autoptimize 2.7.6 - Authenticated Arbitrary File Upload (Metasploit)	WebApps	PHP	SunCSR Team
2021-01-08	+			Apache Flink 1.11.0 - Unauthenticated Arbitrary File Read (Metasploit)	WebApps	Java	SunCSR Team
2021-01-08	+			Cockpit Version 234 - Server-side Request Forgery (Unauthenticated)	WebApps	Multiple	Metin Yunus Kandemir

Showing 1 to 15 of 43,620 entries

FIRST PREVIOUS 1 2 3 4 5 ... 2908 NEXT LAST

Figure 7: Exploit.db website

ELF binaries

In this course, we will study ELF binary file format which is the default Linux executable format. An executable is made of different sections. Here are the main ones (use **man elf** to see them all)

Name	Description	Flags
.text	Holds the executable instructions of the program	allocated & executable
.rodata	Holds the read only data	allocated
.data	Holds initialized data that contributes to program's memory	allocated & write
.bss	Holds uninitialized data that contributes to program's memory	allocated & write

Table 1: ELF Sections

Memory

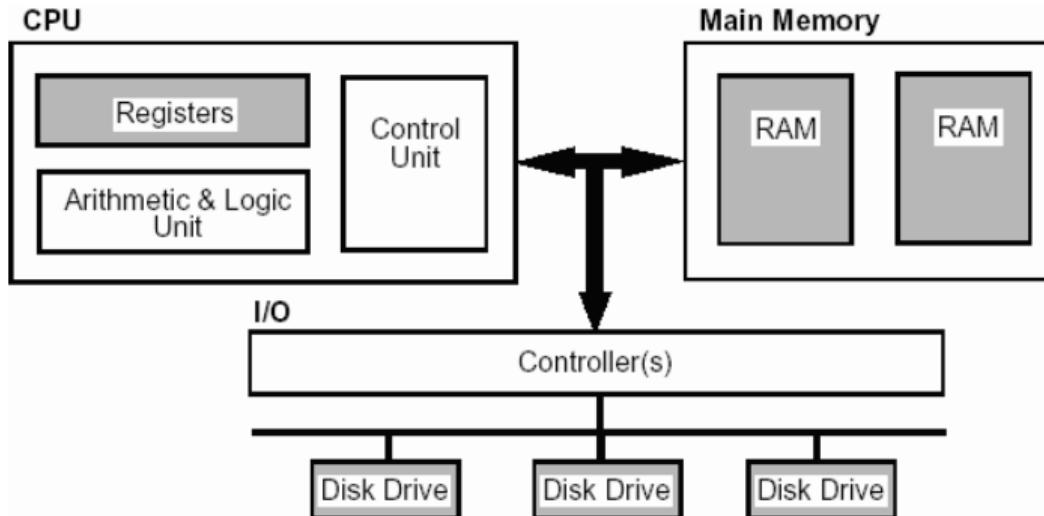


Figure 8: RISC architecture : the memory

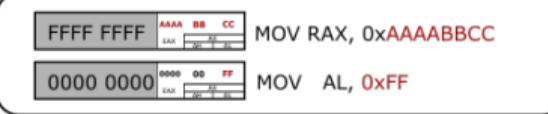
Memory: The registers

Essential for data manipulation, registers are places where a program store temporary values during the execution. Registers also store information about the execution flow (EFLAGS).

Working registers

	63	31	15	0
RAX	EAX	AH	BB	CC
RBX	EBX	BH	BL	
RCX	ECX	CX	CH	CL
RDX	EDX	DH	DL	
RBP	EBP	BP	PL	
RSI	ESI	SI	SL	
RDI	EDI	DI	DL	
RSP	ESP	SP	SL	
R8				
R9				
R10				
R11				
R12				
R13				
R14				
R15				

Moving values to registers



RIP: Register Instruction Pointer
Points on the current instruction

RFLAGS: Flags register
Contains boolean values about
the execution state

Memory : The stack and the heap

When it comes to store data, the stack and the heap are really useful. Stack is mainly used to store local data and variables with a known size whereas heap will be used when variables are declared through allocations (malloc, realloc...).

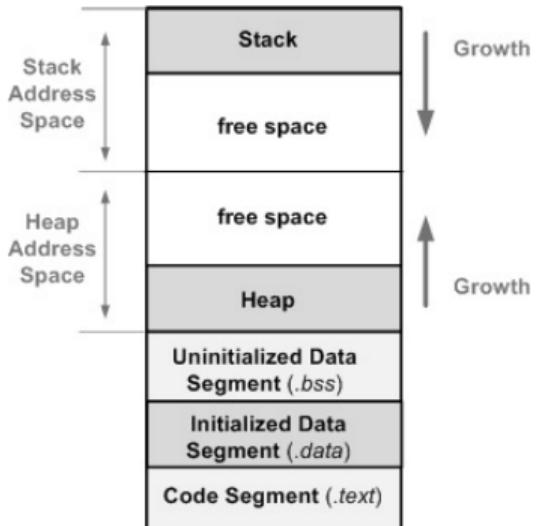


Figure 9: Memory layout

A stack frame

When you enter a new function using `call` instruction, the stack will create a new frame.

A stack frame will always contains the following elements:

- Return address : the next asm instruction right after the call
- Caller's EBP: the previous function EBP value
- Local variables and saved register content

A stack frame: exploit_me() function

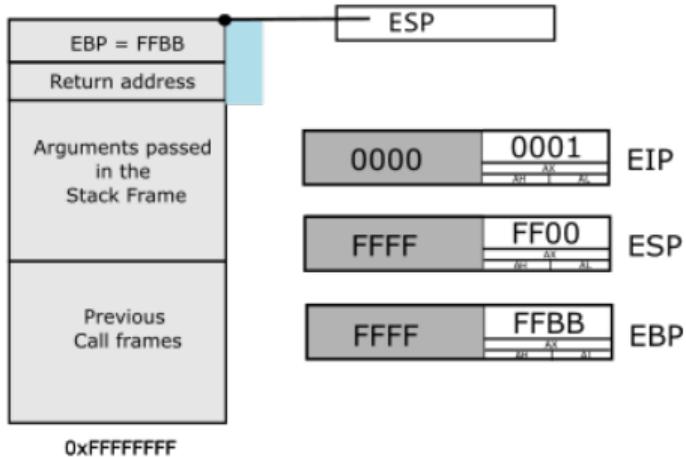
To understand the stack working, we will study the exploit_me function. This function just take an argument input and copy it to a 50 bytes char array.

	0x01	push	ebp
	0x02	mov	ebp, esp
	0x03	sub	esp,72
	0x04	sub	esp,8
void exploit_me(char* arg){	0x05	push	[ebp+8]
char buf[50];	0x06	lea	eax, [ebp-58]
strcpy(buf,arg);	0x07	push	eax
}	0x08	call	strcpy
	0x09	add	esp, 16
	0x0A	mov	ebp,esp
	0x0B	pop	ebp
	0x0C	ret	

Figure 10: C code and ASM disassembly

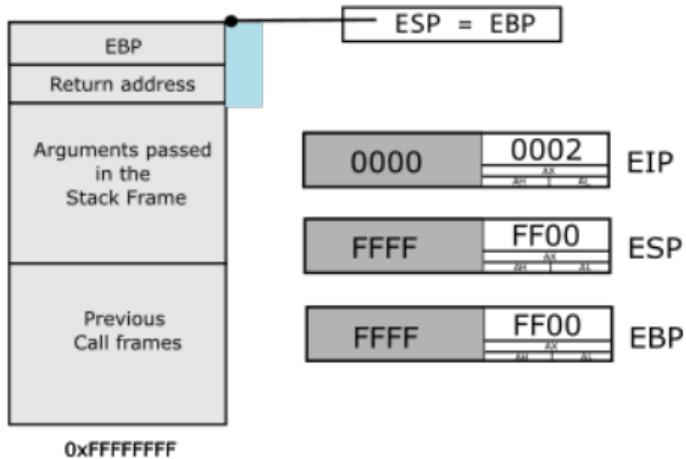
A stack frame: exploit_me() function

0x01	push	ebp
0x02	mov	ebp, esp
0x03	sub	esp,72
0x04	sub	esp,8
0x05	push	[ebp+8]
0x06	lea	eax, [ebp-58]
0x07	push	eax
0x08	call	strcpy
0x09	add	esp, 16
0x0A	mov	ebp,esp
0x0B	pop	ebp
0x0C	ret	



A stack frame: exploit_me() function

0x01	push	ebp
0x02	mov	ebp, esp
0x03	sub	esp,72
0x04	sub	esp,8
0x05	push	[ebp+8]
0x06	lea	eax, [ebp-58]
0x07	push	eax
0x08	call	strcpy
0x09	add	esp, 16
0x0A	mov	ebp,esp
0x0B	pop	ebp
0x0C	ret	

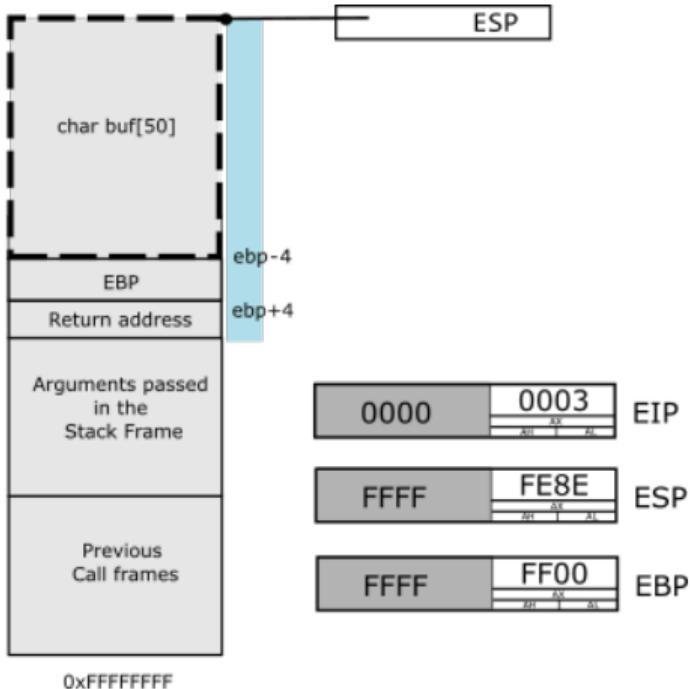


A stack frame: exploit_me() function

0x01	push	ebp
0x02	mov	ebp, esp
0x03	sub	esp,72

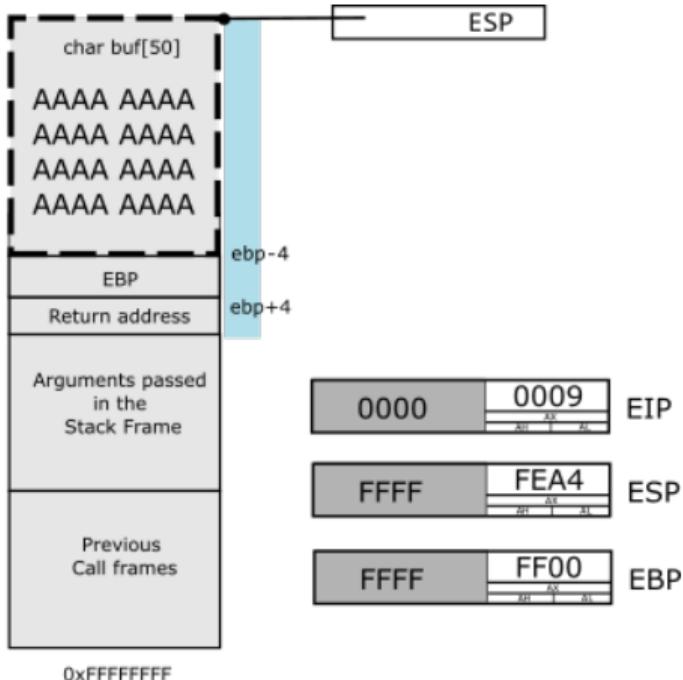
0x04	sub	esp,8
0x05	push	[ebp+8]
0x06	lea	eax, [ebp-58]
0x07	push	eax
0x08	call	strcpy
0x09	add	esp, 16

0xA	mov	ebp,esp
0xB	pop	ebp
0xC	ret	



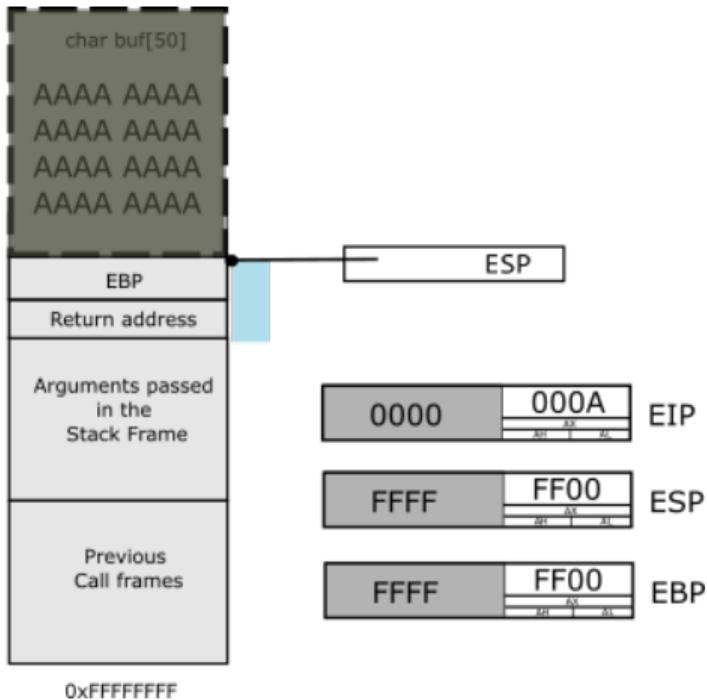
A stack frame: exploit_me() function

0x01	push	ebp
0x02	mov	ebp, esp
0x03	sub	esp,72
0x04	sub	esp,8
0x05	push	[ebp+8]
0x06	lea	eax, [ebp-58]
0x07	push	eax
0x08	call	strcpy
0x09	add	esp, 16
0x0A	mov	ebp,esp
0x0B	pop	ebp
0x0C	ret	



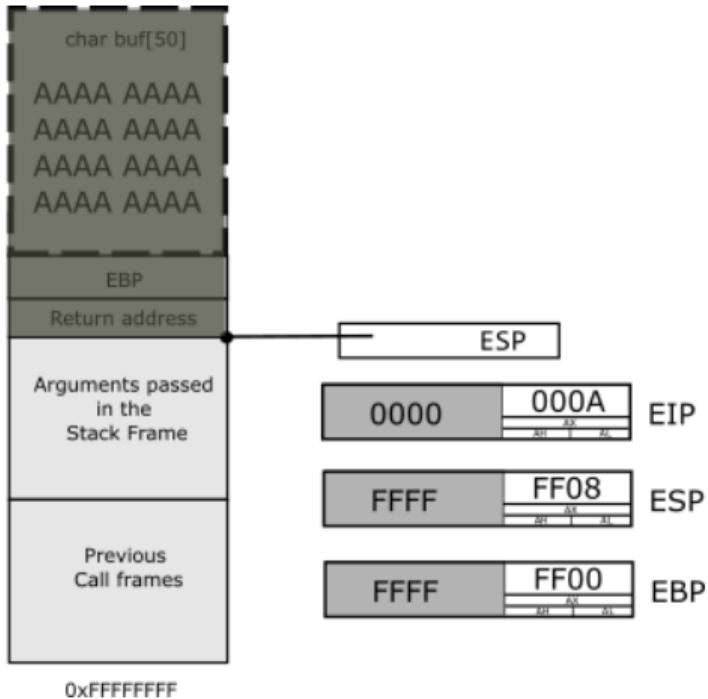
A stack frame: exploit_me() function

0x01	push	ebp
0x02	mov	ebp, esp
0x03	sub	esp,72
0x04	sub	esp,8
0x05	push	[ebp+8]
0x06	lea	eax, [ebp-58]
0x07	push	eax
0x08	call	strcpy
0x09	add	esp, 16
0xA	mov	esp,ebp
0xB	pop	ebp
0xC	ret	



A stack frame: exploit_me() function

0x01	push	ebp
0x02	mov	ebp, esp
0x03	sub	esp,72
0x04	sub	esp,8
0x05	push	[ebp+8]
0x06	lea	eax, [ebp-58]
0x07	push	eax
0x08	call	strcpy
0x09	add	esp, 16
0x0A	mov	esp,ebp
0x0B	pop	ebp
0x0C	ret	



Break: 15 minutes

Break time.

- Open your VM
- Install gdb-peda (<https://github.com/longld/peda>)
- Download <https://cutt.ly/IjmnCJI>
- run gdb ./demo.exe, set breakpoints and inspect the registers

Remark

Basic commands for gdb are :

disas <function name> : disassemble a function

b * <breakpoint address>: set a breakpoint at a given address

ni: go to next instruction (**EIP+1**)

c: continue the execution until next breakpoint

r: run or restart the program

i r: inspect register

x/xw <register or address>: see address content



Binary exploitation: buffer overflow

Software vulnerabilities

A lot of vulnerabilities exist in pentesting software field. Here are some of them you may have heard about :

- Stack-based buffer overflow
- Heap-based overflow
- Format string vulnerability
- Integer overflow

In the next slides, we will focus on "Stack-based buffer overflow" vulnerability which is probably the most known.

Buffer overflow: introduction

This type of vulnerability arises **when more data are inserted in a local variable** than the actual variable size. A stack-buffer overflow happen and data from the stack are overwrite by user input.

Let's reuse our previous exploit_me() function.

```
void exploit_me(char* arg){  
    char buf[50];  
    strcpy(buf,arg);  
}
```

Figure 11: exploit_me() function

Buffer overflow: introduction

What does happen when an argument longer than 50 characters is passed to the function ?
Let's say arg is 'A'x80 without any spaces.

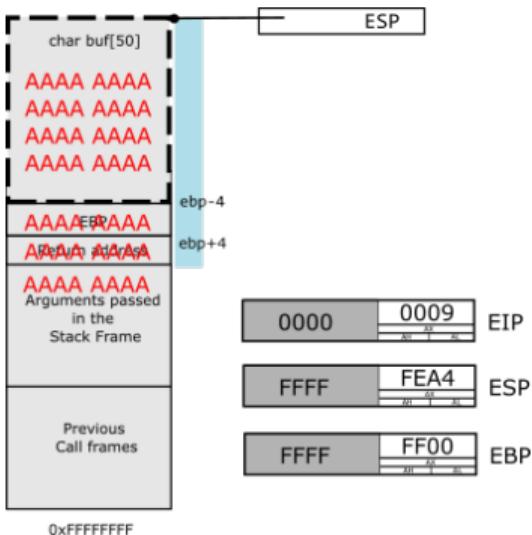


Figure 12: The stack frame is overwritten by the passed argument

Live demonstration

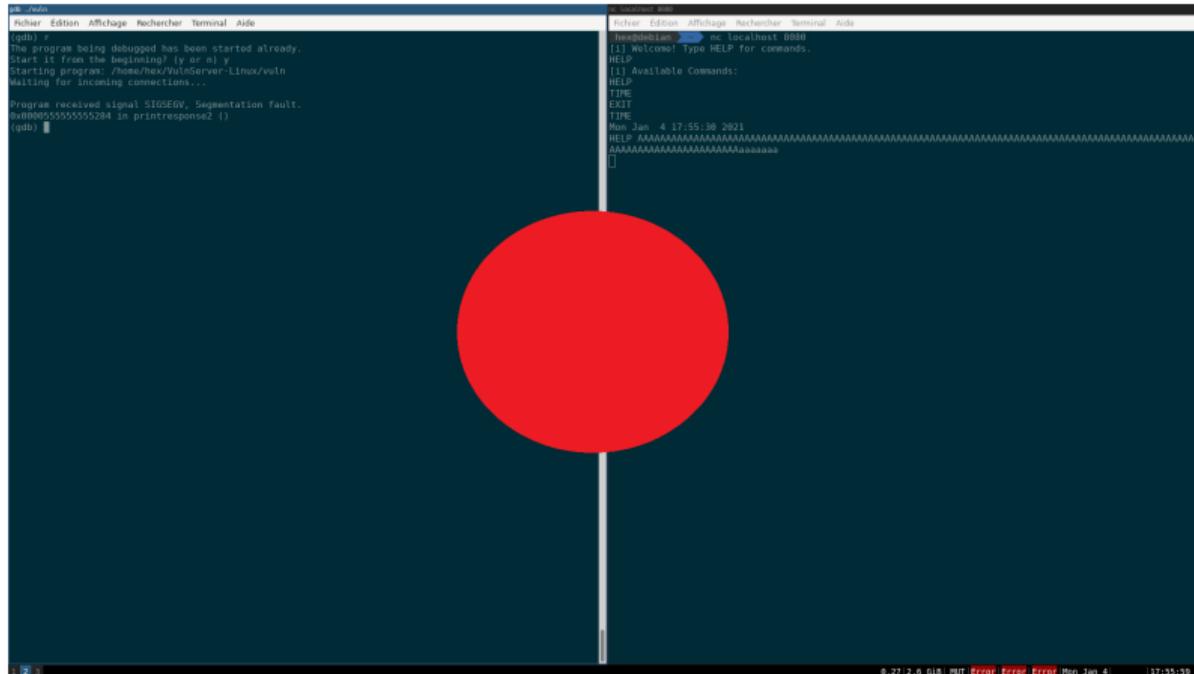


Figure 13: Do you wanna see an exploit ?

Buffer overflow: exploitation

A methodology exists to find and exploit a classic buffer overflow. This one consists in three parts : find, get control and exploit.

Here are the basics steps :

- Fuzzing
- Find a crash
- Find the EIP offset
- Jump to ESP
- Slide into the payload injection
- Get a shell

These steps will be covered in the next slides.

Fuzzing: explanations

Fuzzing is a testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. Using an automated fuzzer, or doing it manually, the goal is to find a crash while the program is running. After the crash has been found, a debugger should be used in order to identify where and when the program has crashed.

```
(gdb) r
Starting program: /home/hex/VulnServer-Linux/vuln
Waiting for incoming connections...

Program received signal SIGSEGV, Segmentation fault.
0x000055555555284 in printresponse2 ()
(gdb) ^C
```

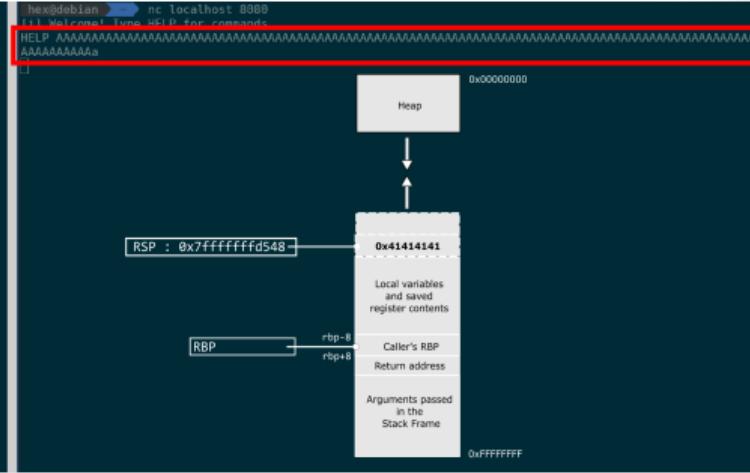
Figure 14: Typical buffer overflow crash

Remark

Boofuzz is a Python library which is really useful to automate fuzzing process for online targets.
Pwn library and random inputs can also be useful.

Fuzzing: The segmentation fault

```
[gdb] r
Starting program: /home/hex/VulnServer-Linux/vuln
Waiting for incoming connections...
Program received signal SIGSEGV, [Segmentation fault].
0xb000055555555284 in printresponse () {__PRETTY_FUNCTION__= "printresponse ()", __FUNCTION__= "printresponse", __LINE__= 1}
[gdbs] i r
rax          0x7fffffff5d00      146737488344326
rbx          0x0                   0
rcx          0xa614141        174145857
rdx          0x4                   4
r8           0x7fffffd5d0      146737488344528
r9           0x7fffffd5d0      146737488344432
rbp          0x4141414141414141 0x4141414141414141
rsp          0x7fffffd548        0x7fffffd548
r10          0x0                   0
r11          0x0                   0
r12          0x7fffffd5f15f0    -2561
r13          0x7ffff7f6ea60      140737353542240
r14          0x555555555150      9382499223586
r15          0x7fffffd2e250      146737488347280
rp           0x0                   0
fp           0x555555555284      0x555555555555284 <printresponse ()@146737488344326>
oflags        0x162802      [ IF RF ]
ls           0x33                  51
ss           0x2b                  43
ds           0x0                   0
ts           0x0                   0
fs           0x0                   0
os           0x0                   0
[gdbs] x/w $r0
$0 = 0x7fffffd548
[gdbs] x/d $r0
$0 = 1467374883444141
```



Controlling the execution flow: cyclic pattern

Now that we have found an entry point inside the program, the next step will be to redirect the execution flow of the program (EIP) where we want it.

First of all, we need to find and control EIP. For that a cyclic pattern can be used to determine the EIP's offset.

```
hex@debian ~ % pwn cyclic 100  
aaaaaaaaaaaaaaaaaaaaaaafaaaaahaaaaaaajaaakaaaalaaamaaaanaaaapaaaqaaaaraaas  
aaaaaaaaaaaaaaaawaaaaayaaaa  
hex@debian ~ %  
  
hex@debian ~ % pwn cyclic -l aabb  
101  
hex@debian ~ %
```

Figure 15: Cyclic pattern location

Once we have the offset, we can create a custom string using this length. Characters right after this string will be considered as the new return address.

Remark

pwn can be installed using python3-pip : **pip3 install pwntools**

Controlling the execution flow: controlling EIP

EIP is now under control.

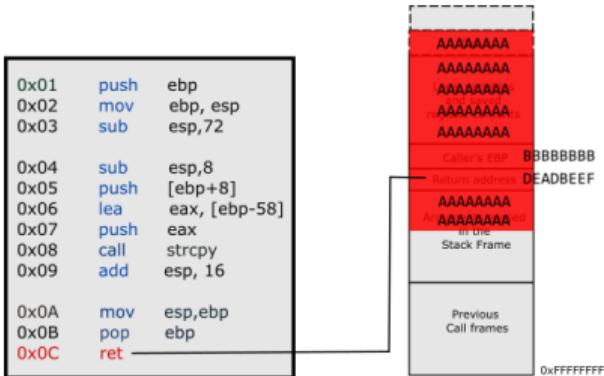


Figure 16: EIP points to 0xdeadbeef

Nevertheless, 0xdeadbeef is a false address. Let's find something we can really jump into.

Controlling the execution flow: jmp ESP

In our demonstration program, we are controlling ESP.

(you can see it using `x/xw $esp` in gdb-peda)

So we need to find a `jmp ESP` instruction inside the program.

Then, using this address we can redirect EIP to ESP thanks to this string:

"A"*offset + jmp esp address + "C"*a relatively big number

Remark

Using GDB-Peda you can type `jmpcall esp` to find one.

Controlling the execution flow: nop sliding

Let's talk about the most useful instruction in ASM.

The NOP (x90) instruction.

This instruction does nothing.

But, it's very useful to slide into another interesting place.

Controlling the execution flow: shellcode

At the end of the nop slide, you can insert a shellcode.

A shellcode is an ASM piece executing some useful functionalities such as a shell.

You can find a shellcode library here : <http://shell-storm.org/shellcode/>

Your string should look like this:

"A"*offset + jmp esp address + "NOP" slide + shellcode

Controlling the execution flow: shellcode

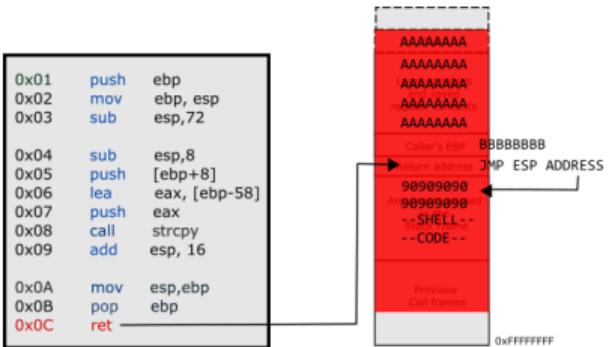


Figure 17: Sliding to shellcode

Once the shellcode has been read, it's executed.



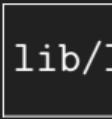
Binary exploitation: workshop

Workshop

- You will find the exercices on https://github.com/Hexliath/VA_BE
- Your report has to be sent to alexandre.thiroux@xx before the next supervised work. It will be marked (1 coefficient).

Questions ?

x



lib/logos/.png



Introduction to low-level software vulnerabilities

Lecture

Alexandre Thiroux¹

¹A company

January & February 2022

