CIT Cyber Security Cell

root@cit:# BootCamp Training

Day 6 : Rev & Binary Exploitation

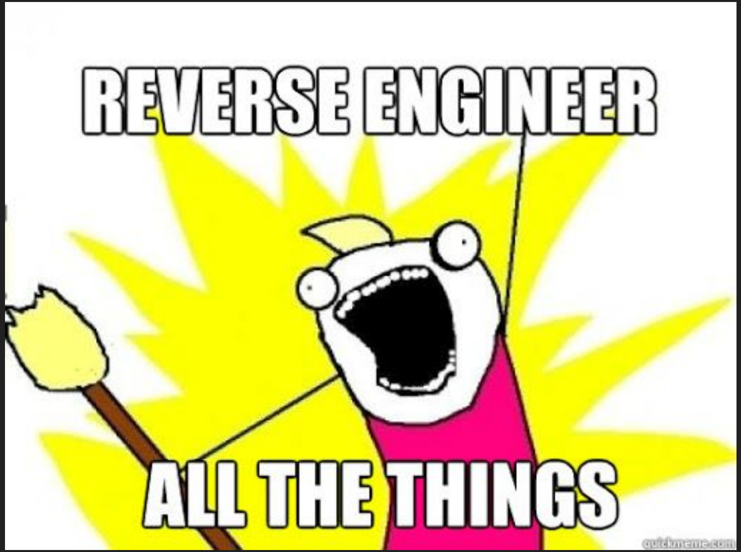
~ [CCSC] CIT Cyber Security Cell ~
OUSSAMA RAHALI
OMAR AOUAJ

CLUB INFORMATIQUE & TÉLÉCOM

cat README.md

Presentation outline

1. Reverse Engineering :
   1.1 Previously on CCSC
   1.2 Disassembling/Decompiling
   1.3 Why reversing (practical demo)
   1.4 Challenges
2. PWN = Binary Exploitation



Warning !

We are just beginners in reverse engineering and binary exploitation

So ... !! ;)

`Previously on CCSC`

Reverse Engineering is the process of decorticating an entity in general. In cybersecurity, it's analyzing what a program does by viewing its assembly code or source code.

Assembly code of a binary is the set of instructions composing it which the processor executes step by step.. It's a description of what actually happens inside the machine when a binary is executed (moving registers, adding one to another, xoring, checking if the value of a register is null…)

`Previously on CCSC`

```
┌─root@kali ~/Downloads
└─# cat script.c
#include <stdio.h>

void main(){
        puts("Hello CIT");
}#
```

```
 <main>:
55                      push   rbp
48 89 e5                mov    rbp,rsp
48 8d 3d c4 0e 00 00    lea    rdi,[rip+0xec4]        #
e8 eb fe ff ff          call   1030 <puts@plt>
90                      nop
5d                      pop    rbp
c3                      ret
0f 1f 84 00 00 00 00    nop    DWORD PTR [rax+rax*1+0x0]
00
```

when the processor receives 0x55,
it'll interpret it as `push rbp`

via objdump, gdb, rdr2, ida, ghidra...

Mostly, without knowing the source code, you need to know what the program does via its assembly code.

Difference between disassembling and decompiling

Disassembling:

compiled program -> machine code

tools: IDA, Ghidra, GDB, Binary Ninja, Radare2, Hopper…

Decompiling:

compiled program -> pseudo source code

tools: IDA, Ghidra…

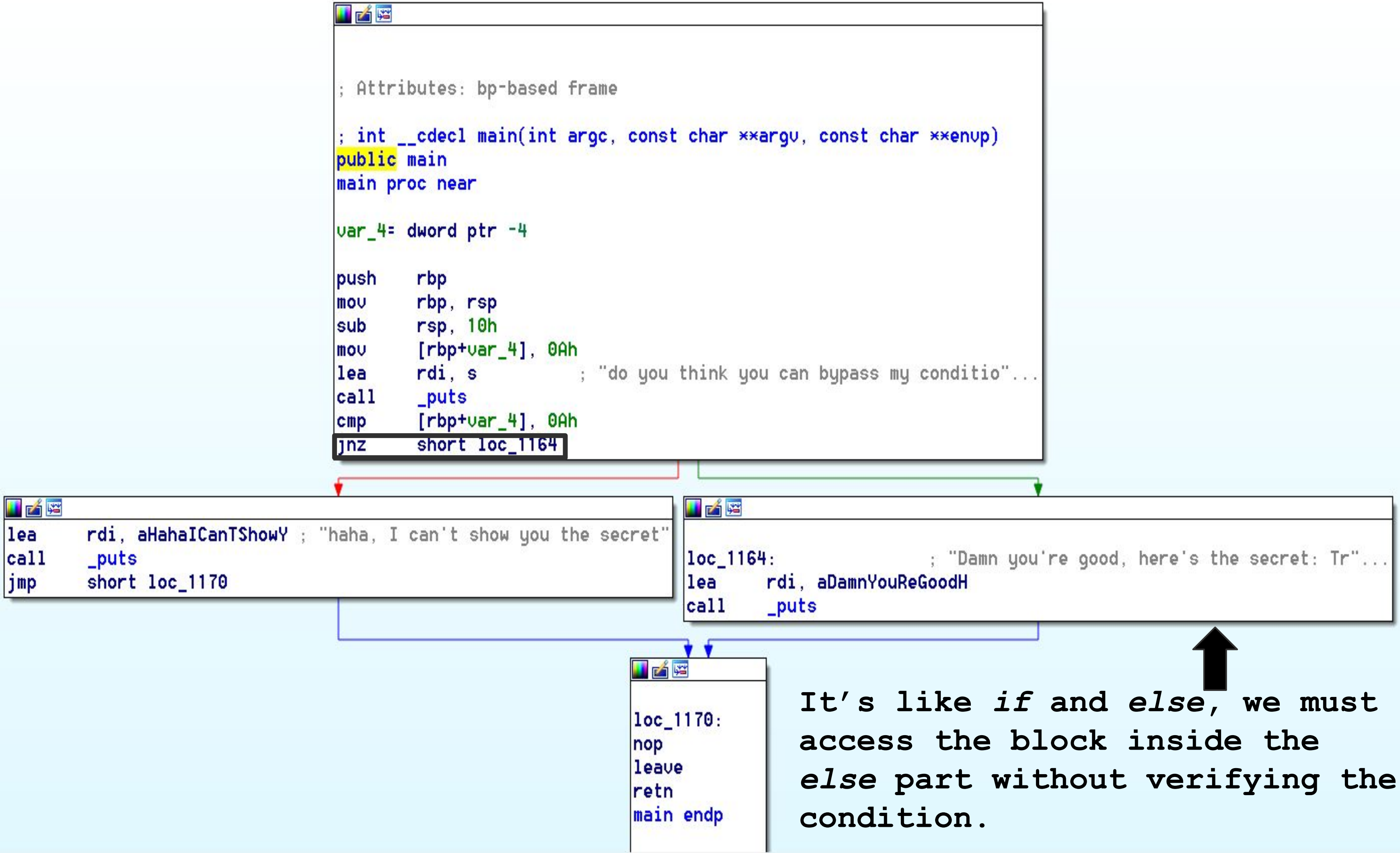
Why reversing?
Demo (control flow of a program)

Suppose we have a script, we don't know its source code.

```
┌─root@kali ~/Downloads
└─# gcc script.c -o script
┌─root@kali ~/Downloads
└─# ./script
do you think you can bypass my condition and get the secret??
haha, I can't show you the secret
```

Running `file` command on the binary reveals that it's compiled in a 64-bit architecture environment. We'll open it with IDA 64-bit.

```
┌─root@kali ~/Downloads
└─# file script
script: ELF 64-bit LSB shared object, x86-64
```



It's like *if* and *else*, we must access the block inside the *else* part without verifying the condition.

So let's open it now with gdb so that we can execute the program and follow its flow, we'll use gdb-peda which is an enhanced version of gdb.

```
┌─root@kali ~/Downloads
└─# gdb-peda script
Reading symbols from script...
(No debugging symbols found in script)
gdb-peda$ info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001030  puts@plt
0x0000000000001040  __cxa_finalize@plt
0x0000000000001050  _start
0x0000000000001080  deregister_tm_clones
0x00000000000010b0  register_tm_clones
0x00000000000010f0  __do_global_dtors_aux
0x0000000000001130  frame_dummy
0x0000000000001135  main
0x0000000000001180  __libc_csu_init
0x00000000000011e0  __libc_csu_fini
0x00000000000011e4  _fini
```

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000000000001135 <+0>:     push   rbp
   0x0000000000001136 <+1>:     mov    rbp,rsp
   0x0000000000001139 <+4>:     sub    rsp,0x10
   0x000000000000113d <+8>:     mov    DWORD PTR [rbp-0x4],0xa
   0x0000000000001144 <+15>:    lea    rdi,[rip+0xebd]        #
   0x000000000000114b <+22>:    call   0x1030 <puts@plt>
   0x0000000000001150 <+27>:    cmp    DWORD PTR [rbp-0x4],0xa
   0x0000000000001154 <+31>:    jne    0x1164 <main+47>
   0x0000000000001156 <+33>:    lea    rdi,[rip+0xeeb]        #
   0x000000000000115d <+40>:    call   0x1030 <puts@plt>
   0x0000000000001162 <+45>:    jmp    0x1170 <main+59>
   0x0000000000001164 <+47>:    lea    rdi,[rip+0xf05]        #
   0x000000000000116b <+54>:    call   0x1030 <puts@plt>
   0x0000000000001170 <+59>:    nop
   0x0000000000001171 <+60>:    leave
   0x0000000000001172 <+61>:    ret
End of assembler dump.
```

This is the part where it jumps to else block, it reveals its address. YET !!!!...

That address isn’t the real one since the program isn’t yet executed, so we have to do as the following:

We have to set a breakpoint at the main function before running the program (because we need to see the address while it’s being executed).

```
┌─root@kali ~/Downloads
└─# gdb-peda script
Reading symbols from script...
(No debugging symbols found in script)
gdb-peda$ b main
Breakpoint 1 at 0x1139
```

When we run the program, it will stop at the beginning of the main function and will tell us the values of all the registers, the most important one is RIP since it’s the instruction pointer register, so we have to find the real address of the else block.

```
gdb-peda$ r
Starting program: /root/Downloads/script
[----------------------------------registers-----------------------------------]
RAX: 0x555555555135 (<main>:	push   rbp)
RBX: 0x0
RCX: 0x7ffff7fa7718 --> 0x7ffff7fa9b00 --> 0x0
RDX: 0x7fffffffe1f8 --> 0x7fffffffe4fc ("SHELL=/bin/bash")
RSI: 0x7fffffffe1e8 --> 0x7fffffffe4e5 ("/root/Downloads/script")
RDI: 0x1
RBP: 0x7fffffffe0f0 --> 0x555555555180 (<__libc_csu_init>:	push   r15)
RSP: 0x7fffffffe0f0 --> 0x555555555180 (<__libc_csu_init>:	push   r15)
RIP: 0x555555555139 (<main+4>:	sub    rsp,0x10)
R8 : 0x0
R9 : 0x7ffff7fe2180 (<_dl_fini>:	push   rbp)
R10: 0x0
R11: 0x0
R12: 0x555555555050 (<_start>:	xor    ebp,ebp)
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
```

Disassembling the main function while the program is running will tell us what’s the effective address.

```
555555555136 <+1>:    mov    rbp,rsp
555555555139 <+4>:    sub    rsp,0x10
55555555513d <+8>:    mov    DWORD PTR [rbp-0x4],0xa
555555555144 <+15>:   lea    rdi,[rip+0xebd]        #
55555555514b <+22>:   call   0x555555555030 <puts@plt>
555555555150 <+27>:   cmp    DWORD PTR [rbp-0x4],0xa
555555555154 <+31>:   jne    0x555555555164 <main+47>
555555555156 <+33>:   lea    rdi,[rip+0xeeb]        #
55555555515d <+40>:   call   0x555555555030 <puts@plt>
555555555162 <+45>:   jmp    0x555555555170 <main+59>
555555555164 <+47>:   lea    rdi,[rip+0xf05]        #
55555555516b <+54>:   call   0x555555555030 <puts@plt>
555555555170 <+59>:   nop
```

Since we now know the address, there are two choices we can do:

- Using jump command:

```
gdb-peda$ j *0x555555555164
Continuing at 0x555555555164.
Damn you're good, here's the secret: TrueCCSCWarrior
```

- Affecting the value of the address to the RIP:

```
gdb-peda$ set $rip=0x555555555164
gdb-peda$ c
Continuing.
Damn you're good, here's the secret: TrueCCSCWarrior
```

Here's the script for better understanding, the else condition is impossible, yet we managed to control the program flow to execute the impossible part.

```c
#include <stdio.h>

void main(){
    int c=10;
    puts("do you think you can bypass my condition and get the secret??");
    if(c==10){
        puts("haha, I can't show you the secret");
    }
    else{
        puts("Damn you're good, here's the secret: TrueCCSCWarrior");
    }
}
```

Also, here's the pseudo-code which IDA gave us, so if you made this pseudo-code as a basis of your analysis, you'll never get to know how the program works.
As you can see there isn't even an if statement, so you won't understand what you should do precisely.

CONCLUSION: DO NOT REFER ONLY TO PSEUDO-CODE !!!

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  puts("do you think you can bypass my condition and get the secret??");
  return puts("haha, I can't show you the secret");
}
```

Get your hands dirty

Challenge 1 : Let's see if you can understand assembly!

>> this assembly code is of a function called cit, what does `cit(0x5ce)` return ?

the answer is a hexadecimal value.

Link : https://github.com/volck3r/CCSC_BootCamp_Training/blob/main/rev/challenge1/challenge1.txt

Get your hands dirty

Challenge 2 : decompiling is sometimes the key

>> someone gained access into our server, and used a ransomware to encrypt a very important file, we'll give you the program which he used and the encrypted text.. Can you know what did he do? If yes, can you decode it for us?

flag syntax: ccscCTF{}

https://github.com/volck3r/CCSC_BootCamp_Training/tree/main/rev/challenge2