Write-up

Shellmates Mini-CTF 2018 – SimpleCheck

Amina MEHERHERA
SHELLMATES MEMBER
fa_meherhera@esi.dz

Thanks

Special thanks to KIMOUCHE Mohamed and BOUTHIBA Abderraouf who organized this Mini-CTF and for all the help they gave us and what we learnt from them.

Thanks to ZOUAHI Hafidh and BALI Amina who helped me write this write-ups (my very first ones).

And of course, thanks to all Shellmates members (ntouma haylin[1] ☺).

[1] BALI Amina ☺

Challenge description

Title: SimpleCheck

Category: Reverse Engineering

Description: none

Points: ?

Flag Format: Shellmates{…}

Difficulty: Easy

Author: Raouf or Mohamed ?

Analysis

We are given an ELF 32-bit non-stripped[2] executable file SimpleCheck as the command `file SimpleCheck` shows:

```
mina@Mina:~/Desktop$ file SimpleCheck
SimpleCheck: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=b93d3f2f3bcbdc71f8d0dc279dda37653c18db56, not stripped
```

If we run it, it would ask for a second parameter (password).

```
mina@mina:~/Bureau$ ./SimpleCheck
[-] Usage : ./SimpleCheck <password>
```

If we try a random password, it will display : “never give up,, try harder!”

```
mina@mina:~/Bureau$ ./SimpleCheck password
[-] Never give up,, try harder !
```

First, let’s check the strings of the program:

```
mina@mina:~/Bureau$ strings ./SimpleCheck
/lib/ld-linux.so.2
=?/;
libc.so.6
_IO_stdin_used
puts
printf
__cxa_finalize
strcmp
__libc_start_main
GLIBC_2.1.3
GLIBC_2.0
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
Y[^]
UWVS
[^_]
[-] Usage : %s <password>
[+] Good job,, flag : Shellmates{%s}
[-] Never give up,, try harder !
;*2$"
```

Nothing seems interesting… ☹

[2] Non-stripped binaries have debugging information built into it (symbol table…) so we can find the functions names and other information. Whereas, stripped binaries remove this debugging information from the binary for example instead of finding the function’s name we’ll find its address.

Resolution

First Solution:

Let's try to display the calls to shared libraries functions using the command: ltrace ./SimpleCheck password

```
mina@Mina:~/Desktop$ ltrace ./SimpleCheck  password
__libc_start_main(0x565ab60d, 2, 0xffe5a954, 0x565ab6c0 <unfinished ...>
strcmp("password", "Simple_password")                                    = 1
puts("[-] Never give up,, try harder !"...[-] Never give up,, try harder !
)                                                 = 33
```

Interesting, we are comparing our input to "Simple_password" by calling the function **strcmp.** When we try it as password, it works.

```
[+] Good job,, flag : Shellmates{Simple_password}
```

Flag: Shellmates{Simple_password}

Second Solution:

Another way to solve the challenge is to use gdb peda to debug our program (gdb SimpleCheck).

When disassembling the main, we notice that we check whether we have two arguments, if it is the case we jump to <main+68>

```
gdb-peda$ pd main
Dump of assembler code for function main:
   0x0000060d <+0>:     lea    ecx,[esp+0x4]
   0x00000611 <+4>:     and    esp,0xfffffff0
   0x00000614 <+7>:     push   DWORD PTR [ecx-0x4]
   0x00000617 <+10>:    push   ebp
   0x00000618 <+11>:    mov    ebp,esp
   0x0000061a <+13>:    push   esi
   0x0000061b <+14>:    push   ebx
   0x0000061c <+15>:    push   ecx
   0x0000061d <+16>:    sub    esp,0xc
   0x00000620 <+19>:    call   0x480 <__x86.get_pc_thunk.bx>
   0x00000625 <+24>:    add    ebx,0x19db
   0x0000062b <+30>:    mov    esi,ecx
   0x0000062d <+32>:    cmp    DWORD PTR [esi],0x2
   0x00000630 <+35>:    je     0x651 <main+68>
```

So what is on <main+68>?:

```
   0x0000064a <+61>:    mov    eax,0xffffffff
   0x0000064f <+66>:    jmp    0x6a6 <main+153>
   0x00000651 <+68>:    call   0x57d <fixIt>
```

We are calling the function **fixIT.** We have to discover what this function does, so we'll disassemble it (pd fixIT):

```
gdb-peda$ pd fixIt
Dump of assembler code for function fixIt:
   0x0000057d <+0>:     push   ebp
   0x0000057e <+1>:     mov    ebp,esp
   0x00000580 <+3>:     push   ebx
   0x00000581 <+4>:     sub    esp,0x10
   0x00000584 <+7>:     call   0x6b1 <__x86.get_pc_thunk.ax>
   0x00000589 <+12>:    add    eax,0x1a77
   0x0000058e <+17>:    mov    DWORD PTR [ebp-0x8],0x0
   0x00000595 <+24>:    jmp    0x5bb <fixIt+62>
   0x00000597 <+26>:    lea    ecx,[eax+0x24]
   0x0000059d <+32>:    mov    edx,DWORD PTR [ebp-0x8]
   0x000005a0 <+35>:    add    edx,ecx
   0x000005a2 <+37>:    movzx  edx,BYTE PTR [edx]
   0x000005a5 <+40>:    sub    edx,0x1
   0x000005a8 <+43>:    mov    ebx,edx
   0x000005aa <+45>:    lea    ecx,[eax+0x24]
   0x000005b0 <+51>:    mov    edx,DWORD PTR [ebp-0x8]
   0x000005b3 <+54>:    add    edx,ecx
   0x000005b5 <+56>:    mov    BYTE PTR [edx],bl
   0x000005b7 <+58>:    add    DWORD PTR [ebp-0x8],0x1
   0x000005bb <+62>:    lea    ecx,[eax+0x24]
   0x000005c1 <+68>:    mov    edx,DWORD PTR [ebp-0x8]
   0x000005c4 <+71>:    add    edx,ecx
   0x000005c6 <+73>:    movzx  edx,BYTE PTR [edx]
   0x000005c9 <+76>:    test   dl,dl
   0x000005cb <+78>:    jne    0x597 <fixIt+26>
   0x000005cd <+80>:    mov    eax,0x1
   0x000005d2 <+85>:    add    esp,0x10
   0x000005d5 <+88>:    pop    ebx
   0x000005d6 <+89>:    pop    ebp
   0x000005d7 <+90>:    ret
```

We can notice that this function returns 1, we don't need to know more about it 😉.

Let's see what is after fixIT?

```
   0x00000651 <+68>:    call   0x57d <fixIt>
   0x00000656 <+73>:    test   eax,eax
   0x00000658 <+75>:    je     0x68f <main+130>
   0x0000065a <+77>:    mov    eax,DWORD PTR [esi+0x4]
   0x0000065d <+80>:    add    eax,0x4
   0x00000660 <+83>:    mov    eax,DWORD PTR [eax]
   0x00000662 <+85>:    sub    esp,0xc
   0x00000665 <+88>:    push   eax
   0x00000666 <+89>:    call   0x5d8 <checkIT>
   0x0000066b <+94>:    add    esp,0x10
   0x0000066e <+97>:    test   eax,eax
   0x00000670 <+99>:    je     0x68f <main+130>
   0x00000672 <+101>:   mov    eax,DWORD PTR [esi+0x4]
   0x00000675 <+104>:   add    eax,0x4
   0x00000678 <+107>:   mov    eax,DWORD PTR [eax]
```

We are checking whether eax=0 or not (we know that eax=1), if eax=0 it would jump to main+130 (which is not our case) so the jump is never taken.

We can notice then a call to the function **checkIT,** we certainly want to know what it hides (pd checkIT):

```
Dump of assembler code for function checkIT:
   0x000005d8 <+0>:     push   ebp
   0x000005d9 <+1>:     mov    ebp,esp
   0x000005db <+3>:     push   ebx
   0x000005dc <+4>:     sub    esp,0x4
   0x000005df <+7>:     call   0x6b1 <__x86.get_pc_thunk.ax>
   0x000005e4 <+12>:    add    eax,0x1a1c
   0x000005e9 <+17>:    sub    esp,0x8
   0x000005ec <+20>:    lea    edx,[eax+0x24]
   0x000005f2 <+26>:    push   edx
   0x000005f3 <+27>:    push   DWORD PTR [ebp+0x8]
   0x000005f6 <+30>:    mov    ebx,eax
   0x000005f8 <+32>:    call   0x3f0 <strcmp@plt>
   0x000005fd <+37>:    add    esp,0x10
   0x00000600 <+40>:    test   eax,eax
   0x00000602 <+42>:    sete   al
   0x00000605 <+45>:    movzx  eax,al
   0x00000608 <+48>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x0000060b <+51>:    leave
   0x0000060c <+52>:    ret
```

That's absolutely what we were looking for: a call to **strcmp** (the function that compares between two strings). Now, we make a breakpoint at the call in order to get the arguments (b* checkIT+32):

```
gdb-peda$ b* checkIT+32
Breakpoint 1 at 0x5f8
gdb-peda$ r password
Starting program: /home/mina/Desktop/SimpleCheck password

[----------------------------------registers-----------------------------------]
EAX: 0x56557000 --> 0x1efc
EBX: 0x56557000 --> 0x1efc
ECX: 0x56557024 ("Simple_password")
EDX: 0x56557024 ("Simple_password")
ESI: 0xffffcf40 --> 0x2
EDI: 0xf7fba000 --> 0x1b1db0
EBP: 0xffffcef8 --> 0xffffcf28 --> 0x0
ESP: 0xffffcee0 --> 0xffffd1fa ("password")
EIP: 0x565555f8 (<checkIT+32>:  call   0x565553f0 <strcmp@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565555f2 <checkIT+26>:     push   edx
   0x565555f3 <checkIT+27>:     push   DWORD PTR [ebp+0x8]
   0x565555f6 <checkIT+30>:     mov    ebx,eax
=> 0x565555f8 <checkIT+32>:     call   0x565553f0 <strcmp@plt>
   0x565555fd <checkIT+37>:     add    esp,0x10
   0x56555600 <checkIT+40>:     test   eax,eax
   0x56555602 <checkIT+42>:     sete   al
   0x56555605 <checkIT+45>:     movzx  eax,al
Guessed arguments:
arg[0]: 0xffffd1fa ("password")
arg[1]: 0x56557024 ("Simple_password")
```

Perfect, here we are, seems like our password is: "Simple_password" let's try it ☺ :

```
[+] Good job,, flag : Shellmates{Simple_password}
```

Flag: Shellmates{Simple_password}

What we learn from this task

We learned through this challenge the different commands that can help us in reversing binaries:

- `file filename`: to determine the file type[3].
- `strings filename`: to print the strings of printable characters in files[4]. This command was very useful in this challenge, we got the password easily.
- `ltrace executable_File parameters`: It intercepts and records the dynamic library calls which are called by the executed process and the signals which are received by that process. It can also **intercept and print the system calls** executed by the program[5]. In this challenge, we use could solve the challenge easily using this command (`ltrace ./SimpleCheck password`)
- Debugging a program using **gdb peda**:
  - PEDA (Python Exploit Development Assistance for GDB) enhance the display of gdb: colorize and display disassembly codes, registers, memory information during debugging. It adds commands to support debugging and exploit development too (for a full list of commands use peda help)[6].
  - The command `pd` (or `pdisas`) is a gdb disassemble command, the argument can be a function name (if the file is not stripped) like we did in this challenge (`pd main, pd fixIT, pd checkIT`) or we could rather use an address with the syntax `pd address /NN` (NN is the number of instructions we won't to disassemble).
  - The command `b*` (`breakpoint*`) is used to make program stop in certain points (breakpoints).
  - The command `r` (`run`) is used to start the program being debugged.

[3] man file
[4] man strings
[5] Man ltrace
[6] https://github.com/longld/peda

- The command `c` (`continue`) is used to continue running the program being debugged after a breakpoint.

We learned in this challenge, too, how to trace a program and move from a function to another until you find what interest you (what help you to get the flag).

Thanks for reading ☺