Write-up

Shellmates Mini-CTF 2018 – EASY

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-up (my very first one).

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

[1] BALI Amina ☺

Challenge description

Title: EASY

Category: Reverse Engineering

Description: none

Points: 50

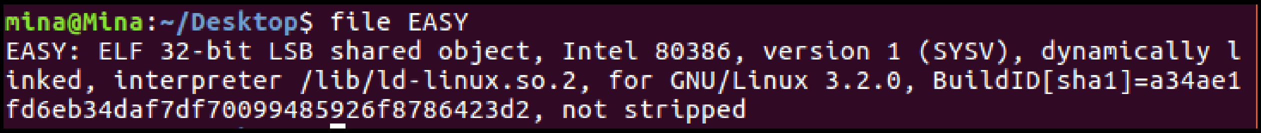
Flag Format: Shellmates{…}

Difficulty: Easy

Author: Raouf or Mohamed ?

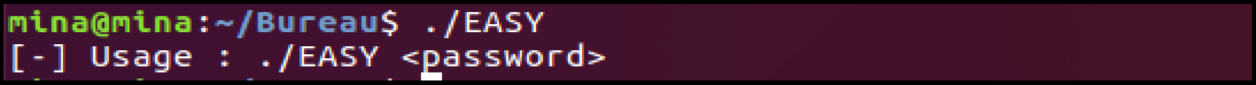
Analysis

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



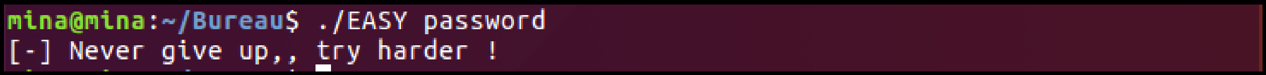
```
mina@Mina:~/Desktop$ file EASY
EASY: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically l
inked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=a34ae1
fd6eb34daf7df70099485926f8786423d2, not stripped
```

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



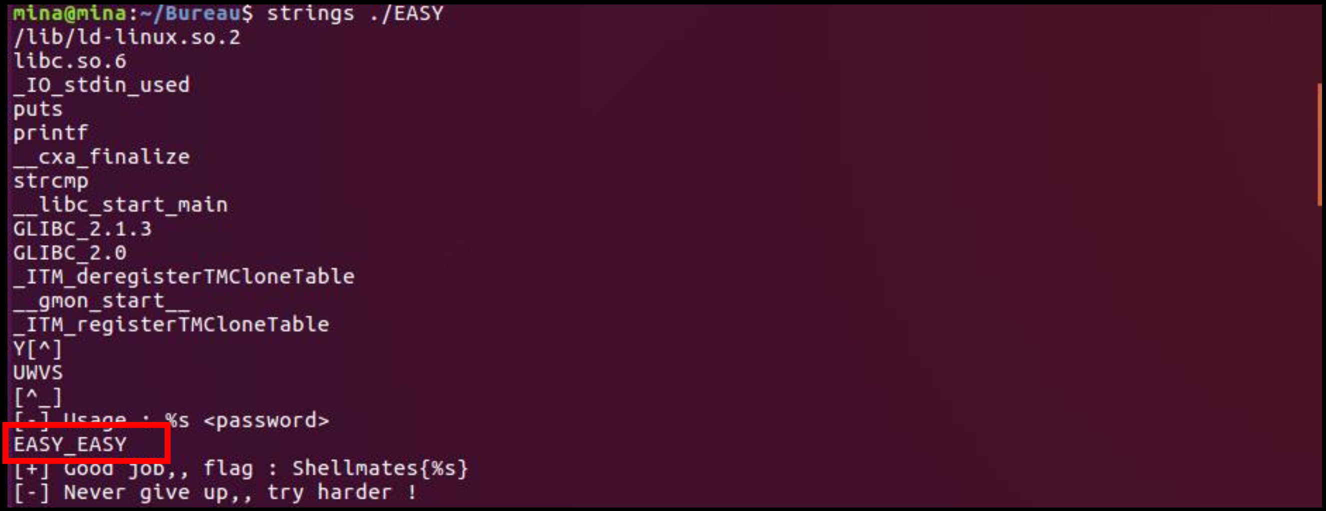
```
mina@mina:~/Bureau$ ./EASY
[-] Usage : ./EASY <password>
```

If we give a random password it would display "Never give up,, try harder!"



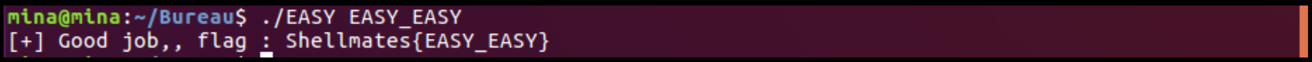
```
mina@mina:~/Bureau$ ./EASY password
[-] Never give up,, try harder !
```

Now, first let's check the list of strings in the program by executing the command: `strings EASY`



```
mina@mina:~/Bureau$ strings ./EASY
/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>
EASY_EASY
[+] Good job,, flag : Shellmates{%s}
[-] Never give up,, try harder !
```

we can notice a string called "EASY_EASY" let's try it:



```
mina@mina:~/Bureau$ ./EASY EASY_EASY
[+] Good job,, flag : Shellmates{EASY_EASY}
```

Oooh it works, "EASY" wasn't that hard ☺

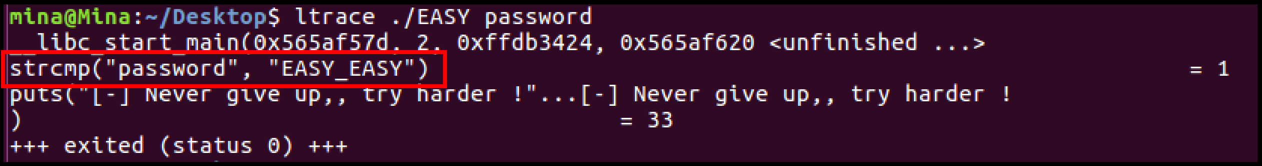
[2] Non-stripped binaries have debugging information built into it (symbol table…) so we can find the functions names and other information. Whereas, stripped binaries remove this debugging information from the binary for example instead of finding the function's name we'll find its address.

Resolution

Beside the first solution using the command strings there are other ways to solve the challenge, I will be talking about two of them:

Second Solution:

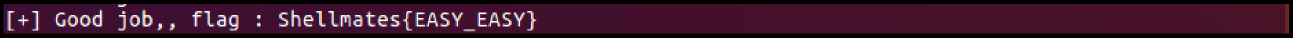
We can use the command ltrace to display the system calls:



```
mina@Mina:~/Desktop$ ltrace ./EASY password
__libc_start_main(0x565af57d, 2, 0xffdb3424, 0x565af620 <unfinished ...>
strcmp("password", "EASY_EASY")                                              = 1
puts("[-] Never give up,, try harder !"...[-] Never give up,, try harder !
)                                                         = 33
+++ exited (status 0) +++
```

We notice a call to strcmp that take our input and "EASY_EASY" as arguments.

Trying it confirm that we found the right password:

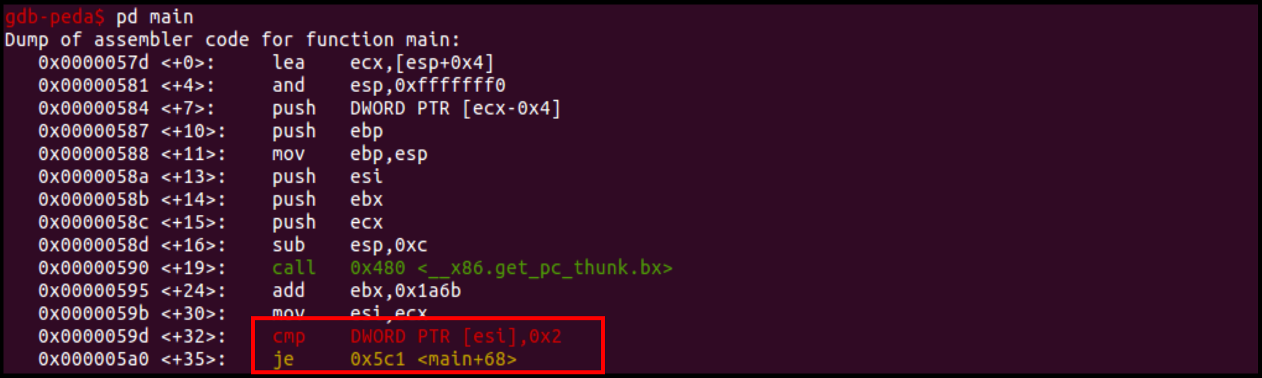


```
[+] Good job,, flag : Shellmates{EASY_EASY}
```

Third Solution:

It's about debugging the program using **gdb peda.** (gdb EASY)

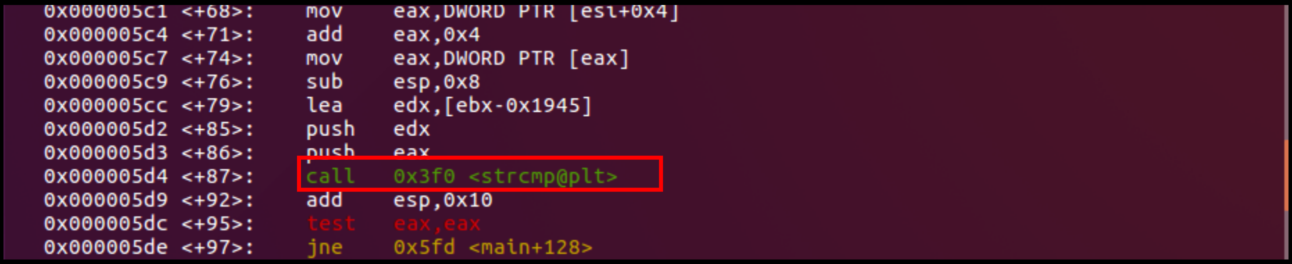
When disassembling the main (pd main) we can notice that the program first checks the number of parameters.



```
gdb-peda$ pd main
Dump of assembler code for function main:
   0x0000057d <+0>:     lea    ecx,[esp+0x4]
   0x00000581 <+4>:     and    esp,0xfffffff0
   0x00000584 <+7>:     push   DWORD PTR [ecx-0x4]
   0x00000587 <+10>:    push   ebp
   0x00000588 <+11>:    mov    ebp,esp
   0x0000058a <+13>:    push   esi
   0x0000058b <+14>:    push   ebx
   0x0000058c <+15>:    push   ecx
   0x0000058d <+16>:    sub    esp,0xc
   0x00000590 <+19>:    call   0x480 <__x86.get_pc_thunk.bx>
   0x00000595 <+24>:    add    ebx,0x1a6b
   0x0000059b <+30>:    mov    esi,ecx
   0x0000059d <+32>:    cmp    DWORD PTR [esi],0x2
   0x000005a0 <+35>:    je     0x5c1 <main+68>
```

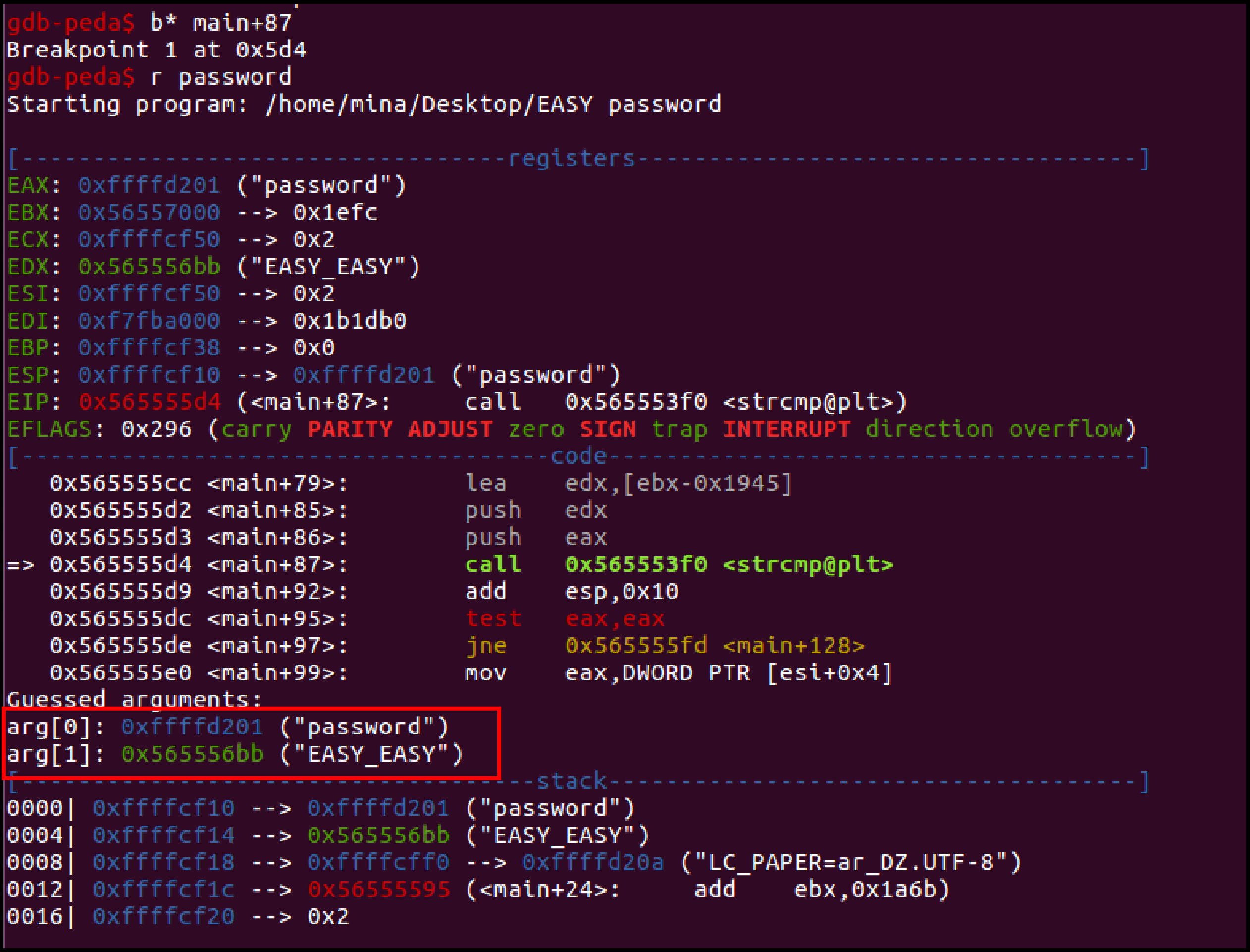
If we have 2 parameters, we jump to <main+68>

Let’s check what’s on <main+68>:


```
   0x000005c1 <+68>:    mov    eax,DWORD PTR [esi+0x4]
   0x000005c4 <+71>:    add    eax,0x4
   0x000005c7 <+74>:    mov    eax,DWORD PTR [eax]
   0x000005c9 <+76>:    sub    esp,0x8
   0x000005cc <+79>:    lea    edx,[ebx-0x1945]
   0x000005d2 <+85>:    push   edx
   0x000005d3 <+86>:    push   eax
   0x000005d4 <+87>:    call   0x3f0 <strcmp@plt>
   0x000005d9 <+92>:    add    esp,0x10
   0x000005dc <+95>:    test   eax,eax
   0x000005de <+97>:    jne    0x5fd <main+128>
```

We notice some instructions and then a call to the function **strcmp** which is used to compare two strings (that are in edx and eax in our case).

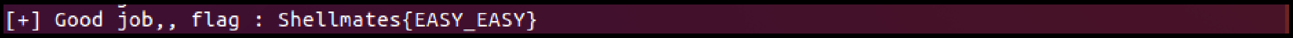
Let’s make a breakpoint in **strcmp** (b* main+87), run the program with a random password (r password) and check what happened:


```
gdb-peda$ b* main+87
Breakpoint 1 at 0x5d4
gdb-peda$ r password
Starting program: /home/mina/Desktop/EASY password

[----------------------------------registers-----------------------------------]
EAX: 0xffffd201 ("password")
EBX: 0x56557000 --> 0x1efc
ECX: 0xffffcf50 --> 0x2
EDX: 0x565556bb ("EASY_EASY")
ESI: 0xffffcf50 --> 0x2
EDI: 0xf7fba000 --> 0x1b1db0
EBP: 0xffffcf38 --> 0x0
ESP: 0xffffcf10 --> 0xffffd201 ("password")
EIP: 0x565555d4 (<main+87>:     call   0x565553f0 <strcmp@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565555cc <main+79>:        lea    edx,[ebx-0x1945]
   0x565555d2 <main+85>:        push   edx
   0x565555d3 <main+86>:        push   eax
=> 0x565555d4 <main+87>:        call   0x565553f0 <strcmp@plt>
   0x565555d9 <main+92>:        add    esp,0x10
   0x565555dc <main+95>:        test   eax,eax
   0x565555de <main+97>:        jne    0x565555fd <main+128>
   0x565555e0 <main+99>:        mov    eax,DWORD PTR [esi+0x4]
Guessed arguments:
arg[0]: 0xffffd201 ("password")
arg[1]: 0x565556bb ("EASY_EASY")
[------------------------------------stack-------------------------------------]
0000| 0xffffcf10 --> 0xffffd201 ("password")
0004| 0xffffcf14 --> 0x565556bb ("EASY_EASY")
0008| 0xffffcf18 --> 0xffffcff0 --> 0xffffd20a ("LC_PAPER=ar_DZ.UTF-8")
0012| 0xffffcf1c --> 0x56555595 (<main+24>:     add    ebx,0x1a6b)
0016| 0xffffcf20 --> 0x2
```

We have the two arguments of strcmp, we are comparing our input with "EASY_EASY"

let's try it:



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

Flag: Shellmates{EASY_EASY}

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) or an address using the syntax pd address /NN (NN is the number of instructions we won't to disassemble).

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

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

Thanks for reading ☺