Francisco Melendez Laureano 801-19-3512 francisco.melendez4@upr.edu

Prof. Rafael A. Arce

CCOM4702 Lab 02 - Program Analysis

1. Where do they live? PIE & NO-PIE

In this exercise we look over the differences of two executables, one that uses the *'Position Independent Executable'* (**PIE**) option and one that doesn't.

A program can be loaded and run at any memory address using the '*Position Independent Executable*' (**PIE**) file format.

Because relative addressing is used during compilation rather than absolute addressing, the code and data sections of an executable file in a **PIE** can be moved to different addresses in memory.

Because the attacker cannot rely on the code and data being loaded at known addresses in memory, this makes **PIE**s more resistant to some attacks, such as buffer overflow attacks…

After creating both executables with **-fPIC -no-pie** and **-pie** respectively , we immediately notice the differences with the file command:

live: *ELF 64-bit* ***LSB pie*** *executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=4a4b95442b6cd14f2a5689372eaf9ecb620ff937, for GNU/Linux 3.2.0, not stripped*

livenopie: *ELF 64-bit* ***LSB*** *executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=a323b6e78199e9d3dc9b5a4851fecfbd421c49c8, for GNU/Linux 3.2.0, not stripped*

PIE:

stack 0x7ffec0132644
stack in foo 0x7ffec0132624
heap 0: 0x55b893dff6b0
heap 1 : 0x55b893e01dd0
foo's address: 0x55b8923c21a9

NOPIE:
stack 0x7ffd30ef44e4
stack in foo 0x7ffd30ef44c4
heap 0: 0xe676b0
heap 1 : 0xe69dd0
foo's address: 0x401196

In the "**PIE**" program, the code and data sections of the executable file are compiled using relative addressing, allowing the program to be loaded at any memory address. As a result, the stack and heap addresses are different from those in the "**No-PIE**" program, and the address of the "**foo**" function is a higher memory address (*0x55b8923c21a9*) compared to the "**No-PIE**" program's (*0x401196*).

In the "**No-PIE**" program, the executable file uses absolute addressing, so the memory addresses are fixed and predictable. The stack and heap addresses are different from those in the "**PIE**" program, and the address of the "f**oo**" function is a lower memory address (*0x401196*) compared to the "**PIE**" program's (*0x55b8923c21a9*).

From a glance, these are the differences in the addresses of the function "**foo**" in the **PIE** and **No-PIE** versions:

PIE – 11a9 <foo>

NOPIE – 401196 <foo>

We make further observations by turning off the '*Address Space Layout Randomization*' (**ASLR**) in Linux with: echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

Afterwards we notice that running the pie or non-pie versions we see no changes relating to their addresses.

PIE:

stack 0x7fffffffdf24

stack in foo 0x7fffffffdf04

heap 0: 0x5555555596b0

heap 1 : 0x55555555bdd0

foo's address: 0x5555555551a9

No-PIE:

stack 0x7fffffffdf14

stack in foo 0x7fffffffdef4

heap 0: 0x4056b0

heap 1 : 0x407dd0

foo's address: 0x401196

Turning the **ASLR** back on we see the **PIE** 'randomizing' the addresses, including the function **foo**'s address:

stack 0x7ffcf1db6c84

stack in foo 0x7ffcf1db6c64

heap 0: 0x5580abca56b0

heap 1 : 0x5580abca7dd0

foo's address: 0x5580aa1031a9

Something similar happens on the No-Pie version, however foo's address stays consistent:

stack 0x7ffcade34a04

stack in foo 0x7ffcade349e4

heap 0: 0x7d06b0

heap 1 : 0x7d2dd0

foo's address: 0x401196

Type	Stack	Heap	Text
PIE	0x7ffcf1db6c84	0x55969f4aa6b0	Change
noPIE	0x7ffc3ff32c84	0x17716b0	Slight-Change

2. Stripped:

We shall now analyze a stripped executable, it has no symbols.

Utilizing readelf we notice the entry point:

Entry point address: **0x40010d**

At that entry point we see the instructions:

ba 0e 00 00 00 **mov** **edx,0xe**

That according to the Linux manual page:

sigprocmask() is used to fetch and/or change the signal mask of
the calling thread. The signal mask is the set of signals whose
delivery is currently blocked for the caller

We also notice that the program loops 5 times, if we were to change it to loop more times, we can edit the

b9 05 00 00 00 mov ecx,0x5

instruction to contain the amount of times we'd like for the loop to occur.

b9 0C 00 00 00 mov ecx,0x5 // loops 12 times.

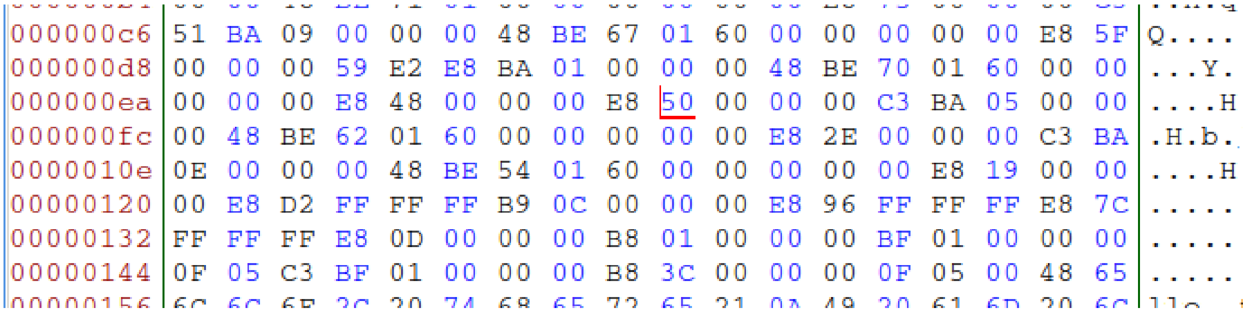
3. Stripped, Re-Loaded - After some observation, we notice that the string doesn't print, "*All done*!" as seen with a:

$ strings -t x stripped

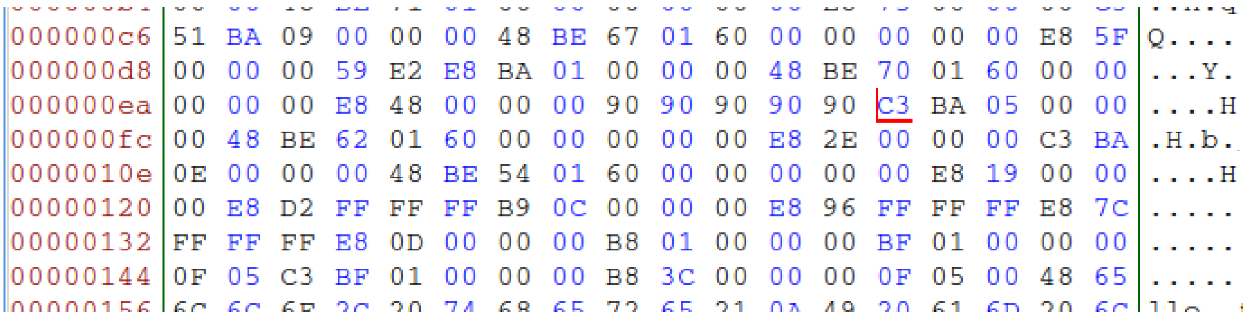
154 Hello, there!
162 I am looping,
171 All done!
17c .shstrtab
186 .text
18c .data

Some observations lead us to notice that the problem was that the program exited earlier before it could make the print, the exit was inside a function that called it after it finished looping.

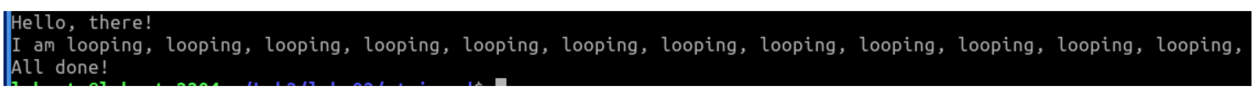
We simply modify it by rewriting the instruction with a series of **NOP**s (**0x90**).



Modified E8 50 00 00 00 -> 90 90 90 90 90



Intended Output:



4. Smash the Stack - In this problem, we work with a file with the following information:
level01: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped

An object dump reveals it to be a quite small ASM file:

level01:   file format elf32-i386

Disassembly of section .text:

```
08048080 <_start>:
 8048080:   68 28 91 04 08              push   0x8049128
 8048085:   e8 85 00 00 00              call   804810f <puts>
 804808a:   e8 10 00 00 00              call   804809f <fscanf>
 804808f:   3d 0f 01 00 00              cmp    eax,0x10f
 8048094:   0f 84 42 00 00 00     je    80480dc <YouWin>
 804809a:   e8 64 00 00 00              call   8048103 <exit>
```

However we see a function called firstly, **0x8049128**, which when we follow it with GDB we notice two comparisons being made:

```
[------------------------------------code-------------------------------------]
   0x80480ba:   xor    ebx,ebx
   0x80480bc:   xor    eax,eax
   0x80480be:   lods   al,BYTE PTR ds:[esi]
=> 0x80480bf:   cmp    al,0x20
   0x80480c1:   je     0x80480bc
   0x80480c3:   sub    al,0x30
   0x80480c5:   cmp    al,0x9
   0x80480c7:   ja     0x80480d3
```

cmp al, 0x20
cmp al, 0x9

And back in <start>

```
EFLAGS: 0x293 (CARRY parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804808a <_start+10>:       call   0x804809f
   0x804808f <_start+15>:       cmp    eax,0x10f
   0x8048094 <_start+20>:       je     0x80480dc
=> 0x804809a <_start+26>:       call   0x8048103
   0x804809f:   sub    esp,0x1000
   0x80480a5:   mov    eax,0x3
   0x80480aa:   mov    ebx,0x0
```

cmp eax, 0x10f

In decimal they're:

0x20 32

0x9 9

0x10f 271

After inputting the correct password we've:

```
kryozek@kry-ftp:Lab2 $ ./level01
Enter the 3 digit passcode to enter: 271 9 32
Congrats you found it, now read the password for level2 from /home/level2/.pass
sh-5.2$
```

5. GDB - Finally we evaluate a file called "**bash_login**". We are to force the program to run **<password_accepted>**.

Running the $ **objdump -s bash_login -M intel**

We notice that it was compiled using some Ubuntu libraries, as such it will not run on other non-Debian Linux Distribution such as my Fedora Linux.

"bash: ./bash_login: cannot execute: required file not found"

When we execute and analyze on Ubuntu we notice some comparisons that will lead us to execute **<password_accepted>** , however, we can *dodge* the deciphering of the conditions and make it so we still *jump* to the function regardless by changing the jump calls,

in,

8048629: eb 05 jmp 8048630 <main+0xb9>

We changed **eb 05 -> eb 00**

So it jumps to **<main + 180>** as long as the first condition is met. Inputting *323232* (a much smaller number will guarantee this condition to be met). Each offset is +5 bytes, so the technique is to find the offset of where the password_accepted call resides and figure out how to call it.

Change from:

—---

8048615: eb 19 jmp 8048630 <main+0xb9>

—---

Into:

—---

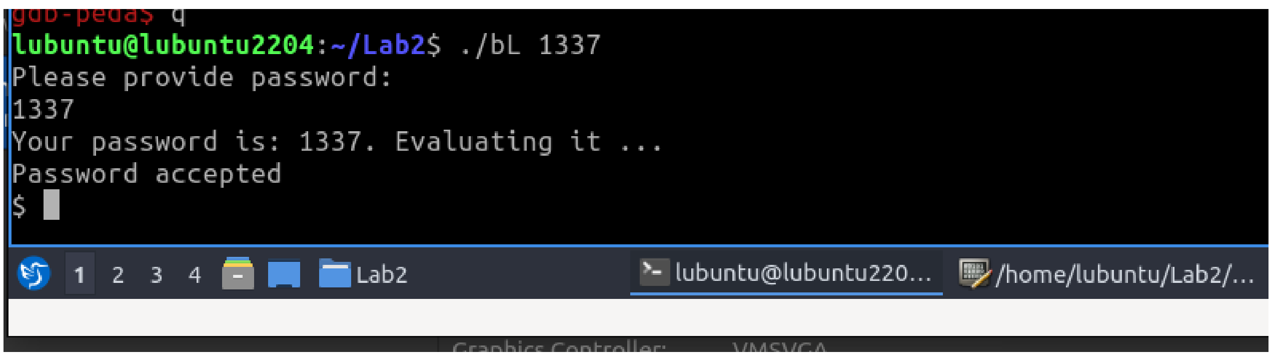
8048615: eb 14 jmp 804862b <main+0xb4>

—---

Utilizing bless…

```
00005c8|83 C4 10 83 EC 0C 6A 02 E8 0B FE FF FF 83 C4 10 8B 45 F0 83|......j....
00005dc|F8 63 7F 07 B8 FF FF FF FF EB 49 8B 4D F0 BA 4F EC C4 4E 89|.c........I
00005f0|C8 F7 EA C1 FA 02 89 C8 C1 F8 1F 29 C2 89 D0 01 C0 01 D0 C1|............
0000604|E0 02 01 D0 29 C1 89 CA 85 D2 74 07 B8 FF FF FF FF EB 14 8B|....).....t
0000618|45 F0 2D 38 01 00 00 83 F8 0B 76 07 B8 FF FF FF FF EB 00 E8|E.-8......v
000062c|1B FF FF FF 8B 4D F4 65 33 0D 14 00 00 00 74 05 E8 AF FD FF|.....M.e3..
0000640|FF 8B 4D FC C9 8D 61 FC C3 66 90 66 90 66 90 90 55 57 56 53|..M...a..f.:
0000654|E8 27 FE FF FF 81 C3 A7 19 00 00 83 EC 0C 8B 6C 24 20 8D B3|.'.........
0000668|0C FF FF FF E8 23 FD FF FF 8D 83 08 FF FF FF 29 C6 C1 FE 02|.....#.....
000067c|85 F6 74 25 31 FF 8D B6 00 00 00 00 83 EC 04 FF 74 24 2C FF|..t%1......
0000690|74 24 2C 55 FF 94 BB 08 FF FF FF 83 C7 01 83 C4 10 39 F7 75|t$,U.......
```

Expected Output:



This works by manipulating the offsets of the jumps, notice the

8048615: **eb 19** jmp 8048630 <**main+0xb9**>

```
-------------------------------------code-------------------------------------]
   0x8048622 <main+171>:        jbe    0x804862b <main+180>
   0x8048624 <main+173>:        mov    eax,0xffffffff
   0x8048629 <main+178>:        jmp    0x804862b <main+180>
=> 0x804862b <main+180>:        call   0x804854b <password_accepted>
   0x8048630 <main+185>:        mov    ecx,DWORD PTR [ebp-0xc]
   0x8048633 <main+188>:        xor    ecx,DWORD PTR gs:0x14
   0x804863a <main+195>:        je     0x8048641 <main+202>
   0x804863c <main+197>:        call   0x80483f0 <__stack_chk_fail@plt>
No argument
```

And voila! *(Do note we can also change conditions above though that'd require a bigger offset)*

🐧