3. Solutions

ASI36

2018

1 Basics (`basics.c`)

Assuming this program has been compiled to an executable named `p`.

1.1 Question 1

This program may have both expected and unexpected behaviors
The expected behaviors are:

1. Printing "Usage: p num1 num2" and exiting due to not enough arguments.

   This can be triggered with the following input `p 1`

2. Printing "You lose"

   Whenever `x == 0`, this will happen. If you only look at the syntactic level, this should be always, since there is no assignment to `x`. This is however not the case as we will see below.

   `p 1 2` will exhibit this behavior

Other behaviors that are "kind of" unexpected are the following:

1. Printing "You win".

   To print this message, one needs to overwrite the value of the local `x` with something other than `0`. Looking at the assembly (`objdump -d -M intel p`), we have the following initialization sequence :

```
59d:       c6 45 e3 00             mov    BYTE PTR [ebp-0x1d],0x0  ; x
5a1:       c7 45 db 00 00 00 00    mov    DWORD PTR [ebp-0x25],0x0 ; t[0-3]
5a8:       c7 45 df 00 00 00 00    mov    DWORD PTR [ebp-0x21],0x0 ; t[4-7]
```

   So `t` ends at `ebp - 0x21 + 0x4` and `x` is located at `ebp - 0x1d`. So there is a no gap between the end of `t` and `x`. If we can write 9 bytes from the start of `t`, we might rewrite `x` as well.

   The number of writes is controlled through `argv[2]`; what we write by `argv[1]`. We want to write something other than `0` (say 1).

   For example, `p 1 8` does that (`p 11 2222` as well).

2. Looping forever

3. crash

   The two behaviors below come from the same problem. It is also possible to overwrite `i`. On my machine it is at `ebp - 0x1c`, right above `x` on the stack — you can locate by putting a value at initialization in it if you wish.

   If you run `p 1 25` (whatever value greater than 9 instead of 25 and whatever value in [0..8] instead of 1 does it). You will loop forever. You can observe it in gdb

```
break main
watch i
continue
```

   You will see the value of `i` loop until `9` then come back to 2 since `t[10]` points to `i` and rewrites it with `1` in our case. Then it is incremented back to 2.

Now if the value you put instead of `1` does not make the index `i` come back to a range between [0..8] then you can either win – if you exit the loop with a value other than 0 for `x`, or provoke a segmentation fault if you jump above `x` and continue overwriting after `i`.

The former is achieved for example by `p 11 12`, the latter `p 14 68`. You may even print you win then provoke a segmentation fault: `p 14 50` does that.

1.2 Question 2

All the expected behaviors are still observable but not the others.

There are two reasons:

1. The order of the locals has been changed: the buffer is now above `x` and `i` and thus cannot change them anymore, e.g., to print "You win" or loop.

2. Only the potential of 3 remains but is replaced by the message whenever you overwrite the bounds of the stack frame.

```
*** stack smashing detected ***: <unknown> terminated
```

   The initialization is now:

```
61c:    8b 50 04                mov    edx,DWORD PTR [eax+0x4]
61f:    89 55 d4                mov    DWORD PTR [ebp-0x2c],edx
622:    65 8b 0d 14 00 00 00    mov    ecx,DWORD PTR gs:0x14   ; canary
629:    89 4d f4                mov    DWORD PTR [ebp-0xc],ecx
62c:    31 c9                   xor    ecx,ecx
62e:    c6 45 e7 00             mov    BYTE PTR [ebp-0x19],0x0 ;x
632:    c7 45 ec 00 00 00 00    mov    DWORD PTR [ebp-0x14],0x0 ; t[0-3]
639:    c7 45 f0 00 00 00 00    mov    DWORD PTR [ebp-0x10],0x0 ; t[4-7]
640:    c6 45 ec 30             mov    BYTE PTR [ebp-0x14],0x30
644:    c7 45 e8 01 00 00 00    mov    DWORD PTR [ebp-0x18],0x1; i
```

2 Take the heap (`h.c`)

The program has a potential vulnerability on the heap, since `p` and `p3` are both dynamically allocated. `f` seems to correctly check against `strcpy` manipulation but the real problem lies the handling of `p3`.

Now let's try to examine what happens right before the `scanf`. Any entry triggering `free(p)` suffices. For example `AAAABBBBCCCCDDDD` as `argv[1]` is long enough.

```
p p
p p3
```

If you add a break at the `scanf` and enter the two lines above, you will see that `p` and `p3` points to the same address: `malloc` has reused the freed space. It means that `system(p)` will execute whatever you enter. So now if you enter, say `fortune` or `sh`, you will execute this program.

3 Format-string exploitation (`fmt.c`)

This exercise is explained in the book "Hacking: the Art of Software Exploitation" in the relevant section about format string exploitation.