Stack

Alföldi Mátyás

June 25, 2020

Contents

stack0

This challange is relative simple, the following code is given:

```
int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];
  modified = 0;
  gets(buffer);

  if(modified != 0) {
    printf("you have changed the 'modified' variable\n");
  } else {
    printf("Try again?\n");
  }
}
```

I just gave the following input for this:
10xa 10xb 10xc 10xd 10xe 10xf 10xg (So basically something that is longer than 64)

stack1

Source code:
https://web.archive.org/web/20170419031559/https://exploit-exercises.com/protostar/stack1/
This time the argument has to be longer than 64, and 0x61626364 has to go into the variable, which is the ascii code for abcd, but because of the endianness has to be reversed.

stack2

Source code:
https://web.archive.org/web/20170419023252/https://exploit-exercises.com/protostar/stack2/
Answer:
Same as above but \n \r \n \r has to be used.

stack3

Source code:
https://web.archive.org/web/20170417130221/https://exploit-exercises.com/protostar/stack3/
For this one we use objdump -d to get the address of the win function, which seems to be 08048424.

Using /bin/echo -e -n '64 filler chars\x24\x84\x04\x08' | ./stack3 does the work

stack4

Source code:
https://web.archive.org/web/20170417130121/https://exploit-exercises.com/protostar/stack4/
During the call to main the location to where the ret should return is pushed into the stack, so after checking with objdump the address of the win function a long enough input has to be generated so that we overwrite that.
win is at: 080483f4
Looking at main the following instructions modify esp:

- push ebp
- and esp, 0xfffffff0
- sub esp, 0x50

And the following changes how long our filler string has to be:

- lea eax,[esp+0x10]
- mov DWORD PTR [esp],eax

This way esp+0x10 gets passed to the gets function, and this means that the buffer is 64 long.
The push ebp modifies esp by 4, and the and esp, 0xfffffff0 by a max of 15, so 64 + 15 + 4 should be the max len needed.
So tests should be done with filler string len 68-84
In the end 76 x a and 080484f4 reversed for the endiannes(in hex) was the amount needed to change where the ret brings us.
(Used /bin/echo -e -n for writing the hex codes into the file)

stack5

Source code:
https://web.archive.org/web/20170419023355/https://exploit-exercises.com/protostar/stack5/
This time we have to change where the ret brings us like before, but we also need to bring our shellcode to the stack, and somehow need to know where the address of it will be, to change where the ret brings us to that.
The shellcode of our choice for the beginning 0xcc(int 3)
Idea 1: after main-s ret there are 5 nop bytes, bringing execution to the first would be simple, only we would have to somehow overwrite that location.(And patching the location wouldn't be using shellcode.) Idea 2: Check

with gdb what the address for the ret is in the stack, overwrite that with its address+4, so that it will jump right after it, and place the shellcode there. (This will only work though, because aslr is disabled, and non executable locations, since the stack is usually non executable.)
Decision: going with Idea 2
We need 68-84 filler chars bffffd30 in hex, and also reversed and 0xccccccccc. 76 filler chars are needed, afterwards 30fdffbfcccccccc (in hex) and if we use gdb stack5, then run <inputFile it stops at our cc.

stack6

Source code:
https://web.archive.org/web/20140405142902/http://exploit-exercises.com/protostar/stack6
Here the return address, can't start with 0xbf(I guess it is the stack), so it has to be solved in another way.
Idea 1 (Very simple example of return oriented programming): After the filler string modify the ret value to point to itself, then the next value will point to after itself, where the shell code will be located(int 3).
Idea 2 (ret2libc): Because there is no aslr, I could also use print system, and search where I find the string /bin/sh or /bin/bash in memory, but this wouldn't work with aslr enabled.
(print system outputs 0xb7ecffb0, and find for /bin/sh gives us 0xb7fba23f)
So we have to place 0xb7ecffb0 first after the filler chars, then a value to which system should return, and lastly 0xb7fba23f