CA647 Secure Programming Lab 06

Darragh O'Brien B.Sc. Ph.D. FHEA
School of Computing
Dublin City University

¶

2023-24

Overview

In this lab we will:

- Derive a hexadecimal representation of some shellcode suitable for use in a buffer overflow attack
- Test the derived shellcode
- Use the shellcode in a buffer overflow attack
- Derive some code which, rather than launching a shell, causes the file named *abc* to be renamed to *def*

A. Deriving shellcode

Having studied the lecture notes you are now in a position to produce some shellcode of your own. I suggest you adopt the following approach:

- Write a simple C program called sc.c which calls `execve` to launch a shell. Compile and execute the program to verify it does indeed launch a shell.
- Now disassemble, using `gdb`, the `execve` function inside the program. Verify the assembly instructions match (approximately) those presented in the lecture notes.
- Study the assembly and work out what you need to do (in terms of register contents a software interrupt) in order to invoke the `execve` system call directly.
- Write a new C program called dummy.c that simply calls an empty function which is of type `void` and takes no parameters.
- Use `gcc -S` to convert `dummy.c` into assembly code to give you something like dummy.s.
- Now in `dummy.s` you have a framework in which you can embed your assembly instructions. (Your code should go between function's prologue and epilogue which have already been generated by the compiler.)
- Add required data i.e. strings.
- Add labels, `jmp` and `call` instructions.
- Add null-terminating instructions if required.

- When finished you should have something like this.
- Build an executable from `sc01.s`.
- Use `gdb` to convert your assembly instructions to a hexadecimal representation.
- Check your hexdump for nulls and rewrite any instructions that generate nulls.
- When finished you should have something like this.
- Again use `gdb` to convert your assembly instructions to a hexadecimal representation. Verify the hexdump contains no nulls.

B. Testing your shellcode

Try compiling and running your final assembly program. What happens? Why? It's because the code attempts to modify itself i.e. the write that null-terminates the string is carried out in the `.text` section which is read-only, executable. Write permission is absent so a segmentation fault occurs. Simply comment out the line that null-terminates the string, rebuild the executable and it should run fine (and launch a shell).

If you want to see your code run on the stack then you can try it this way. If this does not work, the reason is probably that the kernel you are using is taking advantage of hardware NX (No eXecute) support to prevent code from running on the stack. We need to disable this feature in order to test the code. Here is one way to do it:

- Take a copy of xstack.c. Compile it to an an executable. Run it against the shelltest executable. It manually enables execute permission on the stack by modifying the executable's ELF header. You can then run the program and it will not segfault.

C. Using shellcode in a buffer overflow attack

Take a copy of vul.c. Build a debug executable with:

```
$ gcc -g -mpreferred-stack-boundary=2 -o vul vul.c
```

Follow the advice given above to ensure that the `vul` program runs with executable stack.

Examine the source code to determine where the vulnerability lies. Here is a program exploit.c you can use to exploit the buffer overflow in `vul` to cause it to execute the shellcode you derived above. (You will most likely have to disable stack randomisation in order to get your attack to work.)

The exploit program takes two arguments, a buffer size and an offset. It determines its own stack start address and subtracts the supplied offset from it to generate a candidate return address for the vulnerable program. It launches the vulnerable program passing it our malicious shellcode string with a call to `execlp`.

```
$ ./exploit 1032 1800
Stack: 0xbfffef98
Using address 0xbfffe890
Sending in buffer length: 1032
```

```
Vulnerable stack top at 0xbfffebd4
Vulnerable buffer at 0xbfffe7d4
sh-3.2$
```

D. Deriving rename code

Following the same steps as those required to derive your shellcode above, derive a hexadecimal string which when injected into a vulnerable process causes the file named *abc* to be renamed *def*. You need to know the `rename` system call number, it is 38 (decimal). For future reference you can consult the following system call list.