# CSC Winter 2020 Homework 6
# Buffer overflow attacks

### Due: as specified on D2L

---

## Grading

This homework is will **not** be graded on a scale of 0-10. Instead, it will be worth up to 4 extra credit points (2 points for each of the 2 levels). The extra credit will then be added to the homeworks portion of your score.

## Summary

Over the history of computers that have used the modern-day program stack, these computers have been vulnerable to attack from malicious programmers. In one sort of attack, the program stack is corrupted (the book emphasizes how this can happen with buffer overflows), and therefore control of the program can be altered by overwriting the return address of a function, which is of course stored on the stack.

---

## Lab Exercise

The lab worth up to 2 point of extra credit of your overall grade, for each of the "levels" in the lab. (for a total of 4 points extra credit).

On D2L, you will find a Homework 6 submissions folder. It contains this write-up plus a file called `hw6.tar`. The tar file contains a C file called echo.c and an executable `hex2raw`. The C file should be compiled as it is, without adding any code to it. Optimization level –O1 should be used.

In addition to `main` and `echo` functions, `echo.c` contains functions called `level1` and `level2`. While `main` explicitly calls `echo`, the other functions are not explicitly called. Your task in this lab is to devise input which, when fed into a buffer, overflows the buffer in such a way that either `level1 or level2` is called. Details of each level are described below.

---

## Level 1

This level more or less duplicates what we did in lecture today. Here is the relevant code:

```
#include <stdio.h>

char *echo() {
  printf(“Type something\n”);
  char buffer[32];
  return gets(buffer);
}


int main() {
  echo();
}

int level1() {
  printf("Made it to level 1\n");
}
```

Again, while nothing in the code explicitly calls level1, it is possible to corrupt the stack in such a way as to call level1 (although the program will then crash). Once you have properly completed this level, the output you should see is

[slytinen@cdmlinux hw6]$ ./lab < level1.bin
Type something
Made it to level 1
Segmentation fault

---

## Level 2

Since parameters are generally passed through registers on 64-bit machines, it is somewhat difficult to corrupt them. However, this will be our goal in level2. Here is the prototype for the function you should cause to be called:

```
void level2(int x);
```

We would like to pass the value 0xff as the value of the parameter x (no matter what the programmer actually passed as the parameter). However, since the first parameter of a function is passed in register %rsi, we must do something more devious in order to change the value of that register (since it is not on the program stack).

Specifically, we will look for another part of the application which does what we want it do: namely, it copies 0xff into register `%rsi`. I have placed a function in echo.c which does what we want:

```
int helper() {
  printf("You made it to helper\n");
  return junk(0xff);
```

```
}

int junk(int x) {
  return x & 0x101;
}

int level2(int x) {
  if (x == 0xff) {
    printf("Made it to level 2 with x == 0xff!\n");
    exit(0);}
  else {
    printf("Made it to level2, but with x = %d :-(\n", x);
  }
}
```

Helper moves the value 0xff into register %rsi, since it is passed as a parameter. Therefore, if we execute `helper`, then our desired number is in the correct register. We then need to further corrupt the stack so that the program "returns" to level2, which prints a message that x has the correct value (and then crashes again).

Once you have completed this, the output should be

Once you have properly completed this level, the output you should see is

```
[slytinen@cdmlinux hw6]$ ./echo < level2.bin
Type something
You made it to helper
Made it to level 2 with x = 0xff
Segmentation fault
```

Please turn in (a) a screenshot of executing your lab on the 2 different levels; and (b) the .hex files that you used to produce the outputs.