# Lab1 Report

John Dale, Dani Kasti, Thomas Greeley

Lab1 simulated a buffer overflow attack on a Nios II processor. To simulate this attack, the Quartus Prime Lite software was used to design hardware for the Nios II FPGA board. Altera Quartus Prime allows circuits to be designed on the Altera FPGA boards, assisting in the design, simulation, synthesis, and downloading of the bitstream. Additionally, it contains a tool, QSYS, that adds hardware peripherals to the board, such as ethernet and serial ports. QSYS was important in connecting the NIOS II/e processor, JTAG UART, SDRAM Controller, and other hardware components together. The Eclipse software was used to develop and run our C program on the Nios II processor located on the Altera FPGA board. The program we ran on the Nios II demonstrated buffer overflow, as explained below.

A buffer is a region in computer memory that temporarily stores data, including registers, the stack, and variables defined in a program. Buffer overflow is the result of a buffer's predefined size being violated, resulting in the overwriting of other parameters. Buffer overflow can be a malicious attack. If the predefined size of the buffer is violated, attackers can load malicious code into the buffer and redirect the return address. This modification of the stack pointer can alter the program routine. This can allow the malicious adversary to overwrite parameters and inject executable code.

Buffer overflow attacks can be executed through many channels and vulnerabilities, one example is with Smart Cards that use ISO 7816-3 T=0 protocol exchange. This is how the data exchange works :

Reader -> card: CLA INS P1 P2 LEN

Card -> reader: INS

Card <-> reader: LEN data bytes …

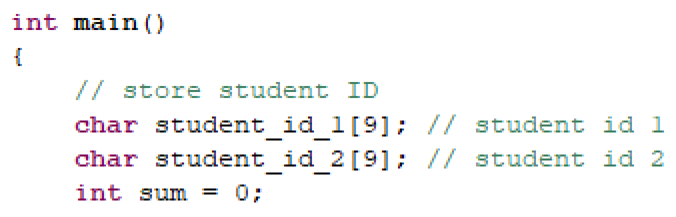
Card -> reader: 90 00

In the communication between the card and the card reader, the LEN value can be used to cause a buffer overflow. The card reader protocol explicitly trusts the length of the LEN value, and an attacker can thus provide a longer value than expected and cause a buffer overflow. This can give the malicious adversary access to RAM content after the resulting buffer, for example, the memory that contains data on secret keys. Other types of buffer overflow attacks are Stack

Overflow or Stack smashing. The Stack contains the return address for active function calls; a Stack Overflow attack is performed by overflowing the Stack, causing the code to derail and could be used to run malicious code in a different area of the computer. Stack smashing is when the Stack is overflown, causing the program to derail and crash.

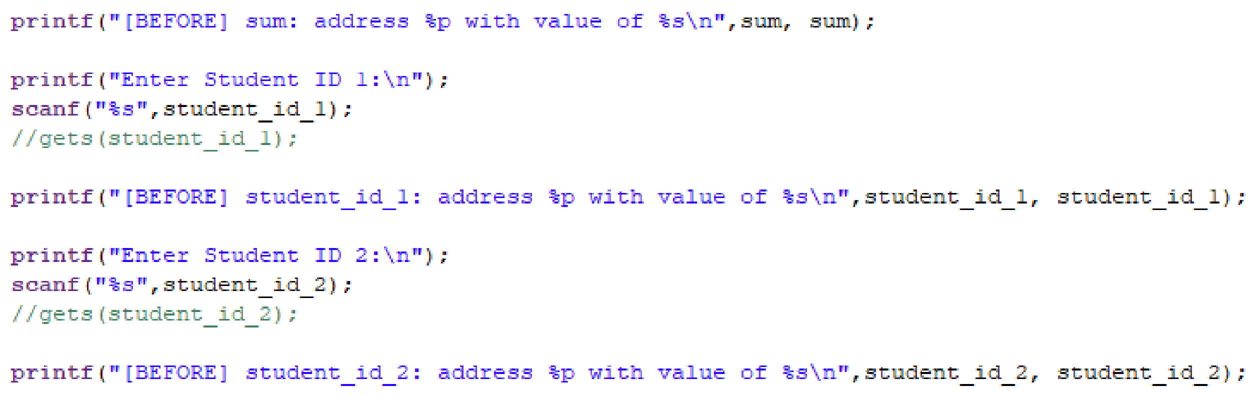
Buffer overflow can be prevented through multiple methods. One solution is avoiding standard library functions, such as gets() and strcpy(), which are not bound-checked. Higher-level programming languages, such as Java, C#, or Python, can also solve this issue, as these languages have bound checking built into them. Another solution is to use a modern operating system that checks for memory violations by checking the size of the arrays stored in the buffer during runtime. This prevents buffer overflow attacks. Finally, another method would be configuring the memory management unit to disable code execution on the stack.

To demonstrate buffer overflow in Lab1, we initialized two char arrays (buffers) to a size of 9 bytes in the main function of our code. Each array has 8 bytes reserved for the student ID, and 1 byte reserved for the null character (\0).



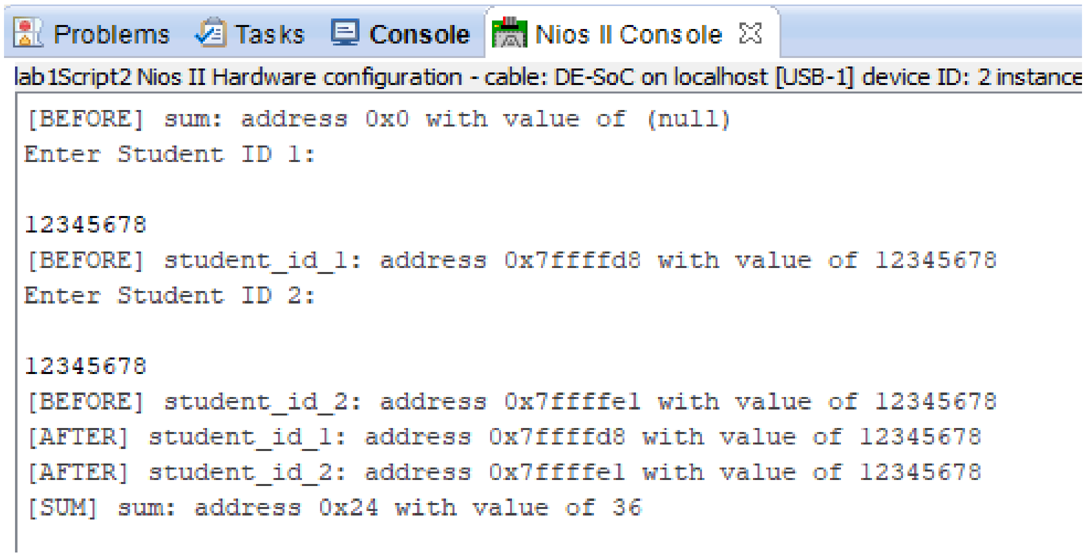
```c
int main()
{
    // store student ID
    char student_id_1[9]; // student id 1
    char student_id_2[9]; // student id 2
    int sum = 0;
```

We prompt the user to enter student_id_1 and student_id_2 , and the library function scanf reads each character and stores it into the student_id_1 and student_id_2 buffer.



```c
printf("[BEFORE] sum: address %p with value of %s\n",sum, sum);

printf("Enter Student ID 1:\n");
scanf("%s",student_id_1);
//gets(student_id_1);

printf("[BEFORE] student_id_1: address %p with value of %s\n",student_id_1, student_id_1);

printf("Enter Student ID 2:\n");
scanf("%s",student_id_2);
//gets(student_id_2);

printf("[BEFORE] student_id_2: address %p with value of %s\n",student_id_2, student_id_2);
```

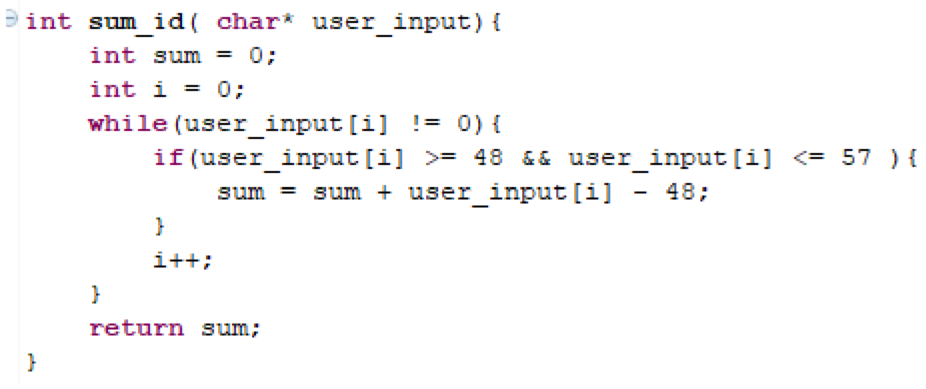
If we do as the program expects and enter a 8 digit student_id number, the console prints out the correct student_id values and sum.



```
lab1Script2 Nios II Hardware configuration - cable: DE-SoC on localhost [USB-1] device ID: 2 instance
[BEFORE] sum: address 0x0 with value of (null)
Enter Student ID 1:

12345678
[BEFORE] student_id_1: address 0x7ffffd8 with value of 12345678
Enter Student ID 2:

12345678
[BEFORE] student_id_2: address 0x7ffffe1 with value of 12345678
[AFTER] student_id_1: address 0x7ffffd8 with value of 12345678
[AFTER] student_id_2: address 0x7ffffe1 with value of 12345678
[SUM] sum: address 0x24 with value of 36
```

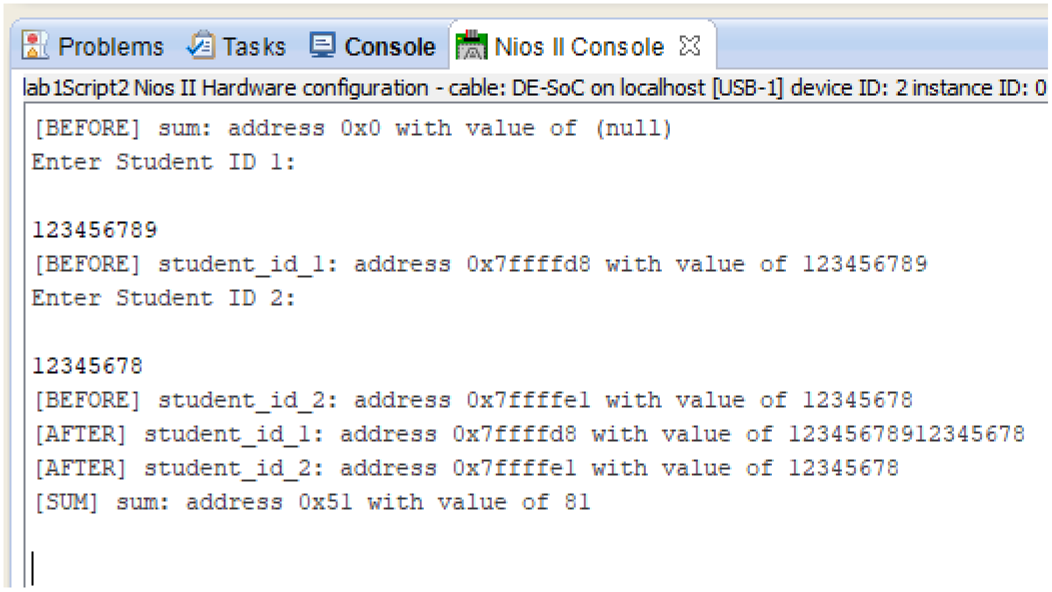
(Sum is correct (8+7+6+5+4+3+2+1 = 36), and both student_ids are the same as entered)

The student ids are summed through a function call to sum_id, which begins by initializing the sum to 0, and an iterative variable i is set to 0. Then a while loop is initialized that reads the array located at the pointer user_input. The while loop checks that the byte is not equal to the null character; if it is not, we then check that its ASCII value is an integer. If the byte contains an ASCII value representing an integer, we convert the ASCII value to an integer value and add it to the sum. We then iterate to the next byte and continue the while loop until a null character is found. Then the while loop ends, and the sum is returned.



```
int sum_id( char* user_input){
    int sum = 0;
    int i = 0;
    while(user_input[i] != 0){
        if(user_input[i] >= 48 && user_input[i] <= 57 ){
            sum = sum + user_input[i] - 48;
        }
        i++;
    }
    return sum;
}
```

To overflow the buffer, we enter nine numbers for the student_id_1. The main function then prompts the user to input student_id_2. The scanf library function reads and stores the values entered into the student_id_2 buffer. When we print out the contents of the student_id_1 and student_id_2 buffers, the result is an incorrect student_id_1 value. This is a result of buffer overflow.



```
lab1Script2 Nios II Hardware configuration - cable: DE-SoC on localhost [USB-1] device ID: 2 instance ID: 0
[BEFORE] sum: address 0x0 with value of (null)
Enter Student ID 1:

123456789
[BEFORE] student_id_1: address 0x7ffffd8 with value of 123456789
Enter Student ID 2:

12345678
[BEFORE] student_id_2: address 0x7fffe1 with value of 12345678
[AFTER] student_id_1: address 0x7ffffd8 with value of 12345678912345678
[AFTER] student_id_2: address 0x7fffe1 with value of 12345678
[SUM] sum: address 0x51 with value of 81
```

(Sum is incorrect (should only sum student_id_1 value), and the printed student_id_1 value is not the same as was entered)

As shown above, the buffer value is not the same as the value entered by the user. This is because the scanf library function failed to check the length of the string as we entered it. Thus we broke the "wall" between buffer1 and buffer2 by overwriting the null escape character at the end of the student_id_1 buffer. Thus, when printf read student_id_1's value, it also read into the student_id_2 buffer's values, which is why the output of the student_id_1 contains both ids imputed by the user. (the student_id_1 buffer was overflowed)