Keith Ecker & Andy Achenreiner

639 Exercise 3.1

1. **C:**

#include <stdio.h>

int main(){

int BUF\_SIZE = 10;

char buff[BUF\_SIZE];

gets(buff);

}

// The overflow happens when the user input is copied into the buffer.

// The program ran fine with small overflows, but started to exit when overflows

// were around 40 characters. It exits with the following.

// \*\*\* stack smashing detected \*\*\*: <unknown> terminated

// Aborted (core dumped)

**Java:**

Public class Main {

Int[] array = new int[5]’

Int check = 0;

Scanner console = new Scanner(System.in);

Int counter = 0;

While(console.hasNextInt()){

Array[counter] = console.nextInt();

Counter++;

}

If(check == 0){

System.out.println(“No access”);

}

Else{

System.out.prinln(“Access granted”):

}

}

The idea of this program is to overflow the array to update the check variable. Java does more checking than c so when I try to update an index that is outside of the array, I get an exception and the program terminates.

**Python:**

BUF\_SIZE = 10

buff = [None]\*BUF\_SIZE

user\_input = input()

for i in range(len(user\_input)):

buff[i] = user\_input[i]

#

# The overflow happens when the letters of the user input

# are copied into the buffer. The program responds by throwing

# an Index error and then exits.

#

1. CODE

#include <stdlib.h>

#include <stdio.h>

long glob\_buff[20];

int main(int argc, char\* argv[]){

int OVERFLOW\_SIZE = atoi(argv[1]);

printf("Overflow size: %i\n", OVERFLOW\_SIZE);

int BUFF\_SIZE = 20;

long stack\_buff[BUFF\_SIZE];

long \*heap\_buff = malloc(sizeof(long)\*BUFF\_SIZE);

long \*buff\_to\_use = heap\_buff;

for(int i=0; i<OVERFLOW\_SIZE; i++){

int index = BUFF\_SIZE + i;

buff\_to\_use[index] = i;

printf("%i\n", i);

}

}

EXPERIMENTATION SUMMARY

The above code was executed using the different overflow amounts of

1, 2, 100, 1000, 10000 for each of glob\_buff, stack\_buff, and

heap\_buff.

RESULTS

Global variable code output

Overflow size: 1

0

Overflow size: 2

0

1

Overflow size: 100

0

1

2

3

...

99

Overflow size: 1000

0

1

2

3

...

483

Segmentation fault (core dumped)

Overflow size: 10000

0

1

2

3

...

483

Segmentation fault (core dumped)

Stack variable code output

Overflow size: 1

0

Overflow size: 2

0

1

Overflow size: 100

0

1

2

3

4

5

Overflow size: 1000

0

1

2

3

4

5

Overflow size: 10000

0

1

2

3

4

5

Heap variable code output

Overflow size: 1

0

Overflow size: 2

0

1

Overflow size: 100

0

1

2

...

99

Overflow size: 1000

0

1

2

...

999

Overflow size: 10000

0

1

2

...

9999

CONCLUSIONS

For the global variable, the code appears to run without issues for the smaller overflows, however with the 1000 and larger the overflow runs into an area of memory it cannot go and a segfault ensues.

For the stack variable, the code appears fine for overflows of 6 and smaller. For an overflow larger than that, the for loop seems to automatically exit after 6 iterations regardless of the value in OVERFLOW\_SIZE. I tried putting a print statement after the for loop, and it will execute, so code after the for loop will run despite this early and unexpected end of the for loop.

For the heap variable, the code appeared fine for overflows of the given sizes.

1. When compiled and ran this program gets stuck in an infinite loop from i = 7 to i = 9 and repeats. The output looks like this:

global[0]

global[1]

global[2]

global[3]

global[4]

global[5]

global[6]

global[7]

global[8]

global[9]

global[7]

global[8]

global[9]

global[7]

global[8]

global[9]

global[7]

global[8]

global[9]

This is because the value for i is overwritten to 7 when the global array is overflowed. When i reaches 10 the value set in global[i] = 7 is out of bounds for the array as it is only of size 10 (0-9). This then overwrites the value of the integer i to 7 since the variable is declared next to the global[] array and the data is written in consecutive data after the array. Since is set to 7 the program is able to run as normal until i reaches 10 again, then the overflow process begins again.