Joseph Camacho-Terrazas
09/16/2020
Programming #3

The reason that on the return of f(), the first printf() is skipped is that the return address of f() is being modified. The line A[6]=A[6]+10 is where the return is changed. Adding the 10 will cause the program to skip the printf() instruction. As an experiment I changed the 10 to play around with the return addresses. The first thing I noticed is that you must put multiples of 10, as any other number gives me an illegal instruction error, since there's more than likely no instruction at that address. Then I found that if it's a valid address (meaning you're not skipping trying to access outside of the function) you can skip as many instructions as you want and even jump back and create a loop. To fix this, I changed A[6] to A[8], and the first printf() appeared as intended. I believe this is happening because A[8] is allocating an extra 64 bit word in memory, so this counters the action of the +10. One interesting thing is that allocating the extra 64 bits will seemingly ignore any offset I put. I tried negative and positive offsets, and this allocation seemed to always keep the program running properly.

The reason we hit a runtime error when adding variables to f() is that we've run out of allocated word space. Each word is 64 bits, while an integer variable is 32 bits. In my experiments, adding variable “a” did not cause a seg fault. This is because we have filled all of the words allocated through A[6]. Upon adding variable “b”, we receive a seg fault. This is because there is no place to store our new 32 bit integer variable. To fix this, we need to allocate another word by incrementing A[6] to A[8]. This will give us another 64 bits to store two integers. I experimented by adding more variables and incrementing A[n] to avoid the error and still skip over the first print statement. This pattern continues throughout my tests, therefore I can infer that for every odd-numbered variable you add, you need to allocate another 64 bits. In my code I have split the variables into 64 bit groups to keep track of the allocations.

Joseph Camacho-Terrazas
09/16/2020
Programming #3

```
/*
*Joseph Camacho-Terrazas
*09/14/2020


* Program to demonstrate how to over write the
* return address inside of function
* we will use a global variable to store
* the address we want to go to on return
* and we will use an array in the function to
* seek the location and replace with the new value
*/

#include <stdio.h>

//dummy function which makes one important change

void f() {

    unsigned int *A;
    int i;

    //adding variables will break the program
    //this is because we're allocating a 64 bit word, but we're going over the li
mit

    int a;
    //need to allocate another word to fit these 2 variables (A[8])
    int b;
    int c;
    //need to allocate another word to fit these 2 variables (A[10])
    int d;
    int e;
    //need to allocate another word to fit these 2 variables (A[12])
    unsigned int f;
    unsigned int g;

    a = 51;
    b = 52;
    c = 53;
    d = 54;
    e = 55;
```

Joseph Camacho-Terrazas
09/16/2020
Programming #3

```
    f = 101;
    g = 102;


    A =(unsigned int *) &A;

    for (i=0;i<=10; i++)
       printf("%d %u\n",i,A[i]);

    //A will allocate the space for variables
    //Adding a number in multiples of 10 to A[n] will change the return address a
nd skip instructions
    //+20 will skip both prints, but +0 will not skip anything. A negative number
 will call previous instructions
    //to offset this, we can allocate extra words (A[12]) and this will set the r
eturn address correctly and the program will no longer skip the print

    //A[10]=A[10]+0; //this will print "I called f"

    //A[10]=A[10]+20; //this will skip both print statements

    //A[14]=A[14]+10; //this will print "I called f"

    //this will skip the print statement after f is called
    A[12]=A[12]+10;
    printf("A is %u \n",A);

    for (i=-4;i<=10; i++)
    printf("%d %u\n",i,A[i]);
}

int main() {

    int A[100];
    unsigned int L[4];
    L[0]=100;
    L[1]=200;
    L[2]=300;
    L[3]=400;

    for (int i=0; i < 100; i++) A[i]=i;

    printf("main is at %lu \n",main);

    printf("f is at %lu \n",f);
```

Joseph Camacho-Terrazas
09/16/2020
Programming #3

```
    printf("I am about to call f\n");
    f();
    printf("I called f\n");

out: printf(" I am here\n");

}
```

```
./program3
main is at 4195906
f is at 4195655
I am about to call f
0 4265784168
1 32766
2 102
3 101
4 55
5 54
6 53
7 52
8 51
9 9
10 4265784656
A before offset is 4265784168
A after offset is 4265784168
-4 4195845
-3 0
-2 4196297
-1 0
0 4265784168
1 32766
2 102
3 101
4 55
5 54
6 53
7 52
8 51
9 9
10 4265784656
 I am here
jterrazas@babbage:~/Documents/programs/CS 471/Program3>
```