Collin Gros
09-18-2020
cs-471

PROGRAMMING #3

The first printf after f is being skipped because the return address to return back to main after f is finished is modified in f to skip 10 addresses ahead (enough for the printf following f to be skipped). Adding variables to f pushes the return address down further, so indexing A at 6 no longer represents the return address. I guess that since the program is running on a 64-bit operating system, gcc allocates enough space to support accepting 2 variables at a time. This means that the program will only runtime error after every 2 variables that are added. When the first variable is added, nothing changes as to where the return address is located, but when the second is added, the return address is shifted (meaning accessing the segment of memory at 6 may cause a segmentation fault). Something we discovered in the lab help session is that if you subtract 10 instead of add 10, the program goes into an infinite loop because the return address is always subtracted enough to go back to the function call for f. In fact, I discovered if you change 10 to 20, you back up enough to call the "i am about to call f" printf statement.

Added variables must be in assignment statements (I guess it's because the compiler does some optimization removing dead code from unused variables).

code:

```
/*
	collin gros
	09-17-2020

	using cooper's code to understand why a print statement is being
	jumped in main

	(Shaun Cooper 2020 Sept.)
*/

#include <stdio.h>


/*
	pre: printf statement must be after this function call so that
		you can see the effect of modifying the return address
	post: modifies the return address so that the printf statement
		following this function is skipped
	in: none
	out: print statements revealing memory around f
*/
void f()
{
	unsigned int *A;
```

```
	int i, a = 0, b = 0;

	// pointing A to its own address
	A = (unsigned int *) &A;
	// print memory around A
	for (i = 0; i <= 10; i++) {
		printf("\t%d\t%u\n", i, A[i]);
	}

	// backs up enough to call f infinitely
	//A[6] = A[6] - 20;
	//A[6] = A[6] - 10;

	// increments return address by 10 to skip the printf statement after
	// the f call
	//A[6] = A[6] + 10;

	// changed to 8 because we added 2 variables (2 bytes), so the
	// RA shifted by 2 bytes
	A[8] = A[8] + 10;

	printf("\tA:\t%u\n", A);

	// print memory around A (to see modified changes)
	for (i = -4; i <= 10; i++) {
		printf("\t%d\t%u\n", i, A[i]);
	}
}


/*
	pre: none
	post: none
	in: none
	out: exit status/ print statements that demonstrate how changing
		a return address using a pointer is possible
*/
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:\t\t%lu\n", main);
        printf("f:\t\t%lu\n", f);

        printf("i ama bout to call f\n");
        f();
        printf("i called f\n");

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