HW #7: Buffer Overflow

CS 392/6813: Computer Security
Fall 2009

[50pts]

Do either Part A or Part B, as per your liking. The HW is due at 11am on 12/18/09. You have to work in teams of three each (as you did for HW6).

For Part A, you will need to give a demo starting 11am on 12/18/09 (no need to submit anything on MyPoly); and you must finish by 11am on 12/18/09.

For Part B, turn in your submission on MyPoly, as you have been doing for prior homeworks.

**PART A:**

Consider the following program “vuln.c”:

```c
//BEGIN:A program to illustrate buffer overflow
//vulnerability

void func(char *str) {
    char buffer[24];
    int *ret;
    strcpy(buffer,str);
}

int main(int argc, char **argv) {
    int x;
    x = 0;
    func(argv[1]);
    x = 1;
    printf("x is 1");
    printf("x is 0");
}

//END
```
The program is very similar to one of the programs we studied in the class. Clearly, it has a static buffer overflow vulnerability. In this exercise, you have to modify the normal execution flow of the program in such a manner that the instruction `{printf("X is 1");}` is skipped. You have to achieve this in two different ways:

1. **[25 points]** Assume that you have write access to the program, i.e., you can modify the source code. Modify the function `func()` in such a manner that the address to which the program returns after executing `func()` is changed so that the instruction `{printf("X is 1");}` is skipped. Use the pointer `*ret` defined in `funct()` to modify the return address appropriately.

2. **[25 points]** Assume that you don’t have write access to the program, but you only have an executable access. Now, you have to exploit the buffer overflow vulnerability and execute the program by passing an argument “argv[1]” in such a manner that the return address is modified so that the instruction `{printf("X is 1");}` is skipped. You can fill up the buffer with NOP instruction (opcode 0x90) wherever needed. You can write an exploit program that runs the above program (“vuln.c”) with an arbitrary input of your choice (use `execl()` function to do so), for testing.

**Instructions**

You can use the `objdump` utility to get a map of the code section of the executable file. The command is “`objdump –d <executable_file_name>`”. This will list all the address locations, opcodes and assembly code of the instructions, for all functions the program calls. For the exercise, you are mainly interested in the main function and the address locations of its various instructions.

**Demo**

You have to demo and explain how you did your exercise to the TA and/or to me.
PART B:

Consider the following program “vuln2.c” on a machine with a “Big Endian” architecture. Assume ASCII value of ‘A’ is 65.

```c
//Begin
void func(char *str)
{
    char buffer[8];
    int a = 0x1a2bc3d4;
    strcpy(buffer,str);
}

int main(int argc, char **argv)
{
    func(argv[1]);
}
//End
```

1. [10pts] On the following page is the stack layout corresponding to the function func(). Fill up, at the appropriate places in the stack shown below, the values (in hexadecimal) of local variables “int a” and “char buffer[8]”, when the program is run with argv[1] as “ABCDE”
2. **[20pts]** You want to use “vuln.c” program to execute a set of instructions starting at address 0x41414141 (in the code section of the executable). Write the pseudo-code for a “wrapper” program to achieve this goal.

3. **[20pts]** Traditionally, the stack grows in a direction opposite to that of memory. Let’s say that the memory grows upwards, and the stack grows downwards. Since a local buffer would be located below the return address, an overflow of the buffer may lead to modifying the return address. A potential way to avoid such a buffer overflow is to follow a convention wherein both the stack and the memory grow in the same direction, say, both grow downwards. Argue whether or not this new convention can be used to prevent buffer overflow vulnerabilities.