IT215 Lab 2 – Buffer overflow (18.1.16)

Goal: In this lab you will learn about buffer overflow, a *classic* exploitation technique, typically applied to C programs.

To get checked off on this lab, the following exercises must be completed and shown to your lab TA by the end of the lab. If anything is not clear, please ask the TA for help.

Exercise 1

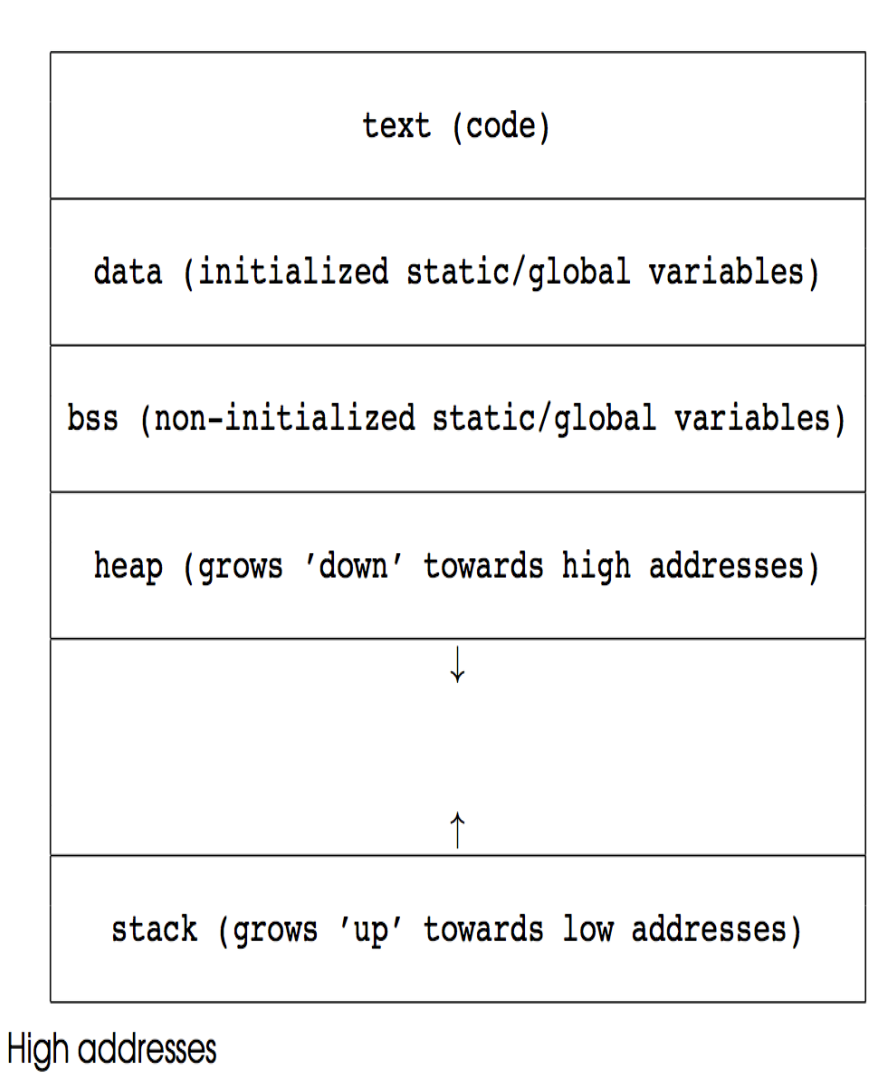
Consider the following C program.

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

int main(int argc, char *argv[]) {
	int value = 5;
	char buffer1[8], buffer2[8];

	strcpy(buffer1, "one"); // copy "one" in the first buffer
	strcpy(buffer2, "two"); // copy "two" in the second buffer

	// show location and content of buffers and of variable 'value'
	printf("[BEFORE] buffer2 is at location %p and contains %s\n",buffer2,
buffer2);
	printf("[BEFORE] buffer1 is at location %p and contains %s\n",buffer1,
buffer1);
	printf("[BEFORE] value is at location %p and contains %d
0x%08x\n",&value, value, value);

	strcpy(buffer1, argv[1]); // copy first argument into buffer1 (no check
on length!!)

	// show again location and content to see what has happened
	printf("[AFTER] buffer2 is at location %p and contains %s\n",buffer2,
buffer2);
	printf("[AFTER] buffer1 is at location %p and contains %s\n",buffer1,
buffer1);
	printf("[AFTER] value is at location %p and contains %d
0x%08x\n",&value, value,value);
}
```

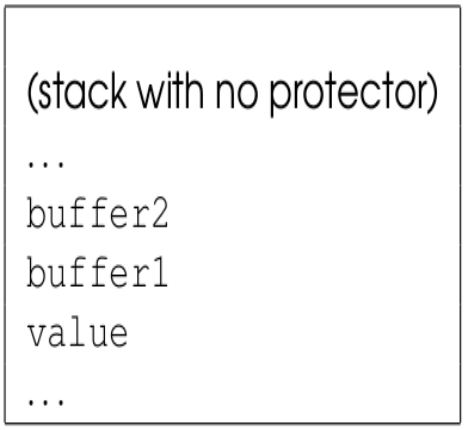
Try to execute the program passing a string of gradually increasing size. For example: ‘A’, ‘AA’, ‘AAA’, and so on.

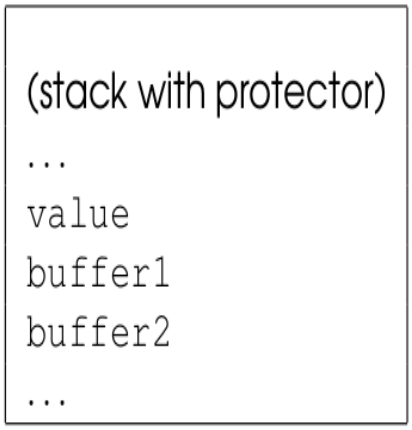
- Observe the overflow in the output;
- Increase the length until the program crashes;
- Try to compile the program with option -fno-stack-protector and observe the difference

To understand what happens in the above example it is good to recall how programs are mapped into memory:



In this example, buffer1 is on the stack at a smaller address than buffer2. Stack protector, implemented by gcc compiler, moves any other variable (value, for example) at even smaller addresses, before buffers, so that overflows cannot affect them. You can check this by giving the -fno-stack-protector option, which disables stack protector. Notice that with this option also the position of the two buffer is swapped (but this may vary depending on the compiler version).





Exercise 2

In this exercise, you will mount a buffer overflow attack on your own program. By doing this exercise, you will learn a lot about machine-level programming.

Download the file bufbomb.c and compile it to create an executable program. In bufbomb.c, you will find the following functions:

```
1  int getbuf()
2  {
3      char buf[12];
4      getxs(buf);
5      return 1;
6  }
7
8  void test()
9  {
10     int val;
11     printf("Type Hex string:");
12     val = getbuf();
13     printf("getbuf returned 0x%x\n", val);
14  }
```

The function getxs (also in bufbomb.c) is similar to the library function gets, except that it reads characters encoded as pairs of hex digits. For example, to give it a string “0123,” the user would type in the string “30 31 32 33.” The function ignores blank characters. Recall that decimal digit x has ASCII representation 0x3x.

A typical execution of the program is as follows:

```
 unix> ./bufbomb
Type Hex string: 30 31 32 33
getbuf returned 0x1
```

Looking at the code for the getbuf function, it seems quite apparent that it will return value 1 whenever it is called. It appears as if the call to getxs has no effect. Your task is to make getbuf return -559038737 (0xdeadbeef) to test, simply by typing an appropriate hexadecimal string to the prompt. Here are some ideas that will help you solve the problem:

- Use OBJDUMP to create a disassembled version of bufbomb. Study this closely to determine how the stack frame for getbuf is organized and how overflowing the buffer will alter the saved program state.
- Run your program under GDB. Set a breakpoint within getbuf and run to this breakpoint. Determine such parameters as the value of %ebp and the saved value of any state that will be overwritten when you overflow the buffer.
- Determining the byte encoding of instruction sequences by hand is tedious and prone to errors. You can let tools do all of the work by writing an assembly code file containing the instructions and data you want to put on the stack. Assemble this file with GCC and disassemble it with OBJDUMP. You should be able to get the exact byte sequence that you will type at the prompt. OBJDUMP will produce some pretty strange looking assembly instructions when it tries to disassemble the data in your file, but the hexadecimal byte sequence should be correct.
- The GCC compiler implements a security mechanism called Stack Guard to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection when you are compiling the program using the switch -fno-stack-protector. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

  ```
  gcc -fno-stack-protector example.c
  ```

- To make the buffer overflow attack work, we have to allow executing any code that is stored in the stack. To mark an executable program example as requiring executable stack, you may use the following command:

  ```
  execstack -s example
  ```

Keep in mind that your attack is very machine and compiler specific. You may need to alter your string when running on a different machine or with a different version of GCC.