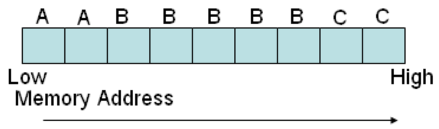
CSC 437/539 Computer Security

Instructor: Dr. Natarajan Meghanathan

Question Bank for Module on Buffer Overflow Attacks

1) Give a brief overview of the buffer overflow attack and its consequences.

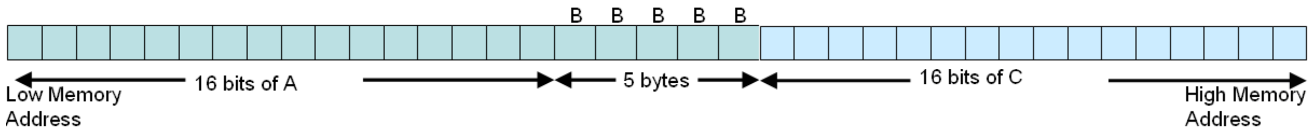
2) Consider the following layout for the memory: Assuming A, B and C are declared respectively as 2-byte integer, 5-byte character array and 2-bye integer. Let the initial values of A, B and C are respectively as follows: A = 4312, B = "0000", C = 1816

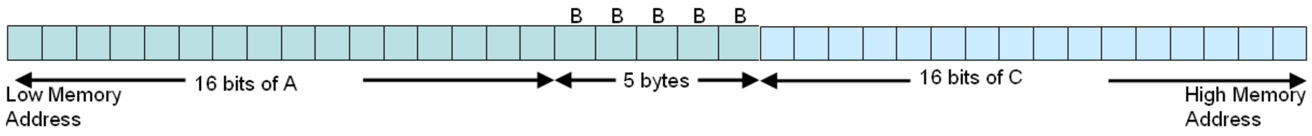


Note that the last character of a string is the terminating character '\0' and assume its ASCII value is 0. Assume the ASCII value of 'R' to be 82.

Find the value of the integers A and C after we set B = "RIVER", assuming the processor used is:
(i) BIG-ENDIAN

Initial Values for A = 4312, C = 1816

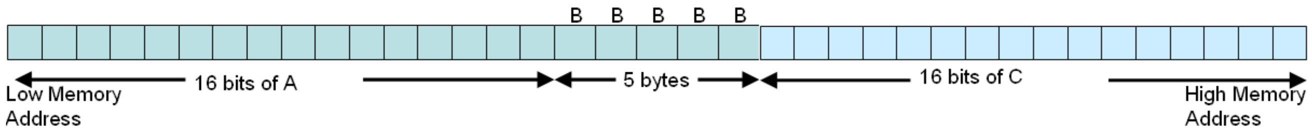


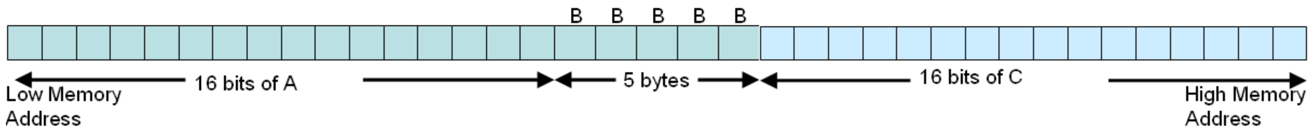


Final Values for A = _______________, C = _______________

(ii) LITTLE-ENDIAN

Initial Values for A = 4312, C = 1816





Final Values for A = _______________, C = _______________

3) Consider the following C code. Assume the following: (i) architecture used is Little-endian; (ii) 32-bit memory addresses and a 4-byte stack boundary; (iii) the largest assigned high end memory address of the stack is 0x8abf4d40. Use this as the starting address to show the structure of your stack frames.
For (c) and (d), do not need to store the characters of the user inputs in their ASCII values in the stack. Just show them as characters itself.

```
void foo ( ){
   char c[8];
   gets(c);
   puts(c);
}

int main( ){
    foo( );
}
```

(a) Show the structure of the stack before the call to the foo( ) function.

(b) Show the updated structure of the stack after the call to the foo( ) function.

(c) Assume the user passes a 7-byte character input "JACKSON" using the gets( ) function. Show the structure of the stack before and after executing the gets( ) function in foo.

(d) Assume the user passes the string “ATTACKERCODE\x8C\35\xC0\x90” as input using the gets( ) function. Show the structure of the stack before and after executing the gets( ) function in foo.

(e) If you were to pass the input for part (d) from command line, how would you pass the input. Assume the name of the executable is *demo*.

4) Consider the following C code. Assume the architecture used is Big-endian. Assume the largest assigned high end memory address of the stack is 0x8abf4d40. Use this as the starting address to show the structure of your stack frames. There is no need to store the characters of the user inputs in their ASCII values in the stack. Just show them as characters itself.

```
int main(int argc, char *argv[ ]) {
        int valid = FALSE;
        char str1[8];
        char str2[8];

        str1 = "JACKSON";
        gets(str2);

        if (strncmp(str1, str2, 8)) == 0)
             valid = TRUE;

        printf("buffer: str1(%s), str2(%s), valid (%d)\n", str1, str2, valid);

}
```

(a) Show the structure of the stack frame for the main( ) function before the gets( ) call is executed.

(b) Show the structure of the stack frame for the main( ) function after the user passes "JACKSON" as input the gets( ) function, and also write the values that will be printed out based on the printf( ) function.

(c) Show the structure of the stack frame for the main( ) function after the user passes "JACKSONSTATE" as input the gets( ) function, and also write the values that will be printed out based on the printf( ) function.

(d) Show the structure of the stack frame for the main( ) function after the user passes "COMPUTERCOMPUTER" as input the gets( ) function, and also write the values that will be printed out based on the printf( ) function.

5) Explain the idea of "NOP sledding" used to launch a buffer overflow attack.

6) Explain the tradeoffs in the programming languages C and Java with respect to safe usage and access to low-level resources.

7) Compare the tradeoffs between Python and Assembly language with respect to their vulnerability to Buffer Overflow Attacks. Justify your answer.

8) What are the four entities that are stored in the stack frame? Define each of them.

9) What is the purpose of the stack pointer register and the frame pointer register? How are they related to the contents of the stack frame?

10) What does the following **gdb** commands do?

a) disas main   b) s   c) x/16xw $esp   d) break 10

11) What is shell-code based stack smashing? How is it typically done and what are its typical side effects?

12) What is the fundamental difference between the compile-time and run-time defense strategies against buffer overflow attacks. What are their pros and cons, if any? Mention two examples for each strategy.

13) Using a simple C code as example, explain the working of the **Libsafe** library to prevent stack smashing buffer overflow attacks.

14) Explain in detail the working of each of the following strategies to prevent buffer overflow attacks? What are their strengths and weaknesses? Also, justify whether they are compile-time or run-time defense strategies?

a) StackGuard
b) ReturnAddressDefender (RAD)
c) Executable Address Space protection
d) Address space randomization
e) Guard pages