EC312 Security Exercise 4

Part 1: Initial Set-Up

Your instructors have prewritten three of the C programs that you will use for this lab, and have placed them in the ec310code directory. The programs you will use today are named sx4a.c, sx4b.c and sx4c.c and these three programs are sitting in the ec310code directory:

We need to copy these files to the work directory. To copy them from the ec310code directory to the work directory, carefully enter the following at the home directory prompt:

```
midshipman@EC310:~ $ cp ec310code/sx4a.c work
```

Make sure you are at your home directory!

Now carefully enter the following two lines at the home directory prompt

```
midshipman@EC310:~ $ cp ec310code/sx4b.c work
midshipman@EC310:~ $ cp ec310code/sx4c.c work
```

If all went well, you should have a copies of sx4a.c, sx4b.c and sx4c.c in your work directory.

Verify that you have sx4a.c, sx4b.c and sx4c.c in your work directory by changing to the work directory: cd work and then listing the files in the work directory: ls.

If you do not have sx4a.c, sx4b.c and sx4c.c in in your work directory STOP and ask your instructor or lab tech for help. Otherwise, proceed to Part 2.
Part 2: Fun with Navy

You should now be in the work directory:

```
midshipman@EC310:~/work $
```

Examine the program `sx4a.c` using nano. The C program is shown below:

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

int main( )
{
    char school[ 5 ] ;
    school[ 0 ] = 'N' ;
    school[ 1 ] = 'a' ;
    school[ 2 ] = 'v' ;
    school[ 3 ] = 'y' ;
    school[ 4 ] = 0 ;
    printf( "%s\n" , school );
}
```

Save your program (Control-o) and exit nano (Control-x) and then compile your program using `gcc -g sx4a.c` and then run your program

```
./a.out
```

to confirm it executes as expected.

If your program is not working **STOP** and ask your instructor for help. Otherwise, proceed to Part 3.

---

Part 3: Wa?

Now, using nano change the line

```c
char school[ 5 ] ;
```

to read

```c
char school[ 10 ] ;
```

and rerun your program.

**Question 1:** What effect did this modification have on your program’s output? **Explain.**

Now, we would like to modify our program so that it prints out: Wavy instead of Navy. Change one line of code in your program to accomplish this. Do not use `strcpy` for this question. Verify your solution works.

**Question 2:** What change did you make to your program?
Now change the line

```c
school[ 2 ] = 'v' ;
```

to read

```c
school[ 2 ] = 0 ;
```

(Note that the line of code contains the number zero, not the letter oh.)

**Question 3:** What output is produced? **Why is this output produced?**

Now, change the line

```c
school[ 2 ] = 0 ;
```

back to its original form:

```c
school[ 2 ] = 'v' ;
```

but change the array declaration line

```c
char school[ 10 ] ;
```

to read instead:

```c
char school[ 2 ] ;
```

So, we are telling C that we intend that our character string only have two characters (and remember, one of them should be the NULL character.

Run the program.

**Question 4:** What was printed out? Did the program realize you only wanted the array to have two items? **Explain.**

If you feel very confident in your answer to Question 4, go on to Part 4. Otherwise, **STOP** and ask your instructor for help.

---

### Part 4: Out of Bounds

Examine the program `sx4b.c` using nano. The C program is shown below:

```c
#include <stdio.h>
#include <string.h>
int main()
{
    int count[3] = { 11 , 12 , 13 };
    int j;
    for ( j = 1 ; j <= 3 ; j = j + 1 )
    {
        printf("The next number is %d\n" , count[j] );
    }
}
```

Compile and run the program.
Question 5: What was printed out? **Explain.**

Question 6: How would you fix the program so that it prints out:

The next number is 11
The next number is 12
The next number is 13

Make this fix to your code and have your instructor verify that your program works properly.

After you have Question 6 signed off, proceed to Part 5.

---

**UNIX Tips and Tricks**

UNIX provides another feature to lessen the amount of typing that busy midshipmen have to perform.

UNIX attempts to complete our commands for us, using a feature called *tab completion*. To use tab completion, you type in part of a command, and then hit the tab key. UNIX will attempt to complete the command for you (or may partially complete the command).

For example, suppose I have been editing the file `smith_4_1.c`, and have opened and closed this file several times already. If I now type:

```
midshipman@EC310-VM:~ $ nano s
```

and then hit the Tab key, UNIX will automatically complete the command for me:

```
midshipman@EC310-VM:~ $ nano smith_4_1.c
```

Similarly, I find that if I type

```
midshipman@EC310-VM:~ $ ./a
```

and then hit the Tab key, UNIX will automatically complete the command for me:

```
midshipman@EC310-VM:~ $ ./a.out
```

Tab completion is very useful. Keep in mind that if UNIX cannot decide how to complete your command (since you have not provided enough characters to start with), enter another character or two and press Tab again.
Part 5: Joys of strcpy

Examine the program sx4c.c using nano. The C program is shown below:

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char slogan[16] = "Cyber 2 is fun!" ;
    printf("\n%s\n", slogan);
}
```

Compile and run the program.

**Question 7:** Why did I choose to make the array size equal to 16. There are only 15 characters in the string *Cyber 2 is fun!*

Carefully add the following two lines to your program above, right before the closing brace.

```c
strcpy( slogan , "Cyber rocks!" );
printf("\n%s\n", slogan);
```

Compile and run the program using:

```
gcc -g sx4c.c
./a.out
```

**Question 8:** What was printed out?

Your friend is confused. He looks at the length of the two strings:

```
Cyber 2 is fun!

---

Cyber rocks!
```

and wonders why the second item printed out wasn't *Cyber rocks!*un!. In other words, he wonders what happened to the un! that finished the string *Cyber 2 is fun!*, since those last few characters were not overwritten.

**Question 9:**

To answer your friend’s question, use the debugger to look at the memory before and after the copy command. Run the debugger:

```
gdb -q ./a.out
```

At the prompt enter:

```
set dis intel
list
disassemble main
```
Examine the memory at location bffff800 using:

```
x/16xb 0xbffff800
```

you will see the data stored at the 16 bytes following the address bffff800. These bytes represent the string slogan. To decode it you can use the ascii table or type in:

```
x/16c 0xbffff800
```

Enter these values on the table on your answer sheet.

Now look at the slogan after the string copy command by entering:

```
break 9
continue
x/16c 0xbffff800
```

Enter these values on the table on your answer sheet.

Now that you have looked at the memory before and after the string copy command, what is the answer to your friend's question? **Explain.**
EC312 Security Exercise 4

Name:

Question 1:

Question 2:

Question 3:

Question 4:

Question 5:

Question 6:

Instructor/Lab Tech signature

Question 7:

Question 8:
Breakpoint line 7 | Breakpoint line 9
---|---
bffff800 | |
bffff801 | |
bffff802 | |
bffff803 | |
bffff804 | |
bffff805 | |
bffff806 | |
bffff807 | |
bffff808 | |
bffff809 | |
bffff80a | |
bffff80b | |
bffff80c | |
bffff80d | |
bffff80e | |
bffff80f | |