Lab 1: Control Hijacking Exercises

Set Up:

In this Lab, we will be demonstrating how Buffer Overflow mechanics work by modifying an input to a vulnerable C code. A compiled application called "echo-app" will run and ask you for an "input". This is how you will cause the overflow and get a shell ($).

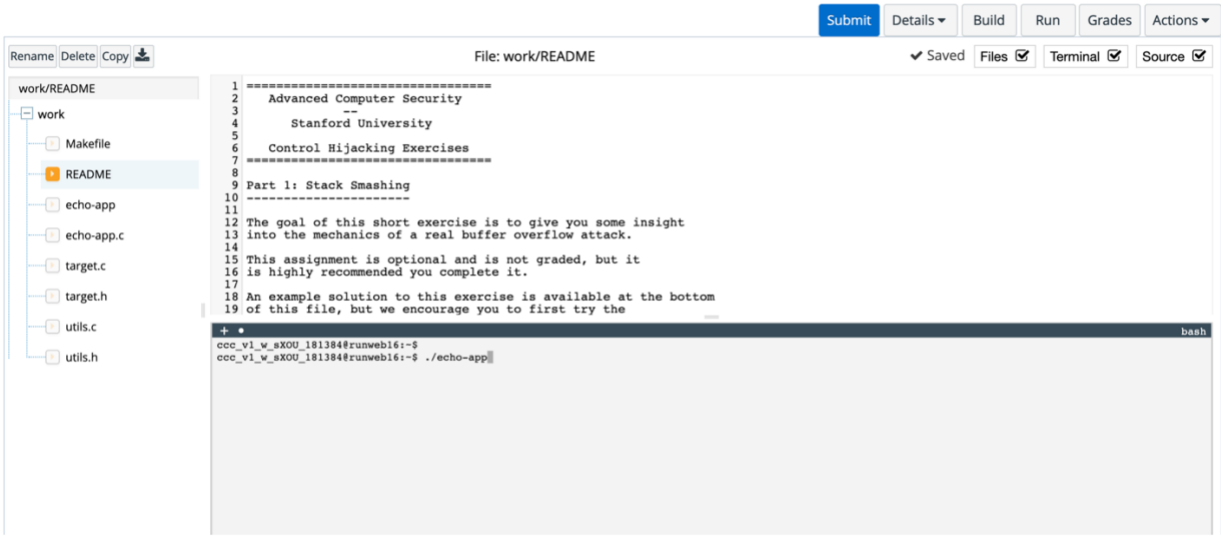
The app echoes what you type, it can stop and error out (segmentation fault) if you do not pass an expected input. Once you find the correct size to overflow the buffer by constructing the right overflowing input, you get a shell in the host while the app is running.

NOTES:

- *This assignment is optional and is not graded, but it is highly recommended you complete it.*
- *An example solution to this exercise is available at the bottom of this file, but we encourage you to first try the exercise yourself. We know you can do it!*
- *YOU DO NOT HAVE TO MODIFY ANY CODE FILE, THE LAB IS COMPLETED AS SOON AS YOU GET THE SHELL.*

Files:

There are seven or eight (in case there is a compiled app) files in your work area on the left panel. You will also see a preview of the files on the top right, and the terminal window on the bottom right – this is the area where you will work. You can uncheck or select the panels that you want.



This is a description of each file in your work area:

Makefile : Specifies rules for compiling the code. You should not modify this file.

README: Lab instructions inside the virtual environment.

echo-app.c and echo-app : The source code and the compiled code.

target.(c|h): Contains code for a 'library call' called echo. You should not modify this file.

utils.(c|h): Contains some helper routines. You do not need to read nor understand this file.

Stack Smashing Lab

Objective:

Your goal in this exercise is to provide an input to the echo-app command-line application that will result in the execution of the function 'start_sh' in target.c.

A successful exploit means you get a shell while echo-app is running. That is, instead of seeing a prompt that looks like this:

```
ccc_v1_w_sXOU_181384@runweb16:~$
ccc_v1_w_sXOU_181384@runweb16:~$ ./echo-app
Welcome! I echo what you say. Enter Ctrl-C to stop the program.

Function digest:
  bar      is at address: 0x80485cf
  start_sh is at address: 0x80485e7
  echo     is at address: 0x80485fa

------------------------------
Your input: hello
You said: hello
------------------------------
Your input:
```

you should see:

```
ccc_v1_w_sXOU_181384@runweb16:~$
ccc_v1_w_sXOU_181384@runweb16:~$ ./echo-app
Welcome! I echo what you say. Enter Ctrl-C to stop the program.

Function digest:
  bar      is at address: 0x80485cf
  start_sh is at address: 0x80485e7
  echo     is at address: 0x80485fa

------------------------------
Your input: hello
You said: hello
------------------------------
Your input: .[redacted]08
You said:[redacted]
$
```

Shell

You can enter "exit" in the shell to exit.

Following hints will also be useful for Lab 2 and Lab 4.

Hint #1

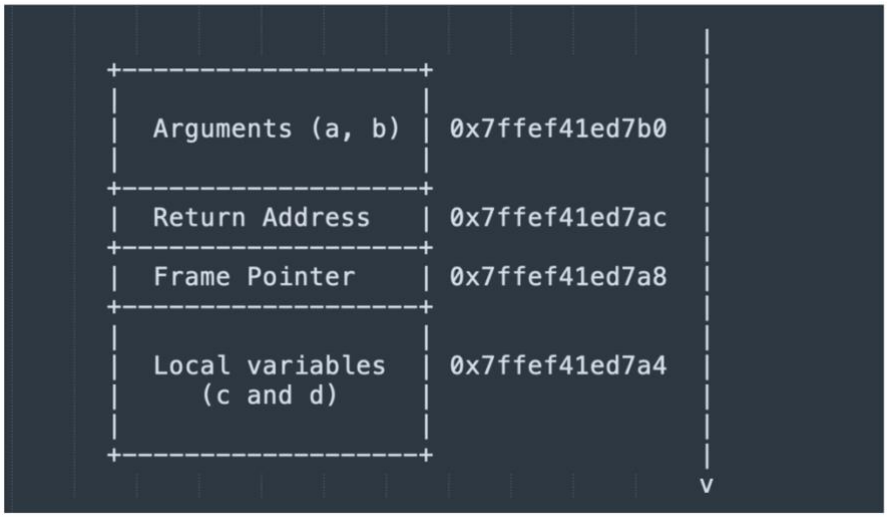
Study the stack frame layout explained in the videos. You could also do some reading online to get more understanding of the topic.

Keep in mind the stack frame layout during x86 function calls. These details vary across architectures and standards, but at the very least you should remember that the stack grows downwards from higher addresses to lower addresses, and that arguments go first followed by the return address, the stack frame pointer, and then other artifacts needed during the function (i.e. local variables). You can experiment with this assumption in mind.

For example, a function like

```
void foo(int a, int b) {
  int c = a + b;
  int d = 2 * d;
  printf("%d and %d\n", c, d);
}
```

could have a stack frame that looks like this:



The order of local variables on the stack do not need to be the order in which they were declared, and there may be some other artifacts below the frame pointer.

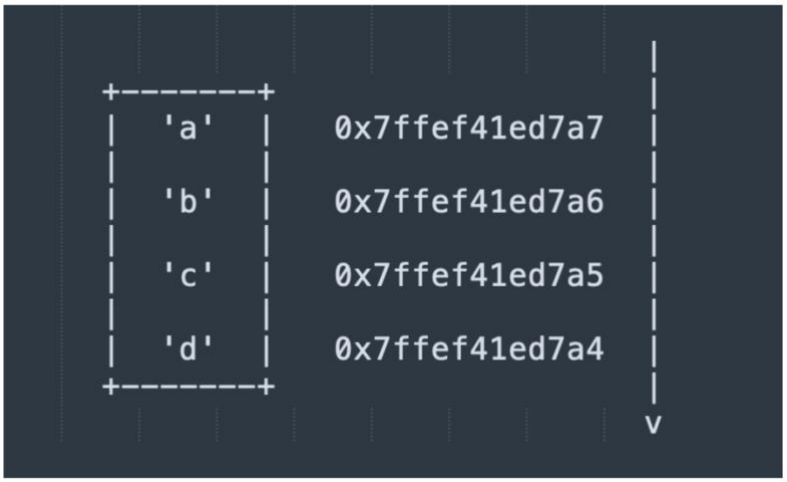
The arguments 'a' and 'b' may be in registers or on the stack. You can rest assured (at least in x86) that the return address will be above the frame pointer and that the frame pointer will be above the local variables and maybe some other local structures.

Hint #2

Strings are written 'upwards' in the x86 stack. For example, code that declares the string 'abcdefghijkl' on the stack could place it like this:

Hint #3

The environment you will be working with has little endian byte order (basic memory explanation). This means that bytes will be arranged on the stack as shown below (continuing with the string example from above):



Hint #4

In this lab, you can specify specific bytes by writing them in hex and preceding them with **‘ \x '**.

For example, if you want to specify the byte value corresponding to 0xab, you can enter **'\xab'**

Environment

You can build the code by pressing the BUILD button on the top bar. If there are no compilation errors, you can invoke the exploit directly from the command line:

./echo-app

You can also run the code by pressing the RUN button. Note that the echo-app prints the location of certain functions at runtime -- you must use this information to construct your exploit.

If at any point you would like to download the StarterCode into your working area again, you can select Actions -> Reset assignment -- this will overwrite the files.

The next page includes a sample solution…

Sample Solution

Your input: .123.123.123.123.123.123\xGH\xEF\xCD\xAB
Where 0xABCDEFGH is the address of the start_sh function.

For example, if the function digest says that start_sh is at address 0x80485e7:

```
Function digest:
  bar      is at address: 0x80485cf
  start_sh is at address: 0x80485e7
  echo     is at address: 0x80485fa

------------------------------
Your input: hello
You said: hello
```

How would YOU construct your input? Try it!