CS 240
Laboratory 8 Assignment
Introduction to Buffer Overflow

Read the partial description of a buffer overflow assignment described below, and answer the questions at the end. You only need to hand in the answers on the final page.

The buffer overflow assignment helps you develop a detailed understanding of the call stack organization by deploying a series of buffer overrun attacks on a vulnerable executable file called `laptop`.

Goals

- To understand the procedure call abstraction and the details of its implementation with the stack discipline.
- To understand the far-reaching impacts of system design choices, especially through security implications of the call stack in a language that does not enforce memory safety.
- To understand the principles of buffer overrun vulnerabilities through practice exploits in a controlled environment.
- To scare yourself a bit when realizing that the same kind of vulnerability you exploited probably exists somewhere in the software powering your healthcare, transportation, utilities, and more.

Repository

Your task is craft exploit strings that accomplish four increasingly sophisticated buffer overrun attacks when provided as input to the vulnerable `laptop` executable.

Your starter repository will contain the following files:

- `questions.txt`: file for English descriptions of your exploits
- `exploit1.hex`, `exploit2.hex`, `exploit3.hex`, `exploit4.hex`: files for Exploits 1-4
- `hex2raw`: utility to convert human-readable exploit descriptions written in hexadecimal to raw bytes
- `id2cookie`: utility to convert user ID to unique “cookie” value
- `Makefile`: recipes to test your exploits
- `laptop`: executable you will attack
- `laptop.c`: important parts of C code used to compile `laptop`

Designing Stack Exploits

In the assignment, you will **design stack exploits** by taking advantage of a system function `Gets()` that is poorly designed.

`Gets()` takes as a parameter a pointer to an array of bytes.

The bytes that are accepted from a call to the function are stored on the stack (this storage on the stack is what we refer to as the **buffer)**.

Although the function is intended to only accept up to a specific number of bytes, nothing in the code prevents it from accepting a larger number of bytes when the array is longer than intended.

So, the **buffer** (the space in the stack that is reserved to store the bytes that are accepted) "overflows", and can overwrite other values on the stack (such as return addresses).

When the values of the extra bytes that over-write the stack are chosen carefully, the program will continue executing, but in a different way than it would normally. The process of choosing the extra bytes is called **designing an exploit.**

Formatting Exploit Strings

We ask you to construct your expoits by expressing them as a string which represent the values of the bytes in your exploit.

Remember that each ASCII character in a string is represented by one byte. For example `'A'` is represented by the byte value also described by the hexadecimal number value `0x41`.

If you had to design your string by hand to represent the numerical bytes you need to for your exploit, you would have to do some tricky things like expressing the asci value for un-type-able characters or determining the byte encoding of x86 instructions.

So, we provide some tools to help you encode your exploit string.

A tool called `hex2raw` will take your exploit string and encode it properly for use in the exploit.

- The input to `hex2raw` is a human-readable text description of a byte sequence where each byte is written as pair of hexadecimal digits. Successive bytes may be separated by spaces.
- The output of `hex2raw` is a raw byte sequence, where each byte has the hexadecimal value described by the corresponding pair of characters in the input.

Suppose we want the raw sequence of bytes whose values are the hexadecimal numbers:

```
0x01 0x02 0x03 0x04
```

Our string to represent it would be:

```
"01 02 03 04"
```

In asci, this is:

```
0x30 0x31 0x20 0x30 0x32 0x20 0x30 0x33 0x20 0x30 0x34
```

So, `hex2raw` will accept our string `"01 02 03 04"` and output the desired 4-byte sequence

```
0x01 0x02 0x03 0x04
```

> **NOTE: do not use `0A` in your exploit strings!**
>
> Your exploit string must not contain 0A, since this is the ASCII code for newline (`'\n'`).
>
> When `Gets()` encounters this byte, it will assume you intended to terminate the string input, and will ignore the rest of your values.
>
> `hex2raw` will warn you if it encounters this byte value.

Using `hex2raw` to create the exploit bytes

- Using VSCode, create a file for each exploit (for example, call the file for Exploit 1 `exploit1.hex`)
- Type the series of hexadecimal byte values you want into the file (for the earlier example, you would enter the values `01 02 03 04`
- Then run:

```
$ ./hex2raw < exploit1.hex > exploit1.bytes
```

The shell's *input redirection* symbol `<` instructs the command-line shell to use the contents of `exploit1.hex` as standard input to `hex2raw`, instead of looking for input from the keyboard. The shell's *output redirection* symbol `>` instructs the command-line shell to store the standard (printed) output of `hex2raw` into a file called `exploit1.bytes`. Input and output redirection (`<` and `>`) are general features of the command-line shell that can be used independently and with any executable command.

Running your exploit

Once the exploit string byte sequence is stored into the file `exploit1.bytes`, run `laptop` with the contents of the file `exploit1.bytes` as input:

```
$ ./laptop -u your_cs_username < exploit1.bytes
```

If you update your exploit string specification in `exploit1.hex`, you must always run `hex2raw` again to translate the new version to a byte sequence in `exploit1.bytes` to use this new exploit with the `laptop`.

SYNOPSIS: Running and Testing Exploits

To run an individual exploit:

1. Write the exploit string in the file `exploit1.hex`
2. Translate it to raw bytes with `hex2raw`

   ```
   $ ./hex2raw < exploit1.hex > exploit1.bytes
   ```

3. Run it directly (possible for Exploits 1 and 2):

   ```
   $ ./laptop -u your_cs_username < exploit1.bytes
   ```

   or run it under `gdb` (required for Exploits 3 and 4):

   ```
   $ gdb ./laptop
   (gdb) run -u your_cs_username < exploit1.bytes
   ```

Questions

1. What files from the starter repository will you modify as part of the assignment, and why?

2. What is the purpose of `hex2raw`?

3. Why shouldn't you use the value **0A** in your exploit strings?

4. What would you put in your exploit file if you wanted the 8-byte value 0x0000000000400ff3 to be made into raw bytes, but in the order from least significant to most significant byte?

5. Which exploits will run alone without **gdb**? Which exploits work only under **gdb**?

6. What are the steps for running an exploit?