CS 240
Laboratory 8 Assignment
Buffer Overflow

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

This 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`

Formatting Exploit Strings with `hex2raw`

Constructing exploits involves tricky tasks like writing untypeable characters and determining the byte encoding of x86 instructions. Use the techniques below to simplify your job.

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`.

While your exploits will be delivered under the guise of strings, they will embed sequences of bytes encoding addresses, numbers, or other non-character data.

It is hard enough to map each desired byte value in your exploit back to a character by hand, but often, the specific bytes required do not even correspond to any typeable or printable ASCII characters, making it "difficult" to type your exploit string on a keyboard or view it on the screen.

So, **Do not try to encode your exploit by hand!**

We have provided a tool called `hex2raw` to encode exploit strings:

- 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`

Given the input `01 02 03 04`, the `hex2raw` utility will output the desired 4-byte sequence.

To run `hex2raw`, type the series of hexadecimal byte value descriptions you want in a file (*e.g.*, `exploit1.hex` for Exploit 1).

Following our example, we could save the string `01 02 03 04` into the file `exploit1.hex` using Emacs. 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.

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
```

Naturally, as with compiled source code, if you update your exploit string specification in `exploit1.hex`, you must run `hex2raw` again to translate the new version to a byte sequence in `exploit1.bytes` to use this new exploit with the `laptop`.

Warning: do not use `0A`

Your exploit string must not contain byte value `0x0A` (`0A` in `hex2raw` input) at any intermediate position, since this is the ASCII code for newline (`'\n'`). When `Gets()` encounters this byte, it will assume you intended to terminate the string input. `hex2raw` will warn you if it encounters this byte value.

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 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 is 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?