CS240 Laboratory 11
Buffer Exploits

In the stack lab, we investigated how the stack is used for procedure calls. The next assignment builds on this knowledge of the stack to help you understand some security weaknesses in system code.

The call frame for a procedure includes space on the stack needed to store parameters for calls to sub-procedures.

We saw an example of this in *phase_2* of the last assignment and lab. Before calling *read_six_numbers*, the *phase_2* code allocated space in the stack to store the six integer values entered by the user.

In this lab, you will design 4 exploits strings. Each one assumes that the you enter a string which is longer than the space which has been allocated to store it on the stack (this is called a “buffer overrun”.

Knowledge of how the call stack works to store parameters and return to the calling program, along with understanding the ways that system code to accept an input string is flawed, can be used to get the program to run in ways not intended by the designer.

Repository

- descriptions.txt you will add your description of exploits
- exploit1.hex, exploit2.hex, exploit3.hex, exploit4.hex: you will fill with hex values for bytes to be used for each exploit
- hex2raw: utility to convert your human-readable exploit files from hex to raw bytes
- id2cookie: utility to convert user ID to unique "cookie" value
- Makefile: used to compile your code
- umbrella: executable you will attack
- umbrella.c: important parts of C code used to compile umbrella

Cookie

Most attacks in this assignment will require you to make a unique 8-byte *"cookie"* value show up in places where it ordinarily would not. This value will also determine the exact behavior of your executable.

To create your personalized cookie, run make cookie and enter **your Bitbucket ID**.

This will print your cookie in hex and record your Bitbucket ID and cookie in the files id.txt and cookie.txt.

Umbrella executable

reads a string from standard input with the function `getbuf()` ,which is called by `test()`:

```
unsigned long long getbuf() {
        char buf[36];
        // ...
        unsigned long long val = (unsigned long long)Gets(buf);
        // ...
        return val % 40;
}
```

calls `Gets()`, to which it passes as an argument the address of its local array `buf`, which is ***allocated on the stack*** with space for 36 `char`s.

`Gets()` reads a string from standard input, terminated by a newline character (`'\n'`)

`Gets()` then stores the characters of the string, followed by a null terminator (`'\0'`) starting at the memory address given by its argument, `buf`.

`Gets()` does not check whether there is enough space on the call stack to store the entire string. If the string is too long, it will overrun the frame and overwrite other values on the stack.

`getbuf()` looks like this in **gdb**:

```
0x0000000000400dd0 <+0>:     push   %rbp
0x0000000000400dd1 <+1>:     mov    %rsp,%rbp
0x0000000000400dd4 <+4>:     sub    $0x30,%rsp
0x0000000000400dd8 <+8>:     lea    -0x30(%rbp),%rdi
0x0000000000400ddc <+12>:    callq  0x400cb0 <Gets>
```

In exploit 1, you will design the string you enter to be long enough to overwrite the return address on the stack so that instead of returning to the `test()` calling program, you return to another function called `smoke().`

You will need to determine the address of `smoke()`and make it part of your exploit string to accomplish this.

Create Exploit Strings

Using emacs or another text editor, create a file containing a byte sequence where each byte is written as pair of hexadecimal digits (i.e **exploit1.hex**). Successive bytes may be separated by spaces.

If you need a specific number of filler bytes in your exploit string, number the values so it is easy to keep count (for example, 16 filler bytes could be):

01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16

WARNING: DO NOT USE `0A` for any byte (since it will be interpreted as the end of the string)

Create Byte File with hex2raw

Once you have created the hex file, use the following command to create the bytes file:

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

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.

Run umbrella from the command prompt (exploit 1 and 2)

Use the bytes file produced by hex2raw

```
$ ./umbrella -u your_bitbucket_username < exploit1.bytes
```

Run umbrella under gdb (exploit 3 and 4):

```
$ gdb ./umbrella
[... gdb startup output ...]
gdb) run -u your_bitbucket_username < exploit1.bytes
```