CS 2110: Lecitation 16

Name:

Monday, July 16th, 2018

Objective: Exploit unsafe code by stack smashing!

Introduction: `hex2ascii`

Someone not well-versed in how the stack works has written a program called `hex2ascii` (source file and `Makefile` to build it are provided, so type `make` to build `hex2ascii`). When `hex2ascii` is run, it will prompt the user for a string of hex digits, then convert them to ascii. For example:

```
preston@arbitrage:$ ./hex2ascii
Input a hex string to convert to ASCII: 48656c6c6f21205e5f5e<enter>
Hello! ^_^
```

For convenience, this can also be done by using the contents of a file as input:

```
preston@arbitrage:$ ./hex2ascii < input.txt
Input a hex string to convert to ASCII:
Hello! ^_^
```

Note that if using a file, there must be a newline at the end to simulate a user pressing enter in the `hex2ascii` program.

Vulnerability

Notice that in the provided code, a fixed-size char buffer is provided to store the converted input before printing it after the user presses enter. Recall that important metadata about the current function's invocation, such as the old frame pointer, return address, etc. are stored below the local variables. What if someone decided not to play nice, and wrote more input than the buffer could hold?

This kind of buffer overflow vulnerability is one of the most basic, and common causes of stack smashing in the real world. One example is the Twilight Hack, a hack that was used to run homebrew on Nintendo Wii by using a hacked save file that gave Epona a really long name in The Legend of Zelda: Twilight Princess.

If the hacker correctly calculates the location of the return address on the stack, a sufficiently long input containing a return address to some other location in memory (the input used to do this is called a "payload") could cause the current function to "return" somewhere else. Scary!

Assignment

Modify `input.txt` so that the function `smash` in the code is run. You may not edit `hex2ascii.c` to accomplish this. Note that the addresses of the functions will be different for each student, and so will your payload.

Here are the general steps you will need to complete this lab. See the next section for information on how to accomplish these steps:

1. Find the address of the `smash` function
2. Find the address that `read_chars` will return to
3. Find out how far after `buffer` starts that the return address for the `read_chars` invocation is located
4. Write a payload of appropriate length to store in `buffer` and overwrite that return address with the address of `smash` instead. Note that the address is 32 bits for 32-bit Linux, and 64 bits for 64-bit Linux.

Gathering Data with GDB

`gdb` is the most important tool you can use for this! Remember that to run the program through `gdb`, you use `gdb ./hex2ascii`

The suggested use of `gdb` is to set a breakpoint after the declaration of the character buffer in `read_chars()`, then start gathering data about where the current function invocation will return, where you want it to return instead, the contents of the stack, and how much data you must write to accomplish this.

Example of setting a breakpoint (on line 53) and running:

```
b 53 r
```

You can print the address of a function just like any variable! Example: `p main`

You can see the addresses of where all current function invocations will return by using the backtrace command. Here is an example, with a breakpoint in the `convert_char()` function:

```
(gdb) bt
#0 convert_char(c1=89'Y',c2=10'\n') at hex2ascii.c:20
#1 0x000000000040073ainread_chars() at hex2ascii.c:38
#2 0x00000000004007afinmain() at hex2ascii.c:56
```

This means that once `convert_char()` returns, it will return to address `0x000000000040073a` which is in the `read_chars()` function, and the `read_chars()` function will return to address `0x00000000004007af` when it returns to `main`.

To analyze stack data, print out the contents at some memory location. The suggested use is starting at the address of the buffer since you know the return address is somewhere below that.

Example printing memory starting at variable `my_var` (16 zero-padded long values):

```
x/16zg &my_var
```

Remember that composite values like pointers are stored and read as a whole in little endian! This means that the address of main in this example:

```
(gdb) p main
$1 = {int (void)} 0x400792 <main>
```

Although displayed as (padded with 0s):

```
00 00 00 00 00 40 07 92
```

Is actually stored in memory in little endian:

```
92 07 40 00 00 00 00 00
```

Keep this in mind when writing your payload in `input.txt` as the order of the bytes must be reversed to correctly redirect the return address.

Submission

Please submit these two things:

- `input.txt`, which should contain the appropriate payload for the stack smash
- a screenshot of your terminal after running `./hex2ascii < input.txt`