Introduction

In this assignment, you will first write a simple HTTP client that can download files off the Internet using the HTTP protocol. Then, you will [collectively] analyze the source code of an existing HTTP server to identify vulnerabilities including those that lead to buffer overflows. Finally, you will develop a remote attack on this server to exploit these vulnerabilities.

This is an individual assignment and all code must be written by yourself. However, the analysis of the HTTP server to identify buffer overflow vulnerabilities is a *collective*, class-wide effort with extra credit opportunities. See the instructions below.

This assignment brings together many of the concepts you've learnt in this course and asks you to write a fairly realistic program. START EARLY! The major concepts that this assignment expects you to be familiar with are:

- Socket programming
- String and string manipulation
- Pointers and dynamic memory allocation
- File I/O
- Error handling
- Memory layout of the stack for function calls

The first part of this assignment will take a solid 10 hours of work, and the second part will take a similar amount. Expect to spend lots of time in `gdb`.

Part I: HTTP Client

The HyperText Transfer Protocol (HTTP) is the original protocol of the web and is used by browsers (i.e. clients) to talk to HTTP servers on the internet. In its simplest form, an HTTP client sends a request to the server that looks like this:

```
GET / HTTP/1.1
Host: www.example.com
```

Here, the client is performing a "GET" request for the "/" object using the "HTTP/1.1" protocol. The header "Host" is also supplied indicating the server the client expects to talk to a server that may serve multiple websites. The HTTP server usually listens on port 80, so the client must first establish a connection to the `www.example.com` on port 80 and then send the request. Each line of the request is terminated by `"\r\n"` (i.e. CRLF), and the whole request is terminated by blank line ending in `"\r\n"` .

The server will then send a response which consists of two parts, a header and a body:

```
HTTP/1.0 200 OK
Content-Length: 13
Content-Type: text/html

<html>
<body>
Hello, world!
</body>
</html>
```

The headers and the body are again separated by a blank line ending in `"\r\n"`. The first line is called the status line and indicates the server version ("HTTP/1.0"), a status code (200), and a string equivalent of the status ("OK").

The syntax of these requests and responses were detailed in RFC 1945 (for HTTP/1.0) and RFC 2616 (HTTP/1.1). Most modern web servers and browsers use HTTP/2.0 and use encryption (using transport layer security or TLS) to prevent eavesdropping.

In this assignment, you will develop a very simple HTTP/1.1 client. It will have the ability to connect to simple servers like the `tiny.c` server in the textbook and the Python HTTP server (`python3 -m http.server`). Since your client will not implement TLS, it will be unable to communicate with most real-world servers. If you'd like to see your program work on a website, try connecting to `neverssl.com` port 80 *once your client is working on tiny*.

Setup

Download the Tiny web server. Compile it using `make` and run it as follows:

```
./tiny 5678
```

Here, `5678` is a randomly chosen number between 1000 and 65535. If you're using the cycle machines, somebody may already be using the port number and you should use a different port number if you encounter a `bind` error.

You can press CTRL+C at any time to stop the Tiny web server.

Once `tiny` is running and listening for connections, switch to another terminal (i.e. open another connection to the cycle machines). If you're using a personal Linux desktop, just open another terminal to run the other commands (`download` and `buf`).

Client Implementation

Write a program `download.c` to download a file from the Internet using HTTP. Your program will be invoked like this:

```
./download localhost 5678 /index.html output.html
```

It will then connect to `localhost` port 5678, and download the `/index.html` file, saving it to `output.html`. For your testing, substitute 5678 with whatever port the `tiny` you started is listening on. Note that tiny must be running on the *same* machine you're running this command on. **NOTE:** The `index.html` files does not exist and you'll get a 404 error. Use `/home.html` if you want to test.

First, parse the command line to extract the server (`argv[1]`), port (`argv[2]`), remote file (`argv[3]`) and output file (`argv[4]`). The server, remote file and output file can be arbitrary strings, but check that the port number is a valid integer between 1 and 65535. Look at (and use) the code in `buf.c` where needed. If the command line is malformed or missing arguments, print a message ("Incorrect usage"), and exit with a code of 1.

Second, perform name resolution on the server to get its IP address. You must use `getaddrinfo` to do this, do *not* use `gethostbyname`. Read the example in the lecture slides or in the manual page and adapt the code. The server is the `node` argument, `service` is the port number as a string, and the `hints` data structure should set `ai_family` to `AF_INET` and `ai_socktype` to `SOCK_STREAM` to avoid getting IPv6 addresses. If `getaddrinfo` fails for any reason, use `gai_strerror` to obtain an error message and print it out before exiting with a code of 2.

Third, use `connect` to open a connection to the server using the *first* address returned by `getaddrinfo`. If the `connect` fails, print an error message (e.g. "Connection failed") and exit with a code of 3.

Fourth, construct the request using the string functions and send the result to the server using `write`.

Fifth, read the response from the server. Assuming the server will always return a well-formed response, separate out the response into a header and body part. Store the code the server responded with. Write the body to the output file. Write the header to a file that has the same name as the output file but with ".header" suffixed to it. Thus, in the example above, the body will be stored in `output.html` and the headers will be stored in `output.html.header`.

If any OS error, *other than* `EINTR`, occurs in sending the request or reading the response, print an error message and exit with a code of 4. If either `write` or `read` fail with an error code of `EINTR`, retry the operation again. You must read the *complete* server response

from the socket until you receive an EOF (or a non-`EINTR` error). Partial reads will cause the `tiny` server to non-deterministically fail with a `SIGPIPE` error.

Finally, close the connection, clean up all memory. If the server returned a status code of 200 to 299, exit with a return code of 0 otherwise exit with a code of 5.

Your program should never crash (assuming a well-formed server response), should not leak memory (checked using ASAN), and behave as specified above.

Part II: Bug hunting in tiny.c (Collaborative)

In this part of the assignment, you will analyze the source code of the Tiny web server presented in Chapter 12 of the textbook. Although it was not perhaps not their intention, this web server is a great example of many security bugs. Your analysis should identify these bugs. This is a *collective* effort and you are allowed to collaborate with *anybody who is a student in the class* to examine the source code of tiny. Once you have found these bugs, post them on Blackboard:

1. with the relevant excerpt of the source code,
2. an explanation of what the bug is in plain language (English),
3. source code changes to fix the bug (C source code)
4. A list of collaborators with explanation of what each person did (or "No collaborators" if you did this alone).
5. (optional) a justification that the bug is "interesting" (see below)

The bugs we're looking for are basic errors such as:

1. Not validating user input.
2. Assuming library functions will always succeed, and therefore not checking for errors.
3. Using stack-allocated fixed-size buffers.
4. Using unsafe C string functions.

Each person who posts a bug first with the first 3 components will get a extra credit (1%) for this assignment, capped to 3%. If 3 or more interesting bugs and bug fixes are found, the whole class will receive a 1% extra credit on this assignment. Here, "interesting" is defined as a bug that can be triggered by a remote user/client and causes the server to 1) crash, or 2) go into an infinite loop, or 3) something I find interesting. You should attempt to make a case that the bugs you find are interesting in your post. Note the SIGPIPE issue noted above is a bug, but is out of the scope for this part of the assignment.

An easy way to locate these vulnerabilities is to download and compile the Tiny web server on a machine with a recent GCC/clang compiler (e.g. the `cycle` machines). Read the warnings carefully to identify potential vulnerabilities. Focus on `tiny.c`, though bugs exist elsewhere too in the Tiny source code and are within scope.

Part III: Attacking tiny.c

Armed with the list of bugs from the collective analysis, now you must *individually* develop a buffer overflow attack for `tiny.c`.

You will send a request that overflows one of the fixed-size stack-allocated buffers. On most modern systems, protections are enabled that will cause the web server to shut down (see below) leading to what appears to be a denial-of-service attack. However, on older systems (or with the protections disabled), the server will actually execute instructions at an address controlled by the attacker (i.e. you).

IMPORTANT: Most modern systems use a stack canary and will detect the buffer overflow and the tiny web server will be terminated with a message like this (these messages will appear in the terminal that the tiny program is running):

```
*** stack smashing detected ***: <unknown> terminated
```
or:

```
*** buffer overflow detected ***: ./tiny terminated
```
Therefore, for the purpose of this assignment, we will disable these mitigations:

Edit the `Makefile` and replace the line for `CFLAGS` with the following line:

```
CFLAGS = -g -Wall -I. -fno-stack-protector
```
Delete `tiny` (`rm tiny`), and recompile. Note that I also disabled optimization which simplifies the attack.

Buffer-overflow attack

Write a program `buf.c`, that accepts two parameters, one a port and another a 64-bit integer representing a memory address. Your program must construct a request that causes Tiny to begin execution at that memory address -- i.e. you have taken control of execution. This program will be called like this (from another terminal, and assuming you have already started `tiny`):

```
./buf 5678 0xaabbccddeeff1122
```

The `buf` program should connect to port 5678 on `localhost` and send a request *smaller than 10000 bytes* that causes a buffer overflow in the tiny web server, causing it to execute code at address `0xaabbccddeeff1122`. Use your understanding of the bugs in tiny to construct the exact request that will cause this behaviour to occur. You can reuse code from `download.c`.

If your request triggers the attack, `tiny` will crash like this:

```
Segmentation fault (core dumped)
```

Now, to verify your request is actually overwriting the return address on the stack, run `tiny` under GDB as follows:

```
$ gdb --args ./tiny 5678   # change the port number if needed
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
 [omitted]

(gdb) run

 [omitted, tiny will wait for payload to be sent from client
 separately, in another terminal, as before, send the request]

Program received signal SIGSEGV, Segmentation fault.

 [omitted]

(gdb) p $rip
$1 = (void (*)()) 0x555555556bf9
(gdb) bt
#0  [omitted]
#1  0xaabbccddeeff1122 in ?? ()

 [omitted]
```

As you can see, the return address has been changed to `0xaabbccddeeff1122`. The other addresses (such as `$rip`) will be different.

At this point, with some more time, you could get the tiny web server to potentially execute anything you wanted. However, the assignment requires you to only reach this point.

How to solve this

First, create a request that causes a buffer overflow. Then, use `gdb` to detect where the buffer overflow occurred (this needs the `-g` flag to `CFLAGS`). Once you have identified the function(s?) with the buffer overflow, *think* how you would construct a request to exploit it. Alternatively, compile `tiny.c` with address sanitizer enabled and use its error messages to guide you.

Finally, construct a request that causes the return address to be overwritten with a user-supplied value. Note this second stage will only work if the stack canaries are disabled as noted above.

You may want to read up on what (obsolete) HTTP/1.0 requests look like. See especially Section 4, and everything before that you need to understand that section.

Ask questions on Blackboard as early as possible.

Other helpful notes

You can use the `curl` program to talk to `tiny` as well. Assuming `tiny` is listening on port 8591:

```
curl http://localhost:8591/
```

Should connect and show you the response body. You can also view the headers:

```
curl -i http://localhost:8591/
```

should show you the headers that the server sends. **Update:** Originally the instructions had `-I` which gives you `501 Not implemented`.