Name: Brian Jon Stout
Date: 24JUN2017
Current Module: Weird week with Liam
Project Name: Ashti

Project Goals

The goal of this project is to write ashti web server. The web server should be multiplex and handle multiple clients at the same time. The web server does not have to be robust, but at a minimum support HTTP codes 200, 404, and 500.

Considerations

- Server must be able to respond properly to a web browser
- Server must be able to respond to GET requests
- Server must be able to reply with codes 200, 404, or 500
- Server must be able to server web pages

Initial Design

The main file is called ashti.c. The support files are files.h, request.h, and response.h.

Ashti is responsible for setting up the multiplex tcp server, and forking off processes when a client accepts. It maintains the proper current working directory for the server, and ties the requests with the responses.

Files contains so miscellaneous functions for checking if directories or files exists. It also has a function that tests a path to see if it contains path traversal.

Request contains functions that process what the server receives from the browser. As of now it only supports GET request, but it's set up in a way to expand. There's a function that checks for headers and return -1 if it's not valid. The other function processes a GET request and returns a FP to the page it wants if it exists.

Response contains functions that decide how to send the web page back to the client. It picks the correct header to send back, and then loops through all the data it can get from the fp passed by the request functions. There's five functions, three of which are specifically for error responses. The first main function is the get_response, which responds with a normal web page. The second function is the cgi_response, which opens up an executable file and sends the output from that file.

The expected format for the web server's folder structures is that a cgi-bin folder and a www folder exists in a root directory the binary is also in the same directory of. The default root_directory is called www_root. If you specify a path to a root directory the binary can be anywhere on the server filesystem where it has the proper permissions.

Data Flow

The program starts by checking the argument values. An argc of one means the binary needs to be in the same directory as www_root. If it is not it will error out. If argc is two than the program knows the user provided a path to the directory. The directory must contain a www folder for web pages, and if executables are placed in the cgi-bin folder. In hindsight, if those folders didn't exist in the specified directory the program should have created them for user friendliness, but for now we'll just assume the user specifies a directory already containing this format, because otherwise the server will not find regular web pages.

The program changes the current working directory to the directory of wherever the use specified or left as default. This is done to simplify path operations.

After this the program starts the steps to start a TCP multiplex server using code that was borrowed from the examples created in class. The code is exactly the same up until the fork process, which runs the run_server function.

The run_server function is where the client/server send and receive data, and then that data is processed appropriately. It starts by memory setting a buffer to avoid some errors. The client's request is placed in the read_buf, sized 512 since that should be larger enough to keep any get request. If a browser attempts to make a request of more than 512 bytes it may continue to keep sending data after the server has closed the connection already.

After this, the program runs strtok to get the first words of the request. Currently the server only supports the GET request. The program sends that token to the determine_request function which strncmps the token to an array of words to determine which request it is. GET returns a value of 1, and invalid response returns a response of -1. The number is used in a switch for the program to determine which steps it needs to take afterwards. Right now anything except a one results in a 500 error.

If given more time the server could have had more robust error reporting here. All the cases would be covered as not implemented (0, and 2-7 options.) and a result of -1 would result in a 500 server error/bad data.

With the switch option of 1 the server knows it's processing a get request. It strtoks the token again and passes it to the detect_path_traversal function. This function is responsible for detecting path traversal attempts, however it appears that most browsers eliminate path traversal attempts on it's own. However if a path traversal attempt is sent through netcat, this function will eliminate it. The detect_path_traversal function does so by simply looking for two dots in a row. Absolute paths do not need to be checked for since the other functions automatically search in the cgi-bin relatively or the www folder relatively.

If a path traversal is detected, the server will print out a 400 error and break out of the switch statement. If this is all good it'll move on to the get_request function. Get_request() attempts to determine three possibilities. The first is that the client is making a homepage request (just a '/'.) The second is that the client is making a cgi-bin request. The third is that the client is attempting to pull a different webpage from the www folder.

On the first the server fopen's www/index.html, the default location for the home page. The second the program returns NULL for the fp, and sets the cgi_bin_request boolean to true. The third the program creates a custom path to www + the token, and then fopens that path. After this the function returns the file pointer back to ashti.

Back at run_server function in ashti, the filepointer is assigned to fp in ashti. The bool value is checked, and if it is the program runs the cgi_response function. If it isn't the program checks if the file pointer is null, if it is, it runs the error_404 function, if it isn't it runs the get_response function.

The cgi_response function first determines if the binary being request even exists in the cgi bin, if it does it creates a process stream with popen. If the process stream is successful (and it pretty much always is unless a system call fails) it will send a standard http 200 ok header to the client, and then send the stdout of the process to the client. The program then checks to see if the buffer ever had any data in it, in the event that the program didn't execute properly and created a shell error. This is done because popen doesn't actually return any errors from the binary itself, because it always opens up the shell properly, and the shell is what runs the process and redirects it's stdout back to our program. It might be possible to redirect the error stream for the shell back but I'm not sure how to do this, but if anything was read to the buffer at all, the first value shouldn't be null since the fgets stops at null, it wouldn't read a null into it.

The get_response function reads the data from the fp passed by the get_request function. It's a similar process to cgi_response in that it sends a 200 ok header, then fgets the rest of data at the fp from that stream.

The error_404 function is a hard coded way for the program to send a 404 error to the client. It checks at the start to see if a 404.html file exists in the error folder (an optional folder next to www and cgi-bin) and if it does it loads that up and sends it instead.

If the switch receives invalid input, it defaults to the error_500 function, which does something similar to the error_404 function but sends text related to the 500 response code instead. Ideally all of this would have been packaged up into one error function that can be passed a token, a specific error, and then generate a more useful error message for the client, but it would have been much more complicated and time consuming when other features needed to be completed at the time.

At the end of all this the connection is closed and the function returns a 0 which the fork returns to successfully complete and close the fork process.

Communication Protocol

TCP, HTTP

Potential Pitfalls

- Not processing browser requests correctly
- Not having enough memory to receive the GET request
- Path traversal attacks
- Not handling missing files properly

Test Plan

User Test:

1. Connect from a browser
2. Connect from a browser requesting a specific page
3. Connect from a browser requesting a nonexistent page
4. Connect from a browser requesting a cgi-bin binary
5. Connect from a browser requesting a nonexistent cgi-bin binary
6. Attempt path traversal through the browser
7. Attempt path traversal through netcat (since the browser isn't working)

Test Cases:

1. Browser loads up default home page, images won't get loaded
2. Browser loads up that page from the www folder
3. Browser reports a 404 error, isn't centered though, can't figure my html out
4. Server returns the stdout from the python file
5. Browser reports a 404 error
6. Browser eliminates path traversal, and it's impossible to access anything outside the www folder or cgi-bin
7. Server reports path traversal and a 400 error is returned

Conclusion

More time with the program would have been nice. It was kind of cool attempting to make a server because it kind of took away the magic of how web servers and clients worked and all you really need is the RFC for http and you could code a web server in almost any language that supports networking sockets. There was one piece of reachable memory that I could never get rid of but that's just how it is. There was some sketchy parts with popen and file pathing that might result in some unnecessary 404s but I honestly couldn't figure out how to error check popen properly. This project around I wanted to focus more on planning the server architecture out before coding, and it kind of worked. The initial plan didn't really work out because I had to chop up the original GET() function that handled all the get request stuff into smaller parts to make it manageable (get_request, get_response.) Of course I could have made a GET() function that just did all of the switches logic in that portion but I didn't want to go into argument passing hell, there's a fine line there. On the other hand the server is set up in a way that's pretty easily expandable beyond the excessive logic in the switch statement and the way the server handles errors (easily fixable with more time.)

I also couldn't get pictures to work… unfortunately. No idea where to even start there.