Due Date: Check course website for due date.

This project must be done in groups of 2 students. Please read the syllabus for instructions, deadlines, penalties, and accommodations regarding group formation and management.

1 Introduction

This assignment introduces you to the principles of internetwork communication using the HTTP and TCP protocols, which form two of the most widely used protocols in today's Internet.

In addition, the assignment will introduce you to existing methods for securely representing claims between parties, using the example of JSON Web Tokens as described in RFC 7519 [2].

Last but not least, it will provide an example of how to implement a concurrent server that can handle multiple clients simultaneously.

2 Functionality

The goal of the project is to build a small personal web server that can serve files, stream MP4 video, and provides a simple token-based authentication API.

The web server should implement persistent connections as per the HTTP/1.1 protocol. HTTP/1.1 is specified in a series of "request for comments" (RFC) standards documents (RFC 7230-7237), though the earlier RFC 2616 [1] provides a shorter read.

You should use code we provide as a base from which to start. To that end, fork the repository at `https://git.cs.vt.edu/cs3214-staff/pserv`. Be sure to set your fork to be private!

2.1 Serving Files

Your web server should, like a traditional web server, support serving files from a directory (the 'server root') in the server's file system. These files should appear under the root (`/`) URL. For instance, if the URL `/private/secure.html` is visited and the root directory is set to a directory `$DIR` that contains the directory `private`, the content of the file `$DIR/private/secure.html` should be served. You should return appropriate content type headers, based on the served file's suffix. Support at least .html, .js, .css, and .mp4 files; see `/etc/mime.types` for a complete list.

Make sure that you do not accidentally expose other files by ensuring that the request url's path does not contain .. (two adjacent periods), such as `/public/../../../../../etc/passwd`.[1]

You should return appropriate error codes for requests to URLs you do not support.

2.2 Authentication

You must, at a minimum, support a single user that will authenticate with a username and password. If the user is authenticated, they should have access to the secure portion of your server, which are all files located under `/private`. Otherwise, such access should be denied.

Your server should implement the entry point `/api/login` as follows:

- When used as the target of a POST request, the body of the request must contain

  ```
  {"username":"user0","password":"thepassword"}
  ```

  where 'user0' is the name of the user and 'thepassword' is their password. If the password is correct, your server should respond with a JSON object that describes claims that the client can later use to prove it has successfully authenticated.

  Send (at least) the following claims: (a) `sub` - to describe the subject (the principal as which the server will recognize the bearer of the claim), (b) `iat` - the time at which the claim was issued, in seconds since Jan 1, 1970, and (c) `exp` - the time at which the claim will expire.

  For example, a claim may look like this:

  ```
  {"exp":1523737086,"iat":1523650686,"sub":"user0"}
  ```

  Returning the claims in the response, however, is not sufficient. The client must also obtain a signature from the server that certifies that the server issued the token (i.e., that the user's password was correct and thus the user has successfully authenticated).

  This signature is obtained in the form of a JSON Web Token, which the server should return as a cookie to the client. You may choose an appropriate signing mechanism (either HMAC or using a private/public key pair using RSA, suggested is HMAC). You may use the jansson and libjwt libraries which are installed as part of the provided code. Check out the files `jwt_demo_hs256.c` and `jwt_demo_rs256.c` for examples.

  See MDN for documentation on the format of the `Set-Cookie` header which you must follow. Make sure to set the cookie's path to / so that the cookie is sent along for all URIs. You may choose a suitable cookie name such as `auth_token`.

[1] Technically, RFC 3986 suggests that you remove those dot segments using an algorithm, but for the purposes of this project we'll demand that the client apply this algorithm and reject any URLs for which it hasn't.

You should also set an expiration time for the cookie via the `Max-Age` attribute, which you should set to the expiration time (in seconds) of the token. Your cookie should also be HTTP-only (set the `HttpOnly` attribute).

If the username/password does not match, your server should return 403 Forbidden.

- When used in a GET request, `/api/login` should return the claims the client presented in its request as a JSON object if the user is authenticated, or an empty object {} if not.

  Be sure to validate tokens before deciding whether the client is authenticated or not; do not accept tokens that have expired or whose signature does not validate.

  You should implement this without storing information about which cookies your server has issued server-side, but rather simply by validating the token the client presents.

  Your server should implement the entry point `/api/logout` as well. When used in a POST request, your server should ask the client to clear the cookie from its cookie store by returning a `Set-Cookie:` header for the cookie in which the `Max-Age` attribute is set to `0`.

  The type of "stateless authentication" can be used to provide a simple, yet scalable form of authentication. Unlike in traditional schemes in which the server must maintain a session store to remember past actions by a client, the presented token contains proof of past authentication, and thus the server can directly proceed in handling the request if it can validate the token. Moreover, this way of securely presenting claims allows authentication servers that are separate from the servers that provide a resource or service: for instance, if you log onto a website via Google or Facebook, their authentication server will present a signed token to you which you can later use to prove to a third server that Google or Facebook successfully authenticated you.

  However, such stateless authentication also has drawbacks: revoking a user's access can be more difficult since a token, once issued, cannot be taken away. Thus, the server either has to keep revocation lists (in which case a session-like functionality must be implemented), or keep token expiration times short (requiring more frequent reauthentication or a token refresh scheme), or by changing the server's key (which invalidates all tokens for all users). For this assignment, you do not need to implement revocation.

  We recommend you read the Introduction to JSON Web Tokens tutorial by Auth0. Note that JSON Web Tokens are not the only technology that make uses of cryptographically signed tokens. Others include PASETO and IRON Session.

2.3 Supporting HTML5 Fallback

Modern web applications exploit the History API, which is a feature by which JavaScript code in the client can change the URL that's displayed in the address bar, making it appear to the user that they have navigated to a new URL when in fact all changes to the page were driven by JavaScript code that was originally loaded. This is also known as "client-side routing," see React Router for how this is accomplished in the popular React.js framework.

When a URL that was modified in this way is bookmarked and later retrieved, or if the user refreshes the page while the modified URL is displayed, a request with this URL will be sent to the server, but it does not correspond to an existing server resource. In this case, the server should be programmed to return a "fallback" resource rather than 404. When your server is run with the `-a` flag it should return this fallback resource, specifically the file `index.html` in its root directory. (As a sidenote, this ability is provided by the `nginx` server using the try_files directive.)

When HTML5 fallback is enabled, your server should return `index.html` in its root directory also for requests to `/`. Otherwise, you may select a suitable response for requests to other, existing directories. (Some servers disallow such accesses, others can be configured to return an HTML page with a listing of the files in a directory.)

2.4 Streaming MP4

To support MP4 streaming, your server should advertise that it can handle Range requests to transfer only part (a byte range) of a file. You should send an appropriate `Accept-Ranges` header and your server should interpret `Range` headers sent by a client.

To support a basic streaming server, it is sufficient to support only single-range requests such as `Range: bytes=203232-` or `Range: bytes=500-700`. Be sure to return an appropriate `Content-Range` header. Browsers will typically close a connection (and create a new one) if the user forwards or rewinds to a different point in the stream.

To let clients learn about which videos are available for streaming, your server should support an entry point `/api/video`. GET requests to this entry point should return a JSON object that is a list of videos that can be served, in the following format:

```
[
  {
    "size": 1659601458,
    "name": "LectureVirtualMemory.mp4"
  },
  {
    "size": 961734828,
    "name": "Boggle.mp4"
  },
```

```
  {
    "size": 1312962263,
    "name": "OptimizingLocking.mp4"
  },
  {
    "size": 423958714,
    "name": "DemoFork.mp4"
  }
]
```

Use the `opendir(3)` and `readdir(3)` calls to list all files in the server's root directory (or a subdirectory, at your choosing), selecting those that carry the suffix `.mp4`. Use the `stat(2)` system call to find the size of each file.

2.5 Multiple Client Support

For all of the above services, your implementation should support multiple clients simultaneously. This means that it must be able to accept new clients and process HTTP requests even while HTTP transactions with already accepted clients are still in progress. You **must use** a single-process approach, either using multiple threads, or using an event-based approach.[2]

If using a thread-based approach, it is up to you whether you spawn new threads for every client, or use a thread pool. You may modify or reuse parts of your thread pool implementation from project 2, if this is useful.[3]

To test that your implementation supports multiple clients correctly, we will connect to your server, then delay the sending of the HTTP request. While your server has accepted one client and is waiting for the first HTTP request by that client, it must be ready to accept and serve additional clients. Your server may impose a reasonable limit on the number of clients it simultaneously serves in this way.

2.6 Robustness

Network servers are designed for long running use. As such, they must be programmed in a manner that is robust, even when individual clients send ill-formed requests, crash, delay responses, or violate the HTTP protocol specification in other ways. *No error incurred while handling one client's request should impede your server's ability to accept and handle future clients.*

[2]For the purposes of this project, a multi-process approach is not acceptable.

[3] Please note, however, that the fork-join thread pool was implemented with a different goal in mind and that some aspects here do not apply to this project, notably the fork-join aspect. We recommend trying a thread-based approach first.

This semester we will be using a research prototype of a new fuzzing software to test your server software. Instructions for how to do this will be separately provided.

2.7 Performance and Scalability

We will benchmark your service to figure out the maximum number of clients and rate of requests it can support. Note that for your server to be benchmarked, it **must obtain a full score in the robustness category first**. We will publish a script to benchmark your server. A scoreboard will be posted to compare your results with the rest of the class.

2.8 Protocol Independence

The Internet has been undergoing a transition from IPv4 to IPv6 over the last 2.5 decades. To see a current data point, Google publishes current statistics on the number of users that use IPv6 to access Google's services. This transition is spurred by the exhaustion of the IPv4 address space as well as by political mandates.

Since IPv4 addresses can be used to communicate only between IPv4-enabled applications, and since IPv6 addresses can be used to communicate only between IPv6-enabled applications, applications need to be designed to support both protocols and addresses, using whichever is appropriate for a particular connection. For a TCP/UDP server, this requires accepting connections both via IPv6 as well as via IPv4, depending on which versions are available on a particular system. For a TCP/UDP client, this requires to identify the addresses at which a particular server can be reached, and try them in order. Typically, if a server is reachable via both IPv4 and IPv6, the IPv6 address is tried first, falling back to the IPv4 address if that fails.

Ensuring protocol independence requires avoiding any dependence on a specific protocol in your code. Fortunately, the socket API was designed to support multiple protocols from the beginning as its designers foresaw that protocols and addressing mechanisms would evolve. For instance, the `bind()` and `connect()` calls refer to the addresses passed using the type `struct sockaddr *` which is an opaque type that could refer to either a IPv4 or IPv6 address.

To implement protocol independence, you need to avoid any dependence on a particular address family. Accordingly, you should use the `getaddrinfo(3)` or `getnameinfo(3)` functions to translate from symbolic names to addresses and vice versa and you should avoid the outdated functions `gethostbyname(3)`, `getaddrbyname(3)`, or `inet_ntoa(3)` or `inet_ntop(3)`.

Tutorials on how to write protocol independent network code are given in this resource and in the code for the textbook's 3rd edition. However, neither tutorial is fully correct and will require (minor) adaptations.

Ensuring that your server can accept both IPv4 and IPv6 clients can be implemented using two separate sockets, one bound to either family. Two separate threads can then be devoted to these sockets to accept clients that connect using either of the two protocol families.

However, the Linux kernel provides a convenience feature that provides a simpler facility for accepting both IPv6 and IPv4 clients. This so-called dual-bind feature allows a socket bound to an IPv6 socket to accept IPv4 clients. Linux activates this feature if /proc/sys/net/ipv6/bindv6only contains 0. You may assume in your code that dual-bind is turned on. [4]

Our starter code uses protocol independent functions, but it is tested with IPv4 only. Augmenting it to implement protocol independence is part of your assignment.

2.9 Choice of Port Numbers

Port numbers are shared among all processes on a machine. To reduce the potential for conflicts, use a port number that is $10,000$ + last four digits of the student id of a team member.

If a port number is already in use, `bind()` will fail with `EADDRINUSE`. If you weren't using that port number before, someone else might have. Choose a different port number in that case. Otherwise, and more frequently, it may be that the port number is still in use because of your testing. Check that you have killed all processes you may have started on the machine you are working on while testing. Even after you have killed your processes, binding to a port number may fail for an additional 2 min period if that port number recently accepted clients. This timeout is built into the TCP protocol to avoid mistaking delayed packets sent on old connections for packets that belong to new connections using the same port number. To override that, you may use setsockopt() with the `SO_REUSEADDR` flag to allow address reuse.

3 Strategy

Make sure you understand the roles of DNS host names, IP addresses, and port numbers in the context of TCP communication. Study the roles of the necessary socket API calls.

Since you may be using a multi-threaded design, use thread-safe versions of all functions.

Familiarize yourselves with the commands `wget(1)` and `curl(1)` and the specific flags that show you headers and protocol versions. These programs can be extremely helpful in debugging web servers.

[4] I should point out, however, that this will make your code Linux-specific; truly portable socket code will need to resort to handling accepts on multiple sockets.

Refresh your knowledge of `strace(1)`, which is an essential tool to debug your server's interactions with the outside world. Use `-s 1024` to avoid cutting off the contents of reads and writes (or recv and send calls). Don't forget -f to allow strace to follow spawned threads.

4 Grading

4.1 Coding Style

Your service must be implemented in the C language. You should follow proper coding conventions with respect to documentation, naming, and scoping. You must check the return values of all system calls and library functions.

Your code should compile under `-Wall` without warnings, the use of the `-Werror` flag as part of CFLAGS should have become a habit by now, as is the use of git for revision control.

4.2 Submission

You should submit a .tar.gz file of the `src` directory of your project, which must contain a Makefile. Your project should build with 'make clean all' This command must build an executable named 'server' that must accept the following command line arguments:

- `-p port` When given, your web service must start accepting HTTP clients and serving HTTP requests on port 'port.' Multiple connection must be supported.
- `-R path` When given, 'path' specifies the root directory of your server.
- `-s` Silent mode (for benchmarking). When given, your server should suppress any output to standard output.
- `-e sec` Specify the expiration time for the issued JWT in seconds. Your server must enforce this expiration time.
- `-a` HTML5 Fallback mode. When given, requests for non-existing resources should be responded to as if the request had been for `/index.html`.

Please test that 'make clean' removes all executables and object files. Issue 'make clean' before submitting to keep the size of the tar ball small. Please use the submit.py script or web page and submit as 'p4'. Only one group member need submit.

Further submission instructions are posted on the course website.

This project will count for 120 points.

Good Luck!

References

[1] Roy Fielding, Jim Gettys, Jeff Mogul, H. Frystyk, L. Masinter, P. Leach, and Tim Berners-Lee. Rfc 2616: Hypertext transfer protocol – http/1.1. http://www.w3.org/-Protocols/rfc2616/rfc2616.html.

[2] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt), 2015. RFC7519.