CS 3214: Project 4

Personal Web and Video Server

Help Session: Wednesday April 30th, 2025 - 7:00pm EST

Thomas Tran (thomastran@vt.edu)

Topics

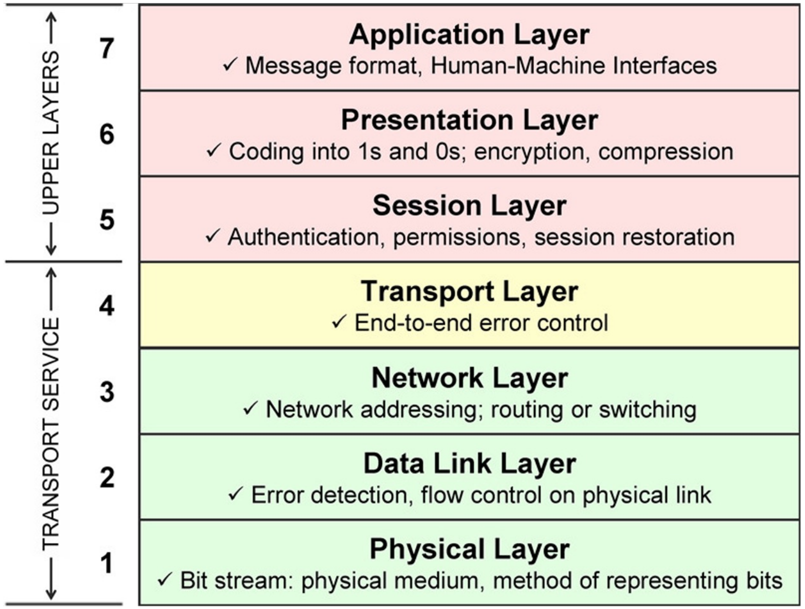
- Overview of a Web Server (prerequisite knowledge)
  - OSI, TCP, HTTP, JSON, JWT
- Basics / Getting Started
- Web Server Design
  - Serving Files
  - Authentication
  - Robustness, Performance, & Scalability
  - IPv6
  - MP4 Streaming
- Logistics and Grading
- Fuzzing!

Overview of Web Server

Prerequisite Knowledge:
OSI, TCP, HTTP, JSON, JWT

OSI Model

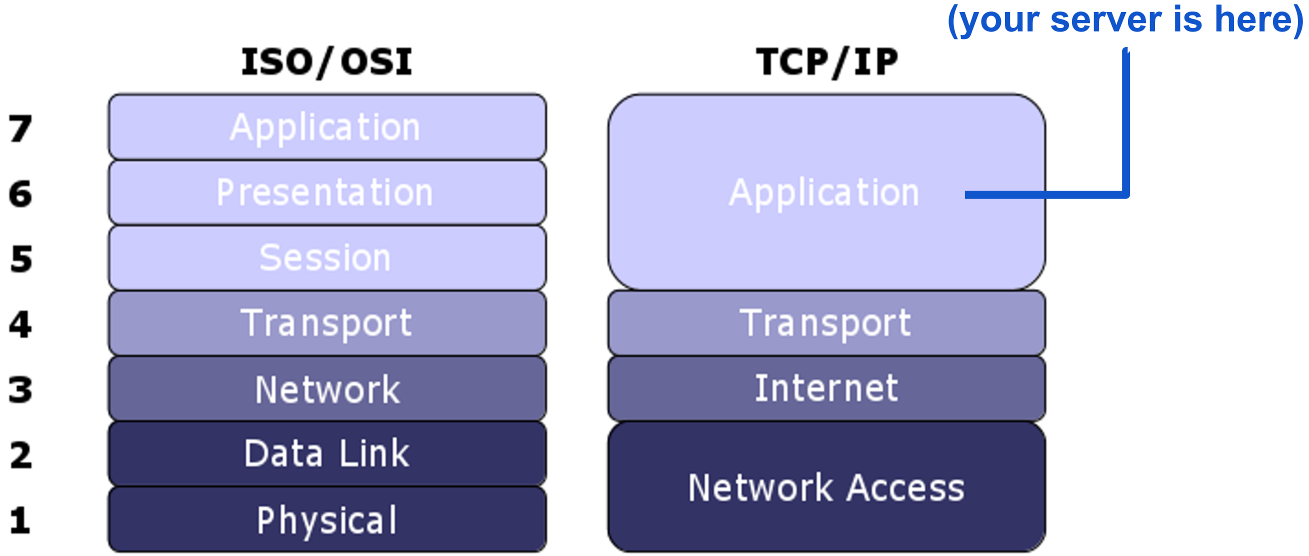
- Network “Stack”



UPPER LAYERS	7	**Application Layer** ✓ Message format, Human-Machine Interfaces
	6	**Presentation Layer** ✓ Coding into 1s and 0s; encryption, compression
	5	**Session Layer** ✓ Authentication, permissions, session restoration
TRANSPORT SERVICE	4	**Transport Layer** ✓ End-to-end error control
	3	**Network Layer** ✓ Network addressing; routing or switching
	2	**Data Link Layer** ✓ Error detection, flow control on physical link
	1	**Physical Layer** ✓ Bit stream: physical medium, method of representing bits

OSI Model

- Slightly more modern approach



Socket Programming

- Medium through which programs access network
- System calls:
  - `socket()`: create the socket file descriptor
  - `bind()`: assign to (local) address and port
  - `listen()`: start queueing incoming requests
  - `accept()`: connect to a client, return new socket

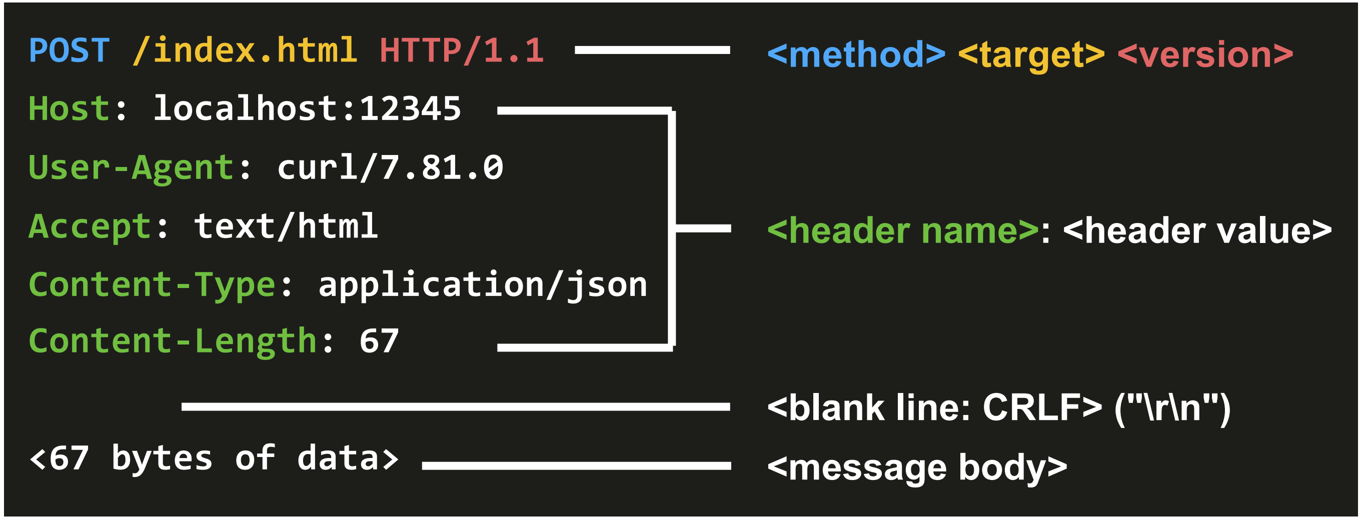
All sockets by default are blocking

HTTP

- Hypertext Transfer Protocol
- Exists in the application layer of the OSI model
  - Normally takes place over TCP/IP connections
- Developed at CERN in 1989 and governed by W3C (World Wide Web Consortium)
- **Request** and **Response** messages use verbiage to denote intent
  - GET, POST, PUT, DELETE
  - Stateless

HTTP Requests

Version 1.1 requests are structured as follows:



```
POST /index.html HTTP/1.1
Host: localhost:12345
User-Agent: curl/7.81.0
Accept: text/html
Content-Type: application/json
Content-Length: 67

<67 bytes of data>
```

<method> <target> <version>

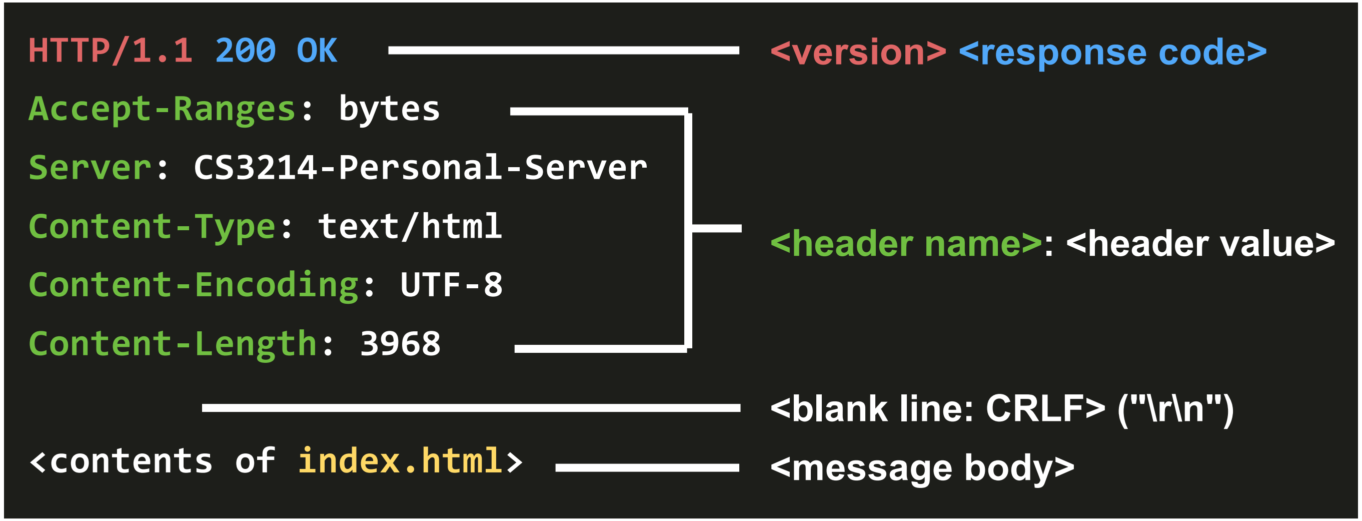
<header name>: <header value>

<blank line: CRLF> ("\r\n")

<message body>

HTTP Responses

Version 1.1 responses are structured as follows:



```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Server: CS3214-Personal-Server
Content-Type: text/html
Content-Encoding: UTF-8
Content-Length: 3968

<contents of index.html>
```

<version> <response code>

<header name>: <header value>

<blank line: CRLF> ("\r\n")

<message body>

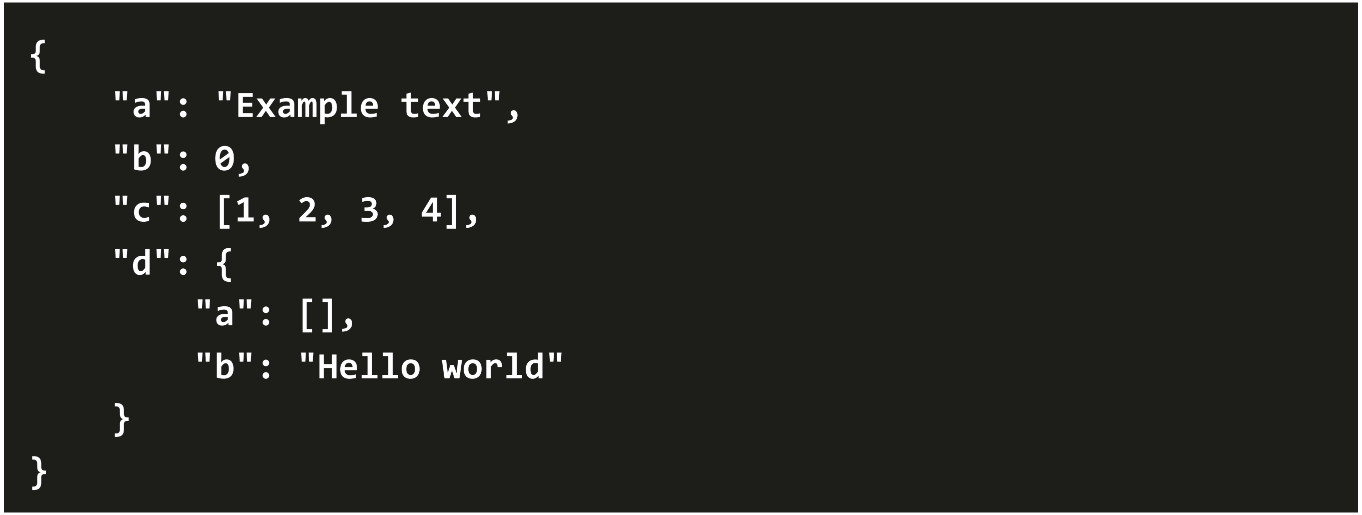
HTTP Standard

- Each line ends in:
  - CR: carriage return, **`\r`**
  - LF: line feed, **`\n`**
- Has version and status
- Optional header fields
- Blank CRLF, then message content (if any)
- HTTP status codes

JSON

“Javascript Object Notation”

Key, value store in a well-defined format



```
{
    "a": "Example text",
    "b": 0,
    "c": [1, 2, 3, 4],
    "d": {
        "a": [],
        "b": "Hello world"
    }
}
```

JSON Web Tokens

- JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties
- Debugged on main website: https://jwt.io
- Three parts:
  - **Header**
  - **Payload**
  - **Signature**

Example JWT

Encoded JWT token is delimited by dots

```
eyJ0eXAiOiJKV1QiLCJhbGciOi
JIUzI1NiJ9.eyJleHAiOjE2OTc
yNzE2MDAsImlhdCI6MTY5NzE4N
TIwMCwic3ViIjoidXNlcjIwMjM
ifQ.qtaLIlrQ23PemNtCeEMOla
P3vaWtfXbYJQfWEzbPy30
```

```
{
  "typ": "JWT",
  "alg": "HS256"
}
{
  "exp": 1697271600,
  "iat": 1697185200,
  "sub": "user2023"
}
HMACSHA256 signature
```

You'll see this later!

Basics / Getting Started

Getting Started

- Fork / clone the repo
  - Set to private!
- **Use the provided `./install-dependencies.sh` to set up the project libraries**
- Build the Svelte frontend & add some videos
  - Make sure `npm` and `node` are ones in `~cs3214/bin`!

```
$ git clone <your fork of cs3214-staff/pserv.git>
$ cd pserv && ./install-dependencies.sh
$ cd svelte-app && npm install && npm run build
$ cd ../tests && ./build.sh
$ cd ../src && make
```

Getting Started

- Understand the code
    - The front-end (Svlete App), files, etc. is handled for you
- What do we write?
    - Any files you like, modifying `http.c` heavily
    - Hint: You’re only messing with 4 files! ✨
- Handle
    - Authentication
    - IPv4 and IPv6 dual support
    - HTML5 Fallback
    - Multi-client support
    - MP4 streaming

Provided Base Code

- Base code already supports:
  - HTTP request parsing,
  - HTTP response building,
  - File mime-type guessing,
  - Serving one client at a time.

Alright, then where do I start?

- Get a feel for static file serving first (GET request to `/something.txt`).
- Start with minimum requirements (`200 OK` response to GET `/api/login`, multiple simultaneous connections) .
- Move to IPv6 support, then authentication functionality.

HTTP Transaction Struct

- Base code parses request headers into structs (think Project 1)
- The information is inside a buffer (struct bufio)
- http_process_headers processes it and stores important info in struct http_transaction
- You should store extra information such as:
  - **`Authentication token`**
  - **`Request range`**
  - **`Content Type`**
- Store as an offset or value? Up to you!

Parsing Arguments

- Already supported for you!
- Supports the following program arguments:
  - `-p <port number>` defines the port to bind()
  - `-R <path>` defines the server root to use
  - `-a` enables HTML5 fallback

(... plus a few more!)

Testing in browser

- Use SSH tunneling

On local machine:

```
$ ssh -L <port>:localhost:<port> <pid>@rlogin.cs.vt.edu
```

(if connecting to a specific host, use **`<host>.rlogin`** in place of **`localhost`**)

On rlogin, start server normally:

```
$ ./server -p <port> -R <root data dir>
```

Open browser to **`localhost:<port>`**

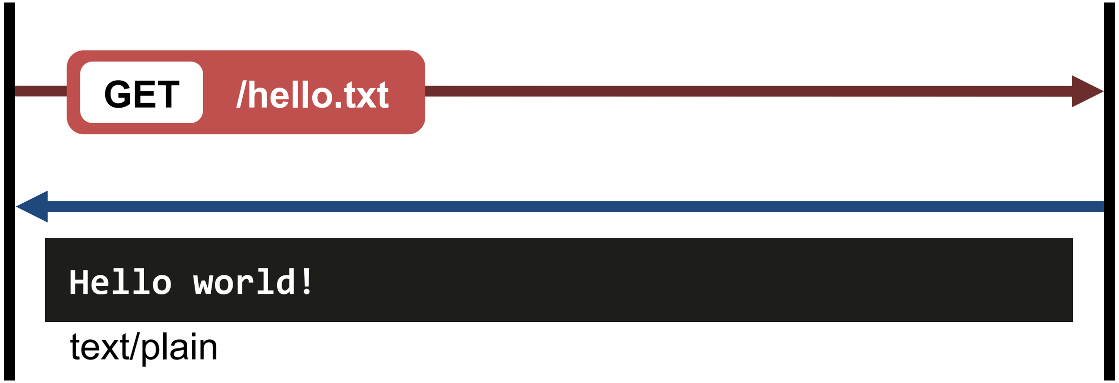
Demo

Getting started
Common pitfalls

Web Server Design

Authentication & Higher-Level Design (and curl)

Serving Static Files



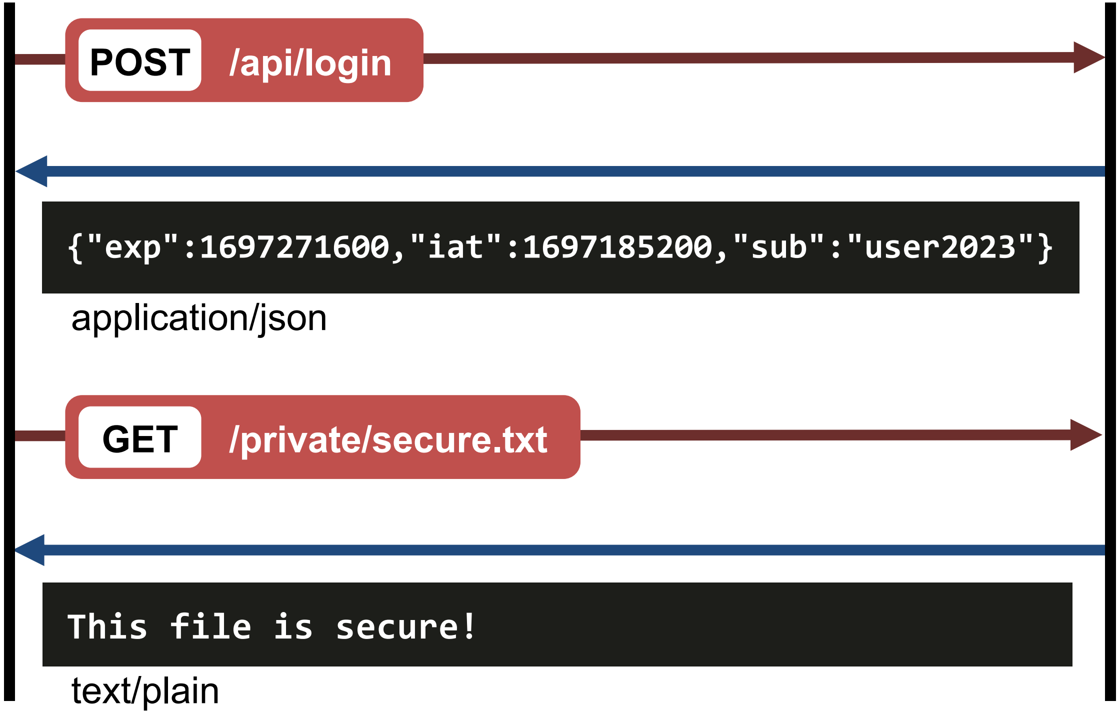
Client asks for the contents of hello.txt

Server opens hello.txt and responds with its contents and type.

- Serve any file in the root directory
    - Be mindful of security vulnerabilities in the provided path (what about '.' and '. .'?)

GET /../../private/passwords.txt

Authentication



Client supplies correct login information

```
{"exp":1697271600,"iat":1697185200,"sub":"user2023"}
```

Server returns authentication token

Client supplies token in a cookie

```
This file is secure!
```

Server looks at cookie, verifies the token, and provides the private file

Auth. Credentials

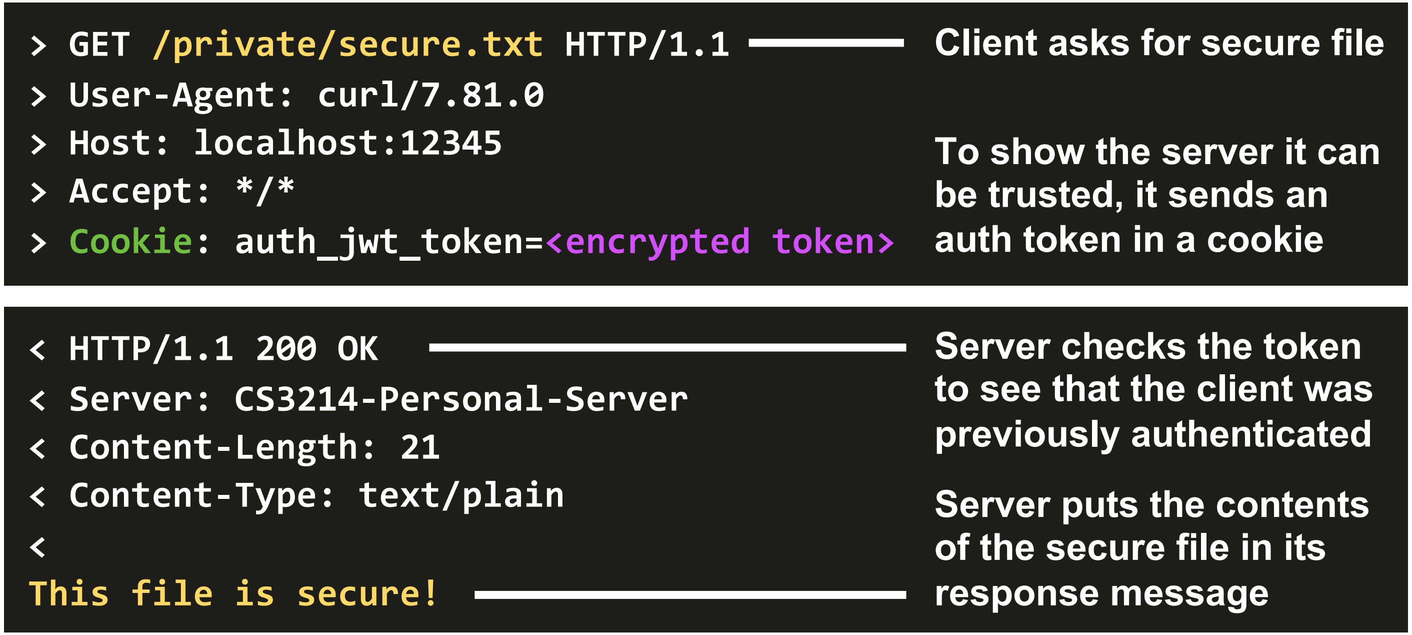
- Only need to handle a single user:

```
{"username":“<USER_NAME>","password":“<USER_PASS>"}
```

- Hardwiring credentials in source code is often bad practice.
- Hard-coding will **not pass** testing!
- The autograder supplies **environment variables**:
  - **`USER_NAME`**
  - **`USER_PASS`**
  - **`SECRET`**
- Use env to supply these to the unit tests.

Secure File Auth.

Checking for the presence of a cookie in the HTTP header



```
> GET /private/secure.txt HTTP/1.1
> User-Agent: curl/7.81.0
> Host: localhost:12345
> Accept: */*
> Cookie: auth_jwt_token=<encrypted token>
```

Client asks for secure file

To show the server it can be trusted, it sends an auth token in a cookie

```
< HTTP/1.1 200 OK
< Server: CS3214-Personal-Server
< Content-Length: 21
< Content-Type: text/plain
<
This file is secure!
```

Server checks the token to see that the client was previously authenticated

Server puts the contents of the secure file in its response message

HTML5 Fallback

- Should a request be sent on every click?
  - “Client-side routing” - updates via JS code
- Clients can change URL in the address bar
  - What if the “fake” URL is bookmarked?
- Policy for a Svelte application (**request** → **fallback**):
  1. Existing file/API → as is
  2. `/` (server root) → `index.html`
  3. `/some/path` → `/some/path.html`
  4. else: `200.html`

Quick Sidenote: curl

- Debugging tool for HTTP requests
- Arguments include urls to query and flags
    - Great way to see the request and response flow between a client and server
    - Helps debug hanging and malformed headers
    - Can chain URLs together
- Flags:
    - `-v`: verbose mode
    - `-0` / `--http1.0`: use HTTP 1.0
    - `--path-as-is`: do not truncate dot dot sequences

curl Examples

Send a POST request with body

```
$ curl -X POST -d \
'{"username":"user2023","password":"passwordf23"}' \
localhost:12345/api/login
```

View headers

```
$ curl -I localhost:12345/private/secure.txt
```

Manually set session cookies

```
$ curl -v --cookie "auth_jwt_token=token" \
localhost:12345/private/secure.txt
```

Demo

Talking to a server using curl

Web Server Design

Robustness, Performance, & Scalability

Multithreaded Servers

- Client threads:
  - Should not bring down / block the whole server
- Ideal case:
  - All threads are doing productive work all the time, like in a threadpool
  - Must be mindful of latency
- Be mindful of return values!

Spawning Threads

- Look for inspiration in literature and other server implementations, like NGINX and Apache
- Suggestions:
  - Repurpose threadpool
  - Epoll set
  - Thread-per-client-connection
- Be mindful of the underlying hardware
- Web servers can be “embarrassingly” parallel because HTTP is stateless
- **DO NOT write a forking/process-based server.**

EPoll

- Asynchronous event listener handling `accept()` and `recv()`
- Threads execute an event loop where they call `epoll_wait()`
  - Kernel returns an array of ready file descriptors
  - Thread is responsible for cleaning up dead connections (and freeing related memory)
  - For best performance, vary number of threads and max size of event array

Web Server Design

IPv6 and Version Conformance

IPv4 versus IPv6

- IPv4
  - Looks like: 192.168.1.30
- IPv6
  - Looks like: 2001:db8:85a3::8a2e:370:7334
- Study the differences between network structures and attributes
- Server must support both IPv4 and IPv6 connections
  - Rlogin supports dual-binding

Version Differences

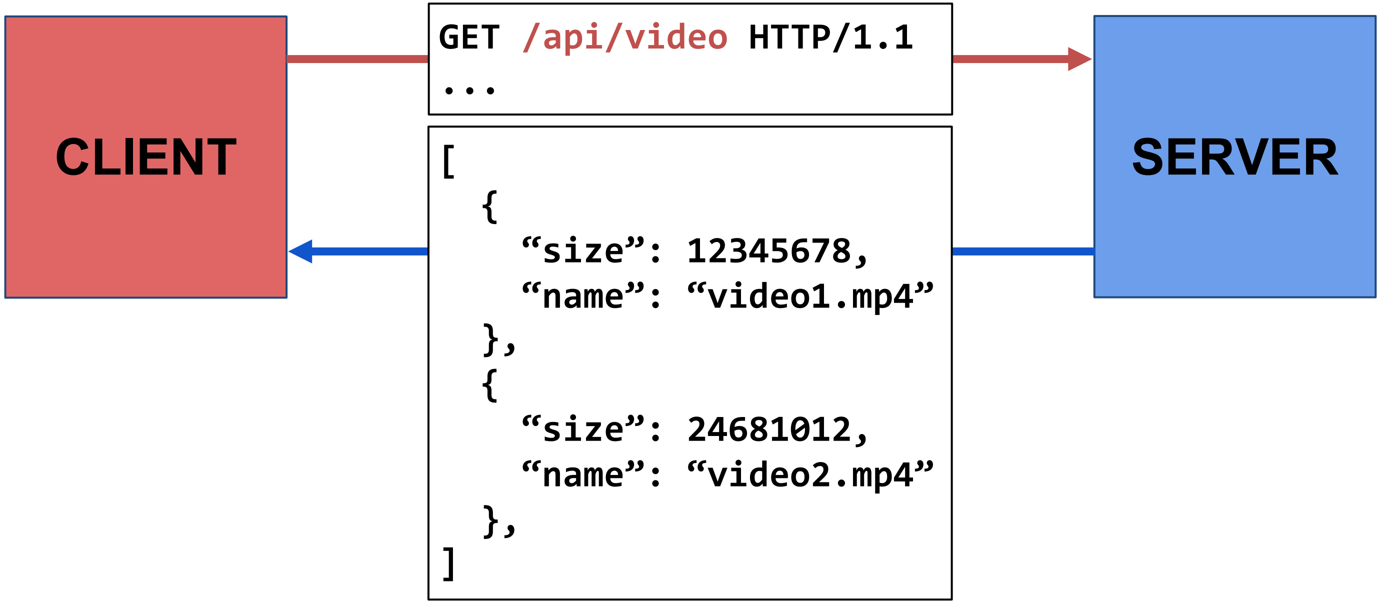
- Persistent connections:
  - HTTP 1.1 by default keeps the connection alive
  - HTTP 1.0 by default closes the connection
  - The connection header is respected
- Additional status states added
- Host header not required for HTTP 1.0, but required for HTTP 1.1

Web Server Design

MP4 Streaming

Video API Endpoint

- Your server will support the /api/video endpoint.
  - Upon GET request, send back a JSON array of videos.

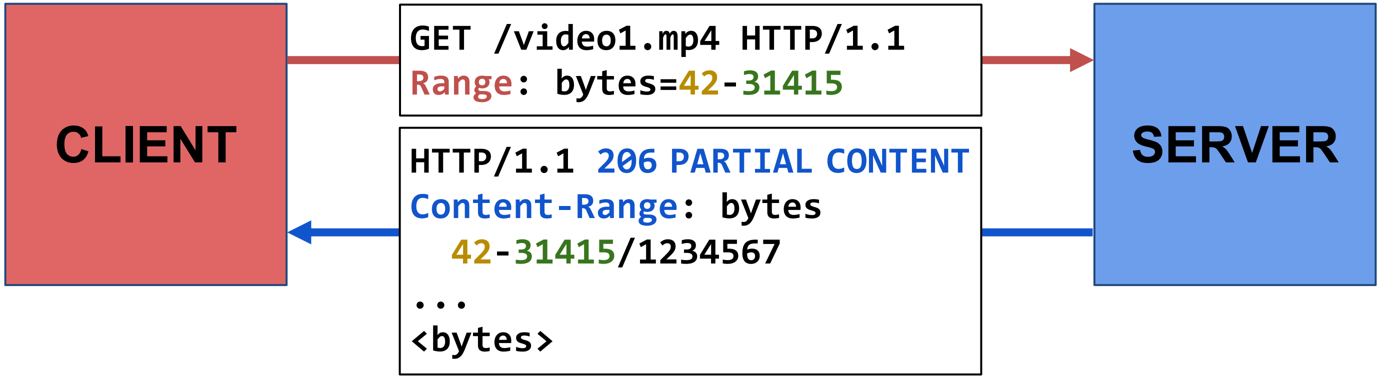


```
GET /api/video HTTP/1.1
...
```

```
[
  {
    “size”: 12345678,
    “name”: “video1.mp4”
  },
  {
    “size”: 24681012,
    “name”: “video2.mp4”
  },
]
```

Range Requests

- Your server will send the `Accept-Ranges` header and accept `Range` headers sent by clients.
  - `Range` header means: “give me bytes `A`-`B` of this file”
- The server responds with a `206 PARTIAL CONTENT` status code and a `Content-Range` header.



```
GET /video1.mp4 HTTP/1.1
Range: bytes=42-31415
```

```
HTTP/1.1 206 PARTIAL CONTENT
Content-Range: bytes
  42-31415/1234567
...
<bytes>
```

Project Logistics

Grading and Advice

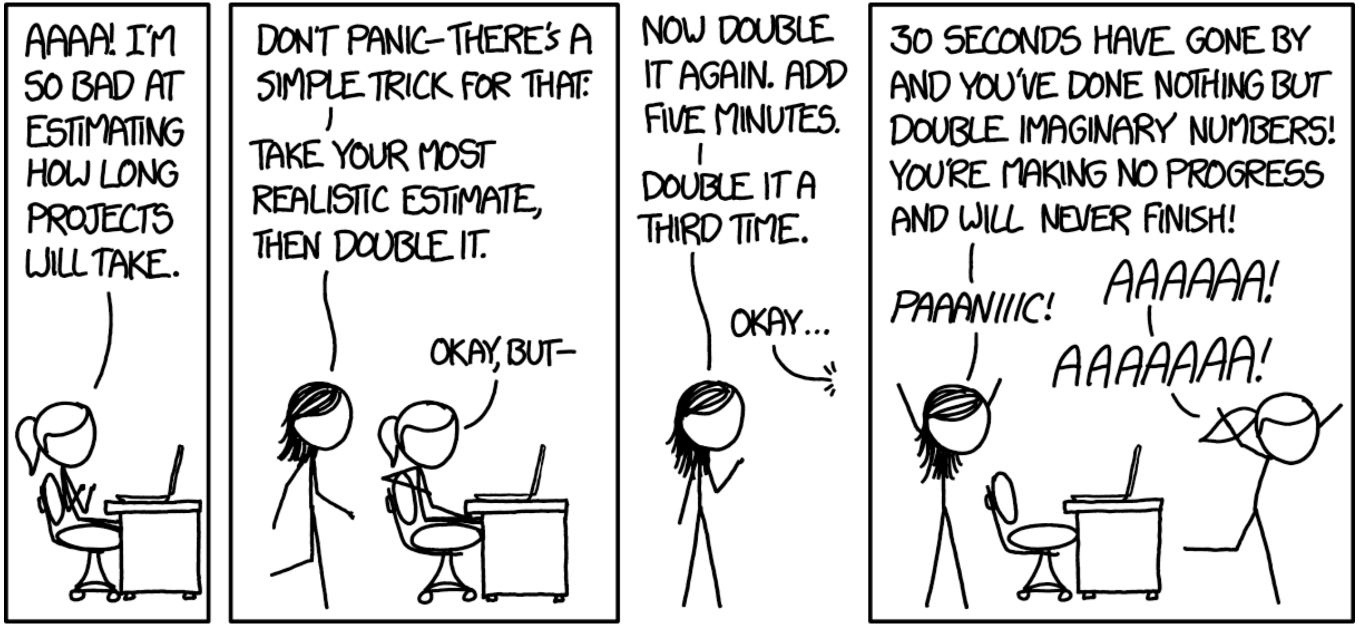
Debugging

- The usual: **`gdb`**, **`strace`**, etc.
- Use curl to simulate interactions
  - HTTPie
  - Postman
- Hexdump function (**`hexdump.c`**)
- Fuzzing utilities

Very relevant skills for life outside of CS 3214

Start Early!

Hard due date: TBD


AAAA! I'M SO BAD AT ESTIMATING HOW LONG PROJECTS WILL TAKE.
DON'T PANIC—THERE'S A SIMPLE TRICK FOR THAT:
TAKE YOUR MOST REALISTIC ESTIMATE, THEN DOUBLE IT.
OKAY, BUT—
NOW DOUBLE IT AGAIN. ADD FIVE MINUTES.
DOUBLE IT A THIRD TIME.
OKAY...
30 SECONDS HAVE GONE BY AND YOU'VE DONE NOTHING BUT DOUBLE IMAGINARY NUMBERS! YOU'RE MAKING NO PROGRESS AND WILL NEVER FINISH!
PAAANIIIC!
AAAAAA!
AAAAAAA!

Logistics

- Please submit code that compiles
- Test using the driver before submitting!
  - Run the tests individually when debugging
  - Run them all at once to see how you'll be graded
- "Passing" a test means that you get the correct result without crashing, within the time limit
  - A failing test can **crash the rest of its section**!
- Full scores required on some sections for others to run:
  - Minimum → auth/extra → malicious → benchmarks
- Benchmarks will be run **after the deadline**
- Benchmarked scores will be the median of 3 runs, assuming you pass all of them

Logistics: Test Points

- Grade breakdown (125 points total):
  - 95 points via `server_unit_test_pserv.py`
    - 25 points Minimum Requirements
    - 20 points Authentication Functionality
    - 5 points HTML5 Fallback
    - 10 points Video Streaming
    - 5 points IPv6 Support
    - 15 points Extra Tests
    - 15 points Robustness (**malicious** tests)
  - 20 points via `server_bench.py` (5 tests × 4 points)
  - 10 points via documentation & version control
- 15 **extra-credit** points via `fuzz-pserv.py`
- 10 **extra-credit** points via superb performance (e.g. EPoll)

Scoreboard

Just like projects 2 and 3, you can submit your performance results to the scoreboard.

```
~cs3214/bin/sspostresult.py
```

See the course website for detailed rules and instructions.

Great way to see how well your server is doing.

> I think this should be a fun project and you'll learn something new, even if you're already an experienced web programmer.
>
> – Dr. Back

Where to start

Concepts

- Read the project spec (Take notes!)
- Understand the starter code (Write comments! Look up system calls!)

Implementation

- Start with serving static files
- Move to authentication (`/api/login`)
- Move to serving `/api/video` and `Range` requests
- Save performance for last (easier debugging)

Helpful Links

The Project Home Page

Socket Programming

- `socket()` man page
- `bind()` man page
- `listen()` man page
- `accept()` man page

HTTP

- Mozilla Documentation - Message Formats

Fuzzing

(Not required, but fun 🙂)

What is Fuzzing?

Fuzzing is a software security testing technique: give a program some unexpected input, with the intention of crashing it or altering its behavior.

It’s a great way to find bugs and security vulnerabilities in our programs. Bugs in web servers are dangerous!

Enter AFL++

AFL++ is a source-code-guided fuzzer that can efficiently find bugs in C programs.

- Originally only works with programs reading from STDIN/files. It runs *forever* until stopped, getting smarter as it goes.
- We've created a library to allow it to work with network sockets, and a series of scripts for you to easily "fuzz" your server.

AFL++ GitHub Repo

AFL++ Website

("We" meaning Dr. Back and Connor Shugg. This was part of a VT CS research project for Connor Shugg's MS thesis.)

AFL++ and your server

Tools have been provided to enable the fuzzing of your servers. Once you’ve got a functional server, give it a whirl!

- **Step 1**: run `~cs3214/bin/fuzz-pserv.py`
  - Let it run. See if it finds some issues!
- **Step 2**: `output_dir/fuzz-rerun-gdb.sh`
  - Run this with the “crash files” or “hang files” discovered by the fuzzer to debug your issues.

(This is an excellent bug-finding and bug-reproducing system!)

Demo

Fuzzing a buggy server

Fuzzing Documentation

Markdown Documentation (multiple locations):

- On the course site
- In the base code repo (check `sfi/`)

Fuzzing Extra Credit

Using the fuzzer allows you to earn extra credit - up to extra points. You get more points the better your server does while the fuzzer is attacking it:

- **Stage 1:** getting the fuzzer running. (**+5**)
- **Stage 2:** fuzzer finds zero bugs in **15 seconds**. (**+2**)
- **Stage 3:** fuzzer finds zero bugs in **2 minutes**. (**+2**)
- **Stage 4:** fuzzer finds zero bugs in **10 minutes**. (**+2**)
- **Stage 5:** fuzzer finds zero bugs in **1 hour**. (**+4**)

Questions?

Thank you for attending!