HTTP

HTTP is plain-text - everything is human readable

To anyone/anything able to read network traffic

Not great security

- Personal data
- Passwords

Can be altered as well as read

- MITM - "man in the middle" attacks

Can be copied and sent again

- "Replay attack"

HTTPS

"S" for Secure

Uses public-key encryption

Headers/bodies are encrypted

- Prevents reading

Sender each way can be validated

- Prevents alteration

Basic Encryption

"Ciphers" are simple encryption

"Corpus" (message) is altered by applying a set of rules

Decryption is applying the same rules in reverse

- ROT-13 (shift english alphabet 13 characters)
- "book codes" (convert letters/words to positions on page/text of a book)

Problems with basic encryption

- Depends on a shared secret (the rules or a key)
    - How do you initially exchange the secret?
- No proof sender isn't someone else with the shared secret

Public Key - Behind the scenes

TL;DR: Mathematical magic

Imagine a math problem that is hard to reverse.

- 8134 cubed isn't too hard
- cube root of 5381613501O4 is much harder

Same idea: A math problem that is easier to do in one direction

Public Key Essentials

Two "keys"

- A "public" key - **no secrecy**
- A "private" key - keep it secret

One-way encryption:

- Msg + **Public** key = encrypted value that needs **private** key to read
- Msg + **Private** key = encrypted value that needs **public** key to read

Notice how you need the OTHER key to decrypt

What it means

- No shared secret - public keys are PUBLIC
- Messages encrypted with private key more likely to be legit
- You can "sign" an unencrypted message by attaching an encryption of message/message checksum using your private key

Browsers

- Browsers maintain a list of "trusted" public keys
    - Certificate Authorities (CA)
- HTTPS sites have a private key and a signed "Certificate" from a CA saying that key is theirs
- Browsers CAN be configured with a set key pair
    - usually make one up for short-term use
    - site identity validated, user identity is NOT.
    - ...but user is trusted to be the same user over duration of browser session

Summary - HTTPS

- HTTP is plain-text - insecure
- HTTPS protects information **in transit**
- HTTPS uses public key encryption
    - one-way encryption/decryption
    - Browsers trust a list of CAs
        - HTTPS is only as secure as the CAs
        - CAs validate identity w/signed "cert"
- letsencrypt.org provides free certs

Authentication / Authorization

- Authentication (Auth)
    - Who are you?
        - Think I.D. Card
- Authorization (Authz)
    - What are you allowed to do?
        - Think housekey

Factors

A way of proving auth/authz

- Something you know
    - passwords, PIN
- Something you have
    - keycards, yubikey, RSA token, cellphone
- Something you are
    - fingerprints, iris, face

2FA is "two factor auth", MFA is "multi-factor (2+) auth"

Login

Authenticates, possibly authorizes

- Username
- Password

Send both. Per security discussion, server will compare hashed password+salt to stored salt+hash for that username.

But then what?

Beyond Stateless

Web requests are stateless

How do you let the server know a later request is from someone that has already authenticated?

One Option: Passing Data

You could embed any necessary data in a form

- Each form submits info from previous forms

Pro:

- Works

Cons:

- User can change data
    - Security: NEVER TRUST DATA FROM USER
- Only w/forms or generated links

Option Two: Session Id

Store the data on server

- Associated it with unpredictable "key"
- Key secret from others
- Not secret from user

Stored Data = "session"

- Secret key = "session id"

Now sensitive data not changeable by user

Option 3: Signed Auth Token

A value that says user

- is an identity (auth)
- can do something (authz)

"Sign" the value using Public Key encryption

- User sends signed value (string)
    - Much like session id (bearer token)
    - Not secret from user
    - Is secret from public
- Server can validate using a public cert
- We trust the signer/system, not user

Passing the bearer token is annoying

Still sending via form/link

- More effort to generate dynamic HTML

Solution: Cookies!

Cookies Managed by Browser

- Server sends a `set-cookie` header on response
    - key=value pair
    - Along with some options
    - Including when it "expires"
- Browser saves this info
- On later requests
    - Browser sends a `cookie` header
        - With key=value pair
        - Automatically
    - Server can read this cookie

Cookies are just a header

Notice how we didn't change HTTP for this

- Just set a header
- Server treats like a header
- Browser does the extra work

Cookie Security Management

- Browsers store cookie
    - Associate with "origin" and "path"
        - origin = protocol + domain + port
        - path - Don't use this, not worth it
    - Cookies only sent to origin server requests
- Cookies editable by user
    - Generally use for session id only
- Cookies end when browser closed
    - Unless they have an Expiration Date
        - "Remember this computer"

Cookie Best Practices

- Set `HttpOnly' flag
  - Unless using with client JS
- Set `Secure` flag
  - In production
  - Dev might be done in http vs https
- Default to soon-expiring cookies
  - Shared computers are a thing
  - Session ID is EVERYTHING
- Set SameSite option value
  - Normally `Strict`

Removing a Cookie

- Cookie is stored on BROWSER
- Server might have associated data
    - But doesn't know what Browser has
- Server sends a response
    - Includes a `set-cookie` header
        - Removes value
        - Sets expires date to past
    - Server libraries have convenience methods

Session Id and Cookies

When user successfully auths, server will:

- Create a random string (session id = sid)
- Connect any auth and authz info with sid
    - Often a DB entry
    - This course: just keep in memory
    - Set cookie with this sid

Later Request

- Browser automatically sends the sid cookie
    - Server can read sid from `req`
    - Server can read session data using sid
    - Server can read OTHER data w/session data

Example:

- Session object holds username (by sid)
- Full user data NOT in Session
- User object holds full user data (by username)

Session data only lasts between login/logout

- User data outside of session

Validating Auth of a later request

Server gets a request

- Checks for cookie
- Checks the value of cookie to make sure it is valid
- Ensures that user is permitted to do request

Logout

Two parts to logout

- Clean up sid cookie on browser
    - Server sends `set-cookie` to remove
- Remove session data
    - Example: deleting sid from sessions object

Remember: Most users don't logout

- Stale session data will collect
- Server frameworks may manage
    - But "session" is a general concept

Other tokens

Session Id is a "token"

- With random value

Other tokens may

- Contain usable info directly
- Are "signed" to prove who created them

Example: JWT (JSON Web Token) ("jot")

Still a "bearer token"

- Must keep secret

JSON Web Token - JWT

Signed bit of auth info + expire date

Advantages

- No DB check each time used
- Can be passed to others
    - How many 3rd party login systems work
    - Can pass to disconnected servers

Disadvantages

- Good for their lifetime, even if user "logs out"
- Don't want to store changing info in them

JWT Security

- Don't use if you need fast lockout
- Be sure to validate signatures!
    - Use tested libraries
- Generally use Secure and HTTPOnly cookies
- For server-to-server web calls
    - Expect JWT to be sent as `Auth` header

This course will use sid + cookies

- Most prevalent
- Still informs the server-client exchange

We will NOT use passwords!

- We will check for username "dog"
    - Shows when we check
    - Doesn't create false security about security

Express cookie example

```
// express "middleware", this time as an extra library
const cookieParser = require('cookie-parser');
app.use(cookieParser());

// (skipping over other express stuff)
app.get('/', (req, res) => {
  const store = req.query.store;
  if(store) {
    res.cookie('saved', store);
  }

  const saw = req.cookies.saved;
  res.send(`<p>Request had cookie "saved": ${saw}</p>`);
});
```

Steps

1. Inside new project directory:
    - `npm init -y`
    - `npm install express`
    - `npm install cookie-parser`
2. Create the `server.js` (or whatever you call it) file
3. run `node server.js`
4. go to `localhost:3000` in the browser
5. use `?store=SOMEVAL` at end of url to set the cookie
6. DevTools-Network-Headers to see the `Set-Cookie` in the response and the `Cookie` in the request
7. DevTools-Application-Cookies (left) to see cookies

Changing the cookie example

Do you know how to:

- Store the cookie under a different name
    - not `"saved"`?
- Change the expiration time of the cookie?
- Change the name of the query param you are sending to set the cookie value?
    - instead of `"store"`
- Redirect the user to `'/'` (no query param) after setting the cookie?

What is UUID?

- Universally
- Unique
- IDentifier

(Also known as GUID, for "Globally")

UUID variations

- Some have random-ish
    - Others NOT!
- Often factor in date/time
- Some pull in other info bits
- Generated by algorithm, not a central producer
- Attempt to make collision practical impossibility

session ids want unpredictable in addition to unique

- why?

UUID in node

Here's one library:

```
npm install uuid
```

```
const uuidv4 = require('uuid').v4;

const sid = uuidv4(); // sid common name for "session id"
```

UUID as session id in express

```
app.use(express.urlencoded({ extended: false }));

const sessions = {}; // Created outside a route handler

app.post('/session', (req,res) => {
  const username = req.body.username.trim();
  if (!username) { // Give better errors than this!
    res.status(400).send('username required');
    return; // don't allow redirect attempt
  } else if (username === 'dog') { // Simulating a bad password
    res.status(403).send('user account not permitted');
    return;
  }
  const sid = uuidv4();  // uuidv4 is from uuid module
  sessions[sid] = { username }; // Do you know why?
  res.cookie('sid', sid);
  res.redirect('/');
});
```

Session Storage

```
// example of sessions
sessions = {
    'asdf-asdf-asdf-asdf': {
        username: 'Jorts',
    },
    'zxcv-zxcv-zxcv-zxcv': {
        username: 'Jean',
    },
};
```

- Same user can have different sid
- But we want the same data for that user
- Store data by username/user id, NOT by sid
- Look up username by sid
    - look up data by username/user id

Checking the SID in express

```
app.get('/users', (req,res) => {
  const sid = req.cookies.sid;
  if(!sid || !isValid(sid)) {
    res.clearCookie('sid');
    res.sendStatus(401);
    return;
  }

  const { username } = sessions[sid];
  // Do whatever here
});
```

- `isValid()` is a function/check you have to write
  - Do we know this session?
  - `isValid()` is a concept, not a specific requirement

Removing SID to end session

Imagine we have a `/logout` route

- Is this a GET or a POST?
    - When we get to REST, the question changes
- How do we clear the sid cookie?
    - `res.clearCookie('sid');`
    - OR, set cookie to blank value
    - OR, set cookie to immediately expire
- How do we clear the data from the server?
    - Delete this sid from `sessions`

Remember there is data in two places!

`sid` cookie on the browser-side

- `res.clearCookie('sid');` tells browser to delete

`sessions` has the `sid`

- `delete sessions[sid];` will remove that

Deleting in one place will not change the other!