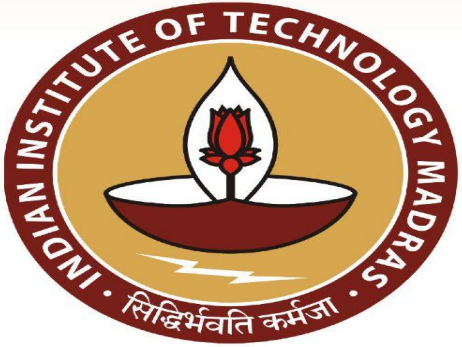


IIT Madras
BSc Degree

Security

Security

- Access Control
- Web-based Mechanisms
- Session management
- HTTPS
- Logs and Analysis

Access Control

What is access control?

- Access: being able to read/write/modify information
- Not all parts of application for public access
  - Personal, Financial, Company, Grades, …
- Types of access:
  - read-only
  - read-write (CRUD)
  - modify but not create
  - …

Examples

- Linux files:
  - owner, group: access your own files, cannot modify (or even read?) others
  - can be changed by owner
  - “root” or “admin” or “superuser” has power to change permissions

Examples

- Linux files:
  - owner, group: access your own files, cannot modify (or even read?) others
  - can be changed by owner
  - “root” or “admin” or “superuser” has power to change permissions
- Email:
  - you can read your own email
  - can forward an email to someone else - this is also access!

Examples

- Linux files:
  - owner, group: access your own files, cannot modify (or even read?) others
  - can be changed by owner
  - “root” or “admin” or “superuser” has power to change permissions
- Email:
  - you can read your own email
  - can forward an email to someone else - this is also access!
- E-commerce login:
  - shopping cart etc visible only to user
  - financial information (credit card etc.) must be secure

Discretionary vs Mandatory

- Discretionary:
  - you have control over who you share with
  - forwarding emails, changing file access modes etc possible

- Mandatory:
  - decisions made by centralized management - users cannot even share information without permission
  - Typically only in military or high security scenarios

Role-based access control

- Access associated with “role” instead of “username”

Role-based access control

- Access associated with “role” instead of “username”
- Example:
  - Head of department has access to student records
  - What happens when HoD changes?

Role-based access control

- Access associated with “role” instead of “username”
- Example:
  - Head of department has access to student records
  - What happens when HoD changes?
- Single user can have multiple roles
  - HoD, Teacher, Cultural advisor, sports club member, …

Role-based access control

- Access associated with “role” instead of “username”
- Example:
  - Head of department has access to student records
  - What happens when HoD changes?
- Single user can have multiple roles
  - HoD, Teacher, Cultural advisor, sports club member, …
- Hierarchies, Groups
  - HoD > Teacher > Student
  - HoD vs sports club member? - no hierarchy here

Attribute-based access control

- Attribute
  - time of day
  - some attribute of user (citizenship, age, …)
- Can add extra capability over role-based

Policies vs Permissions

- Permissions
  - Static rules usually based on simple checks (does user belong to group)?
- Policies
  - More complex conditions possible
  - Combine multiple policies
  - Example:
    - Bank employee can view ledger entries
    - Ledger access only after 8am on working days

Principle of least privilege

- Entity should have minimal access required to do the job
- Example: Linux file system
  - users can read system libraries but not write
  - some files like /etc/shadow not even readable
  - you can install Python to local files using “venv” but not to system path
- Benefits
  - better security - fewer people with access to sensitive files
  - better stability - user cannot accidentally delete important files
  - ease of deployment - can create template filesystems to copy

Privilege escalation

- Change user or gain an attribute
  - “sudo” or “su”
- Usually combined with explicit logging, extra safety measures
- Recommended:
  - do **not** sudo unless absolutely necessary
  - never operate as root in a Linux/Unix environment unless absolutely necessary

Context: Web apps

- Admin dashboards, user access, etc.
- Gradebook example:
  - only admin should be able to add/delete/modify
  - users should have read permissions only on their own data

Enforcing

- Hardware level
  - Security key, hardware token for access, locked doors etc

Enforcing

- Hardware level
  - Security key, hardware token for access, locked doors etc
- Operating system
  - filesystem access, memory segmentation

Enforcing

- Hardware level
  - Security key, hardware token for access, locked doors etc
- Operating system
  - filesystem access, memory segmentation
- Application level
  - DB server can restrict access to specific database

Enforcing

- Hardware level
  - Security key, hardware token for access, locked doors etc
- Operating system
  - filesystem access, memory segmentation
- Application level
  - DB server can restrict access to specific database
- Web application
  - Controllers enforce restrictions
  - Decorators in Python used in frameworks like Flask

Security Mechanisms

For the Web

Types of security checks

- Obscurity (generally very bad idea):
  - application listens on non-standard port known only to specific people

Types of security checks

- Obscurity (generally very bad idea):
  - application listens on non-standard port known only to specific people
- Address:
  - where are you coming from? host based access/deny controls

Types of security checks

- Obscurity (generally very bad idea):
  - application listens on non-standard port known only to specific people
- Address:
  - where are you coming from? host based access/deny controls
- Login:
  - username/password provided to each person needing access

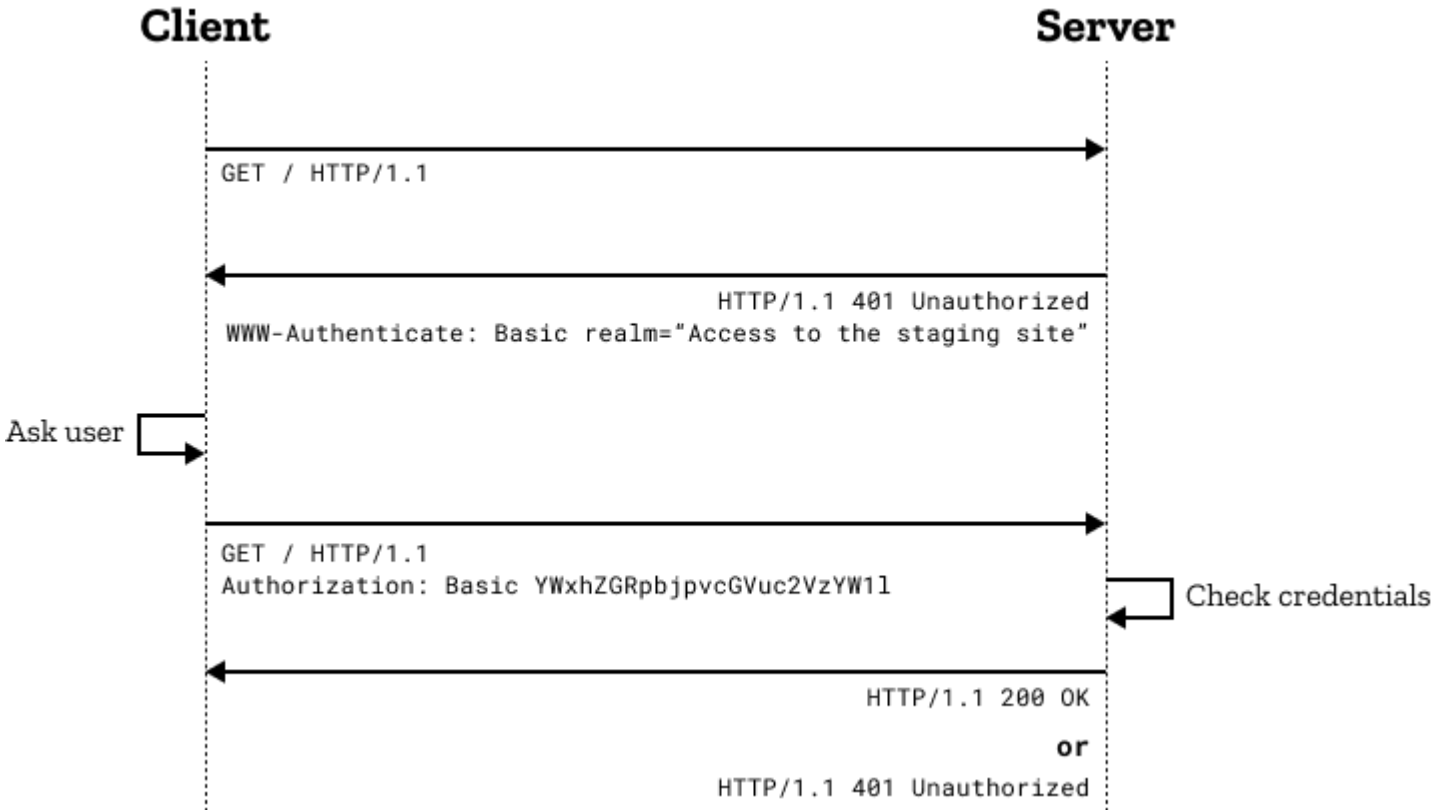
Types of security checks

- Obscurity (generally very bad idea):
  - application listens on non-standard port known only to specific people
- Address:
  - where are you coming from? host based access/deny controls
- Login:
  - username/password provided to each person needing access
- Tokens:
  - access tokens that are difficult/impossible to duplicate
  - can be used for machine-to-machine authentication without passwords

HTTP authentication

Basic HTTP auth:

- Enforced by server
- Server returns “401/Unauthorized” code to client
- Contrast with:
  - “404” - not found
  - “403” - forbidden (no option to authenticate)
- Client must respond with access token as an extra “Header” in next request



Src: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

Problems with HTTP Basic Auth

- Username, Password effectively sent as plain text (base64 encoding)
  - Some minimal security if HTTPS is used (wiretap is difficult)
- Password will be seen in cleartext at server
  - Should not be needed - better mechanisms possible
- No standard process for “logout”

Digest authentication

- Message digest: cryptographic function
  - eg. MD5, SHA1, SHA256 etc.
- One-way function:
  - f(A) = B
  - Easy to compute B given A
  - Very difficult (near impossible) to compute A given B
- Can define such one-way functions on strings
  - String -> binary number

HTTP Digest authentication

- Server provides a “nonce” to prevent spoofing
- Client must create a secret value including nonce
- Example:
  - HA1 = MD5(username:realm:password)
  - HA2 = MD5(method:URI)
  - response = MD5(HA1:nonce:HA2)
- Server and client know all parameters above, so both will compute same
- Any third party snooping will see only final response
  - cannot extract original values (username, password, nonce etc)
  - nonce only used once to prevent replay

Client certificates

- Cryptographically secure certificates provided to each client
- Client does handshake with server to exchange information, prove knowledge
- Keep cert secure on client end
  - Impossible to reverse and find the key

Form input

- Username, Password entered into form
- Transmitted over link to server
  - link must be kept secure (HTTPS)
- GET requests:
  - URL encoded data: very insecure, open to spoofing
- POST requests:
  - form multipart data: slightly more secure
  - still needs secure link to avoid data leakage

Request level security

- One TCP connection
  - One security check may be sufficient
  - other network level issues to consider for TCP security
- Without connection KeepAlive:
  - each request needs new TCP connection
  - each request needs new authentication

Cookies

- Server checks some client credentials, then “sets a cookie”
- Header
  - `Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>; Secure; HttpOnly`
- Client must send back the cookie with each request
- Server maintains “sessions” for clients
  - Remember cookies
  - Can set timeouts
  - Delete cookie record to “logout”
- Client
  - must send cookie with each request

API security

- Cookies etc. require interactive use (browser)
- Basic auth pop-up window

APIs:

- Typically accessed by machine clients or other applications
- Command-line etc. possible
- Use “token” or “API key” for access
    - Subject to same restrictions: HTTPS, not part of URL etc.

Sessions

Session management

- Client sends multiple requests to server
- Save some “state” information
  - logged in
  - choice of background colour
  - …
- Server customizes responses based on client session information

Storage:

- Client-side session: completely stored in cookie
- Server-side session: stored on server, looked up from cookie

Cookies

- Set by server with Set-Cookie header
- Must be returned by client with each request
- Can be used to store information:
  - theme, background colour, font size: simple no security issues
  - user permissions, username: can also be set in cookie
    - must not be possible to alter!

Example: Flask

```
from flask import session

# Set the secret key to some random bytes. Keep this really secret!
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

@app.route('/')
def index():
    if 'username' in session:
        return f'Logged in as {session["username"]}'
    return 'You are not logged in'
```

Example: Flask

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
        <form method="post">
            <p><input type=text name=username>
            <p><input type=submit value=Login>
        </form>
    '''

@app.route('/logout')
def logout():
    # remove the username from the session if it's there
    session.pop('username', None)
    return redirect(url_for('index'))
```

Security issues

- Can user modify Cookie?
  - Can set any username
- If someone else gets Cookie, can they log in as user?
  - Timeout
  - Source IP
- Cross-site requests
  - Attacker can create page to automatically submit request to another site
  - If user is logged in on other site when they visit attack page, will automatically invoke action
  - Verify on server that request came from legitimate start point

Server-side information

- Maintain client information at server
- Cookie only provides minimal lookup information
- Not easy to alter
- Requires persistent storage at server
- Multiple backends possible
  - File storage
  - Database
  - Redis, other caching key-value stores

Enforce authentication

- Some parts of site must be ***protected***
- How?
  - Enforce existence of specific token for access to those views
- Views:
  - determined by controller
- Protect access to controller!
  - Flask controller - Python function
  - Protect function - add wrapper around it to check auth status
    - Decorator!

Example - flask_login

```
from flask_login import login_required, current_user
...
@main.route('/profile')
@login_required
def profile():
    return render_template('profile.html', name=current_user.name)
```

Example - flask_login

```
from flask_login import login_user, logout_user, login_required
...
@auth.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('main.index'))
```

Transmitted data security

- Assume connection can be “tapped”
- Attacker should not be able to read data
- HTTP GET URLs not good:
  - logged on firewalls, proxies etc
- HTTP POST, Cookies etc:
  - if wire can be made safe, then good enough

How to make the wire safe?

HTTPS

Normal HTTP process

- Open connection to server on fixed network port (default 80)
- Transmit HTTP request
- Receive HTTP response

Safety of transmitted data?

- Can be tapped
- Can be altered!

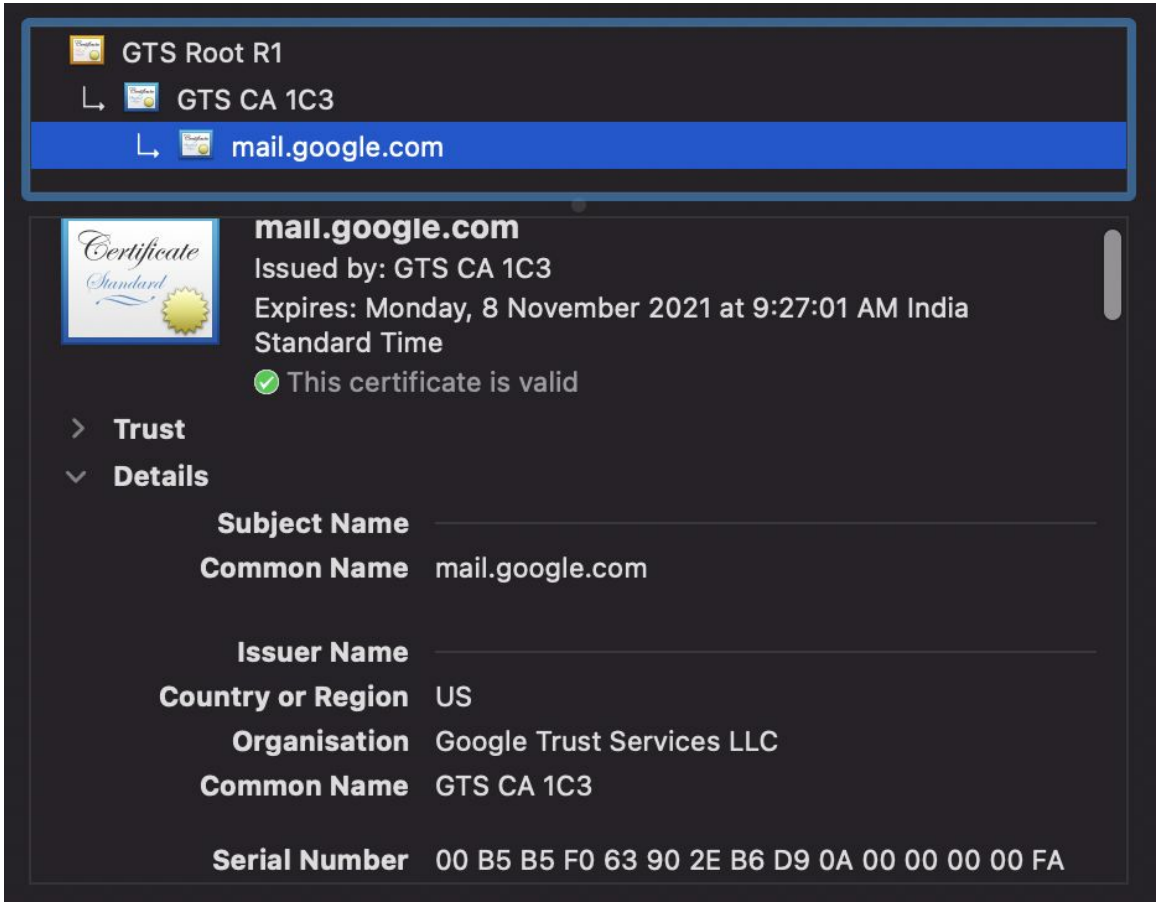
Secure sockets

- Set up an “encrypted” channel between client and server
- How?
  - Need a shared secret - eg. long binary string - this is the “key”
  - XOR all input data with key to generate new binary data
  - Attacker without key cannot derive actual data
- How to set up shared secret?
  - Must assume anything on the wire can be tapped!
  - What about pre-existing key?
  - Secure side channel - send a token by post, SMS

Types of security

- Channel (wire) security
  - Ensure that no one can tap the channel - most basic need for other auth mechanisms etc.
- Server authentication
  - How do we know we are actually connecting to mail.google.com and not some other server?
  - DNS hijacking possible - redirect to another server!
  - Server certificates
  - Common root of trust needed - someone who “vouches for” mail.google.com
- Client certificate
  - Rare but useful - server can require client certificate
  - Used especially in corporate intranets etc.

mail.google.com/n



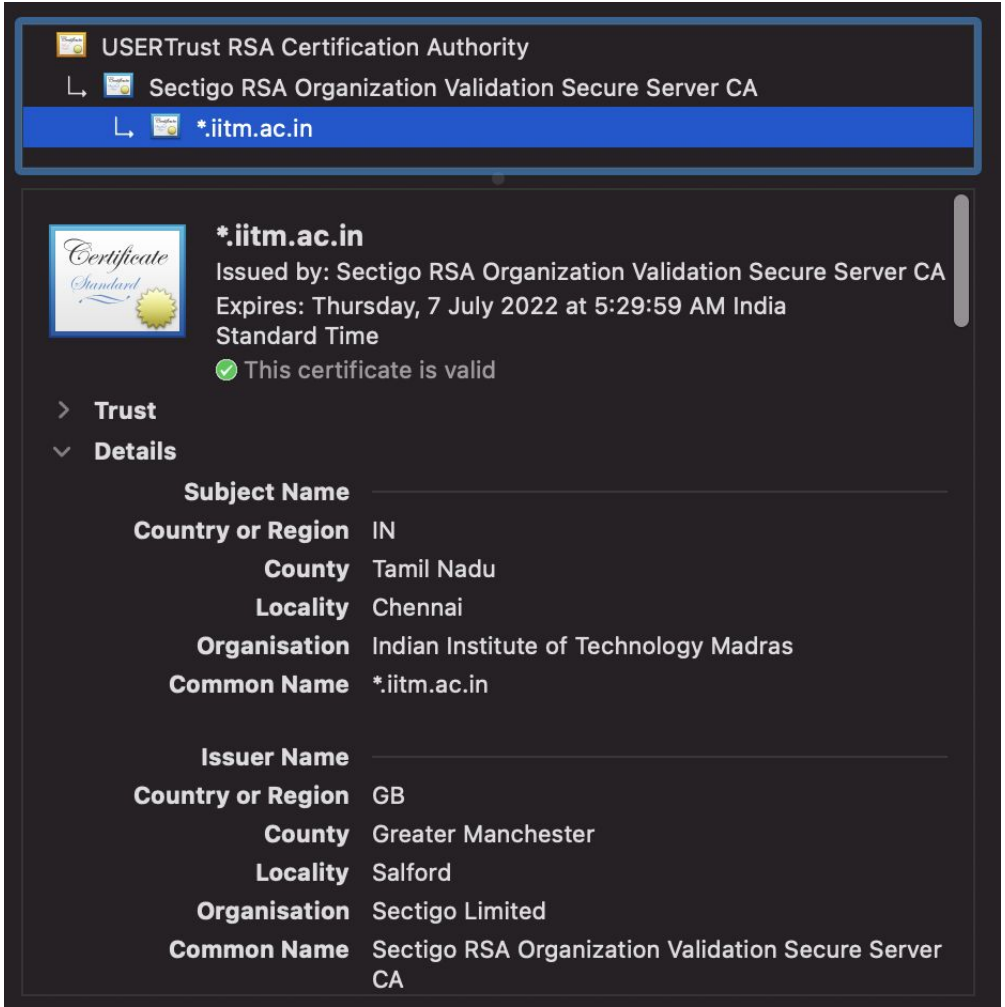
Chain of Trust

- Chain of trust
  - mail.google.com issued certificate by
  - GTS CA1C3 issued certificate by
  - GTS Root R1
- GTS Root R1 certificate stored in Operating System or Browser
  - Do you trust your OS? Do you trust your browser?
- From there on a secure (crypto) chain

Potential problems

- Old browsers
  - Not updated with new chains of trust
- Stolen certificates at root of trust
  - Certificate revocation, invalidation possible
  - Need to ensure OS, browser can update their trust stores
- DNS hijacking
  - Give false IPs for server as well as entries along chain of trust
  - But certificate in OS will fail against eventual root of trust

Wildcard certificates



Impact of HTTPS

- Security against wiretapping
- Better in public WiFi networks

Negative:

- Affects caching of resources (proxies cannot see content)
- Performance impact due to run-time encryption

Logging

What is logging?

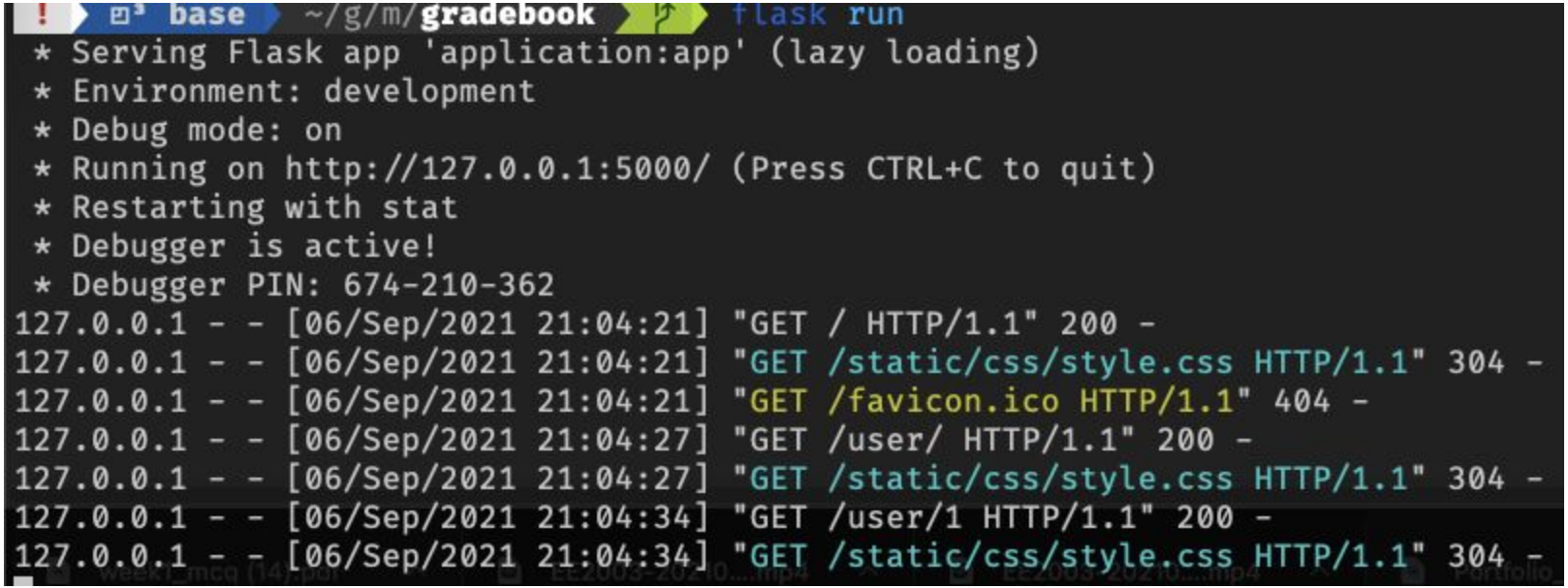
- Record all accesses to app
- Why?
  - Record bugs
  - Number of visits, usage patterns
  - Most popular links
  - Site optimization
  - Security checks
- How?
  - Build into app - output to log file
  - Direct output to analysis pipeline

Server logging

- Built in to Apache, Nginx, …
- Just accesses and URL accessed
- Can indicate possible security attacks:
  - Large number of requests in short duration
  - Requests with “malformed” URLs
  - Repeated requests to unusual endpoints

Application level logging

- Python logging framework
  - Output to file, other “stream” handlers
- Details of application access
  - Which controllers
  - What data models
  - Possible security issues
- All server errors


! base ~/g/m/gradebook flask run
* Serving Flask app 'application:app' (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 674-210-362
127.0.0.1 - - [06/Sep/2021 21:04:21] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [06/Sep/2021 21:04:21] "GET /static/css/style.css HTTP/1.1" 304 -
127.0.0.1 - - [06/Sep/2021 21:04:21] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [06/Sep/2021 21:04:27] "GET /user/ HTTP/1.1" 200 -
127.0.0.1 - - [06/Sep/2021 21:04:27] "GET /static/css/style.css HTTP/1.1" 304 -
127.0.0.1 - - [06/Sep/2021 21:04:34] "GET /user/1 HTTP/1.1" 200 -
127.0.0.1 - - [06/Sep/2021 21:04:34] "GET /static/css/style.css HTTP/1.1" 304 -

Log rotation

- High volume logs - mostly written, less analysis
- Cannot store indefinitely
  - Delete old entries
- Rotation:
  - Keep last N files
  - Delete oldest file
  - Rename log.i to log.i+1
  - Fixed space used on server

Logs on custom app engines

- Google app engine
  - Custom logs
  - Custom reports
- Automatic security analysis

Time series analysis

- Logs are usually associated with timestamps
- Time series analysis:
  - How many events per unit time
  - Time of specific incident(s)
  - Detect patterns (periodic spikes, sudden increase in load)
- Time-series databases
  - RRDTool, InfluxDB, Prometheus, …
  - Analysis and visualization engines

Summary

Security is key to successful applications!

- Requires good understanding of principles
  - Crypto
  - SQL, OS vulnerabilities, …
- Good frameworks to be preferred
- Analyze, Identify, Fix