API systems Task 1

Part 1 – RESTful APIs

A RESTful API is basically an interface that allows companies or users to access and use data. Such data can be used to GET(Read), PUT(Update), POST (Create) or DELETE. REST technology is usually preferred as it uses less bandwidth, which makes it more efficient. RESTful APIs can also be built using PHP, JavaScript and Python, which are some of the most common programming languages.

The Core principles of a REST are, Uniform Interface, Client-Server, Stateless, Cacheable, Layered Systems, Code on Demand.

Uniform Interface: This is basically how the interface is set up. Normally this is set up in a way to make the API easy to use. There are 4 constraints which can help to get a uniform REST interface. These are;

- Identification of resource - this interface should identify each resource that is used in the interaction used between the client and the server.
- Manipulation of resources through representations - the server should respond in a consistent format which clients will use to update data on the server.
- Self-descriptive messages - each response should include enough details for the client to understand in order to know what actions it can take next.
- Hypermedia as the engine of application state – the client starts with one main URI (Uniform Resource Identifiers) and uses links in the responses to navigate and interact with other resources.

A Client-Server: This keeps things separate, allowing the client and server to develop independently. By handling the user interface on the client and data storage on the server, the interface becomes more portable across platforms with simplicity and better scalability.

Stateless: In a stateless system, each client request must include all the details needed for the server to process it. The server does not remember past interactions, so the client is responsible for managing the session state.

Cacheable: The cacheable rule means a response must indicate whether it can be stored or not. If it's cacheable, the client can reuse the response for similar requests within a set time.

Layered system: The layered system design organizes an architecture into stacked layers, where each part only interacts with the only directly connected to it. A simple example of this is the MVC patter, which separates concerns, making development, maintenance, and scaling easier.

Code on demand: This is when a REST allows the clients to gain new features by downloading and running small programs or scripts. This helps clients because they don't need to have all features built in from the start. The server can send parts of a feature as code, and the client just needs to run it.

The process of making a request with an API:

1: Client sends a request to an API. This request includes a URL, sometimes even sending extra details like data or authentication.

2: Server Processes the Request. The API server receives the request and checks what the client is asking for.

3: Server sends a response. The API sends back a response which contains the requested data, or an error message indicating that something went wrong.

4. Client uses the response. The client takes the data and displays it.

- The GET method is used to retrieve data from an API.
- The POST method is used to create new data on the server.
- The PUT method is used for a full update, meaning it replaces all existing data for a resource.
- The PATCH method is used for a partial update, allowing changes to only specific fields.
- The DELETE method is used to remove data from the server.

Query String Parameters are used to send data in the URL, usually for filtering, searching or sorting and are mainly used in GET requests to pass optional data. We used these in class where if there was an error in the input a user made, then a "?" will be added to the URL and specifying the error.

Request Body is used to send larger and more complex data, usually in POST, PUT, or PATCH requests. Instead of being in the URL, this data is sent inside the request in formats like JSON.

Part 2: Authorisation

OAuth 2.0 is a protocol that is used for authorization which is an industry-standard. It works by issuing access tokens to apps after user authentication, to enable a secure API interaction.

Scopes define what parts of a user's data an app can access. They help unsure that apps only get minimum required permissions, without exploiting important data.

Access tokens are keys that allow apps to interact with APIs securely. These keys are short lived and change through time for maximum protection. These tokens are included in API requests to prove authorization.

Client Id is a public identifier for an app, while Client Secret is a private key used to authenticate the app with the OAuth provider.

Part 3: Security Considerations

OWASP API Security Top 10 is a list of the most security risks related to APIs. This helps organizations identify vulnerabilities that could lead to breaches of data and security. These risks are, Broken Object Level Authorization, Broken Authentication, Broken Object Property Level Authorization, Unrestricted Resource Consumption, Broken Function level Authorization,

Unrestricted Access to Sensitive Business Flows, Server Side Request Forgery, Security Misconfiguration, Improper Inventory Management, and Unsafe Consumptions of APIs.

Broken Object level Authorization: This is when the endpoints object identifiers are exposed, which can create security risks. To prevent this vulnerability, every function that retrieves or modifies data which use user provided ID, should include strict authorization checks.

Broken Authentication: Authentication is often implemented incorrectly, making it easier for attackers to exploit weaknesses to impersonate other users. This is why its important that an API properly verifies users, to not make the system vulnerable.

Broken Object Property Level Authorization: This is a combination of 2 issues, which ultimately comes down to the lack of authorization validation at the object property level. This leads to exposure of information.

Unrestricted resource Consumption: Some API requests, such as emails and phone calls, may require payment per request. This may lead to denial of Service or increase of cost.

Broken Function Level Authorization: complicated access control policies with multiple groups roles and unclear admin boundaries can lead to security issues. Hackers could exploit these weaknesses to access user data or admin functions.

Unrestricted Access to Sensitive Business Flows: API at risk of this issue allow certain actions to be done without limits on excessive automated use. This could harm business, even if there is no error in the code.

Server Side Request Forgery: This occurs when an attacker tricks a server into making an un authorized request. This could lead to data leaks, or even unauthorized access.

Security Misconfiguration: This happens when APIs have weak security settings such as credentials and exposed debug information. Attackers can exploit these to gain unauthorised access or extract sensitive information.

Improper Inventory Management: When organizations fail to keep track of all API versions and exposed endpoints, this can become security risks. Attackers may exploit deprecated APIs that lack proper security updates.

Unsafe Consumption of APIs: When an API interacts with a third party API without proper security checks, it can introduce vulnerabilities. If the external API is compromised or behaves unexpectedly, it can expose sensitive data.

We will be addressing 2 core issues in the API, Broken Authentication and Excessive Data Exposure.

Broken Authentication: In order to tackle this we will need to implement OAuth 2.0 for secure and authorised access. With the use of multi factor authentication we will be able to enhance security for user login.

Excessive data exposure: We will be implementing data filtering to ensure only authorised users are able to access a certain type of data. This will ensure that not everyone will be able to access sensitive data.

References:

https://restfulapi.net

https://oauth.net/2/

https://owasp.org/API-Security/editions/2023/en/0x11-t10/