# REST API Security

Dr. Hamed Hamzeh

13/03/2025

# What you will learn

Top 10 OWASP vulnerabilities

RESTful API Authentication Basics

JAX-RS Security

# Web application security vs API security

In traditional web applications, data processing is done on the server side

In this case, the only thing is to add a Web Application Firewall in front of the Server.

Modern API-based applications are very different. More and more, the UI uses APIs to send and receive the data from the backend servers to provide the functions of the application.

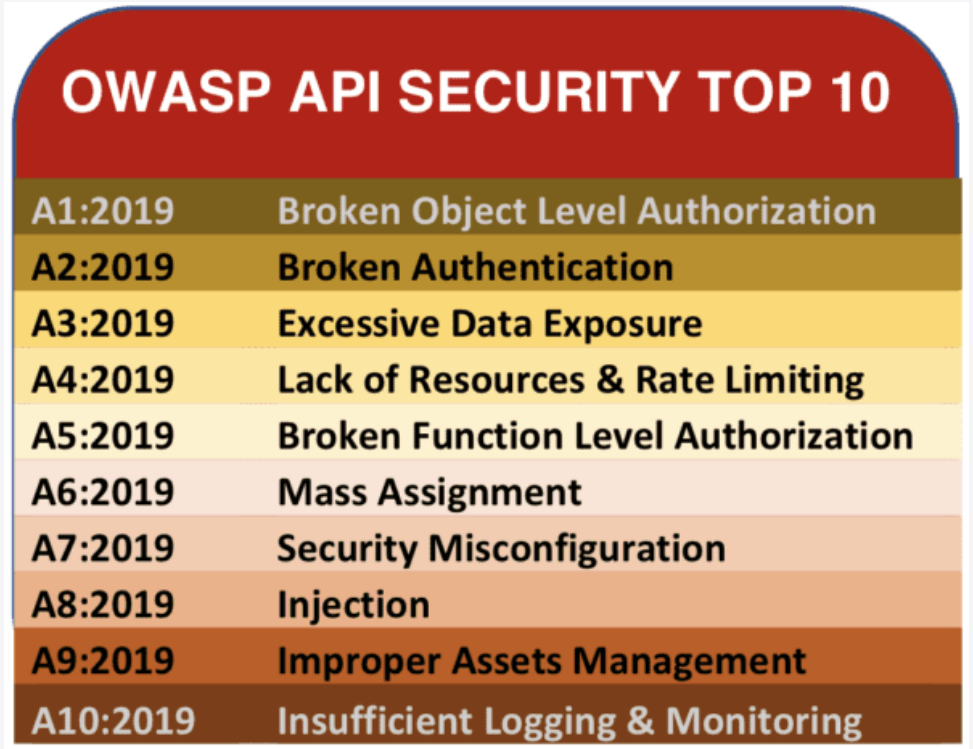
It is now the clients that do the rendering and maintain the state.

# Top 10 OWASP vulnerabilities

# OWASP API Security Top 10 Vulnerabilities

The Open Web Application Security Project (OWASP) is a non-profit, collaborative online community behind the OWASP Top 10.

They produce articles, methodologies, documentation, tools, and technologies to improve application security.



# OWASP and REST API

Authentication and Authorization

Input Validation

Session Management

Error Handling and Logging

Encryption and Transport Security

# Broken object level authorization

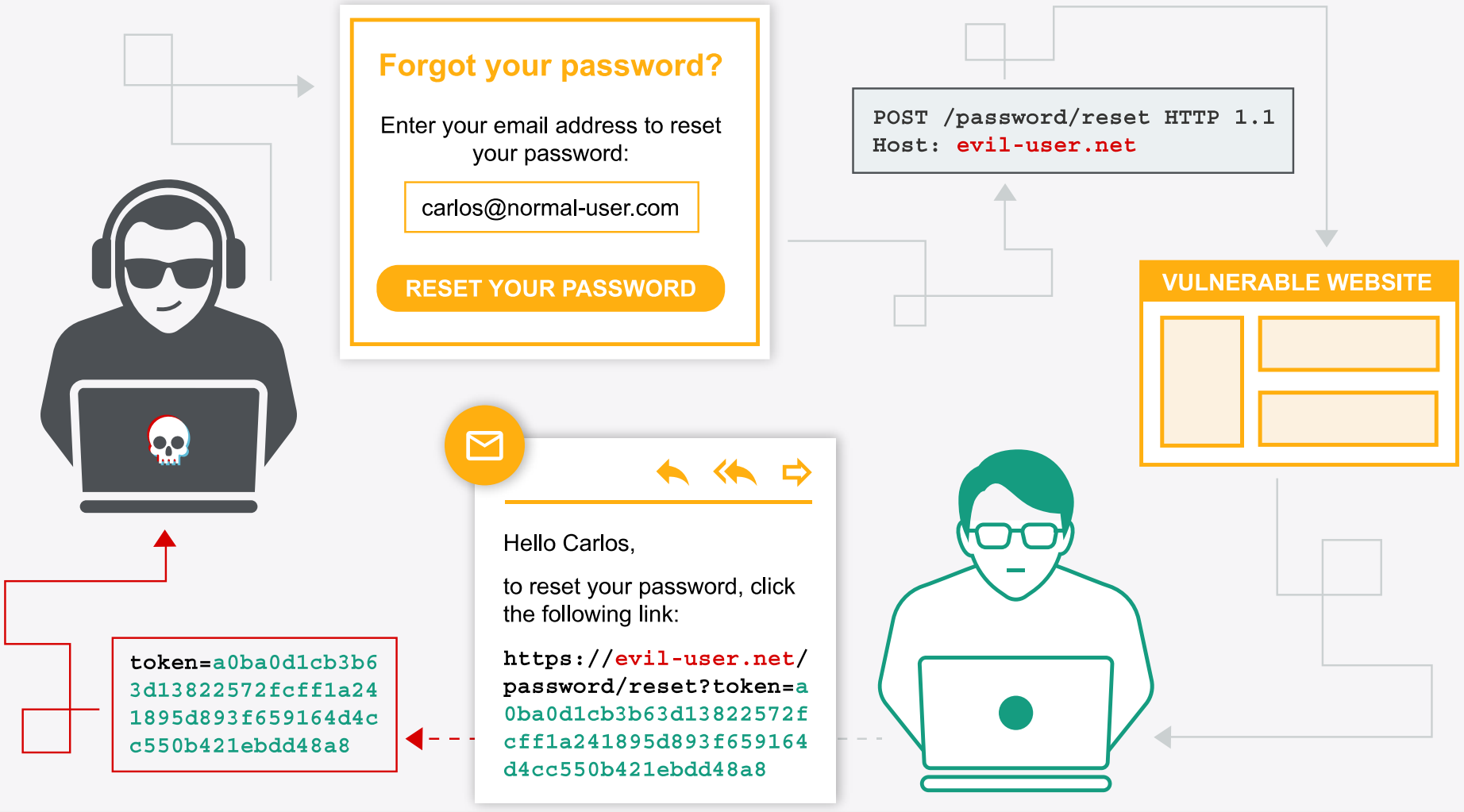
Attackers substitute the ID of their own resource in the API call with an ID of a resource belonging to another user.

The lack of proper authorization checks allows attackers to access the specified resource. This attack is also known as IDOR (Insecure Direct Object Reference).

# How to prevent?

1. Implement authorization checks with user policies and hierarchy.
2. Do not rely on IDs that the client sends. Use IDs stored in the session object instead.
3. Check authorization for each client request to access database.
4. Use random IDs that cannot be guessed (UUIDs).

# Broken authentication



# How to prevent?

- Check all possible ways to authenticate to all APIs.
- APIs for password reset and one-time links
- Use standard authentication, token generation, password storage, and multi-factor authentication (MFA).
- Use short-lived access tokens.
- Use stricter rate-limiting for authentication, and implement lockout policies and weak password checks.

# Excessive data exposure

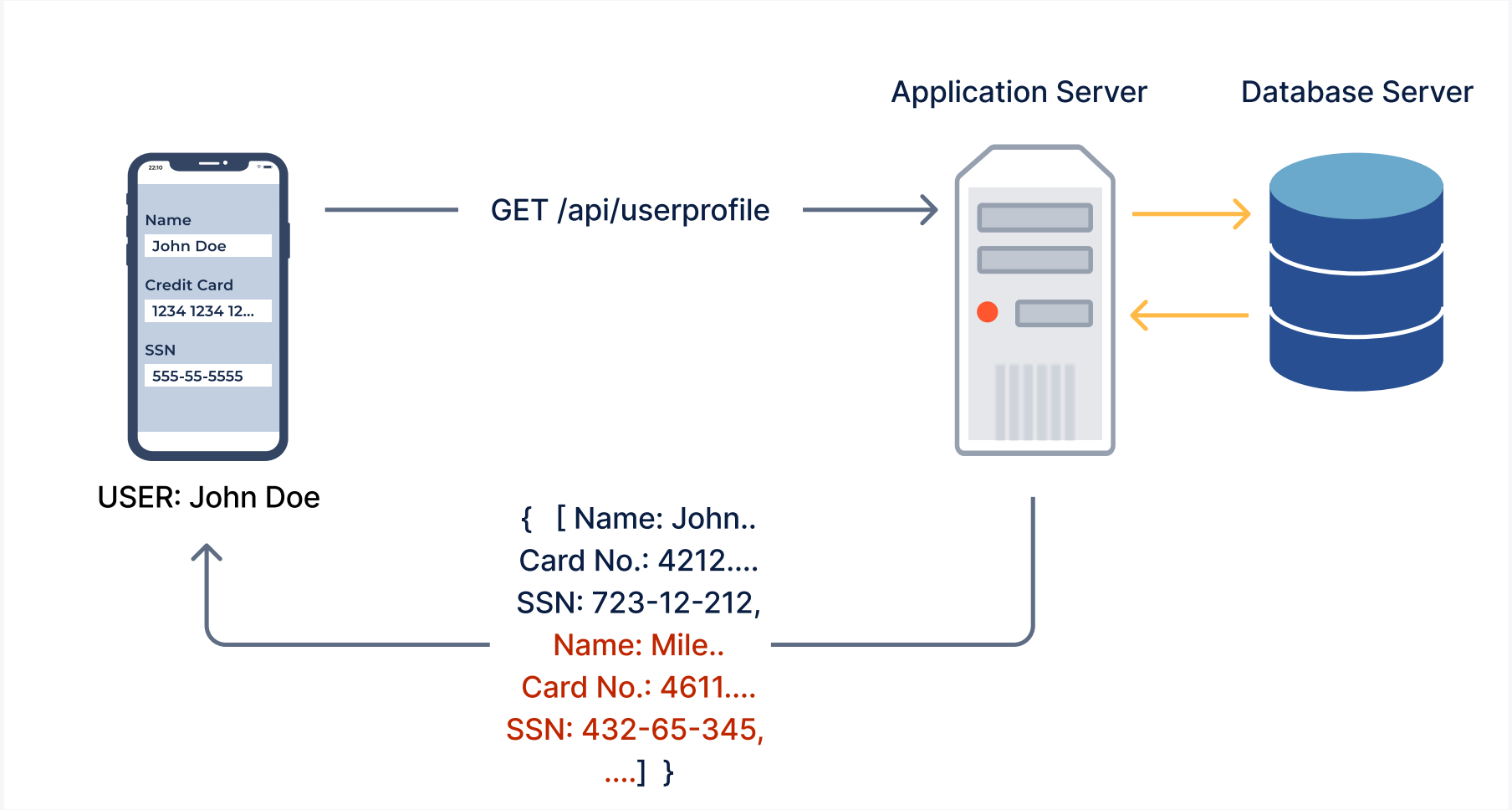
The API may expose a lot more data than what the client legitimately needs, relying on the client to do the filtering. If attackers go directly to the API, they have it all.

The API returns full data objects as they are stored in the backend database.

The client application filters the responses and only shows the data that the users really need to see.

Attackers call the API directly and get also the sensitive data that the UI would filter out.

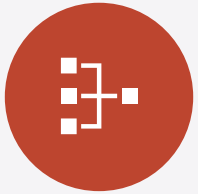
# Excessive data exposure



# How to prevent

Never rely on the client to filter data!

Review all API responses and adapt them to match what the API consumers really need.



Carefully define schemas for all the API responses.

Do not forget about error responses, define proper schemas as well.

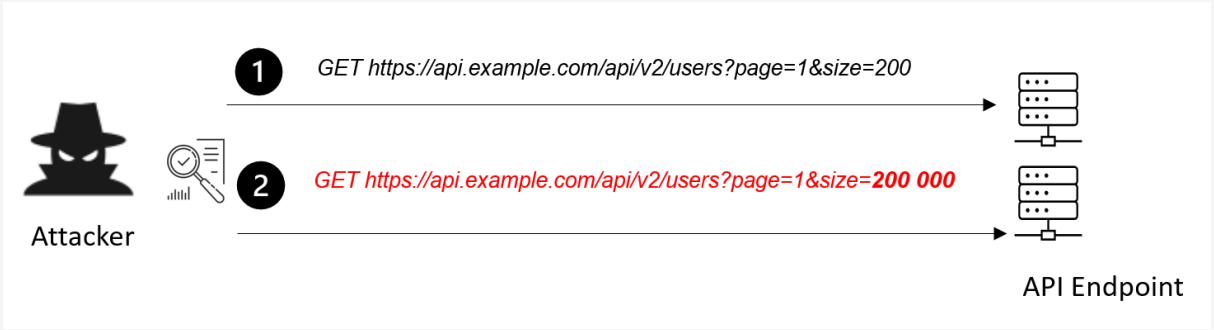
Identify all the sensitive data or Personally Identifiable Information (PII), and justify its use.

Enforce response checks to prevent accidental leaks of data or exceptions.

# Lack of resources and rate limiting attack scenario

An attacker sends a legitimate API request for a single page with 200 items.

The attacker modifies the page size, increasing it from 200 to 200,000.



# Broken function level authorization

Authorization is the process where requests to access a particular resource should be granted or denied.

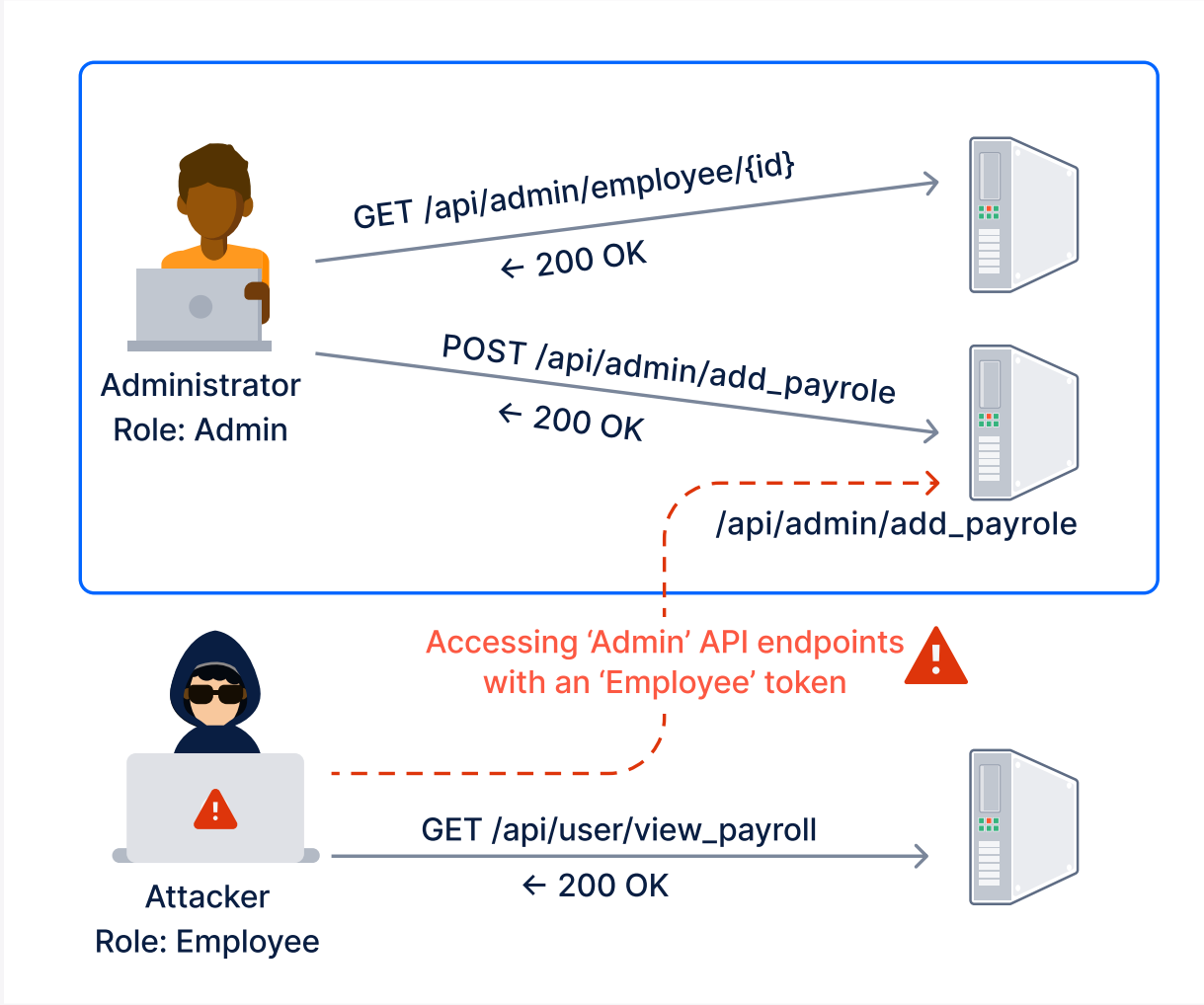
Authorization determines which functionality and data the logged in user (or Principal) may access.

Attackers figure out the “hidden” admin API methods and invoke them directly.

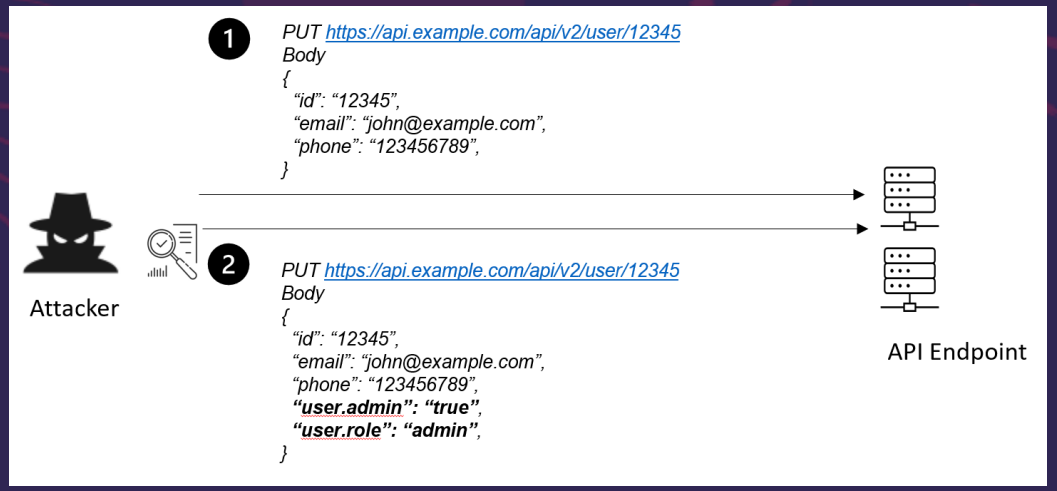
Some administrative functions are exposed as APIs.

Non-privileged users can access these functions without authorization if they know how.

# Broken function level authorization



# Mass assignment

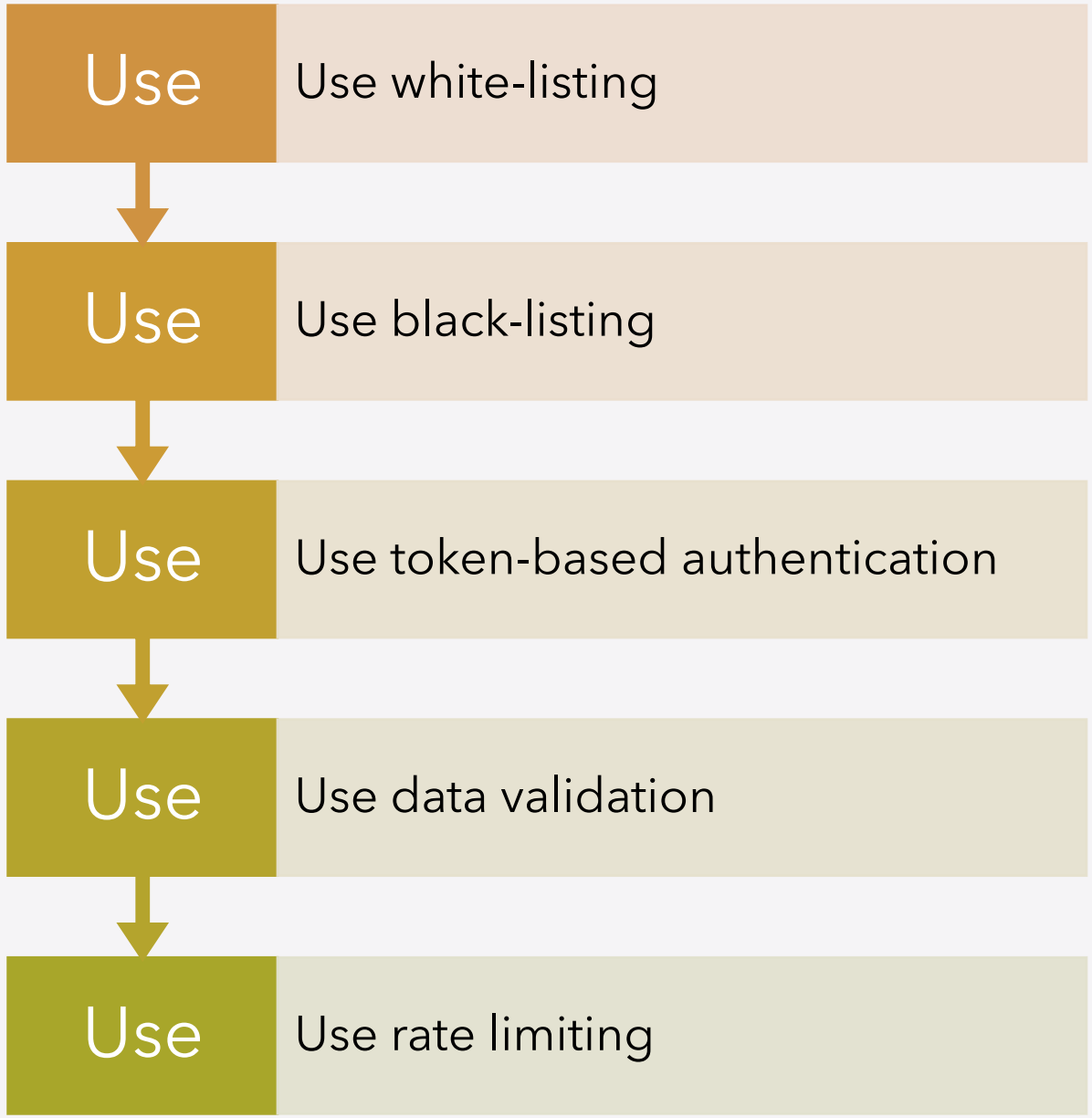



API endpoints gets client input and bind them to code variables or internal object properties without proper filtering can result in mass assignment vulnerabilities.

This allows users to change or overwrite object properties that they are not supposed to.

Depending on the application logic and API design, these illegal inputs can be sent from the client within the API URL paths, HTTP headers, or in the HTTP body itself in different formats

# How to prevent?



# Server Misconfiguration

Since servers are often left with default settings, this often leads to security vulnerabilities.

One way to prevent this problem is by making sure that your server configuration file is up-to-date.

Some examples:

- Improper access control
- Weak authentication
- Exposed sensitive data
- Poor error handling

# How to prevent

Getting a third party to check your server configuration.

You can hire a firm like Ethical Hacking Services or maybe even do it yourself.

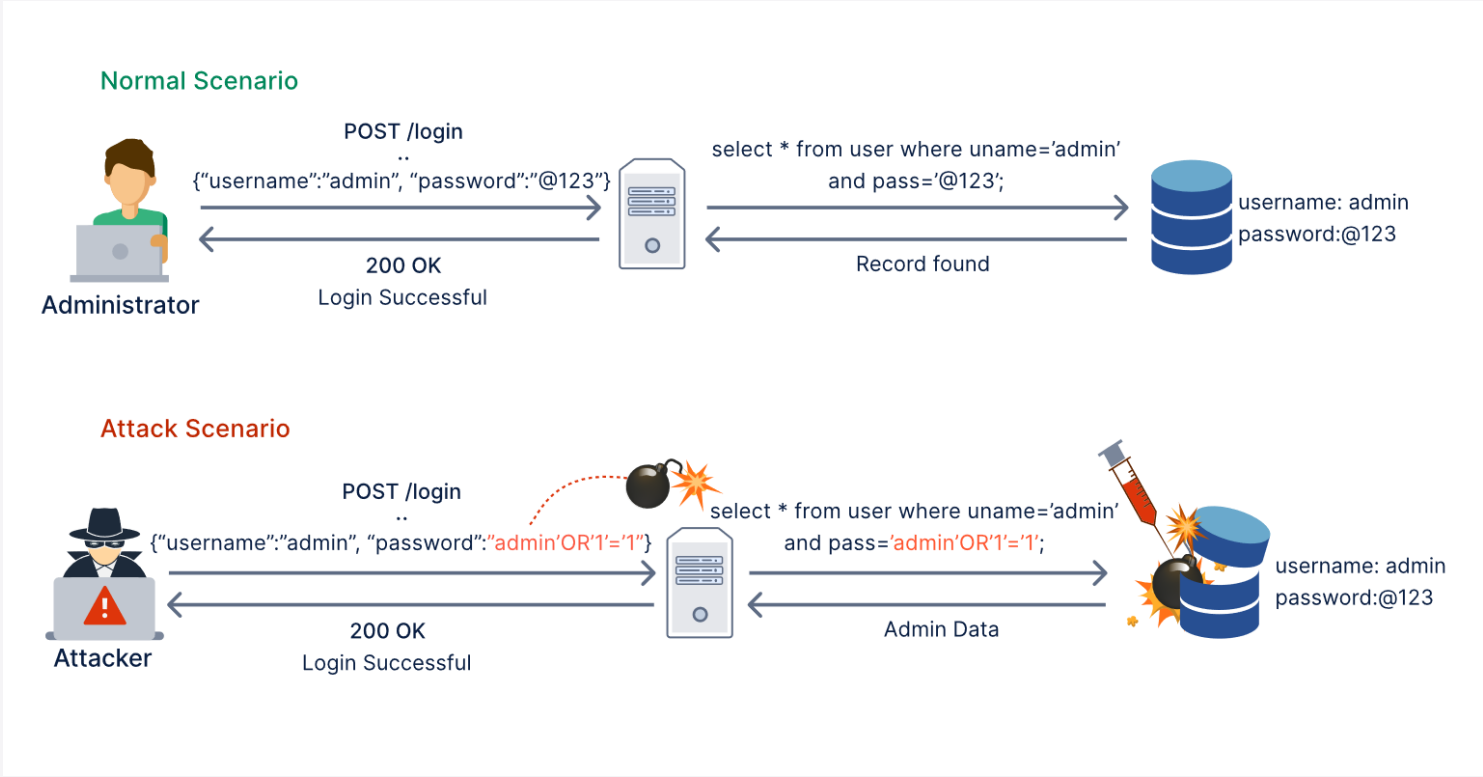
Disable unnecessary features.

Restrict administrative access.

Define and enforce all outputs, including errors.

# Injection

- Attackers construct API calls that include SQL, NoSQL, OS, or other commands, that the API or the backend behind it blindly executes.




- Use Parameterized Statements or Prepared Statements

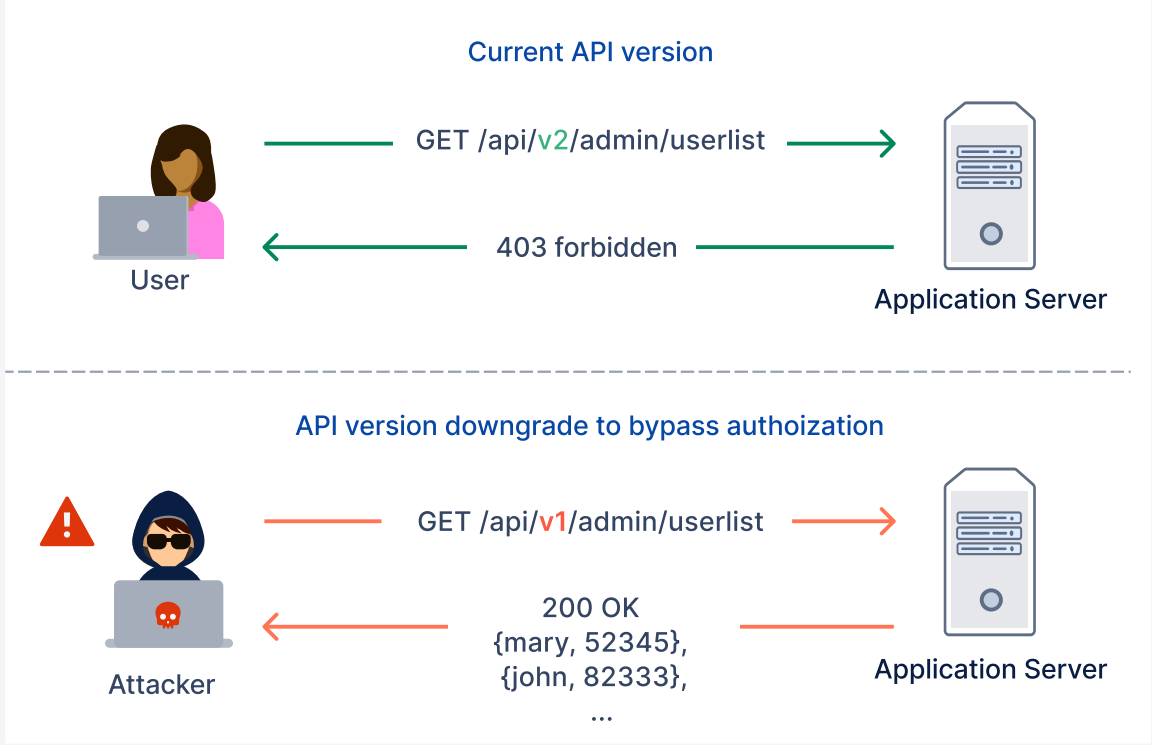
"SELECT * FROM users WHERE username = ? AND password = ?", (input_username, input_password))

- Input Validation and Sanitization
- Implement Web Application Firewalls (WAFs)

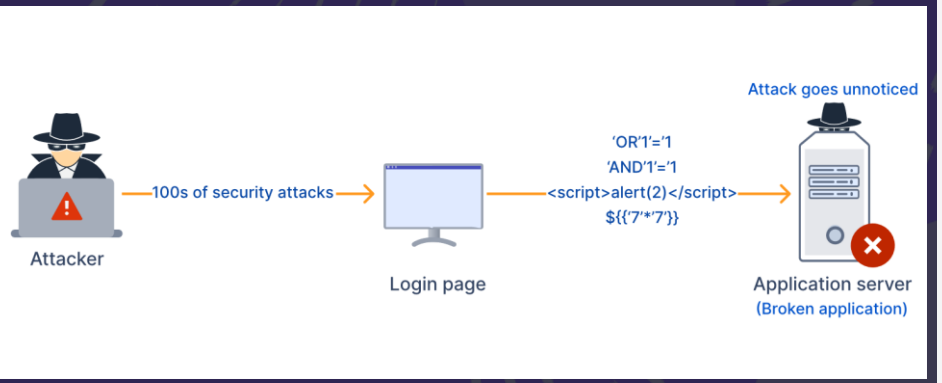
# How to prevent?

# Improper Asset Management

- Attackers find non-production, and/or older versions of the API (for example, staging, testing, beta, or earlier versions) that are not as well protected as the production API, and use those to launch their attacks.



# Insufficient Logging & Monitoring




The access keys of an administrative API were leaked on a public repository.

The repository owner was notified by email about the potential leak, but took more than 48 hours to act upon the incident.

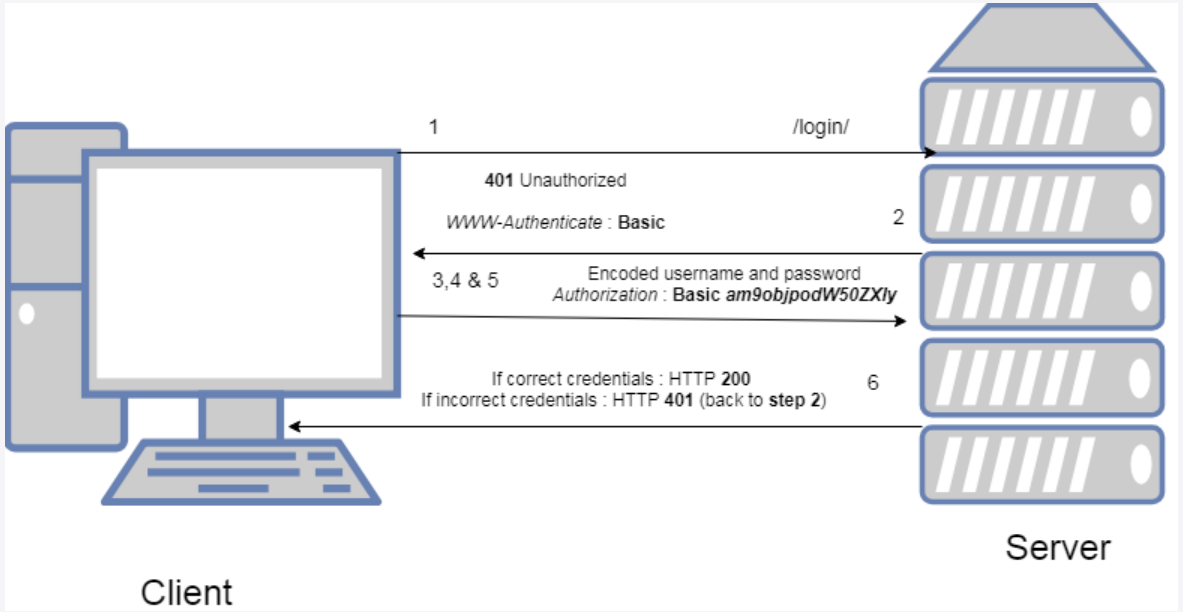
Due to insufficient logging, the company was not able to assess what data was potentially accessed by specific malicious actors.

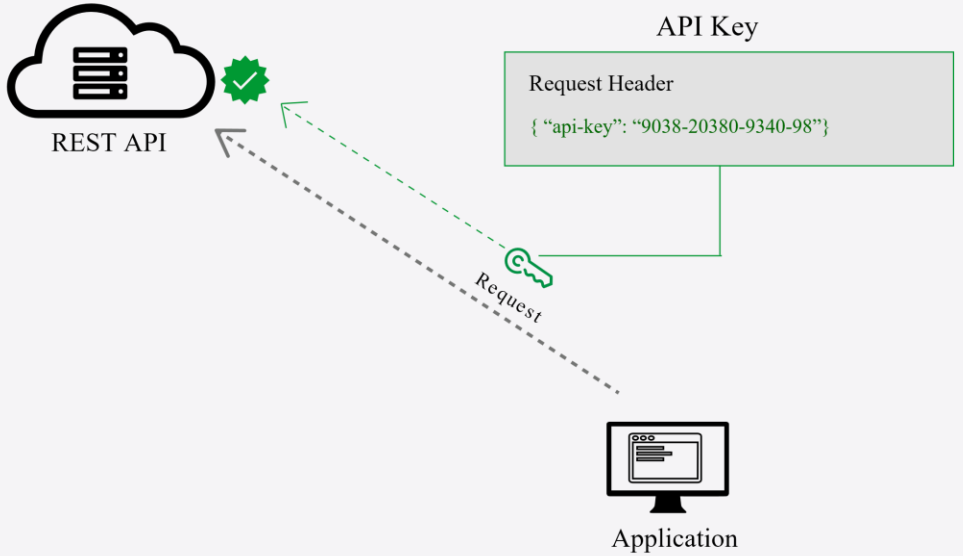
# RESTful API Authentication Basics

# RESTful API Authentication Basics

REST service will expose some kind of data or will allow some kind of interactions with a server.

Basic HTTP authentication is a method of authentication used in HTTP for RESTful services.





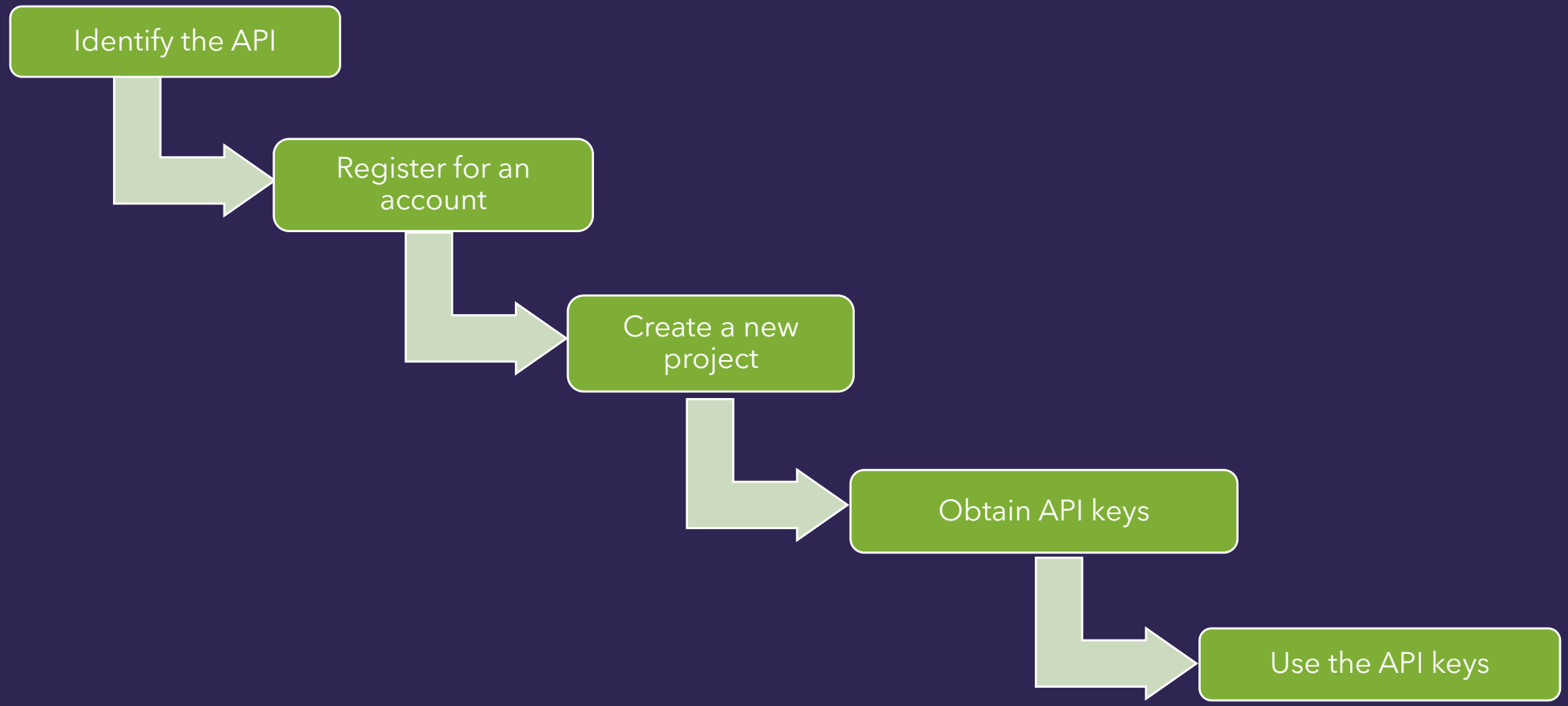

# API Keys

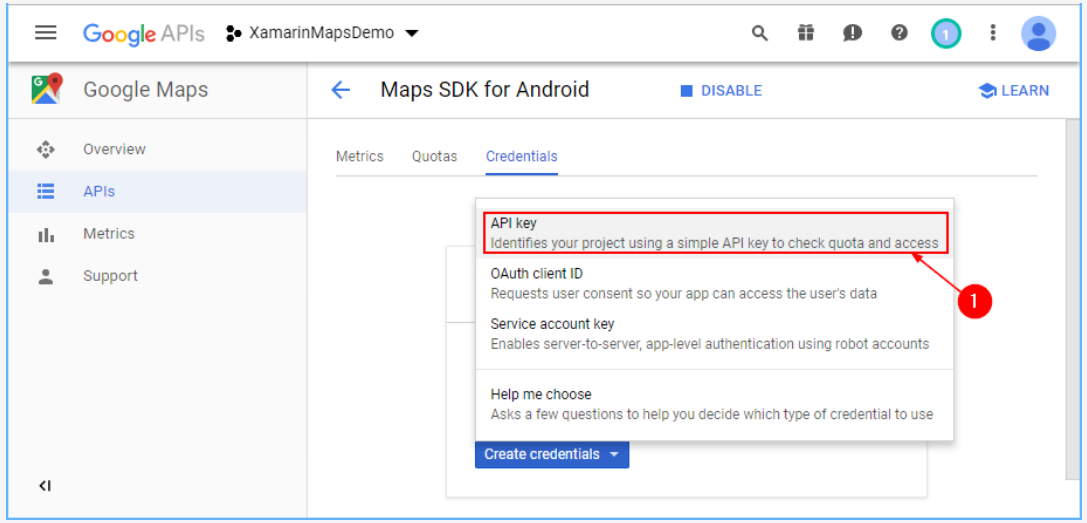
This key works as an ID card for API that allows seamless authenticity-verification of API end-users.

Whether it’s the developer or the application, API key implementation helps in the identification of both with the same ease and perfection.

When an API user makes a request to use an API, the API will ask for an API key and verify it. It’s only after successful verification that API access is granted to the end-user.

# How to get API Keys?






# How to get API Keys?

You need to contact an API key provider to get an API key.

For instance, the Google Maps key asks for a minimum of one API key beforehand.

Once the API key generation is complete, don't forget to limit the API key to prevent overconsumption or usage.

Add the API key to the request as per the need of the hour.

# OAuth (2.0)

is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service.

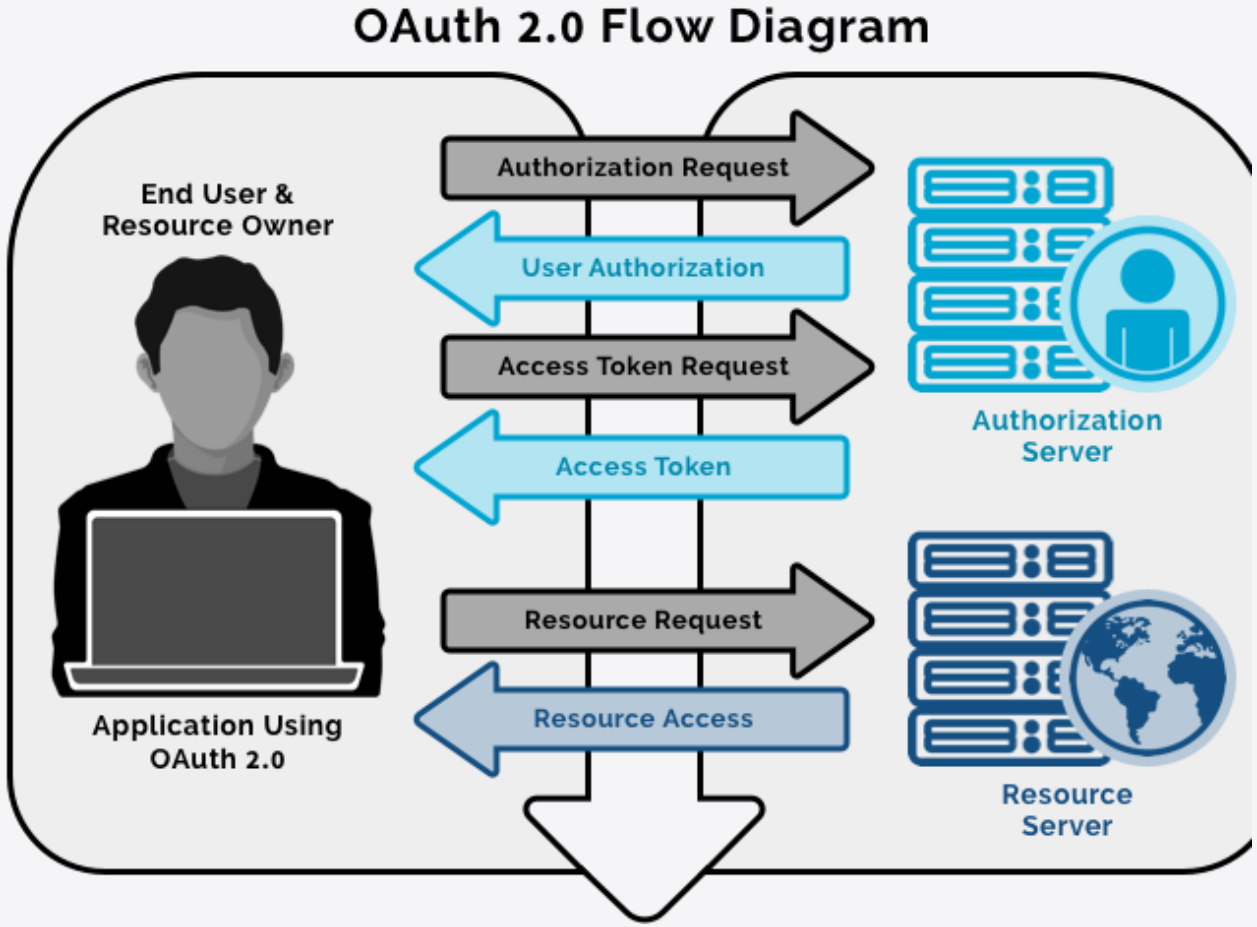
It works by delegating user authentication to the service that hosts a user account and authorizing third-party applications to access that user account.

The most common implementations of OAuth use one or both of these tokens instead:

- **access token**: sent like an API key, it allows the application to access a user's data; optionally, access tokens can expire.
- **refresh token**: optionally part of an OAuth flow, refresh tokens retrieve a new access token if they have expired. OAuth2 combines Authentication and Authorization to allow more sophisticated scope and validity control.

# How does OAuth work?

- **Application asks permission:** The application or the API (application program interface) asks for authorization from the resource by providing the user's verified identity as proof.
- **Application requests Access Token:** After the authorization has been authenticated, the resource grants an Access Token to the API, without having to divulge usernames or passwords.
- **Application accesses resource:** Tokens come with access permission for the API. These permissions are called scopes and each token will have an authorized scope for every API. The application gets access to the resource only to the extent the scope allows.



# OpenID Connect

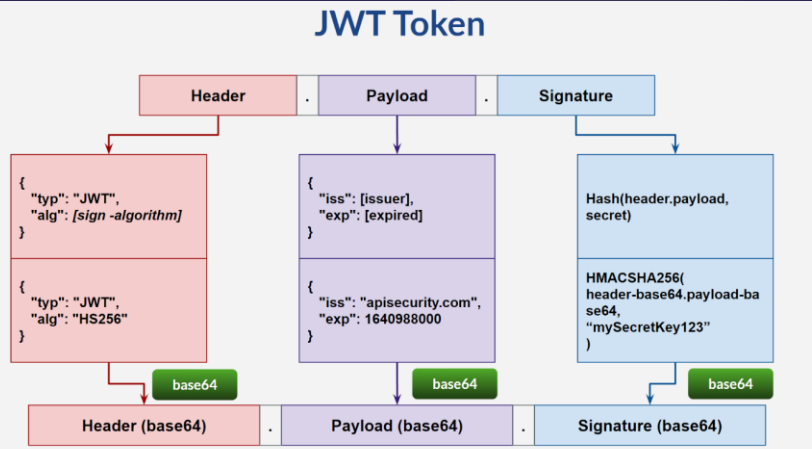

OpenID Connect is a simple identity layer on top of the OAuth 2.0 protocol, which allows computing clients to verify the identity of an end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user.

In technical terms, OpenID Connect specifies a RESTful HTTP API, using JSON as a data format.

OpenID Connect defines a sign-in flow that enables a client application to authenticate a user, and to obtain information (or "claims") about that user, such as the user name, email, and so on.

User identity information is encoded in a secure JSON Web Token (JWT), called ID token.

# JSON Web Tokens




JSON web token (JWT), pronounced "jot", is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

Because of its relatively small size, a JWT can be sent through a URL, through a POST parameter, or inside an HTTP header, and it is transmitted quickly.

A JWT contains all the required information about an entity to avoid querying a database more than once.

The recipient of a JWT also does not need to call a server to validate the token.

# JAX-RS Security

# JAX-RS - Container Managed Basic Authentication

Container Managed Basic Authentication is a method of authentication where the web container, such as Tomcat

To enable Container Managed Basic Authentication in JAX-RS, you need to Configure web.xml

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>secured</web-resource-name>
    <url-pattern>/api/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>MyRealm</realm-name>
</login-config>
```

# JAX-RS - Container Managed Basic Authentication

src/main/webapp/WEB-INF/web.xml

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                             http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" version="3.1">
    <security-constraint>
        <web-resource-collection>
            <url-pattern>/employees/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>EMPLOYEE</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>default</realm-name>
    </login-config>
</web-app>
```

# JAX-RS - Container Managed Basic Authentication Servlet Security

Since JAX-RS runs in a servlet container, we can implement container managed authentication by specifying <security-constraint> and <login-config> elements in web.xml.

In an servlet based application, we usually use @ServletSecurity annotation to specify a security constraint.

```
@Path("/admin")
@ServletSecurity(@HttpConstraint(rolesAllowed = "admin"))
public class AdminResource {
    // Resource methods here
}
```

# JAX-RS - Container Managed Basic Authentication

- Defining Users and Roles:
  - Since we are going to use embedded Tomcat server, we have to define users in a local file:

src\main\resources\tomcat-users.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tomcat-users>
    <role rolename="EMPLOYEE"/>
    <user username="joe" password="123" roles="EMPLOYEE"/>
</tomcat-users>
```

  - Following is the mapping for the local users file with embedded Tomcat plugin in pom.xml:

```xml
<plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.2</version>
    <configuration>
        <path>/</path>
        <tomcatUsers>src/main/resources/tomcat-users.xml</tomcatUsers>
    </configuration>
</plugin>
```

# JAX-RS Security

- The **javax.ws.rs.core.SecurityContext** interface provides information about the security context of a request.

```
@GET
@Path("/secured")
@RolesAllowed("admin")
public Response getSecuredResource() {
    //code to handle secured resource
}
```

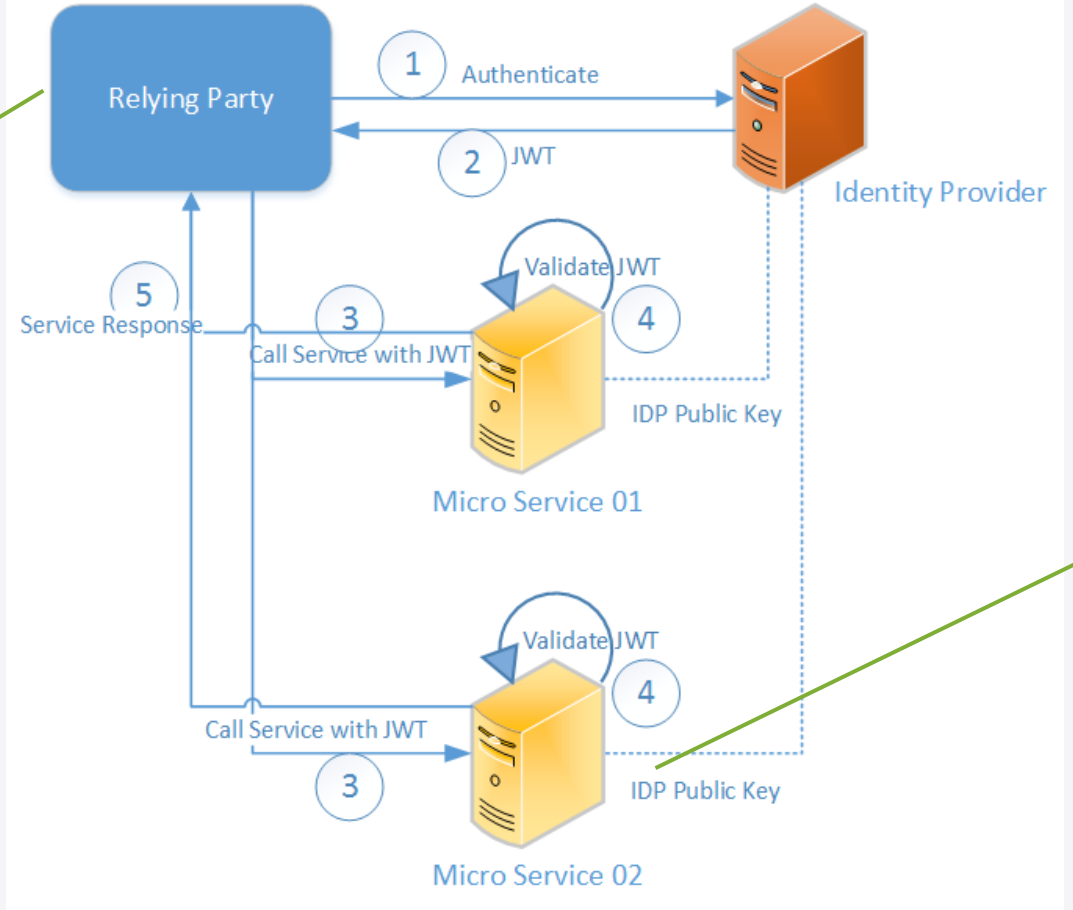
# JAX-RS Security

```java
@Path("/echo")
@Produces({ MediaType.TEXT_PLAIN })
public class EchoAPI {
    @Context
    protected SecurityContext securityContext;

    @GET
    @JWTSecured // limits filter to be applied to just this method
    @RolesAllowed("USER") // just for demonstration, check JWTRequestFilter
    public Response echo(@QueryParam("message") String message) {
        JWTPrincipal p = (JWTPrincipal) securityContext.getUserPrincipal();
        // TODO: inspect principal for fine grain security before proceeding
        return Response.ok().entity(message).build();
    }
}
```

# Secure JAX-RS API with JWT

Relying party (RP) is the web, mobile application that requests the identity token from an OpenID Connect (OIDC) provider.



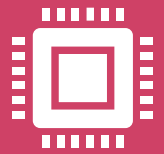

Identity Provider is the central part of the security architecture. There are various open source OIDC solutions available, e.g. Keycloak

# JAX-RS Filter for JWT

```java
@NameBinding
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface JWTSecured {

}


@Provider
@Priority(Priorities.AUTHENTICATION)
@JWTSecured // optional marker to limit the scope of this filter to methods matching wi
public class JWTRequestFilter implements ContainerRequestFilter {
    private Pattern     tokenPattern  = Pattern.compile("^Bearer$", Pattern.CASE_INSENS
    private JWSVerifier jwsVerifier;
    ...
```



A basic solution to secure a set of REST endpoints running in a Java Web container is to install a security filter.

A typical security filter intercepts all the incoming requests and examins authentication and applies authorization logic based on the security context.

The JAX-RS API allows us to subclass javax.ws.rs.container.ContainerRequestFilter to intercept incoming request to a REST endpoint.