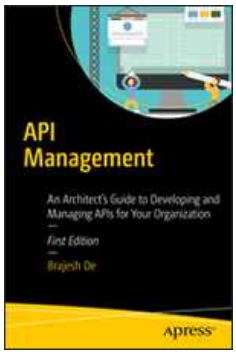

API Management: An Architect's Guide to Developing and Managing APIs for Your Organization

by Brajesh De

Chapter 7: API Security

B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_7

Overview

APIs provide a very good opportunity to build engaging and innovative customer experiences. They help businesses build new channels of integration and partnership. As companies look to expose their assets and data as REST APIs in an effort to provide new customer experiences and expand their business, security becomes a main concern. To a chief security officer at an enterprise, it is of paramount importance to secure the APIs and protect underlying assets from misuse, attacks, or any kind of threat. The security threats to APIs can be of various types. The security model to protect against these threats depends on the type of asset or service exposed and the associated risk. For example, APIs dealing with sensitive financial data over public networks require stronger security measures than APIs dealing with publicly available data over a restricted network. There is no one-size-fits-all approach that can be applied to protect APIs against various threats. This chapter looks at some of the important security threats to consider when building a security solution. It also looks at the various approaches and the security models for protecting APIs against these threats.

The Need for API Security

APIs allow consumers to interact with and access an enterprise's data and services. It is no good to have assets locked down within the enterprise. Exposing assets helps enterprises grow business and revenue. Creating APIs that enable customers to use their assets builds enriching customer experiences and increases customer engagement and loyalty. However, since assets have a business value, they are prone to theft and attacks to gain unauthorized access. APIs that act as the front door to these assets should therefore be secured. An API without security is like keeping the door to your vault open. Since APIs are used by in-house developers, trusted partners, and third-party developers, they should be protected intelligently. Considering that APIs may sit at the edge of the enterprise and can be accessed by a wide variety of customers through different channels, such as mobiles, smartphones, tablets, web apps, connected cars, kiosks, IoT devices, and more, they should be secured thoroughly to prevent any kind of misuse of the underlying assets.

Today, APIs introduce a new form of security threats by hackers. Earlier hackers used to sit behind a console and try out attacks to find vulnerabilities. Due to the programmable nature of APIs, hackers can now use them to automate their attacks and try out different things to find system vulnerabilities. Hence, the security model in the APIs should be able to identify such attacks and reject requests in order to protect the back-end assets.

APIs today form a critical part of any digital strategy. Lack of API security can bring your digital transformation journey to a grinding halt. Hence, having a well-defined strategy for API security is of principal importance.

API Security Threats

APIs provide channel of access to enterprise assets. Hence, they introduce many more types of security threats that were previously non-existent or were not considered a genuine threat. The different API security threats can be broadly classified into the following categories:

- Authentication
- Authorization
- Message or content-level attacks
- Man-in-the-middle attacks
- DDoS attacks (distributed denial-of-service)

APIs allow a new range of third parties to access enterprise assets. Without proper security policies in place, anyone can access these assets—even before a formal relationship has been established with third parties. Apps built by third parties can compromise enterprise security. Hence, it is important to have a proper registration and onboarding process for third-party organizations and app developers. Apps built by third parties should be registered with the API provider before they can use the APIs. Only authenticated systems, apps, and developers should be allowed access to the APIs in order to eliminate any risk of security compromise.

Third-party apps often access information on behalf of the end users. This information may be sensitive and private and can be accessed only after proper authorization has been obtained from the end user. Hence, APIs should be secured to check for the right level of authorization to grant in the request. Access to the resource should be granted only after authorization checks succeed.

Attackers can place malicious content, such as malware, in API requests to attack the system. They can also inject scripts in the request that are executed in the back-end systems. The impact can be devastating. It can corrupt systems and provide an outsider with unauthorized access to sensitive and business critical data. This can put a company's reputation at stake. APIs should be protected to detect any such malware or scripts, or malformed payloads in the request.

In a man-in-the-middle attack, hackers get access to credentials and tokens that can be used to get access to APIs. These credentials and tokens may be harvested and used nefariously. All data should therefore be encrypted in transit and protected from unauthorized access while at rest.

Attackers can launch DDoS attacks from one or more IP addresses via APIs to bring down the system. Since APIs provide a programmatic access to underlying resources, launching a DDoS attack is very easy. An API security model should be able to identify a DDoS attack and take the right action to protect back-end systems.

The next few sections look at how to design a security framework to protect APIs and their underlying resources against various forms of attack.

API Authentication and Authorization

Authentication determines the identity of the end user or the party requesting access to a protected resource. It helps validate who you are.

Authorization determines the access level and permissions of the end user to perform a certain operation. It determines the actions that the client is allowed to perform on the protected resource.

The following are some of the most commonly used forms of authentication and authorization used for API security:

- API keys
- Username and password
- X.509 client certificates and mutual authentication
- SAML
- OAuth
- OpenID Connect

API Keys

An API key identifies the application using an API. It provides a simple mechanism to authenticate the apps. API keys allow an API to determine which applications are using it. API keys are generally long series of random characters typically passed as an HTTP query parameter or header. This makes it easy to use an API key in an API request for application authentication. API keys are also known by other names, such as app ID, client ID, app key, or consumer key.

When a developer registers his app with an API provider, a unique API key is provided to the developer. The developer needs to secretly store this API key and use it in the application requests when making an API call from the application. The key identifies the application making the request and helps the provider monitor which application is making the request. The developer also gets insights into how his application is used by end users.

An API key is normally a long alphanumeric string that is opaque and without any signature or encryption. This makes API keys less secure for authentication purposes. The use of API keys is best for auditing and identification. An API key validation policy in the request flow of the API can help to validate the API key and also capture important metadata information, such as developer, organization, and so forth, related to the application. This may be good for APIs that only need to know who is using it. API keys can also be used to enforce API call quotas for an application (see Figure 7-1).

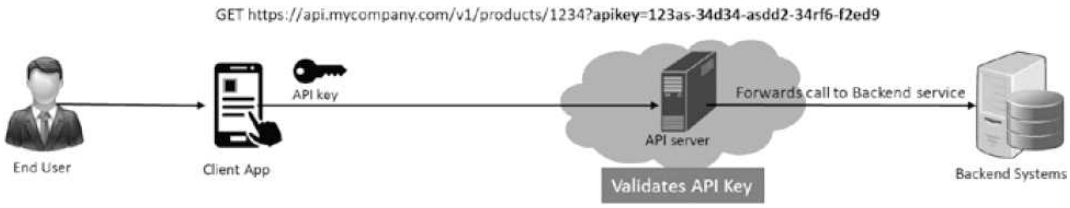

Figure 7-1: API key usage to secure back-end services

Thus, API keys can be also used to filter or turn off access to rogue applications that might be flooding the system with API calls. Providers can revoke an API key to block traffic coming from that application. Where enhanced security is required, API keys can be used to generate tokens using OAuth and OpenID flows.

Username and Password

A username and password is the most common form of authentication and is useful when dealing with sensitive data in an API call. In this form of authentication, the client presents the server with a unique name (username) and a secret code (password). The server validates the username and password against its credential store and provides access to the client only on successful validation.

For a REST API call, the client can pass the credentials (username and password) in an HTTP header using the Basic Authentication scheme. As per this scheme, the client sends the server authentication credentials using an `Authorization` header. The `Authorization` header is constructed as follows:

1. The username and password are combined with a single colon (:).

2. The resulting string is then Base64 encoded.
3. The authorization method and a space (`Basic`) are then entered before the encoded string.

The username *John* and password *John@123*, results in a header that looks like this:

```
Authorization: Basic Sm9objpKb2huQDEyMw==
```

HTTP Basic Authentication is the most common form of authentication; it is supported by nearly all clients and servers. It is easy to implement without the need for any special processing. The client needs to ensure that the password is protected and kept secret. If the client needs to store the password, it must be encrypted in some way to protect it against any attackers reading it from the store. SSL should be used for all client-server communications to protect credentials in transit from eavesdroppers.

API key validation with Basic Authentication can be combined for better API security. For example, the app identity may be sent as API key and the end user credentials may be passed in as Basic Authentication header. The API server may first validate the app identity from the API key and then validate the credential of the end-user accessing the client using the Basic Authentication headers.

X.509 Client Certificates and Mutual Authentication

An X.509 certificate contains a public key that validates an end entity, such as a web server or an application. It is a good alternative to a username/password for authentication purposes in application-to-application communication. The X.509 certificate contains the identity of the subject. The subject information is described as a distinguished name (DN), common name (CN), along with other optional attributes, such as country (C), state (ST), location or address (L), organizational unit (OU), and organization name (O). All of this certificate information is digitally signed by a trusted certificate authority (CA). This helps certify the public key of the subject and ensure that the certificate is not tampered with. The certificate's private key is always kept secret with the user and is never divulged to the signing authority or anyone else.

After the subject receives a signed certificate from the certificate authority, it can be used as identification. It allows secure access to protected APIs. For a mutual authentication using two-way SSL, the API resource server needs to import the client certificate in its trust store. The SSL handshake starts with the API resource server sending its X.509 certificate to the client. After the client app has validated the server certificate, it sends its public key to the API resource server. The server validates the client certificate against the list of certificates present in its trust store. A two-way SSL is established after the mutual authentication by the server and the client is successful. Only then can the app can make an API call. Figure 7-2 shows a high-level view of the message exchanges to establish a two-way SSL and make an API call.

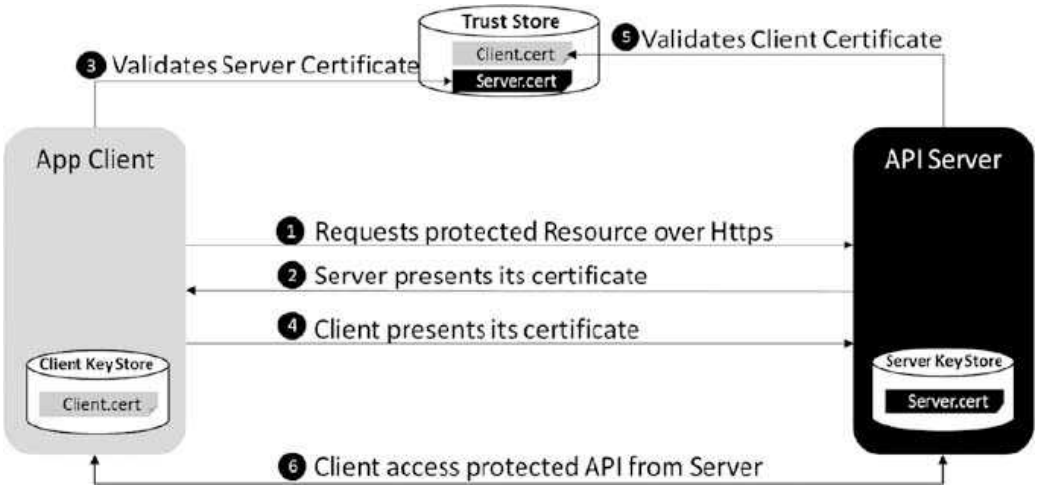

Figure 7-2: Two-way SSL for mutual authentication

OAuth

OAuth 2.0 is a protocol that allows clients to grant access to server resources without sharing credentials. As per the IETF specifications, the OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. For example, a shopping app can use access to its customer data in Facebook. When a customer accesses the shopping app, he is redirected to log in via Facebook. The customer is redirected to the shopping app after he has successfully logged in. The shopping app can now access customer data and can even post status updates on Facebook on behalf of the customer (if authorized to do so).

So what problem is OAuth trying to solve? Let's look at the scenario where a user wants to post some reviews about a product from the shopping app (say, Amazon) to Facebook but doesn't want to type their Facebook password on Amazon. This is possible if the Amazon app is able to store the user's Facebook password somewhere and use that to post on their Facebook page. But why should the user trust Amazon with their Facebook password? Also, what happens when the user changes their Facebook password, which is now stored in multiple locations with different apps? The user now has to manually go and update their Facebook password in all the locations that it is stored, which definitely is not a good user experience.

Instead of storing the Facebook password on every application that wants to access the Facebook account, what if we create a token that is authorized to perform limited actions, such as post on Facebook on their behalf. This token is generated after the end user has authorized Amazon to access their Facebook account.

The token has a defined validity and is understood and recognized by Facebook. So when Amazon presents the token to Facebook within the validity period of the token, it is allowed to access and post reviews on the user's Facebook page. In this way, users do not need to share their Facebook password with every other application that needs to access their Facebook account to post any updates. The access is automatically revoked when the validity of the token expires. The token can be revoked even earlier than its expiry time, if required.

Using OAuth tokens for API security makes APIs more resilient to security breaches, since they don't rely on passwords. In the previous example, if the user finds out that their Facebook password has been compromised, they only need to change it in one place, without impacting other applications that need to access their Facebook account. Those applications continue to access the Facebook account using the access token until it has expired or has been revoked.

OAuth Basic Concepts

To understand OAuth better we will now look at some of the basic concepts involved in the next few sections.

Actors in OAuth

OAuth protocol defines a sequence of message exchanges that need to happen between the various parties to grant the client access to a server resource. The various actors involved are resource owner, client, resource server, and authorization server.

- A **resource owner** is the end user who authorizes an application to access various resources in their account. For example, the user of a Facebook account can be the resource owner. The photos and activities like the posts and likes in the Facebook account are the data owned by the resource owner. The list of resources that an application can access or the operation that an application can do, is determined by the "scope" of the authorization granted.
- A **client** is the application that is trying to get access to a resource owner's account.
- A **resource server** hosts the protected resources of the user. In the API world, it is the server where the API resources are hosted. For example, Facebook is the resource server hosting the APIs to view or edit photos and user activities. Access to the API resources is allowed only after the client has been authorized by the resource owner or the user.
- An **authorization server** validates the identity of the user and then issues the access token to the client, which can be used to get access to resources.

Figure 7-3 shows the various actors involved in an OAuth flow.

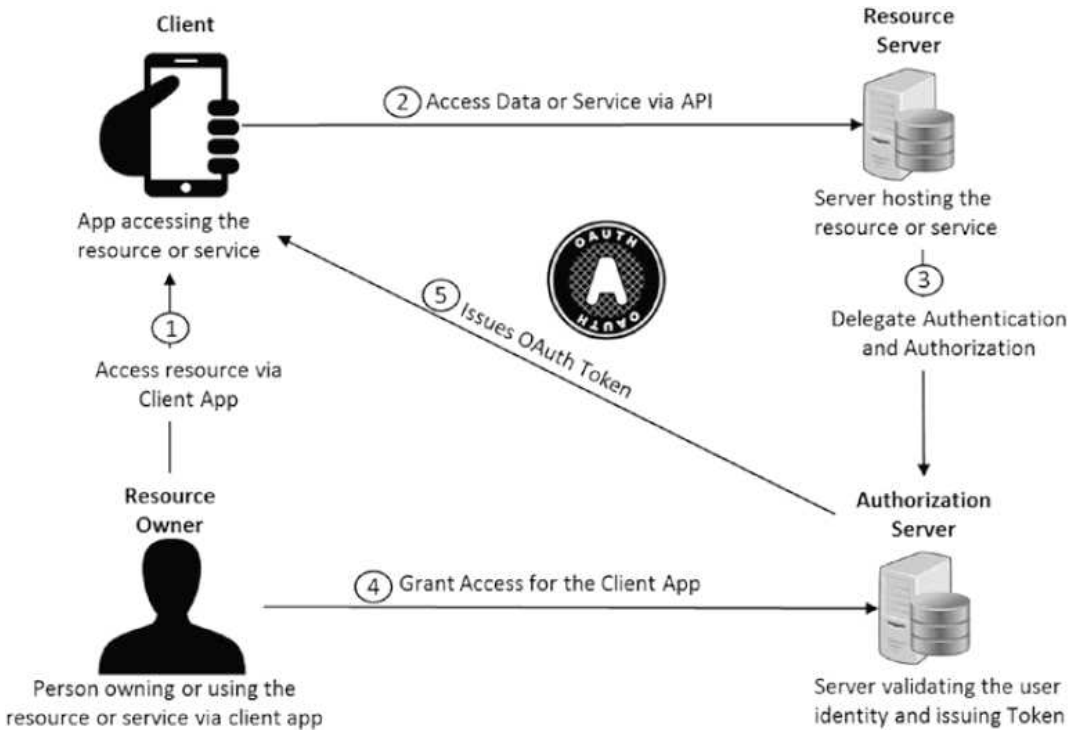

Figure 7-3: The various actors involved in an OAuth flow

In most cases, the server on which the API is hosted acts both as the resource server and the authorization server. An API gateway can play this combined role, as shown in Figure 7-4.

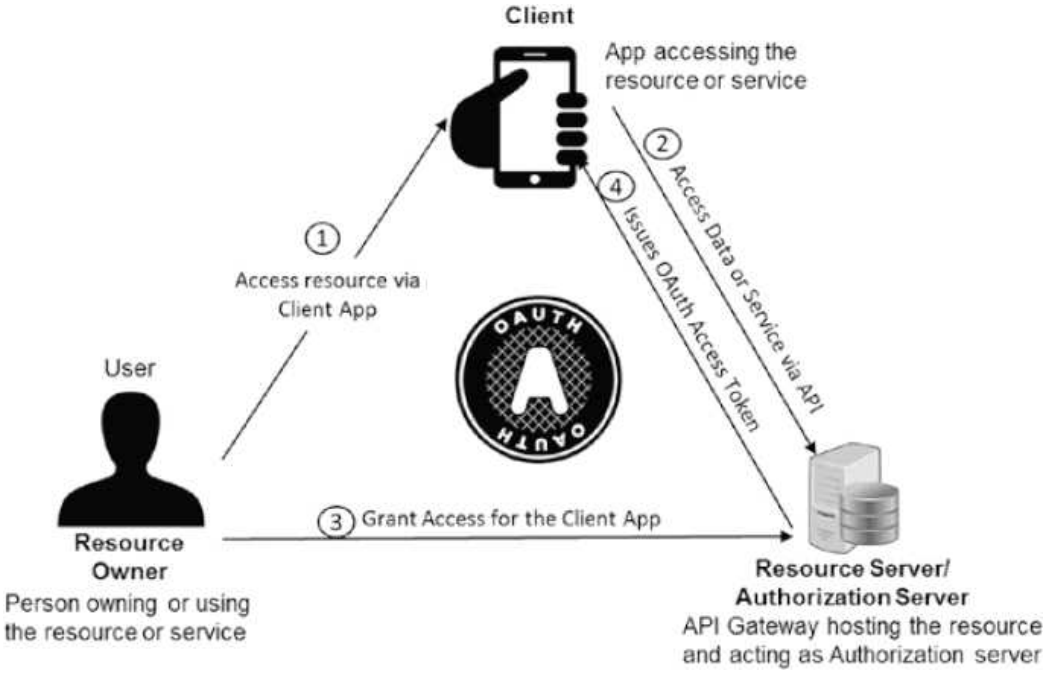

Figure 7-4: Role of API gateway in OAuth

Tokens

Tokens are issued to allow access to specific resources for a specified period of time and may be revoked by the user that granted permission or by the server that issued the token. There are two different kinds of tokens used in the OAuth flow: *access tokens* and *refresh tokens*.

- **Access tokens** allow access to a protected resource for a specific application to perform only certain actions for a limited period of time. They are a long string of characters that serve as a credential. They are generally passed as bearer tokens in an authorization header. An access token can also have restrictions or scope associated with it that specify the API resources that can be accessed using the token. An access token generally has an expiry duration and can be refreshed using refresh tokens for certain grant types. In situations where an access token is compromised, it can be revoked to prevent any further use of that token.
- **Refresh tokens** represent a limited right to reauthorize the granted access by obtaining new access tokens.

Scope

Scope identifies what an application can do with the resources that it is requesting access to. Scope names are defined by the authorization server and are associated with information that enables decisions on whether a given API request is allowed or not. When an application requests an access token, the scope names are optional.

Grant Type

An *OAuth grant type* can be thought of as the interactions that an app goes through to get an access token. OAuth 2.0 defines the following four grant types:

- Authorization code
- Client credentials
- Resource owner password credentials
- Implicit

Each of these grant types have their own pros and cons. The grant type used for generating a token depends on the business use case. One of the important considerations for choosing a grant type is the trust in the app accessing the resource.

Let's now look at each of the grant types in detail and learn about the flows involved for generating an access token for them.

Authorization Code

An *authorization code* is one of the most commonly used grant types. It is considered the most secure because it involves authorization from the end user, who actually owns the resource. The experience of using an authorization code grant type is similar to signing in to an app using a Facebook or Google account. This is sometimes referred to as "three-legged OAuth" since it involved three parties:

- End user

- Client app
- Authorization server

The following is the high-level process involved with the authorization code grant type:

1. Generate an authorization code.
    a. The end user logs in and grants consent to the application to access resources.
    b. The authorization server generates an authorization code that contains the scope information for which authorization was given.
2. Exchange authorization code for access token. The client application exchanges the authorization code for an access token from the authorization server. A refresh token is also generated and given to the client.
3. Use the access token. The client app uses the generated access to make API calls.

Figure 7-5 shows a detailed sequence of flow for generating an access token using an authorization code grant type.

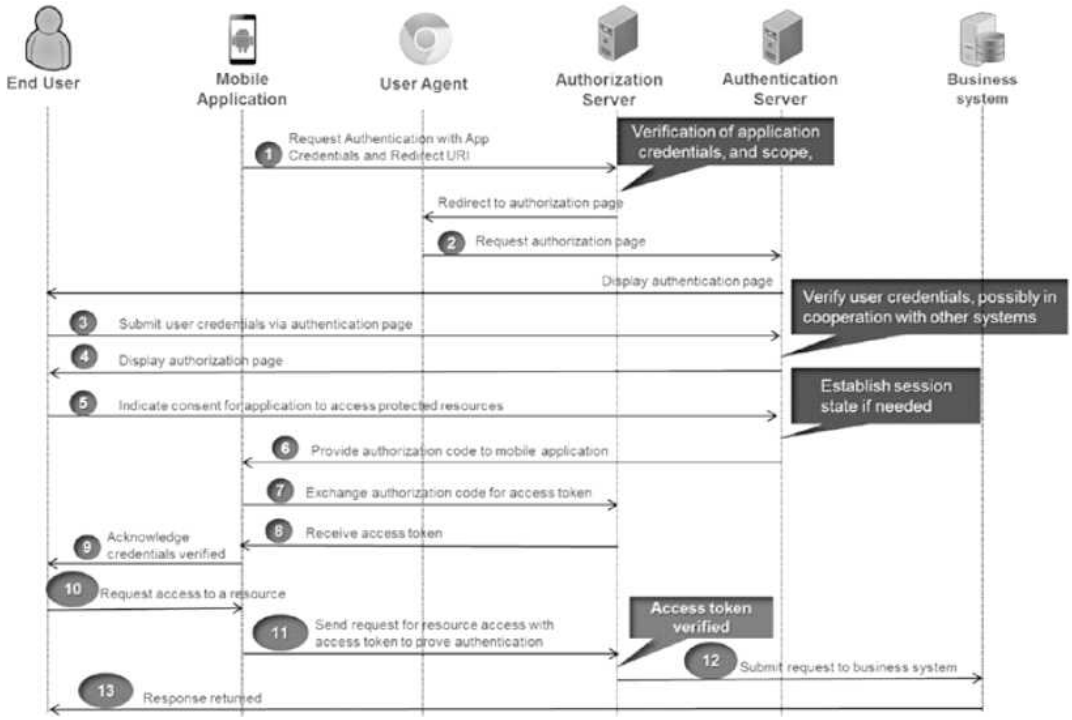

Figure 7-5: Authorization code grant flow for OAuth 2

Client Credentials

The client credentials grant type is suitable for machine-to-machine interaction and does not require any user permissions to access data. The following describes the high-level flow sequence (shown in Figure 7-6).

1. Generate access token.
    a. The client sends a message with its identity and the scope of access required to the authorization server.
    b. The authorization server validates the client's identity and issues an access token.
2. Use the access token. The client app uses the generated access token to make API calls.

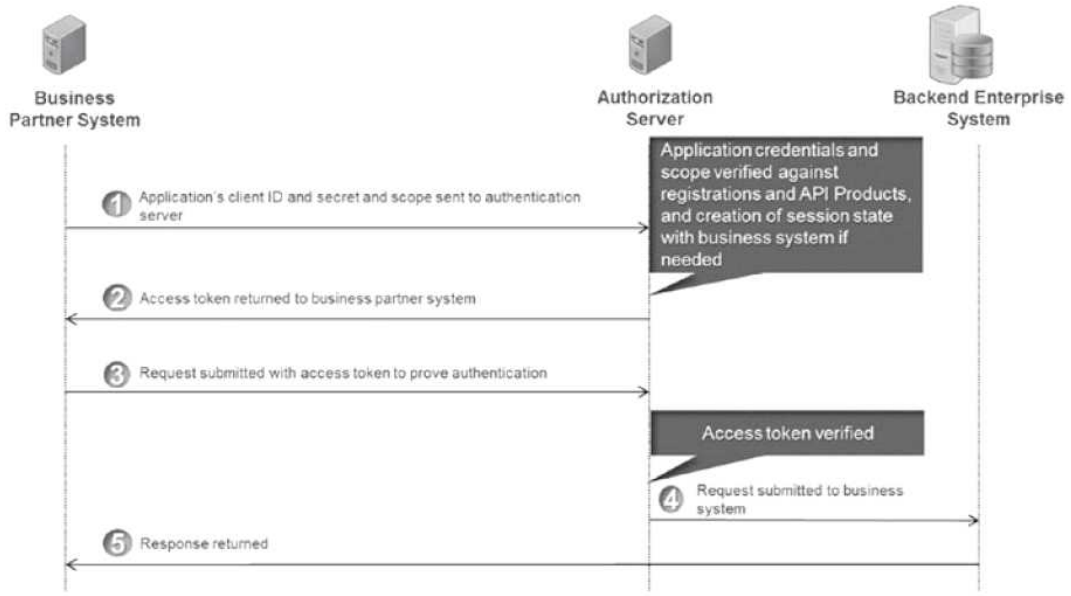

Figure 7-6: Client credential flow

Resource Owner Password Credentials

Resource owner password credentials grant type are used when the end user's credentials need to be authenticated before access can be granted. The following describes the high-level flow sequence (see Figure 7-7).

1. Generate access token.
    a. The client sends a request with its identity, scope, and the user's username and password.
    b. The authorization server validates the client's identity and user credentials.
    c. The authorization server issues an access token.
2. Use the access token. The client app uses the generated access token to make API calls.

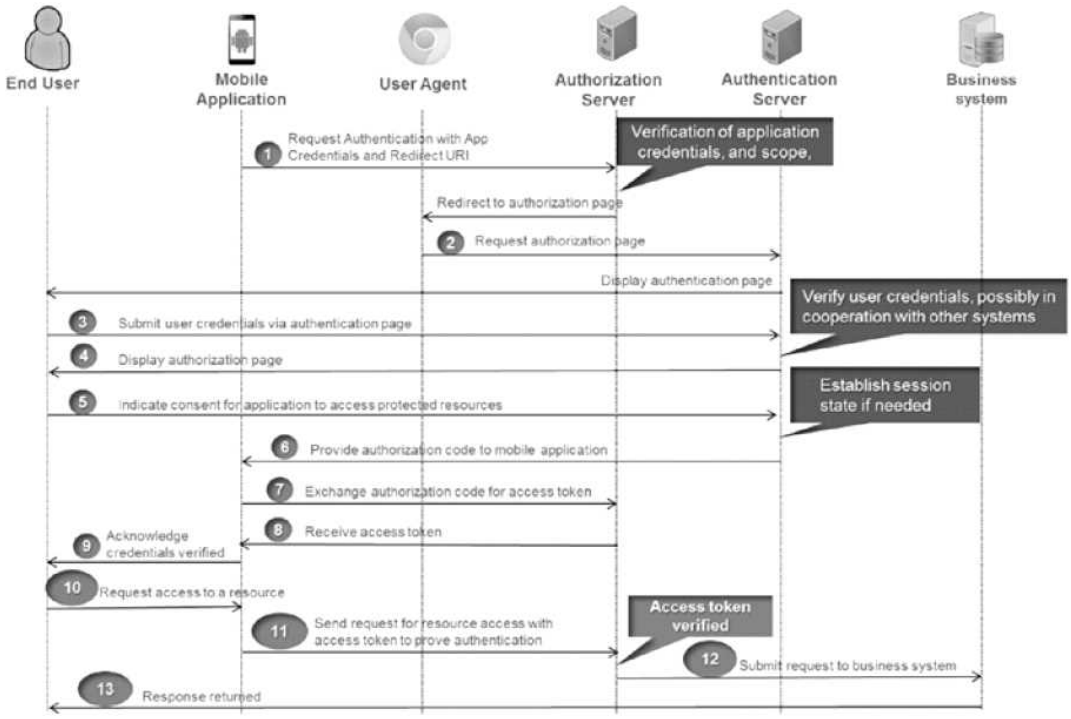

Figure 7-7: Resource owner password credential flow

Implicit

Implicit grant type is used by mobile apps and JavaScript applications running in the web browser. In this flow, the access token URL is given

to the user-agent to be forwarded to the client app via a redirect URL. Since the access token is encoded into the redirect URI, it may be exposed to the user and other applications running on the same device. The identity of the client is also not validated by the authorization server in this flow. Unlike the authorization code flow, where the client makes separate calls for authorization and for the access token, in the implicit flow, the client gets the access token as a result of the authorization request without any client authentication. The resource server only verifies the redirect URI that was originally registered. This makes the implicit flow easy but less secure. No refresh token is generated with implicit flow. Figure 7-8 shows the sequence of flow for the implicit grant type.

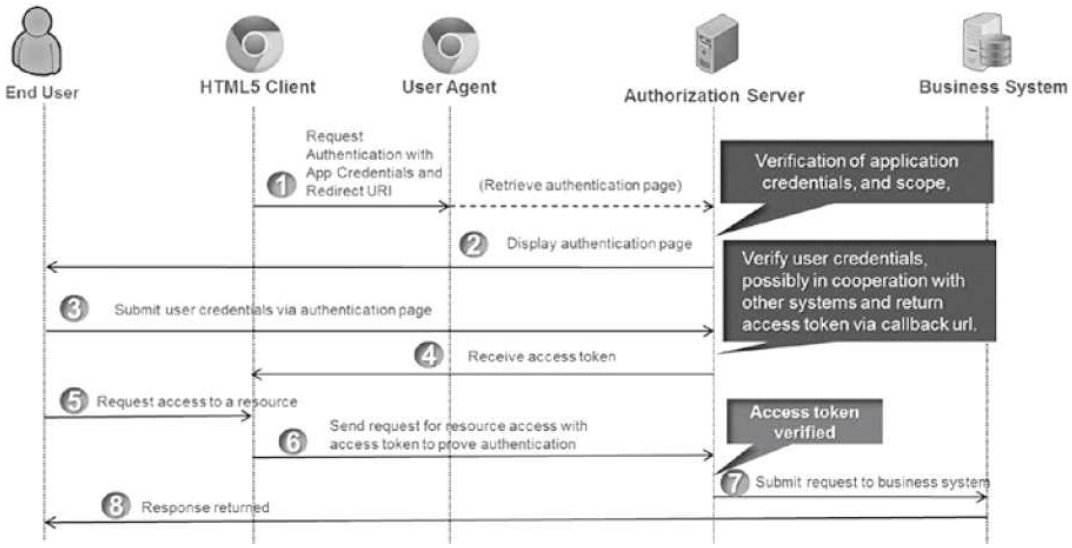

Figure 7-8: Implicit flow

OpenID Connect

OpenID Connect 1.0 is an authentication protocol that builds on top of OAuth 2.0 specs to add an identity layer. It extends the authorization framework provided by OAuth 2.0 to implement authentication. OpenID connect introduces an ID token in addition to the access and refresh tokens provided by OAuth 2.0. The ID token contains the identity information of the end user in JWT format. OpenID Connect defines *identity* as a set of claims or attributes related to an entity, which can be a person, a service, or a machine.

Actors in OpenID Connect

The following are the various actors involved in an OpenID Connect authentication flow:

- **OpenID Connect provider (OP):** An OAuth 2.0 authorization server that provides authentication as a service. It authenticates the end user entity and provides the claims or attributes of the entity to the client.
- **Relying party (RP):** An OAuth 2.0 client that requires end user authentication or claims from the OpenID Connect provider.
- **End user:** The entity that requests identity or claims information from the OpenID provider. The entity can be a human participant, a machine, or a service and is the owner of the resource that the client is trying to access.

Figure 7-9 is a high-level illustration of how different actors in an OpenID Connect protocol interact with each other.

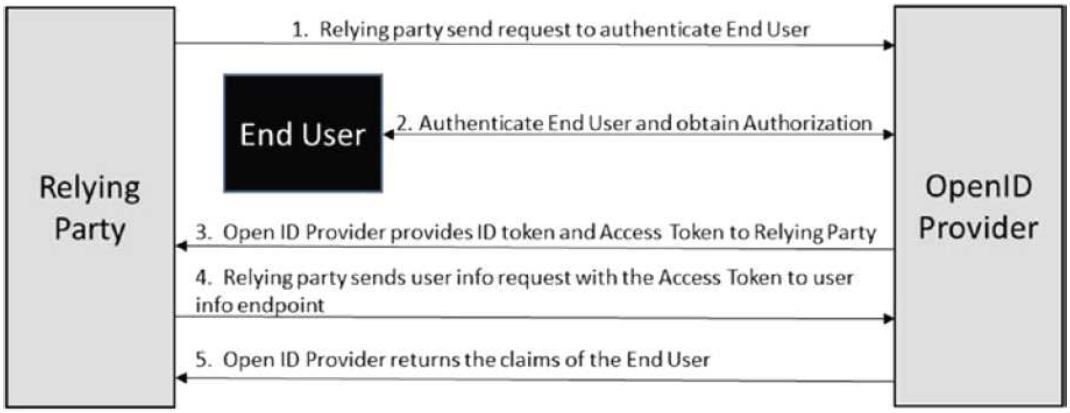

Figure 7-9: Interaction between parties in OpenID Connect flow

By using OpenID Connect, clients can request and receive identity and authenticated session-related information about the end user from a

central identity provider, and validate it. OpenID Connect can be used by clients of all types—including web-based, JavaScript, and native/mobile clients—to create a distributed and federated model for SSO.

ID Tokens

ID tokens are the main enhancements introduced by OpenID Connect on top of OAuth 2.0. An ID token is like an identity card that contains claims information about the authenticated end user. The ID token is represented as a JSON web token that is signed by the OpenID provider. The ID token contains the following claims information related to the end user in JSON format.

- **Subject identifier** (`sub`) is locally unique and asserts the identity of the end user.
- **Issuer identifier** (`iss`) identifies the issuing authority of the token. It is a case-sensitive URL using the `https` scheme. It contains the scheme, host, and optionally the port and path components.
- **Audience information** (`aud`) is what the ID token is intended for. It identifies the relying party and other audiences that can use this token. The OAuth 2.0 `client_id` of the relying party must be present in the audience information.
- An **alphanumeric string** (`nonce`) associates a client session with the ID token to prevent replay attacks. The nonce value is normally passed unmodified from the client authentication request to the ID token. If this value is present in a client authentication request, it must be included in the ID token response by the authorization server acting as the OpenID provider. If the nonce is present in the response, the relying party must validate that the value received in the response is equal to the value passed in the original request.
- The **time** (`auth_time`) when the end user authentication occurred.
- The **authentication context** class reference (`acr`).
- The **time** that the ID token was issued (`iat`).
- The **expiry date** of the ID token (`exp`).
- Optionally, it may contain other details about the entity, such as name and email address.

The following is a sample JSON format of the set of claims in an ID token.

```
{
  "iss": "https://example.server.com",
  "sub": “2340051",
  "aud": "s6Bhdr4k9qt3",
  "nonce": "n-078_WzB3Mj",
  "exp": 1317881970,
  "iat": 1316780970,
  "auth_time": 1311280969,
  "acr": "c2id.loa.hisec"
 }
```

The ID token is a JWT token created from the JSON format of the claims. JWT generally has three parts: a header, a payload, and a signature.

- The **header** specifies the algorithm used for signing and the token type in JSON format, as follows:

```
header = '{"alg":"HS256","typ":"JWT"}'
```

- The **payload** contains the claims in JSON format.
- The **signature** is calculated by Base64 encoding the header and the payload, concatenating them with a period separator, and then applying the signature algorithm on the concatenated string.

```
key = ‘mysecretkey'
unsignedToken  = encodeBase64(header)  + '.' + encodeBase64(payload)
signature  = HMAC-SHA256(key,  unsignedToken)
```

- The ID token is created by concatenating together the Base64- encoded value of the header, payload, and the signature with a period as the separator between them, as follows. This is done so that the token can be easily passed around.

```
token = encodeBase64(header) + '.' + encodeBase64(payload) + '.' + encodeBase64(signature)

eyJhbGciOiJKUzI1NiIsInR47CI6IkpXVCJ9.eyJsb2dnZWRsdKlbkFzIjoiYWRtaW4iLCJpYXQde5eE0MjI3Nzk2Mzh9.gz5raSYS8EXBxLN_oWnHRDgCzcmImMjLiuyu5CSpyHI
```

OpenID Authentication Flows

OpenID performs authentication to log in an end user or to determine if the end user is already logged in. The result of the authentication is securely retuned by the authorization server to the client in an ID token so that the client can rely on it. For this reason, the client is also referred to as the *relying party*. OpenID Connect defines the following three paths or flows for authentication to obtain the ID token:

- Authorization code flow
- Implicit flow
- Hybrid flow

Authorization Code Flow

In this flow's first step, an authorization code is returned directly to the client after authenticating the end user and receiving consent. In the second step, the client exchanges the authorization code to get an ID token and an access token. Since OpenID Connect is built on top of OAuth 2.0, the sequence of message exchange is almost same for both. The main difference being that the end-user is authenticated against an Open ID Provider and an ID token is generated and returned to the client in addition to the access token. The following are the high-level steps for the authorization code flow.

1. The client sends an authentication request to the authorization server containing the client_id, secret, redirect URI, and scope.
2. The authorization server authenticates the end user accessing the client against the identity provider.
3. The authorization server obtains consent and authorization from the end user for the client to access resources owned by the end user.
4. The authorization server sends the end user back to the client with an authorization code via HTTP 302 redirect.
5. The client sends a request using the authorization code to the token endpoint.
6. The client receives a response that contains an ID token and an access token in the response body.
7. The client validates the ID token and passes the access token to retrieve the end user's subject identifier.

An authorization server must implement the following endpoints to support the OpenID connect authorization code flow:

- Authorization endpoint (`/authorize`)
- Token endpoint (`/token`)
- User information endpoint (`/userinfo`)

An *authorization endpoint* is used to authenticate the end user and provide an authorization code to the client. The user agent is sent to the authorization endpoint hosted by the authorization server for authentication and authorization. The authorization request contains the following information:

- `scope:` Mandatory information sent in the request. For OpenID connect flows, this must have the `openid` value. It can also have other values for which the client is requesting access on behalf of the end user.
- `response_type:` This value determines the authorization processing flow to be used. For an authorization code flow, it must have the value of `code.`
- `client_id:` The identifier of the client making the request. The client gets this at the time of registration.
- `redirect_uri:` The redirection URL to which the response is sent. For security reasons, it must match the value of the redirect URI provided by the client at the time of registration to the OpenID provider.
- `state:` An opaque value that is used to maintain the state between the request and the callback. It is typically used to mitigate cross-site resource forgery (CSRF) attacks.

Other request parameters defined by OAuth 2.0 specifications may also be used.

The authorization endpoint must validate all the information sent in the authentication request according to OAuth 2.0 specifications. If the request is valid, the authorization server attempts to authenticate the end user or determines if the end user is authenticated. The method used for authentication is beyond the scope of the OpenID specification. The authorization server may display an authentication user interface to the end user, depending upon the values in the request parameters and the authentication method. The authorization server must authenticate the end user if not already authenticated or if the authenticate request contains the `prompt` parameter with the `login` value. After the end user has been authenticated, the authorization server must obtain consent from the end user before releasing any information to the relying party. The end user consent can be obtained through an interactive dialog with the end user. After the authorization server has successfully

authenticated the end user and received the consent, it responds with a successful authentication response containing the *authorization code* and the *state* information. This information is returned as a query parameter added to the `redirect_uri` specified in the authentication request. The following is a sample response from the authorization server.

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?
code=TplxlOBeZMMYbYS7WxSbIA &state=af2ifjhldbj
```

After successful authentication of the user, the client uses a *token endpoint* to obtain the following:

- ID token
- Access token
- Refresh token

The client or the relying party makes a token request by presenting the authorization code received from the authorization endpoint. The token request can be made using an HTTP POST call over TLS (Transport Layer Security) 1.2, as follows:

```
POST /token HTTP/1.1
Host: myserver.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic bzZCaGRSa4F0MzpnWMFmQmF0M2JZ

grant_type=authorization_code&code=TplxlOBeZMMYbYS7WxSbIA
&redirect_uri=https%3A%2F%2Fclient.example.com%2Fcb
```

The token endpoint must validate the token request, as follows.

1. Authenticate the client and validate its client credentials.
2. Validate that authorization code was issued to the authenticated client.
3. Verify the validity of the authorization code and ensure that it has not already been used.
4. Validate that the `redirect_uri` presented in the token request is same as that included in the authorization request.

After successful validation of the token request received from the client, the authorization server returns a successful token response containing the following:

- ID token
- Access token
- Refresh token

The following is a sample token response containing the three tokens:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "SlAV32hkKG",
  "token_type": "Bearer",
  "refresh_token": "8xLOxBtZp8",
  "expires_in": 3600,
"id_token":"eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogImlzcyI6ICJodHRwOi8vc2Vyd
mVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiMjQ4Mjg5NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3F0MyIsC
iAibm9uY2UiOiAibi0wUzZfV3pBMk1qIiwKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5N
zAKfQ.ggW8hZlEuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96Sr_P6qJp6IcmD3HP99Obi1PRsc
wh3lOp146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJNqeGpegccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1z
ZwgjxqGByKHiOtX7TpdQyHE5lcMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJbOEoRoSK5
hoDalrcvRYLSrQAZZlyuVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4XUVrWOLrLl0nx7RkKU8NXNHqr
vKMzqg"
}
```

The client receiving the token response must validate the received ID token as follows.

- If the ID token is encrypted, the client must first decrypt it using the keys and the algorithms that the client specified during the time of registration with the OpenID provider.
- The client must—at a minimum—validate the following information in the ID token:
  - The issuer identity of the OpenID provider must exactly match the value in the `iss` claim attribute.
  - The audience (`aud`) claim attribute must contain the `client_id` value that was issue to the client by the OpenID provider at the time of registration.
  - The algorithm value (`alg`) must be as negotiated at the time of registration.
  - The expiry (`exp`) claim of the ID token must be greater than the current time.
  - The issued at (`iat`) claim of the ID token is not too far from the current time. The client can decide on the value of this duration.
  - The *nonce* value, if sent in the authentication request, must match with the value received in the ID token.
- Other information, such as `acr` claim and `auth_time` claim, should also be provided by the client.

The *userinfo endpoint* (`/userinfo`) is an OAuth 2.0 protected resource that returns claims about an authenticated end user. The client makes a request to this endpoint using the access token received from the token endpoint to get claims and attribute information about the end user. The end user claims are returned as aname:value pair in a JSON object. All communication to the userinfo endpoint must use TLS. This endpoint must support both HTTP GET and POST methods. The endpoint must be able to accept and process a request containing an accept token in bearer format sent in the authorization header. The following is a sample userinfo request:

```
GET /userinfo
HTTP/1.1 Host: myserver.example.com
Authorization: Bearer SlBV35hkKH
```

On successful processing of the request, the endpoint returns the end user claims in a JSON format, as follows:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "sub": "548286761004",
  "name": "Fred Sweet",
  "given_name": "Fred",
  "family_name": "Sweet",
  "preferred_username": "fred.sweet",
  "email": "fredsweet@myserver.com",
  "picture":"http://myserver.com/fredsweet/fred.jpg"
}
```

Implicit Flow

Implicit flow is mostly used for browser (JavaScript)-based apps. In this flow, the client obtains the ID token and optionally the access token from the authorization endpoint. The authorization endpoint does not perform any explicit client authentication, but uses the redirect URI as an alternative way to verify the client's identity. After the client receives the tokens, it may expose the tokens to the end user and applications using the same user agent. Hence, this flow is used only for untrusted clients to obtain identity tokens. Unlike the authorization code flow, no refresh token is generated in this flow.

The implicit flow consists of the following steps.

1. The client prepares and sends an authentication request to the authorization server.
2. The authorization server authenticates the end user.
3. The authorization server obtains the end user's consent.
4. The authorization server sends the end user back to the client with the ID token and optional access token, if requested.
5. The client validates the token and retrieves the end user's subject identifier.

When the relying party wishes to validate the client, it prepares the authentication request and sends it to the authorization endpoint. The client can send this request either using the HTTP GET or the POST methods. For an implicit flow, the value of the `response_type` parameter in the request must consist of `id_token` and `token` as a space delimited list, as shown in the following example:

```
https://myserver.example.com/authorize?
response_type=id_token%20token
&client_id=c6BhdTkqt3
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&scope=openid%20profile &state=bf1ifjsldkj
&nonce=n-0S7_WzA3Mk
```

After authenticating the end user and obtaining consent, the authorization server responds with the `id_Token` and optionally `access_token`, as follows:

```
HTTP/1.1 302 Found
Location: https://client.myexample.com/cb#
    access_token=TlAV35hkKH
    &token_type=bearer
    &id_token=exJ0 ... NiJ8.eyJ2d ... I6IjJifX0.EfWt4Ru ... Zxso
    &expires_in=3600 &state=af0ifjsldkj
```

Hybrid Flow

The *hybrid flow* is a combination of the authorization code flow and the implicit flow and hence the name. This flow allows the client to make immediate use of the ID token to get access to the client's identity and retrieve an authorization code that can request a refresh token. The refresh token can gain long-term access to back-end resources.

The hybrid flow consists of the following high-level steps.

1. The client prepares and sends an authentication request to the authorization server.
2. The authorization server authenticates the end user.
3. The authorization server obtains end user consent.
4. The authorization server sends the end user back to the client with the authorization code. Depending on the `response_type` parameter, one or more parameters may also be returned.
5. The client requests a response using the authorization code at the token endpoint and received a response containing the ID token and the access token in the response body.
6. The client validates the ID token and retrieves the end user's subject identifier.

In the hybrid flow, the client makes the authentication request to the authorization server. The `response_type` parameter in the request can have the following values:

- `code id_token`
- `code token`
- `code id_token token`

The following is an example request using the hybrid flow that would be sent by the user agent to the authorization server in response to a corresponding HTTP 302 redirect response by the client:

```
GET /authorize?
    response_type=code%20id_token
    &client_id=f6BhdKkqg
    &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
    &scope=openid%20profile%20email
    &nonce=n-2S6_XzA2My &state=afligjslfku HTTP/1.1
Host: myserver.example.com
```

On receipt of the authentication request, the authorization server does the following validations before responding with a code and the ID token.

1. Validates the `scope` parameter present in the request.
2. Validates the `client_id` provided in the request and that the `redirect_uri` is the same as provided by the client at the time of registration.
3. Validates that all the mandatory parameters are present in the request as per the specifications.
4. Authenticates the end user or determine if the end user is already authenticated.
5. Obtains end user consent for the client to access the protected resources.

After successfully processing the authentication request, the authorization endpoint returns the authorization code. Depending on the value in the `response_type` parameter, the authorization endpoint returns the `id_token` and optionally the `access_token` in a response format, as shown in the following example:

```
HTTP/1.1 302 Found
Location: https://client.myexample.org/cb#
code=TplxlOAeZQQYbCS6WxSrIL &id_token=eyK0 ... NiK9.eyW1c
... I0IjIifL5.GeWt1Qu ... ZQsj &state=af9ifjs0dkj
```

Benefits of Integration with an Open Identity Provider

Applications often need to validate the identity of an end user. The following are possible ways to achieve this.

- A local database for user accounts and credentials for each app
- A central identity provider used by all end users to register apps and validate their information

With a local database for each app, end users have to register for each new app that they want to use. Many people find the registration process very tedious and not a good customer experience. For an enterprise providing multiple apps, maintenance of separate user databases brings in additional administrative and operational overhead. Hence, having a central identity provider provides a better option from user experience, as well as maintenance and administrative standpoints. Organizations such as Google and Facebook, which have large registered user bases, provide identity provider services than can be used with OpenID Connect. Organizations can streamline and simplify their customer onboarding and login processes by integrating with identity provider services.

Protecting Against Cyber Threats

In the era of social, cloud, and mobile technologies, where enterprises expose their sensitive data and information via APIs in a zero-trust environment, protecting APIs against malicious attacks is of paramount importance. Adding authentication and authorization to protect APIs is not enough. The API security framework must be able to detect any kind of cyber threat and take necessary actions to protect the back-end resources. To protect its APIs from different types of threats, an organization must build an API proxy in front of the APIs with an API management platform and implement security policies in these proxies to protect against such threats. Some of the most common types of threats are as follows:

- Injection threats
- Insecure direct object reference
- Sensitive data exposure
- Cross-site scripting (XSS)
- Cross-site resource forgery
- Bot attacks

The next few sections go into detail about each of these threats and exposes options of protecting against them.

Injection Threats

Injection threats are common forms of attacks, in which attackers try to inject malicious code that, if executed on the server, can divulge sensitive information. Malicious code can be in any of the following forms:

- XML and JSON bombs
- Script injection attacks

XML and JSON Bombs

Attacks using XML and JSON bombs try to use structures that overload the parsers thereby crash the service. Parsing corrupt or extremely complex XML/JSON payloads with long list of elements and attributes or long tag names and values or multiple levels of nesting can easily use up system resources—such as memory and CPU—and thus induce an application-level DOS attack. Such attacks can be mitigated by using XML and JSON threat protection policies.

XML threat protection policies can be used to check the message payload for the following, and reject the message if any of the allowed limits are exceeded:

- The length of the names of elements, attributes, and namespace prefixes
- The length of the values of elements, attributes, and namespace prefixes
- The node depth of an element
- The number of attributes in an element
- The number of namespaces defined for an element
- The number of child elements for an element

JSON threat protection policies can be used to check the message payload for the following, and reject the message if any of the allowed limits are exceeded:

- The length of a property's name within a JSON object
- The length of a property's string values within a JSON object
- The container depth of the JSON object
- The number of entries allowed in the JSON object of an element
- The number of array elements entries allowed within a JSON object

Script Injection Attacks

Script injection attacks can be in various forms

- SQL injection
- Script injection

SQL Statement Injection

SQL statement injection is a technique in which a hacker presents a malicious SQL query to an application's input parameter. This can be dangerous if the application takes this input in the request to directly query into the database. For example, an API (`/employees?EmpName=<Employee Name>`) that provides the details of an employee from the employee database. This API is implemented in a way to execute the following SQL statement in the database:

```
"select * from Employees where employeename =" + queryparam.EmpName + ";"
```

In this situation, if an attacker invokes the API with the following parameters, the effect can be catastrophic:

```
/employees?EmpName=Lary;drop table Employees;
```

SQL statements like the following can be used by hackers to bypass authentication:

```
select userid FROM customerdata WHERE username = ' ' OR 1 = 1
-- customer_passwd = 'abcd';
```

Hence, it is important that any API that accepts input that can be inserted into an SQL database must be protected against SQL injection attacks. Regular expressions that match certain SQL keywords can be used to detect malicious SQL content in the API request.

Script Injections

Script injections can be in various forms: JavaScript injection, XPath injection, or Java exception injection.

JavaScript is a powerful technology that modifies and sends data. If such scripts are injected though an API, they can reveal sensitive data. For example, hackers can get an unsuspecting user to execute a script in an API request to get access to their authorization token or cookies. The token or the cookie can then be used to log in to the system and steal sensitive information. This kind of attack is known as a *cross-site scripting* (XSS) attack.

XPath injections are also used by hackers to gain unauthorized access to sensitive stored in an XML format.

Input data in API request parameters should be validated and sanitized to harden the APIs and protect against script injection attacks. Regular expressions can detect the presence of malicious JavaScript and XPath in the payload of an API request. However, no regular expression can stop all content-based attacks. Hence, multiple mechanisms should be combined to enable defense-in-depth.

Insecure Direct Object Reference

In an Insecure Direct Object Reference attack, the hacker modifies an existing API request to get access to information. The hacker may try to modify parameters in the request to get a higher level of access. For example, the following API provides access to user account information identified by the account number specified in the URI:

```
GET http://api.myownbank.com/user/account/1234
```

A hacker can attempt to change the account number to get access to a different account. Alternatively, they may try to get admin access to an account using the following URL:

```
GET http://api.myownbank.com/admin/account/1234
```

This kind of attack can be prevented by using OAuth2/OpenID Connect with the right scopes set for the API.

Sensitive Data Exposure

APIs expose internal services and enterprise data. Some of this data may be customer sensitive and highly confidential. Such sensitive data should always be kept private and hence should always be encrypted and masked. Regulatory compliance standards such as PCI, HIPPA, and so forth, require that all sensitive data—such as credit card information and customers' private data—should always be stored in encrypted mode. When sensitive data is sent in an API response, it should be encrypted and tokenized to prevent inadvertent exposure. Again, there may be scenarios where only certain API users may be authorized to view certain information sent in an API response. If the same API is called by another user, some of the response data may have to be either filtered or masked. Sensitive data logged in debug trace should also be obfuscated.

Encryption of data in transit can be achieved by using SSL. Using SSL to encrypt sensitive data is the least any API should do. Another alternative is to selectively encrypt part of API message that contains the sensitive information. This requires the API provider and the client to take on the additional overhead of managing the private/public key. Hence, deployment of APIs that require selective encryption of sensitive data can be complex.

Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is among the top 10 open web application security threats. It is a type of script injection attack that exploits a vulnerability in a web site that the victim visits. The attacker injects malicious code, generally in the form of JavaScript, into otherwise benign and trusted web sites. When a user visits the web site, the malicious JavaScript is delivered to the victim's browser, which appears to be a legitimate part of the web site. The user information or data is compromised when these scripts are executed on the non-suspecting user's browser.

Figure 7-10 shows an example of how an XSS attack is done.

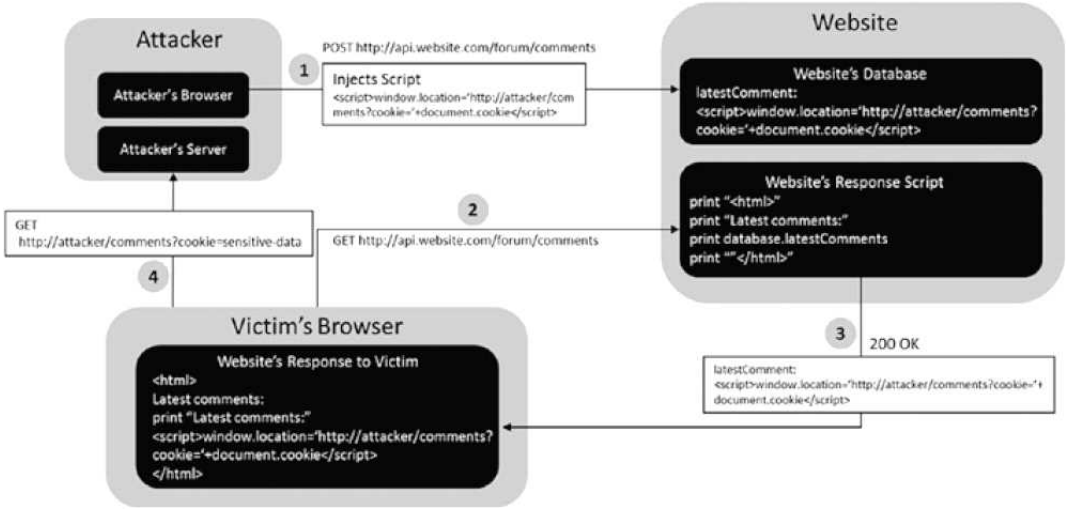

Figure 7-10: XSS attack approach

The following describes the process of an XSS attack.

1. The attacker injects malicious JavaScript into a web site's database using a POST request to submit a form.
2. An unsuspecting victim requests a page from the web site using a GET request.
3. The web site responds to the GET request with the malicious script from its database.
4. The victim's browser executes the malicious script in the response, sending sensitive data to the attacker's server.

Hence, to protect against XSS attacks, all user input for an API request must be encoded and validated. Encoding helps to escape the user input so that the browser interprets it only as data and not as code. Validations must include schema and data type validations, and check for the presence of any malicious scripts. Adding a Content-Security-Policy header in the HTTP response constrains the browser viewing a web page to only use resources (script/stylesheet, etc.) loaded from a trusted site. With a properly defined Content-Security-Policy, even if the attacker succeeds in injecting the malicious code, it will not be executed on the browser, since the attacker's site is not among the list of trusted sites.

Cross-Site Resource Forgery (CSRF or XSRF)

Cross-site resource forgery is a type of attack where a user is tricked into executing unwanted actions on a web application in which they are already logged in. This way, the attacker can target the web application, via victim's already authenticated browser. Using social engineering like email or chats, the victim is tricked into clicking a link that sends a forged request to a server where they are already authenticated. Since the user is authenticated, it is difficult for a web application to distinguish between a legitimate request and a forged one. This type of attack is different from XSS. In XSS, the attacker exploits the trust of the user on a web site; in CSRF, the attacker exploits the trust the web site has for the user. Figure 7-11 shows an example of how an attacker can launch and execute a CSRF attack.

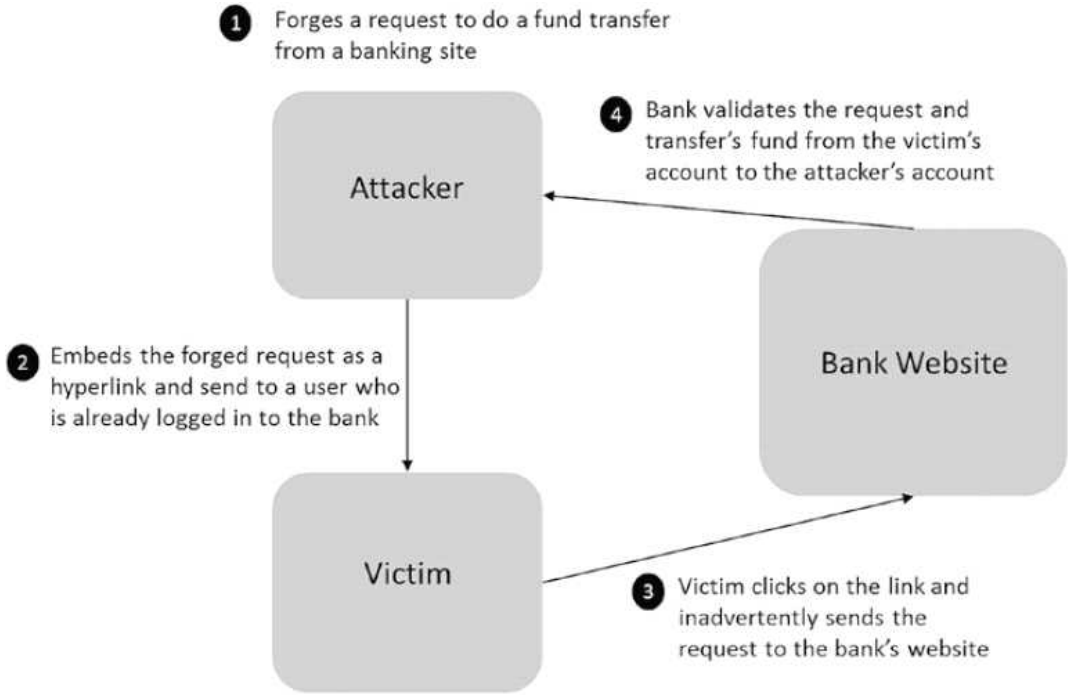

Figure 7-11: CSRF attack approach

An attacker uses CSRF to execute unauthorized fund transfers, change passwords or customer data, and many other things that can be detrimental to both the business and the user. The following techniques can be used to protect APIs against CSRF attacks:

- Use OAuth tokens to validate the requests. Tokens are long alphanumeric strings that are difficult for attackers to guess.
- Use a nonce that is unique for every URL and form, in addition to the standard session.
- Check for a "referrer" header in the HTTP request to ensure that the request has come from the original site.

Bot Attacks

In addition to the known threats, a new type of security vulnerability is arising from the use of automated software programs called *bots*. Since APIs provide a programmable interface, it becomes easier for hackers to target APIs using bots. Bot programs constantly scan the application infrastructure for security vulnerabilities. Bot traffic probes for weakness in APIs, abuses guest accounts with brute force, and uses customer API keys to access private APIs. Bot traffic can be identified by analyzing API traffic and access behavior patterns. Using machine learning and statistical models, an adaptive security system constantly learns "good behaviors," which helps it distinguish "bad behaviors" and enforce dynamic policies that block bots from accessing a protected resource. Bot traffic can be identified in the form of anomalous activities, as follows:

- Logical walk-throughs of the application resource paths by bots.
- Requests originating from a bot network, low-reputation IP address, ISP, or compromised proxies and devices. Malwares installed in rooted devices and PCs may be used to generate bot traffic.
- Unexpected high traffic volumes from certain IP addresses or endpoints.

- High traffic volumes to URIs (resources) that are not generally accessed by end users.
- High rates of form submissions with slight variations in the input parameters. This is one of the common techniques used by bots when applying brute force techniques to get access.
- High error rates on access to resources, especially those that are available to privileged users or applications.

API security strategies must consider how Bot activities can be easily identified through the analysis of API access anomalies. Research has shown that more than 50% of Internet traffic involves bot activities. Retailers and ecommerce service providers that provide dynamic pricing, loyalty programs, financial services, and so forth, are in the radar of bot attacks. Bots are known to target APIs with any valuable or sensitive data. Bot traffic can have a major load impact on API infrastructure, cause performance concerns, and hurt a company's brand and bottom line due to content theft. Advanced API analytics functionality with machine learning capabilities that can identify malicious bot activities should be considered for building a robust and adaptive API security system.

Considerations for Designing an API Security Framework

There are many aspects to consider in building the right API security framework. Some of the most important considerations include (but are not limited to) the following.

- The nature of the asset or the service being exposed as APIs. What is the impact/loss if data gets into the hands of someone who is not supposed to see it or if a service goes down?
- The regulatory compliance requirements for securing an API. Which regulatory standards should be followed for securing an API?
- The authentication requirements for using the API. Is it OK to authenticate only the client? Or is it necessary to authenticate even the end users before they can use the APIs?
- The authorization needs before a client app can access an API resource. Should the end user authorize access to an API before the client app can access it?
- Threats from API consumers. How can consumers and attackers possibly misuse the API and use loopholes to gain unauthorized access?

API Security Threat Model

To come up with the right security strategy for APIs, the security architect must create a threat model for API exposure and consumption. The following are some of the security threats that need to be considered in API security:

- Unauthorized applications and users may imitate that of another app or user
- Denial of service due to rogue apps or inadvertent errors
- Replay attacks
- Man-in-the-middle attacks
- Data tampering
- Malicious data injection attacks
- Theft of credentials, API keys, and tokens
- Network eavesdropping

API Security Recommendations

An API-centric security architecture that enables defense-in-depth security practices must be adopted to protect data and services from API security threats. This approach builds a security capability that includes role-based access control, fine-grained policies for authentication and authorization, and threat protection against malicious payload content and DoS attacks. The following are some API security recommendations for building a robust API security architecture.

- All API communication involving sensitive data must be secured and encrypted using TLS.
- Build a mechanism to detect malicious content injections and defend against such attacks. This protection is ideally built at the beginning of the API request flow at the edge of the network.
- All incoming and outgoing data must be validated and sanitized. Input data type and format validation must be done at a minimum for all APIs that have input request parameters. This prevents any malicious content from entering the system.
- APIs accepting input parameters via HTTP POST or PUT methods must validate the payload. Such validations help detect large payloads or malformed content that can potentially overload computing resources. Replay attacks and message tampering can also be detected early through these validations. Input parameters passed as query parameters in GET methods should also be validated to check for any malicious contents.
- Use a combination of approaches to identify the source of the request. IP address validation may not be sufficient to identify the originator

of a request since IP addresses can be easily spoofed.

- Protection via API key validation can be used only for nonsensitive and read-only data. API key validation identifies the applications and developers making API calls. It also implements API quotas and monitors usage by applications. If the data exposed is non-sensitive and read-only, such as Google Maps APIs, tracking consumer identities through API key validation might be sufficient.
- Use OAuth2 for public or private APIs that are intended for use by native and mobile apps. With OAuth2, the user is not required to share his password with the app and device that he uses. When a user authenticates in an OAuth flow, he enters his credentials in a web browser screen, rather than the application itself. Hence, the application never gets to see the user's password. This becomes a crucial factor when these apps are built by untrusted developers. Since OAuth uses tokens for authorization, API providers can revoke these tokens in any compromise, without the need for users to change their passwords.
- Use OpenID Connect for APIs that need end user identity and authentication. The ID token provided in the OpenID Connect flow can be used by the client or relying party to validate the end user. It can also be used by the API provider to validate the end user trying to get access to a protected API resource.
- Use two-way SSL or TLS with mutual authentication for APIs that are used by a limited number of internal or partner systems authenticating the client. If the API is open to all, maintaining client certificates for a large number of clients to implement two-way SSL may become a real challenge. The Basic Authentication scheme can also be a suitable alternative for authenticating partners.
- All sensitive information must be encrypted in transit using SSL.

Figure 7-12 shows the recommended order in which API security policies must be implemented in an API gateway.

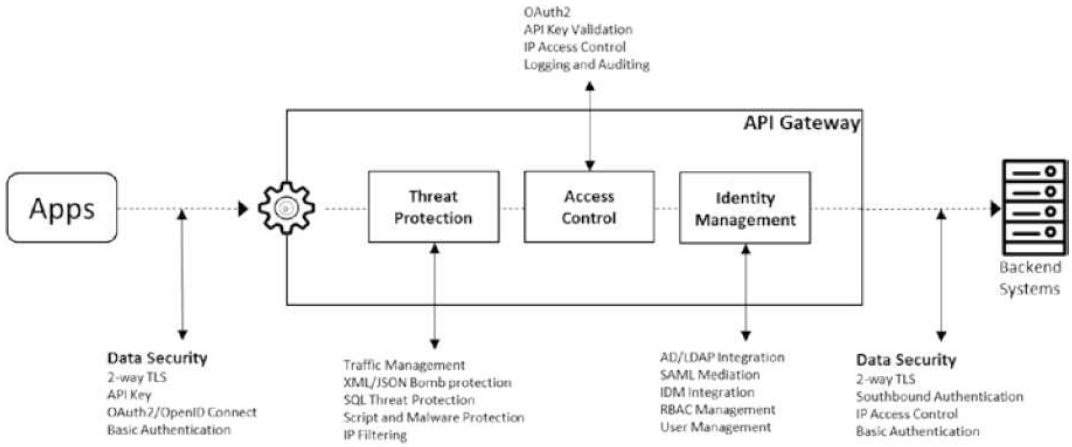

Figure 7-12: Approach for building end-to-end API security