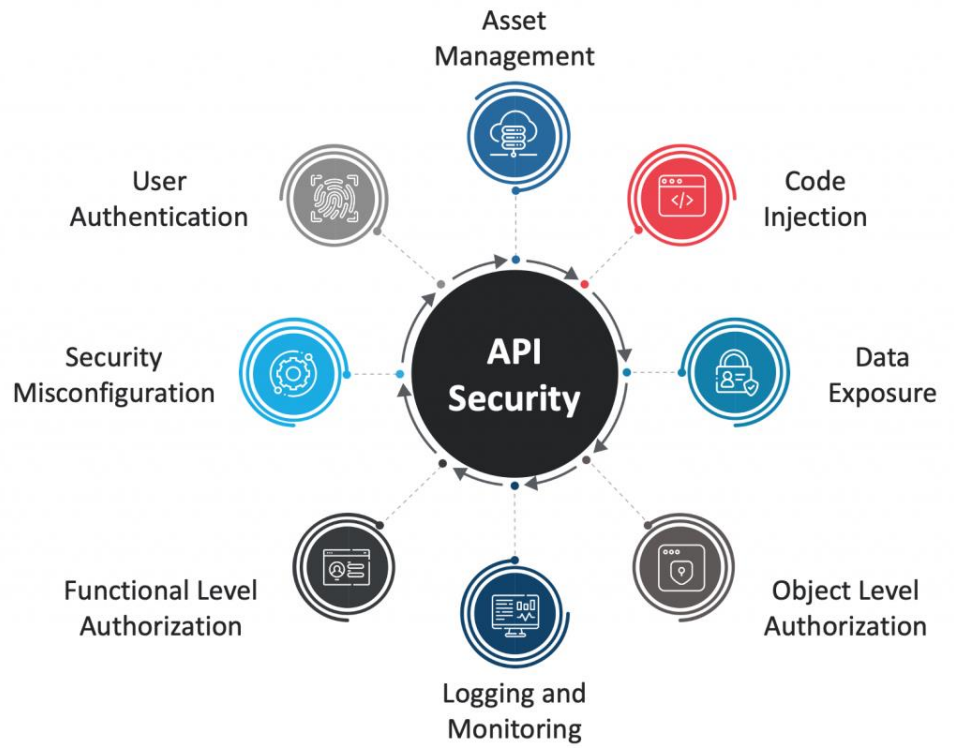
Web Security Best Practices

What is API Security?

An API represents a set of services that allow one program to communicate with another external or internal program. When we talk about API security, we typically refer to securing an application's backend services, including its database, user management system, or other components interacting with the data store.

API security encompasses the adoption of multiple tools and practices to protect the integrity of a tech stack. A robustly secured API covers both an organization's APIs and its services. This includes preventing malicious actors from accessing sensitive information or taking actions you did not intend to perform on your behalf. Unfortunately, while APIs are a crucial part of modern applications, they are a common target of attackers to access sensitive information.

Understanding how third-party applications funnel data through the interface when using APIs is crucial. Furthermore, with APIs increasingly becoming an attack vector, API security measures help security teams to assess security risks and have a comprehensive plan to protect them.



- **Security Practices May Varies:**
    - ✓ 1.May varies from application to application
    - ✓ 2.May varies from developer to developer
    - ✓ 3.May varies from environment to environment
    - ✓ 4.May varies from use case to use case

- **But we have to know:**
    - ✓ 1.The best practices
    - ✓ 2.Know about the security layers
    - ✓ 3.Security placement

Securing APIs with Best Practices for API Security:

The following web API security best practices can help mitigate API attacks and secure APIs:

- **Use throttling and rate-limiting:** Throttling involves setting a temporary state that allows the API to evaluate every request and is often used as an anti-spam measure or to prevent abuse or denial-of-service attacks. There are two primary considerations when implementing the throttling feature: how much data should be allowed per user, and when should the limit be enforced?

  On the other hand, rate-limiting helps administer REST API security by avoiding DoS and Brute force attacks. In some APIs, developers set soft limits, which allow clients to exceed request limits for a brief duration. Setting timeouts is one of the most straightforward API security best practices, as it can handle both synchronous and asynchronous requests. Request queue libraries enable the creation of APIs that accept a maximum number of requests and then put the rest in a waiting queue. Each programming language comes with a queue library directory to implement request queues.

- **Scan for API Vulnerabilities:** To maintain the continuous security of API services, it is vital to enable API automatic scanning, identify vulnerabilities, and mitigate them across software lifecycle stages. Automated scanning tools autonomously detect security gaps by comparing the application's configuration against a known vulnerabilities database. Crashtest Security Suite offers a vulnerability scanner that helps establish a continuous testing process and eliminates the security risk of being hacked through vulnerabilities of an API.

- **Use HTTPS/TLS for REST APIs:** HTTPS and Transport Layer Security (TLS) offer a secured protocol to transfer encrypted data between web browsers and servers. Apart from other forms of information, HTTPS also helps to protect authentication credentials in transit. As one of the most critical practices, every API should implement HTTPS for integrity, confidentiality, and authenticity. In addition, security teams should consider using mutually authenticated client-side certificates that provide extra protection for sensitive data and services. When building a secure REST API, developers should avoid redirecting HTTP to HTTPS, which may break API client security. Adequate steps should also be taken to divert Cross-Origin Resource Sharing (CORS) and JSONP requests for their fundamental vulnerabilities for cross-domain calls.

- **Restrict HTTP Methods to Secure APIs:** REST APIs enable web applications that execute various possible HTTP verb operations. Data over HTTP is unencrypted, and using some HTTP methods may be intercepted and exploited by attack vectors. As a recommended best practice, HTTP methods (GET, PUT, DELETE, POST, etc.) that are inherently insecure should be forbidden.
  If a complete forbidding on their usage is not possible, security teams can also apply policies to vet the use of such methods with a strict allow list, whereby all requests that do not match the list should be rejected. It is also recommended to utilize RESTful API authentication best practices to ensure that the requesting client can use the specified HTTP method on the action, record, and resource collection.

- **Implement sufficient input validation:** In principle, data supplied by the API client should not be trusted blindly since the authentication server may execute a malicious script from unauthorized users or application services. To avoid this, security teams should implement input validation mechanisms on both the client and server sides to prevent unhealthy input. While client-side validation involves interactive indication of errors and advice to a user on acceptable inputs, server-side validation additionally checks the data received to avoid the different types of XSS and SQL Injection attacks.

- **API Security by Using an API Gateway:** An API gateway decouples the client interface from the collection of backend APIs, delivering a centralized resource for consistent availability and scalability of API services. Apart from managing various API services, the API management platform also handles standard functions, including telemetry, rate limiting, and user authentication, to maintain security between internal services. The gateway acts as a reverse proxy gatekeeper that accepts all API calls, coordinates the resources required to service them, and returns the appropriate results post-authentication.

Output Validation:

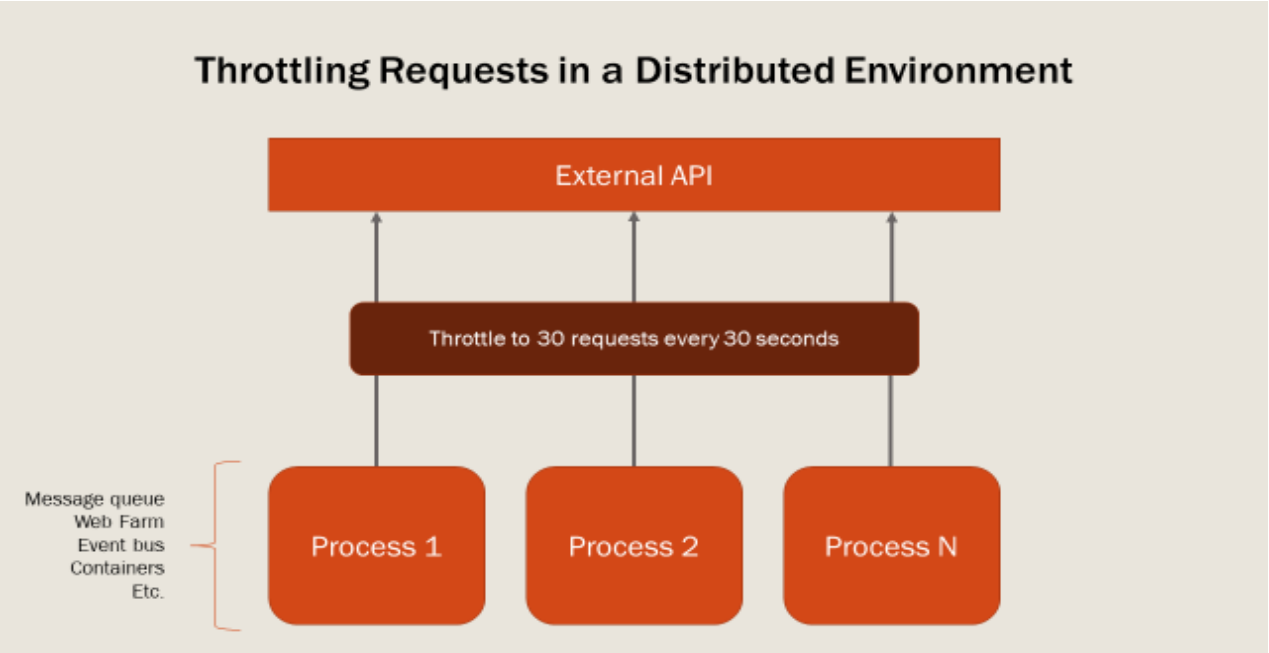
- **Output Header:**
  - ✓ ·Provide proper http response status code.
  - ✓ ·Provide proper content type, file type if any.
  - ✓ ·Provide cache status if any.
  - ✓ ·Authentication token should provide via response header.
  - ✓ ·Only string data is allowed for response header.
  - ✓ ·Provide content length if any.
  - ✓ ·Provide response date and time.
  - ✓ ·Follow request-response model described before.

- **Output Body:**
  - ✓ ·Avoid providing response status, code, message via response body
  - ✓ ·Use JSON best practices for JSON response body.
  - ✓ ·For single result, can use String, Boolean directly.
  - ✓ ·Provide proper JSON encode-decode before writing JSON Body.
  - ✓ ·Follow discussion on JSON described before.

Request Rate limit- Throttling: API throttling allows you to control the way an API is used. Throttling allows you to set permissions as to whether certain API calls are valid or not. Throttles indicate a temporary state, and are used to control the data that clients can access through an API. When a throttle is triggered, you can disconnect a user or just reduce the response rate. You can define a throttle at the application, API or user level.

As a developer, you have control over what applications and which users can use your APIs. Just like permissions, a combination of multiple throttles may be used on a single request. You can even have multiple levels of throttling based on the user. For example, you can restrict sensitive information from external developers, while giving access to the same for internal developers.

We need to make sure our APIs are running as efficiently as possible. Otherwise, everyone using your database will suffer from slow performance. Performance isn't the only reason to limit API requests, either. API limiting, which also known as rate is limiting, is an essential component of Internet security, as DoS attacks can tank a server with unlimited API requests. Rate limiting also helps make your API scalable. If your API blows up in popularity, there can be unexpected spikes in traffic, causing severe lag time.



Need of Throttling:

- ✓ APIs are a gateway to your backend resources and throttling offers you an extra layer of protection for those resources.
- ✓ You can deliver consistent applications by making sure that a single client is not suffocating your applications. Enhanced performance will drastically improve the end-user experience.
- ✓ You can control user authentication and access by rate limiting APIs at various levels—resource, API or application.
- ✓ You can design a robust API that can be leveraged by multiple groups based on their access level. Simplified API monitoring and maintenance can help reduce your costs.

Types of Throttling: Enterprises custom throttle their APIs based on the needs of their organization such as monetization, authentication, security, governance, performance, availability, etc. Here are some general throttling strategies adopted by the industry today to help you decide what your API needs:

- ❖ **Rate-Limit Throttling:** This is a simple throttle that enables the requests to pass through until a limit is reached for a time interval. A throttle may be incremented by a count of requests, size of a payload or it can be based on content; for example, a throttle can be based on order totals. This is also known as the API burst limit or the API peak limit.

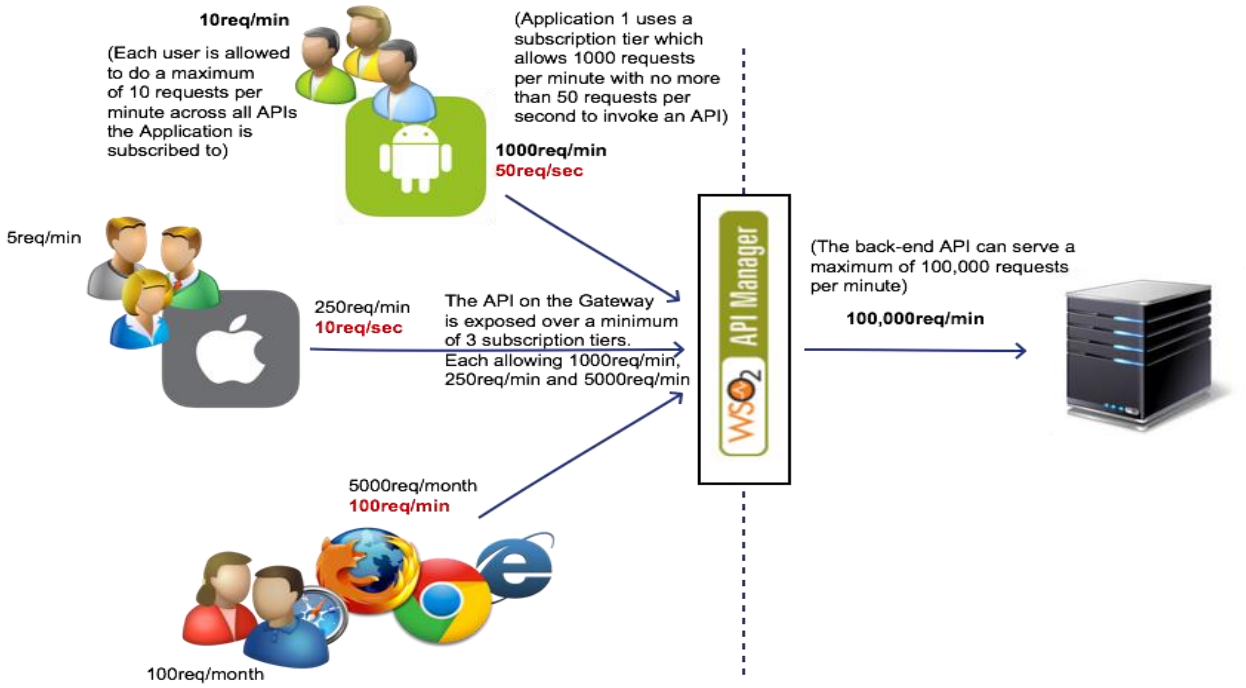
- **IP-level Throttling:** You can make your API accessible only to a certain list of whitelisted IP addresses. You can also limit the number of requests sent by a certain client IP.

- **Scope Limit Throttling:** Based on the classification of a user, you can restrict access to specific parts of the API—certain methods, functions or procedures. Implementing scope limits can help you leverage the same API across different departments in the organization.

- **Concurrent Connections Limit:** Sometimes your application cannot respond to more than a certain number of connections. In such cases, you need to limit the number of connections from a user/account to make sure that other users don't face a DoS (Denial of Service) error. This kind of throttling also helps secure your application against malicious cyber-attacks.

- **Resource-level Throttling (also referred to as Hard Throttling):** If a certain query returns a large result set, you can throttle the request so that your SQL engine limits the number of rows returned by using conditions attributes like TOP, SKIP, SQL_ATTR_MAX_ROWS, etc.

- **Tiers of Throttling:** Throttling can be applied at multiple levels in your organization:
    - ✓ API-level throttling
    - ✓ Application-level throttling
    - ✓ User-level throttling
    - ✓ Account-level throttling



Language	Platform	Library name	Library Sources
C#	ASP.NET	WebApiThrottle, MvcThrottle	Nuget package manager
PHP	Laravel	Laravel Karnel Default	Packagist
JS	Node/Express JS	express-rate-limit	NPM

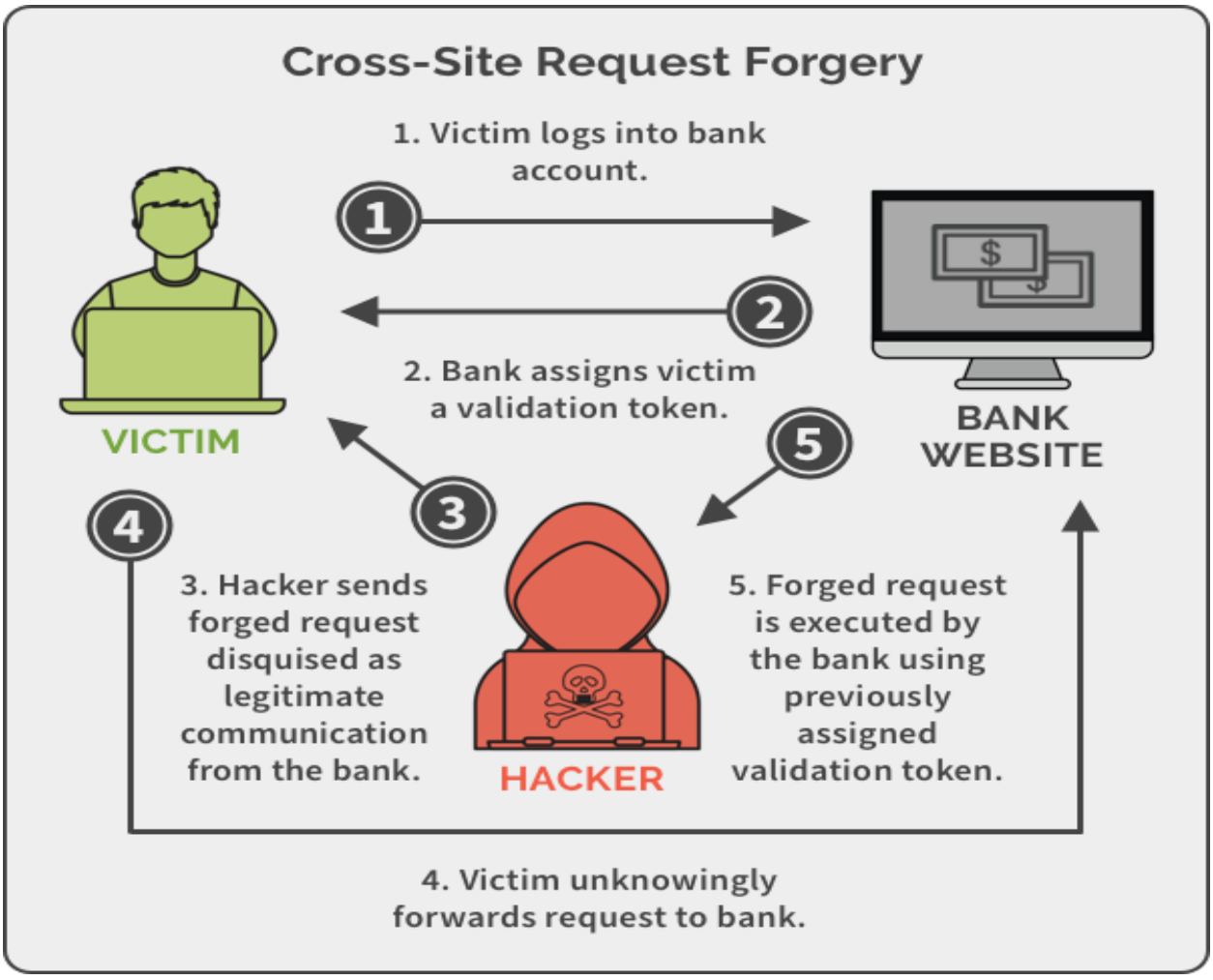
How to Throttle an API / Query: Throttling your API is an extremely sensitive process and it can have a huge impact on customer satisfaction, application performance and security. For that reason, I recommend you use our commercial enterprise solutions that have inherent support for throttling:

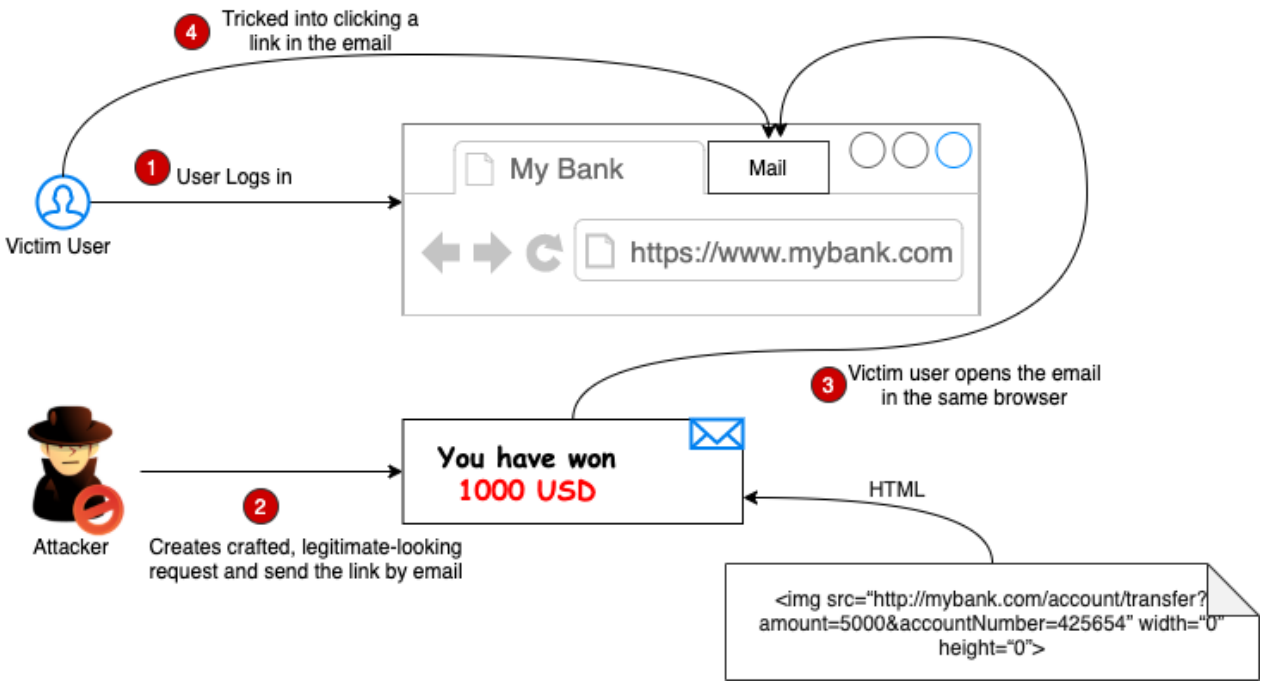
- **Hybrid Data Pipeline:** offers throttled ODBC, JDBC and OData APIs for popular databases such as IBM DB2, Oracle, SQL Server, MySQL, PostgreSQL, SAP Sybase, Hadoop Hive, Salesforce, Google Analytics and many more. To find out more details, you can check out our blog post on Hybrid Data Pipeline's throttling capabilities. Furthermore, OData has built-in functions such as $count, $top and $skip to filter the query results passed back to a client. In turn, pagination can help in avoiding throttling restrictions and possible performance degradation. You can learn more about *OData here*.

- **OpenAccess SDK:** In other cases where you have proprietary enterprise APIs, you can leverage our Open Access SDK to deploy a standard SQL interface—ODBC, JDBC, ADO.NET or OLE-DB. The SDK supports several throttling capabilities to meet your API throttling needs. The best part of Open Access is that it can easily integrate with your existing security and authentication systems, so it can serve as a true extra layer of protection for your enterprise APIs and the underlying backend resources.

CSRF/XSRF Protection: Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

- ✓ Use request-response header to pass CSRF token
- ✓ CSRF token should be unique for every session
- ✓ For self API CSRF token works well.


Cross-Site Request Forgery
1. Victim logs into bank account.
1
2
2. Bank assigns victim a validation token.
VICTIM
BANK WEBSITE
$
3
4
5
3. Hacker sends forged request disquised as legitimate communication from the bank.
HACKER
5. Forged request is executed by the bank using previously assigned validation token.
4. Victim unknowingly forwards request to bank.



Language	Platform	Library name	Library Sources
C#	ASP.NET	AntiCSRF	Nuget package manager
PHP	Laravel	Laravel Default	Packagist
JS	Node/Express JS	npm i csrf	NPM

User Agent Protection: User agent is a request header property, describe client identity like operating system, browser details, device details etc. Moreover, every web crawler like Google crawler, Facebook crawler has specific user-agent name.

- ✓ Using user agent we can prevent REST API from search engine indexing, social media sharing.
- ✓ Can stop subspecies request from who is hiding his identity.
- ✓ We can add user agent along with REST API usage history.

✓ We can add device/OS usage restriction.

Platform	Example User Agent Like
Android web browser	Mozilla/5.0 (Linux; Android 6.0.1; RedMi Note 5 Build/RB3N5C; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/68.0.3440.91 Mobile Safari/537.36
IOS web browser	Mozilla/5.0 (iPhone; CPU iPhone OS 12_3_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.1.1 Mobile/15E148 Safari/604.1
Windows	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.169 Safari/537.36
Mac	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15
Google BOT	Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)
Facebook BOT	facebookexternalhit/1.0 (+http://www.facebook.com/externalhit_uatext.php)

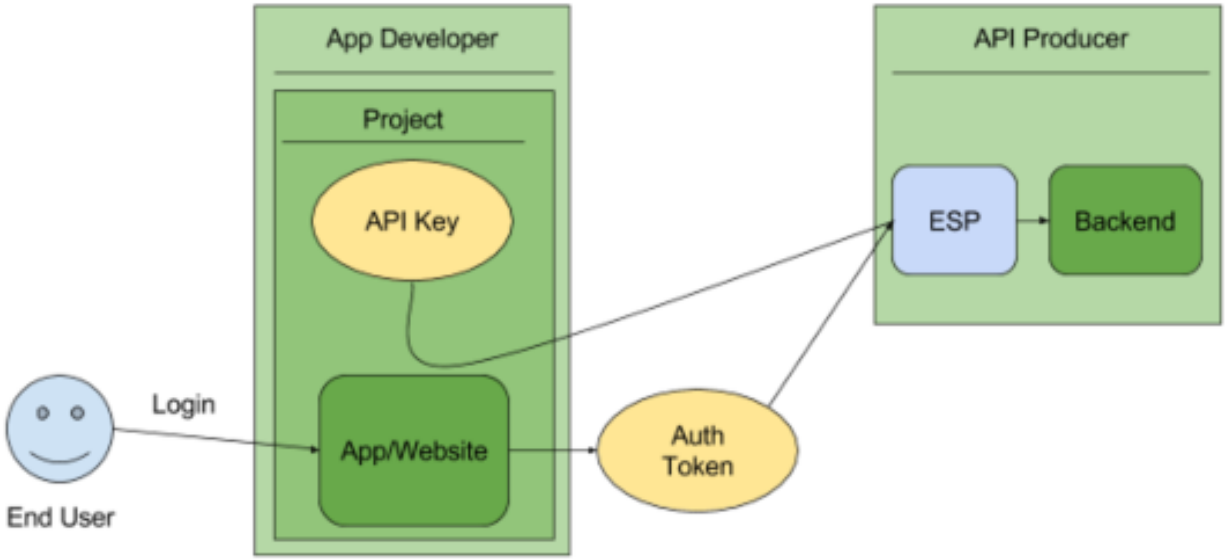
API Key: An API key has the following components, which you use to manage and use the key:

- **String:** The API key string is an encrypted string, for example, AIzaSyDaGmWKa4JsXZ-HjGw7ISLn_3namBGewQe. When you use an API key to authenticate, you always use the key's string. API keys do not have an associated JSON file.

- **ID:** The API key ID is used by Google Cloud administrative tools to uniquely identify the key. The key ID cannot be used to authenticate. The key ID can be found in the URL of the key's edit page in the Google Cloud console. You can also get the key ID by using the Google Cloud CLI to list the keys in your project.

- ✓ This is the most straightforward method and the easiest way for auth
- ✓ With this method, the sender places a **username:password/ ID / Keys** into the request header.
- ✓ The credentials are encoded and decode to ensure safe transmission.
- ✓ This method does not require cookies, session IDs, login pages, and other such specialty solutions.

```
Authorization: Basic bG9sOnNlY3VyZQ==
```



Bearer token: Bearer tokens are a much simpler way of making API requests, since they don't require cryptographic signing of each request. The tradeoff is that all API requests must be made over an HTTPS connection, since the request contains a plaintext token that could be used by anyone if it were intercepted.

It is the predominant type of access token used with OAuth 2.0. A Bearer Token is an opaque string, not intended to have any meaning to clients using it. Some servers will

issue tokens that are a short string of hexadecimal characters, while others may use structured tokens such as JSON Web Token.

Bearer Authentication: Bearer authentication (also called token authentication) is an HTTP authentication scheme that involves security tokens called bearer tokens. The name “Bearer authentication” can be understood as “give access to the bearer of this token.” The bearer token is a cryptic string, usually generated by the server in response to a login request. The client must send this token in the Authorization header when making requests to protected resources:
Authorization: Bearer

The Bearer authentication scheme was originally created as part of OAuth 2.0 in RFC 6750, but is sometimes also used on its own. Similarly, to Basic authentication, Bearer authentication should only be used over HTTPS (SSL).

use of bearer token: Bearer Token A security token with the property that any party in possession of the token (a “bearer”) can use the token in any way that any other party in possession of it can. Using a bearer token does not require a bearer to prove possession of cryptographic key material (proof-of-possession).

Access tokens are used in token-based authentication to allow an application to access an API. For example, a Calendar application needs access to a Calendar API in the cloud so that it can read the user’s scheduled events and create new events.

Once an application has received an access token, it will include that token as a credential when making API requests. To do so, it should transmit the access token to the API as a Bearer credential in an HTTP Authorization header.

How bearer token works: The Bearer Token is created for you by the Authentication server. When a user authenticates your application (client) the authentication server then goes and generates for you a Token. Bearer Tokens are the predominant type of access token used with OAuth 2.0. A Bearer token basically says “Give the bearer of this token access”.

The Bearer Token is normally some kind of opaque value created by the authentication server. It isn’t random; it is created based upon the user giving you access and the client your application getting access.

In order to access an API for example you need to use an Access Token. Access tokens are short lived (around an hour). You use the bearer token to get a new Access token. To get an access token you send the Authentication server this bearer token along with your client id. This way the server knows that the application using the bearer token is the same application that the bearer token was created for. Example: I can’t

just take a bearer token created for your application and use it with my application it won't work because it wasn't generated for me.

OAuth 1.0: In OAuth 1, there are two components to the access token, a public and private string. The private string is used when signing the request, and never sent across the wire.

OAuth 2.0: The most common way of accessing OAuth 2.0 APIs is using a "Bearer Token". This is a single string which acts as the authentication of the API request, sent in an HTTP "Authorization" header. The string is meaningless to clients using it, and may be of varying lengths.

JWT (JSON WEB TOKEN): A JSON web token (JWT) is *JSON Object* which is used to securely transfer information over the web (between two parties). It can be used for an authentication system and can also be used for information exchange. The token is mainly composed of *header, payload, signature*. These three parts are separated by dots(.). JWT defines the structure of information we are sending from one party to the another, and it comes in two forms – **Serialized, Deserialized**. The Serialized approach is mainly used to transfer the data through the network with each request and response. While the deserialized approach is used to read and write data to the web token.

- ✓ Compact and self-contained way for securely transmitting information between parties as a JSON object.
- ✓ Information can be verified and trusted because it is digitally signed.

Uses:

- ❖ **Authorization:** Allowing the user to access routes, services, and resources.
- ❖ **Information Exchange**: Way of securely transmitting information between parties.

JSON WEB TOKEN STRUCTURE:

- ❖ **Serialized:**
  - ✓ Signature
- ❖ **Deserialized:**
  - ✓ Header
  - ✓ Payload

Deserialized: JWT in the deserialized form contains only the header and the payload. Both of them are plain JSON objects.

- ✓ **Header:** A header in a JWT is mostly used to describe the cryptographic operations applied to the JWT like signing/decryption technique used on it. It can also contain the data about the media/content type of the information we are sending. This information is present as a JSON object then this JSON object is encoded to BASE64URL. The cryptographic operations in the header define whether the JWT is signed/unsigned or encrypted and are so then what algorithm techniques to use. A simple header of a JWT looks like the code below:

```
{
    "typ":"JWT",
    "alg":"HS256"
 }
```

The ‘alg’ and ‘typ’ are object key’s having different values and different functions like the ‘typ’ gives us the type of the header this information packet is, whereas the ‘alg’ tells us about the encryption algorithm used.
Note: HS256 and RS256 are the two main algorithms we make use of in the header section of a JWT.
Some JWT’s can also be created without a signature or encryption. Such a token is referred to as unsecured and its header should have the value of the alg object key assigned to as ‘none’.

```
{
    "alg":"none"
 }
```

- ✓ **Payload:** The payload is the part of the JWT where all the user data is actually added. This data is also referred to as the ‘claims’ of the JWT.This information is readable by anyone so it is always advised to not put any confidential information in here. This part generally contains user information. This information is present as a JSON object then this JSON object is encoded to BASE64URL. We can put as many claims as we want inside a payload, though unlike header, no claims are mandatory in a payload. The JWT with the payload will look something like this:

```
{
     "userId":"b07f85be-45da",
     "iss": "https://provider.domain.com/",
     "sub": "auth/some-hash-here",
     "exp": 153452683
 }
```

The above JWT contains *userId,iss,sub,and exp*. All these play a different role as userId is the ID of the user we are storing, ‘iss’ tells us about the issuer, ‘sub’ stands for subject, and ‘exp’ stands for expiration date.

Serialized: JWT in the serialized form represents a string of the following format:

`[header].[payload].[signature]`

all these three components make up the serialized JWT. We already know what header and payload are and what they are used for. Let's talk about signature.

- ✓ **Signature:** This is the third part of JWT and used to verify the authenticity of token. BASE64URL encoded header and payload are joined together with dot(.) and it is then hashed using the hashing algorithm defined in a header with a secret key. This signature is then appended to header and payload using dot(.) which forms our actual token *header.payload.signature*

  Syntax: *HASHINGALGO(base64UrlEncode(header)+ “.” +base64UrlEncode(payload),secret)*

So, all these above components together are what makes up a JWT. Now let’s see how our actual token will look like:

JWT Example:

header:

```
{
  "alg" : "HS256",

  "typ" : "JWT"
}
```

Payload:

```
{
  "id" : 123456789,

  "name" : "Joseph"
}
```

Secret: GeeksForGeeks

JSON Web Token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTIzNDU2Nzg5LCJuYW1lIjoiSm9zZXBoIn0.OpOSSw7e485LOP5PrzScxHb7SR6sAOMRckfFwi4rp7o

Advantage of Bearer tokens: The advantage is that it doesn't require complex libraries to make requests and is much simpler for both clients and servers to implement.

Disadvantage of Bearer tokens: The downside to Bearer tokens is that there is nothing preventing other apps from using a Bearer token if it can get access to it. This is a common criticism of OAuth 2.0, although most providers only use Bearer tokens anyway. Under normal circumstances, when applications properly protect the access tokens under their control, this is not a problem, although technically it is less secure. If your service requires a more secure approach, you can a different access token type that may meet your security requirements.

Language	Platform	Library name	Library Sources
C#	ASP.NET	JwtBearer, jose-jwt	Nuget package manager
PHP	Laravel	firebase / php-jwt	GitHub
JS	Node/Express JS	npm i jsonwebtoken	NPM