**The Architectures Of Information Security**

***Resources from Microsoft Copilot***

Talking about information security, we talk about different kinds of information encryption and decryption methods. This article we will explore the custom or traditional ways of the architechture behind. For web browsing, not talking about ways of information encrytion, we will start talking about the architechture behind of how web browsing handles information security. We will first discuss what “**Private**” and “**Public**” keys is and how it is related to the “**Https**” protocol. Then we will jump into the coding part, and discuss ways and methods of how programmers or developers handles information security during coding. After we will spare some time talking about how **mobile apps** handles it’s own apps security.

We all know what internet is. Internet for short is **W**orld **W**ide **W**eb. We use different kinds of web browsers to access the internet. The contents we access from or submit to must be base on two points. Web security is the way of how to secure these 2 points of connections, so as to avoid anoymous of accessing information between theses 2 points. During the early stage of WWW, we usually access the web site by using “**HTTP**” protocol. When we want to access for instance google home page, we must type in the url: “[http://www.google.com](http://www.google.com/)” in order to access. After years of time changing, if we want to access to google home page, we need to use the protocol “**HTTPS**”, this also becomes a standard for browsing web pages. What is the additional “**S**” character means, it means “**Secure**”. How this is done is base on the terms, “Private” and “Public” Keys. Below briefly discuss these 2 terms:

**What is private and public keys?**

In the simplest terms, private and public keys are like secret code pairs used to secure digital communication and data. They are a part of asymmetric encryption, where two different keys are used: one for encryption and one for decryption.

### Public Key:

* Shared with anyone: This key can be freely distributed.
* Used for encryption: People use your public key to encrypt information they send to you.
* Verification: It's also used to verify signatures made with the corresponding private key.

### Private Key:

* Kept secret: This key should never be shared.
* Used for decryption: You use your private key to decrypt information that others have encrypted with your public key.
* Signatures: You can use it to digitally sign documents, proving the information came from you.

#### A Real-World Analogy:

Imagine a public key as your email address (anyone can send you emails) and your private key as your email password (only you have access to read those emails).

Using both, you can ensure secure communication, authenticity, and integrity in digital interactions.

**By using public and private key terminology, please explain how web browsing process works.**

We will walk through a typical web browsing process involving public and private keys, focusing on the secure HTTPS connection.

### 1. **Requesting a Webpage:**

When you enter a URL into your browser (like https://example.com), your browser sends a request to the web server hosting that site.

### 2. **Server Responds with a Certificate:**

The server responds by sending its public key certificate. This certificate includes the server’s public key and is issued by a trusted Certificate Authority (CA), ensuring the authenticity of the server.

### 3. **Browser Verifies Certificate:**

Your browser verifies the certificate against a list of trusted CAs. If the certificate is valid and trusted, the browser proceeds. If not, you might see a security warning.

### 4. **Key Exchange:**

* Session Key Generation: Your browser generates a session key (a symmetric key for fast encryption).
* Encrypting with Public Key: The session key is encrypted using the server’s public key (from the certificate) and sent back to the server.

### 5. **Server Decrypts Session Key:**

The server uses its private key to decrypt the session key. Now, both the browser and the server share the same session key, which they will use for encryption.

### 6. **Secure Data Transmission:**

Using the shared session key, the browser and the server can now securely exchange information. This ensures that any data sent between them is encrypted and protected from eavesdropping.

### 7. **Browsing Continues:**

You can now securely browse the website, knowing your data (like login credentials, personal information, etc.) is protected by encryption.

Now that we have the basic understanding of how web browsing works, as said; programmers and developers also needs to program the code in order to adapt the changes.

1. Generating **Session Keys** on the client program, for example by using **JSON WEB TOKEN** or other forms of token generation.
2. Send and encrypt the **Session keys** together when making a **HTTPS Request** to the web server in the “**Authorization**” header field.
3. On the Server Side retrieve the **Session keys** from the HTTPS Request method from the “**Authorization**” header field.
4. Now the Server and Client share the same “**Session Keys**” and can communicates in between.

How mobile apps works is a different approach. We don’t have a secure connection or linkage between the Apps and the Server, there is no such things like “**HTTPS**” protocol. How this could be done is mainly base on how you program or handle data from your code. For instance, if you are working on a banking app, I assume you need to deal with transaction data between your app and the banking server. As mentioned before we can make use of “**Session Keys**” generate by the client app (banking app) and encrypted it to the “**HTTPS Request**” method. Why “**HTTPS Request**” method? as discussed this metod is usually works for the **web**. Yes! But that is the only way to secure your data. When your mobile app needs to deal with data or transaction from the banking server, In between the banking server (you can think of the server consists of database, images storage, videos storages...etc) and your client app, you need to setup a **REST FUL API Server**. The RESTFUL Api Server acts a middleware Server in between the client app and the server. For web browsing we use **Web server**, for data transaction we use **RestFul Api Server**. Just like web server, in order to connect to the Api Server, you also need to use the “**HTTPS**” protocol. The same encryption methodologies can be used for securing the data like the Web. Restful API Server also needs to install a **SSL certificate,** usually this certificate is issued by the CA (Certificate Authority), for example in Hong Kong one of the Certificate Authority is the **Hong Kong Post Office**.

Not only we can secure data transaction in between the app and the server. The app you build also is being digital signed and verify in order to be downloaded from the “Google Play Store” or “App Store”. The android app for instance needs to be digital signed when build and will generate an digital signed APK File, Google Play Store only accept digital signed APK file in order for user to download from their Google Play platform.

**The process of installing the certificate key in the Web server**

Installing an SSL/TLS certificate on a web server is a crucial step in securing your website. Here’s a detailed process to guide you through:

1. Generate a Certificate Signing Request (CSR):

This is the first step where you create a CSR from your server. The CSR includes your public key and information about your organization.

Using OpenSSL (for example) for the command prompt please type in:

openssl req -new -newkey rsa:2048 -nodes -keyout yourdomain.key -out yourdomain.csr

This command generates a private key (yourdomain.key) and a CSR (yourdomain.csr).

1. Submit CSR to a Certificate Authority (CA):

Submit the CSR to a trusted CA. They will verify your information and issue an SSL certificate. This may take some time, depending on the level of verification required.

1. **Download the Certificate:**

**Once the CA has issued your certificate, you’ll typically receive several files:**

* Your primary SSL certificate (yourdomain.crt).
* Intermediate certificate(s), if applicable.
* CA root certificate.

1. Install the Certificate on Your Web Server:

The installation process varies depending on the server software you’re using. Here’s a general guide for Apache and Nginx:

#### **For Apache:**

1. Locate Configuration File:
   * Typically found in /etc/httpd/conf/httpd.conf or /etc/apache2/sites-available/default-ssl.conf.
2. Update Configuration File:
   * Add the following lines, replacing with your actual paths:
     + SSLCertificateFile /path/to/yourdomain.crt
     + SSLCertificateKeyFile /path/to/yourdomain.key
     + SSLCertificateChainFile /path/to/intermediate.crt
3. **Restart Apache:**

**sudo service apache2 restart**

#### **For Nginx:**

1. Locate Configuration File:
   * Typically found in /etc/nginx/nginx.conf or /etc/nginx/sites-available/default.
2. Update Configuration File:

Add the following lines, replacing with your actual paths:

**server {**

**listen 443 ssl;**

**server\_name yourdomain.com;**

**ssl\_certificate /path/to/yourdomain.crt;**

**ssl\_certificate\_key /path/to/yourdomain.key;**

**ssl\_trusted\_certificate /path/to/intermediate.crt;**

**}**

1. Restart Nginx:

sudo service nginx restart

1. Verify Installation:

After installation, use online tools like SSL Labs or Postman to verify that your SSL certificate is installed correctly and is working as expected.

1. Update URL to HTTPS:

Ensure that all links and resources on your website use HTTPS. You can also set up redirects from HTTP to HTTPS to ensure all traffic is secured.

### **Summary:**

1. Generate CSR.
2. Submit CSR to CA.
3. Download certificate.
4. Install certificate on web server.
5. Verify installation.
6. Update URL to HTTPS.

By following these steps, you'll have your SSL/TLS certificate properly installed, ensuring secure communication between your server and users.

**The process of installing the certificate key in the Restfull API server**

Installing an SSL/TLS certificate on a RESTful API server is crucial for ensuring secure communication. Here’s how you can do it:

1. **Generate a Certificate Signing Request (CSR):**

Similar to web servers, you’ll start by generating a CSR from your API server. In the Command prompt please type in:

openssl req -new -newkey rsa:2048 -nodes -keyout api.yourdomain.key -out api.yourdomain.csr

1. **Submit CSR to a Certificate Authority (CA):**

Submit the CSR to a trusted CA. The CA will verify your details and issue an SSL certificate.

1. **Download the Certificate:**

Once issued, download your SSL certificate and any intermediate certificates from the CA.

1. Install the Certificate on Your API Server:

The steps vary depending on the software you’re using for your API server. Here are examples for popular frameworks:

For Express (Node.js):

1. Install Dependencies:  
    npm install express https
2. Set Up HTTPS in Your Server Code:

const express = require('express');

const https = require('https');

const fs = require('fs');

const app = express();

const options = {

key: fs.readFileSync('/path/to/api.yourdomain.key'),

cert: fs.readFileSync('/path/to/api.yourdomain.crt'),

ca: fs.readFileSync('/path/to/intermediate.crt')

};

https.createServer(options, app).listen(443, () => {

console.log('API server running on port 443');

});

// Define your API routes here

app.get('/api/hello', (req, res) => {

res.send('Hello, secure world!');

});

**For Flask (Python):**

1. Install Dependencies:

pip install Flask pyOpenSSL

1. Set Up HTTPS in Your Application Code:

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/api/hello')

def hello():

return 'Hello, secure world!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=443, ssl\_context=('/path/to/api.yourdomain.crt', '/path/to/api.yourdomain.key'))

1. Verify Installation:

User tools like SSL Labs or Postman to verify that your API server is properly configured and accepting secure connections.

1. Update Clients:

Ensure that any clients consuming your API endpoints use HTTPS URLs to connect securely.

### **Summary:**

1. Generate CSR.
2. Submit CSR to CA.
3. Download certificate.
4. Install certificate on your API server.
5. Verify installation.
6. Update clients to use HTTPS.

By following these steps, you'll ensure your RESTful API server is secure, protecting data transmitted between clients and your server.