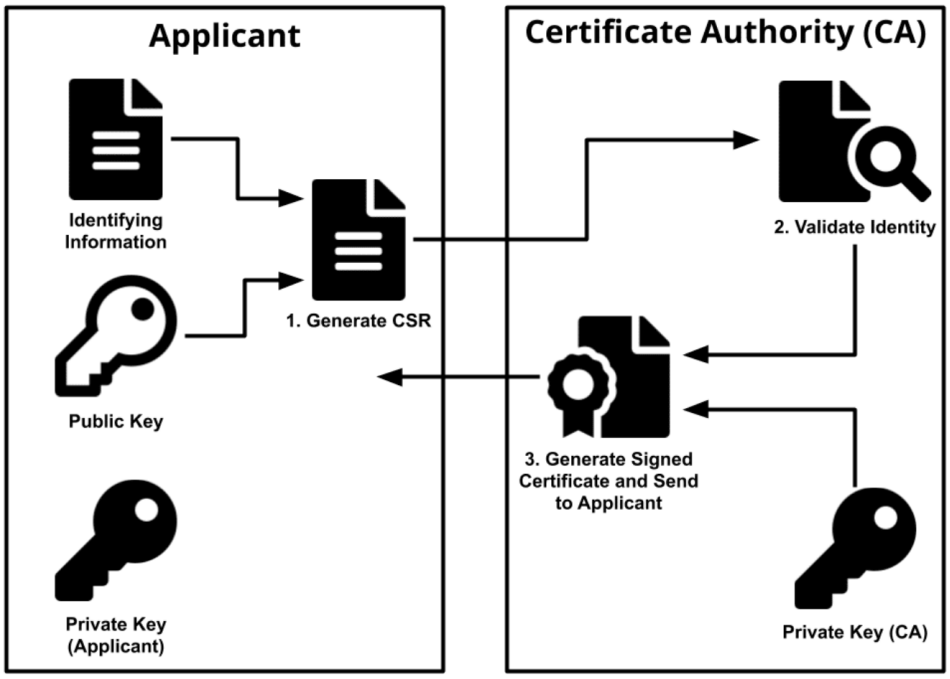
- SSL/TLS
  - Certificate
  - Certificate Authority (CA)
  - Certificate Signing Request (CSR)
  - Certificate Validation
  - Key formats
  - Content
  - Keystore / Truststore
  - JWK / JWT
- Single Sign On (SSO)
- OAuth 2.0
  - Authorization Code Flow
  - Implicit Flow
  - Client Credentials Flow
  - OAuth vs SSO
- CSRF
- CORS
- XSS

SSL/TLS

SSL (Secure Sockets Layer) and its successor, **TLS** (Transport Layer Security), are protocols for establishing authenticated and encrypted links between networked computers. Although the SSL protocol was deprecated with the release of TLS 1.0 in 1999, it is still common to refer to these related technologies as “SSL” or “SSL/TLS.” The most current version is TLS 1.3, defined in RFC 8446 (August 2018).



TL;DR

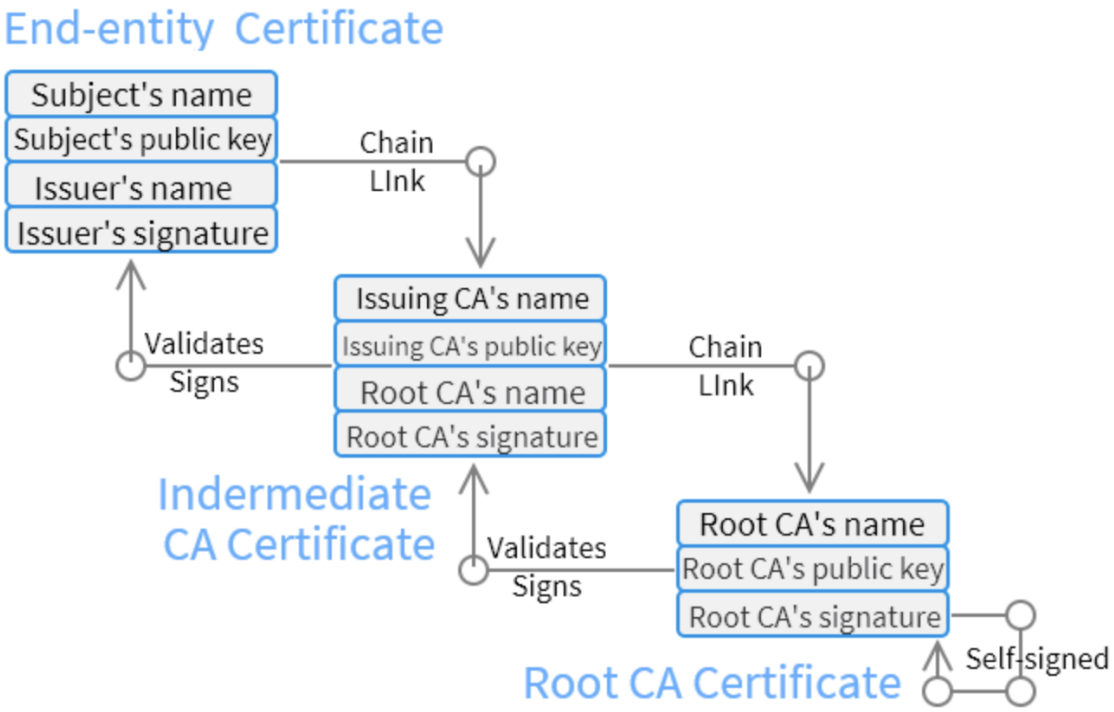
- the private key (that is always kept secret) is used to apply the digital signature to the software/information
- public key is used to verify the identity of the entity that has signed an SSL certificate.

Certificate

A certificate verifies that an entity is the owner of a particular public key.

The **certificate**, in addition to containing the public key, contains additional information such as issuer, what the certificate is supposed to be used for, and other types of metadata.

The use of one or other of these storage formats is really an issue about how your application will store encrypted private keys locally. The vendor who sells you your certificate will never see the private key so he doesn't care what format you use. You send him (the vendor/CA) a PKCS#10 **certificate request** (containing the public key and signed using the private key, but NOT containing the private key) and he sends you back a certificate (which you can store in JKS or in the PKCS#12 file or both, or anywhere else that takes your fancy).



Certificate Authority (CA)

https://www.ssl.com/faqs/what-is-a-certificate-authority/

CA is an entity that stores, signs, and issues digital certificates.

The root CA has a self-signed certificate. Each subordinate CA has a certificate that is signed by the next highest CA in the hierarchy. A certificate chain is the certificate of a particular CA, plus the certificates of any higher CAs up through the root CA.

Certificate Signing Request (CSR)

The **CA** will use the data from the CSR to build your SSL Certificate. The key pieces of information include the following:

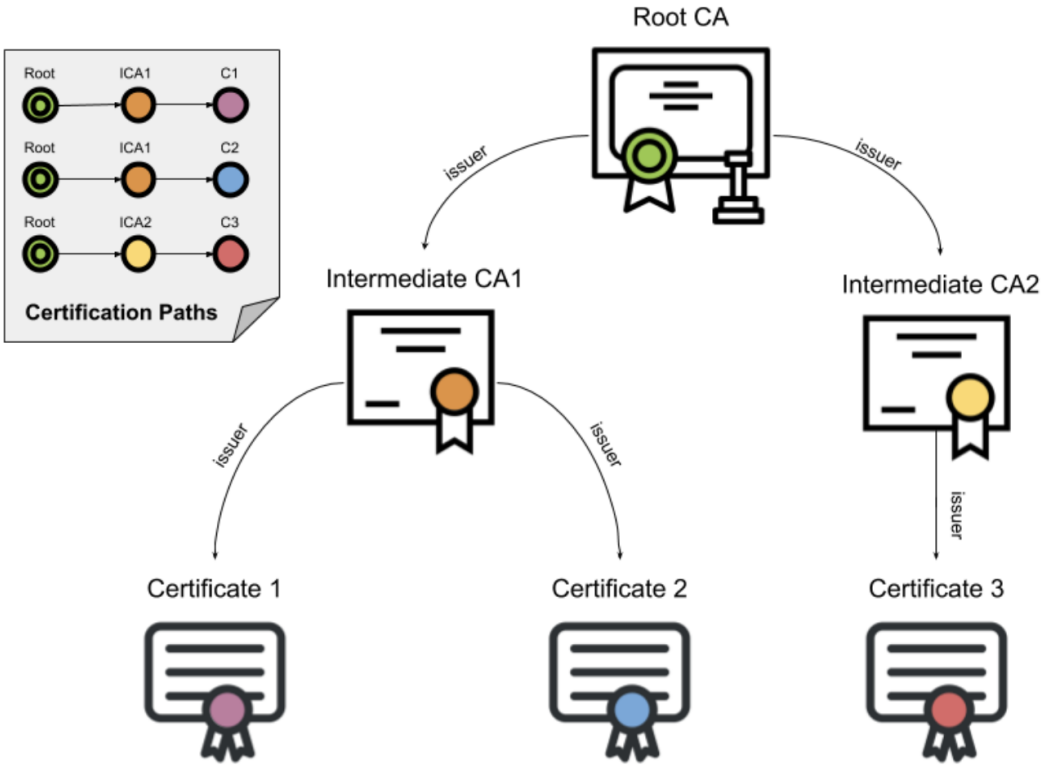
- Common Name (CN)
- Organization (O)
- Organizational Unit (OU)
- City/Locality (L)
- State/County/Region (S)
- Country (C)
- Email Address
- **Public Key**

The CSR itself is usually created in a Base-64 based **PEM** format:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIDVDCCAr0CAQAweTEeMBwGA1UEAxMVd3d3Lmpvc2VwaGNoYXBtYW4uY29tMQ8w
DQYDVQQLEwZEZXNpZ24xFjAUBgNVBAoTDUpvc2VwaENoYXBtYW4xEjAQBgNVBAcT
…
CU1haWRzdG9uZTENMAsGA1UECBMES2VudDELMAkGA1UEBhMCR0IwgZ8wDQYJKoZI
ZSZTusPFTLKaqValdnS9Uw+6Vq7/I4ouDA8QBIuaTFtPOp+8wEGBHQ==
-----END NEW CERTIFICATE REQUEST-----
```

Certificate Validation

https://www.ssl.com/article/browsers-and-certificate-validation/



Browsers are shipped with a **built-in list of trusted roots**. (These are roots from CAs who have passed the browser's stringent criteria for inclusion.) To verify a certificate, a browser will obtain a **sequence of certificates**, each one having signed the next certificate in the sequence, connecting the signing CA's root to the server's certificate.

This sequence of certificates is called a **certification path**. The path's root is called a **trust anchor** and the server's certificate is called the leaf or end entity certificate.

After a candidate certification path is constructed, browsers validate it using information contained in the certificates. **A path is valid if browsers can cryptographically prove that, starting from a certificate directly signed by a trust anchor, each certificate's corresponding private key was used to issue the next one in the path, all the way down to the leaf certificate.**

The following sections show the sequence of checks that browsers perform.

- The browser verifies the certificate's integrity - the signature on the certificate can be verified using normal public key cryptography
- The browser verifies the certificate's validity
- The browser checks the certificate's revocation status
- The browser verifies the issuer

- The browser checks name constraints
- The browser checks policy constraints
- The browser checks basic constraints (a.k.a. path length)
- The browser verifies key usage - browsers reject certificates violating their key usage constraints, such as encountering a server certificate with a key meant only for CRL signing.
- The browser continues to process all remaining critical extensions

Key formats

In cryptography, **X.509** is a standard defining the format of **public key certificates**.

There are several commonly used filename extensions for X.509 certificates. **Unfortunately, some of these extensions are also used for other data such as private keys.**

- **.pem** – (Privacy-enhanced Electronic Mail) Base64 encoded DER certificate, enclosed between "-----BEGIN CERTIFICATE-----" and "-----END CERTIFICATE-----"
- **.cer, .crt, .der** – usually in binary DER form, but Base64-encoded certificates are common too (see .pem above)
- **.p7b, .p7c** – PKCS#7 SignedData structure without data, just certificate(s) or CRL(s)
- **.p12** – PKCS#12, may contain certificate(s) (public) and private keys (password protected)
- **.pfx** – PFX, predecessor of PKCS#12 (usually contains data in PKCS#12 format, e.g., with PFX files generated in IIS)
- **PKCS#7** is a standard for signing or encrypting (officially called "enveloping") data. Since the certificate is needed to verify signed data, it is possible to include them in the SignedData structure. A .P7C file is a degenerated SignedData structure, without any data to sign.
- **PKCS#12** evolved from the personal information exchange (PFX) standard and is used to exchange public and private objects in a single file.

A Java KeyStore (**JKS**) is a repository of security certificates – either authorization certificates or public key certificates – plus corresponding private keys, used for instance in TLS encryption.

Content

https://docs.oracle.com/cd/E19424-01/820-4811/gdzdx/index.html

Every **X.509 certificate** consists of the following sections:

A data section, including the following information.

- The version number of the X.509 standard supported by the certificate.
- The certificate's serial number. Every certificate issued by a CA has a serial number that is unique among the certificates issued by that CA.
- **Information about the user's public key, including the algorithm used and a representation of the key itself.**
- The DN of the CA that issued the certificate.
- The period during which the certificate is valid (for example, between 1:00 p.m. on November 15, 2003 and 1:00 p.m. November 15, 2004).
- The DN of the certificate subject (for example, in a client SSL certificate this would be the user's DN), also called the subject name.
- Optional certificate extensions, which may provide additional data used by the client or server. For example, the certificate type extension indicates the type of certificate—that is, whether it is a client SSL certificate, a server SSL certificate, a certificate for signing email, and so on. Certificate extensions can also be used for a variety of other purposes.

A signature section includes the following information:

- The cryptographic algorithm, or cipher, used by the issuing CA to create its own digital signature.
- **The CA's digital signature, obtained by hashing all of the data in the certificate together and encrypting it with the CA's private key.**

Keystore / Truststore

In most cases, we use a **keystore** and a **truststore** when our application needs to communicate over SSL/TLS.

Usually, these are password-protected files that sit on the same file system as our running application. The default format used for these files was JKS until Java 8.

Since Java 9, the default keystore format is PKCS12. The biggest difference between JKS and PKCS12 is that JKS is a format specific to Java, while PKCS12 is a standardized and language-neutral way of storing encrypted private keys and certificates.

A Java keystore stores private key entries, certificates with public keys, or just secret keys that we may use for various cryptographic purposes. It stores each by an alias for ease of lookup.

Generally speaking, keystores hold keys that our application owns, which we can use to prove the integrity of a message and the authenticity of the sender, say by signing payloads.

Usually, we'll use a keystore when we're a server and want to use HTTPS. **During an SSL handshake, the server looks up the private key from the keystore, and presents its corresponding public key and certificate to the client.**

Similarly, if the client also needs to authenticate itself, a situation called **mutual authentication**, then the client also has a keystore and also presents its public key and certificate.

There's no default keystore, so if we want to use an encrypted channel, we'll have to set **javax.net.ssl.keyStore** and **javax.net.ssl.keyStorePassword**. If our keystore format is different than the default, we could use **javax.net.ssl.keyStoreType** to customize it.

Of course, we can use these keys to service other needs as well. **Private keys can sign or decrypt data, and public keys can verify or encrypt data**. Secret keys can perform these functions as well. A keystore is a place that we can hold onto these keys.

—

A **truststore** is the opposite. While a keystore typically holds onto certificates that identify us, a truststore holds onto certificates that **identify others**.

In Java, we use it to trust the third party we're about to communicate with.

Take our earlier example. If a client talks to a Java-based server over HTTPS, the server will look up the associated key from its keystore and present the public key and certificate to the client.

We, the client, then look up the associated certificate in our truststore. If the certificate or Certificate Authorities presented by the external server isn't in our truststore, we'll get an SSLHandshakeException, and the connection won't be set up successfully.

Java has bundled a truststore called **cacerts**, and it resides in the **$JAVA_HOME/jre/lib/security** directory.

It contains default, trusted Certificate Authorities:

```
$ keytool -list -keystore cacerts
Enter keystore password:
Keystore type: JKS
Keystore provider: SUN
Your keystore contains 92 entries
verisignclass2g2ca [jdk], 2018-06-13, trustedCertEntry,
Certificate fingerprint (SHA1): B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
```

We can override the default truststore location via the **javax.net.ssl.trustStore** property. Similarly, we can set **javax.net.ssl.trustStorePassword** and **javax.net.ssl.trustStoreType** to specify the truststore's password and type.

JWK / JWT

JWT - JSON Web Token (RFC 7519) - is an open standard, useful for Authorization and Information Exchange. JWTs represent “claims”. JWTs are signed using a secret or a public/private key pair.

JWT has the following structure.

- Header
- Payload
- Signature

These pieces are encoded and delimited by periods like in the example below:

Example: xxxxx.yyyyy.zzzzz

Typically the JWT encoded form is used in the Authorization HTTP request header. The value using the “Bearer schema” would look like this and have the JWT information as below:

Authorization: Bearer <encoded JWT>

—

A JSON Web Key (**JWK**), an IETF standard (RFC 7517), is a JSON data structure that represents a cryptographic key.

JSON Web Key (JWK) provides a **mechanism to distribute the public keys that can be used to verify JWTs**.

Data and Signature Sections of a Certificate in Human-Readable Format

Certificate:
Data:
  Version: v3 (0x2)
  Serial Number: 3 (0x3)
  Signature Algorithm: PKCS #1 MD5 With RSA Encryption
  Issuer: OU=Certificate Authority, O=Example Industry, C=US
  Validity:
   Not Before: Fri Oct 17 18:36:25 2003
   Not After: Sun Oct 17 18:36:25 2004
  Subject: CN=Jane Doe, OU=Finance, O=Example Industry, C=US
  Subject Public Key Info:
   Algorithm: PKCS #1 RSA Encryption
   Public Key:
     Modulus:
       00:ca:fa:79:98:8f:19:f8:d7:de:e4:49:80:48:e6:2a:2a:86:
       ed:27:40:4d:86:b3:05:c0:01:bb:50:15:c9:de:dc:85:19:22:
       43:7d:45:6d:71:4e:17:3d:f0:36:4b:5b:7f:a8:51:a3:a1:00:
       98:ce:7f:47:50:2c:93:36:7c:01:6e:cb:89:06:41:72:b5:e9:
       73:49:38:76:ef:b6:8f:ac:49:bb:63:0f:9b:ff:16:2a:e3:0e:
       9d:3b:af:ce:9a:3e:48:65:de:96:61:d5:0a:11:2a:a2:80:b0:
       7d:d8:99:cb:0c:99:34:c9:ab:25:06:a8:31:ad:8c:4b:aa:54:
       91:f4:15
     Public Exponent: 65537 (0x10001)
  Extensions:
   Identifier: Certificate Type
     Critical: no
     Certified Usage:
     SSL Client
   Identifier: Authority Key Identifier
     Critical: no
     Key Identifier:
       f2:f2:06:59:90:18:47:51:f5:89:33:5a:31:7a:e6:5c:fb:36:
       26:c9
  Signature:
   Algorithm: PKCS #1 MD5 With RSA Encryption
  Signature:
6d:23:af:f3:d3:b6:7a:df:90:df:cd:7e:18:6c:01:69:8e:54:65:fc:06:
30:43:34:d1:63:1f:06:7d:c3:40:a8:2a:82:c1:a4:83:2a:fb:2e:8f:fb:
f0:6d:ff:75:a3:78:f7:52:47:46:62:97:1d:d9:c6:11:0a:02:a2:e0:cc:
2a:75:6c:8b:b6:9b:87:00:7d:7c:84:76:79:ba:f8:b4:d2:62:58:c3:c5:
b6:c1:43:ac:63:44:42:fd:af:c8:0f:2f:38:85:6d:d6:59:e8:41:42:a5:
4a:e5:26:38:ff:32:78:a1:38:f1:ed:dc:0d:31:d1:b0:6d:67:e9:46:a8:
d:c4

Certificate In the 64-Byte Encoded Form Interpreted by Software

-----BEGIN CERTIFICATE-----
MIICKzCCAZSgAwIBAgIBAzANBgkqhkiG9w0BAQQFADA3MQswCQYDVQQGEwJVUzER
MA8GA1UEChMITmV0c2NhcGUxFTATBgNVBAsTDFN1cHJpeWEncyBDQTAeFw05NzEw
MTgwMTM2MjVaFw05OTEwMTgwMTM2MjVaMEgxCzAJBgNVBAYTAlVTMREwDwYDVQQK
EwhOZXRzY2FwZTENMAsGA1UECxMEUHViczEXMBUGA1UEAxMOU3Vwcml5YSBTaGV0
dHkwgZ8wDQYJKoZIhvcNAQEFBQADgY0AMIGJAoGBAMr6eZiPGfjX3uRJgEjmKiqG
7SdATYazBcABu1AVyd7chRkiQ31FbXFOGD3wNktbf6hRo6EAmM5/R1AskzZ8AW7L
iQZBcrXpc0k4du+2Q6xJu2MPm/8WKuMOnTuvzpo+SGXelmHVChEqooCwfdiZywyZ
NMmrJgaoMa2MS6pUkfQVAgMBAAGjNjA0MBEGCWCGSAGG+EIBAQQEAwIAgDAfBgNV
HSMEGDAWgBTy8gZZkBhHUfWJM1oxeuZc+zYmyTANBgkqhkiG9w0BAQQFAAOBgQBt
l6/z07Z635DfzX4XbAFpjlRl/AYwQzTSYx8GfcNAqCqCwaSDKvsuj/vwbf91o3j3
UkdGYpcd2cYRCgKi4MwqdWyLtpuHAH18hHZ5uvi00mJYw8W2wUOsY0RC/a/IDy84
hW3WWehBUqVK5SY4/zJ4oTjx7dwNMdGwbWfpRqjd1A==
-----END CERTIFICATE-----

Single Sign On (SSO)

SSO is a high-level concept used by a wide range of service providers (sometimes with confusing differences).

SSO is an authentication / authorization flow through which a user can log into multiple services using the same credentials.

For instance, at your company, you might want to use one set of credentials to access:

- Your internal company website.
- Your Salesforce account.
- Your Atlassian account.
- etc.

Instead of making each employee at your company create different accounts for each of those services they use all the time, you can instead create a single account for each employee that grants them access to all of your company services.

Benefits:

- Users have only a single account and password to remember which gets them into all of their services. This typically makes account management / user data storage simpler for employees, as there's less duplicate data floating around between systems.
- You have a lot more control over user accounts and user data: you retain this information and interface with providers using the *Security Assertion Markup Language* (SAML).

Essentially what happens is this:

- You store your user accounts in your own internal system.
- You create a SAML-compatible interface to talk to various SAML providers (like Salesforce, Microsoft, etc.).
- Your users can then authenticate just once, and log into any of the compatible providers.

OAuth 2.0

Authorization Code Flow

- User must authenticate and return a **code** to the API consumer (called the "Client").
- The "client" of the API (usually your web server) exchanges **the code** obtained in #1 for an access_token, authenticating itself with a **client_id** and **client_secret**
- It then can call the API with the access_token.

What's so good about this scheme: if anyone ever intercepts the code, it is useless without the specific client_id + client_secret pair of the client that this code was granted for.

Implicit Flow

It goes like Authorization Code flow, but step 2 is omitted. So after user authentication, an **access_token** is returned directly, that you can use to access the resource. **The API doesn't know who is calling that API**. Anyone with the **access_token** can access a user data, whereas in the previous example only the web app would.

Client Credentials Flow

This type of grant is commonly used for **server-to-server interactions** that must run in the background, without immediate interaction with a user.

In the client credentials flow, **permissions are granted directly to the application itself by an administrator**. When the app presents a token to a resource, the resource enforces that the app itself has authorization to perform an action since **there is no user involved in the authentication**.

Example:

- Server-1 wants to access some resource located on Server-2
- Server-1 request an **auth-server** which issues a token containing something like:


```
{
   "access_token" :
"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzY29wZSI6WyJhbGwiXSwianRpIjoiYWRkMGIyNjUtMDNiMS00ODgwLWE0MDYtYzU5ODd
mNzZhMTU4IiwiY2xpZW50X2lkIjoiYXpvbiJ9.AbxkCcIGZLvJW34-LNYABkODyIBPVsnwr7P9u5BLGOsP7gg7tSdPacFWqaReL6Ez7jKbHFb
CB8HYcHaDQhvsEjcjwqgGbaJZlj0_u_I2CnlAnXU8vSp97NucBr582lms4ixDYgsEC-WO_yn7G3R64aRmisvOOfO5IQN43rv02Ku19wde86M
g1wa0J9FXNkZt9xyuIS2ttWLGVsER31Qa2VE2wXqyLjFnTGvZs3QpvnWxmxFRXGBQ2E8ljHRtHzQC0YYQ9MxPhDJlgZOPX_novpbD_c1
Lf3qIctc0VELSB2k_5hlh8I_Pn7wGmx8twCFdWE9x-2iJczPU7I93RyED9w",
   "scope" : "all",
   "jti" : "add0b265-03b1-4880-a406-c5987f76a158",
   "token_type" : "bearer"
}
```


- Server-1 caches the token (optional) and sends it in the Authorization header when making request to the Server-2
- Server-2 validates the token received from Server-1 using public key of the **auth-server** (for example, obtained from the jwks-endpoint)
- Server-2 checks if the token scope matches the requested resource scope

OAuth vs SSO

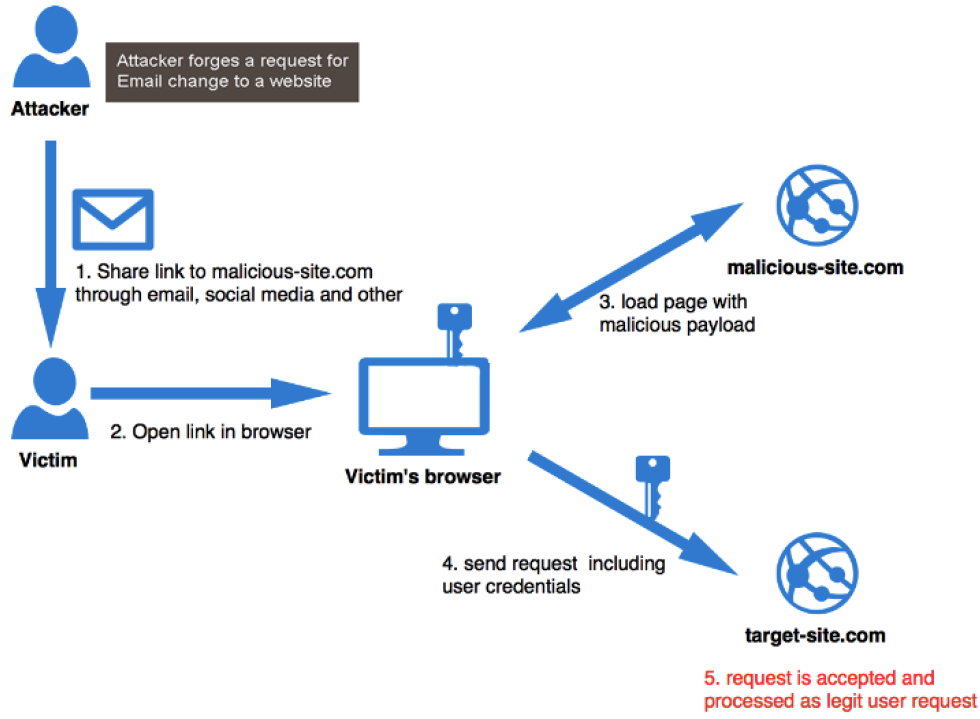
If you want your users to be able to use a single account / credential to log into many services directly, use SSO.

If you want your users to have accounts on many different services, and selectively grant access to various services, use OAuth.

CSRF

Cross-Site Request Forgery - an attack that forces the end user to make unwanted calls to the web application servers where the end user is already authenticated. When a user is authenticated with a website, the session information is created and stored in both server and browser. If any call is made in the browser to the website, all calls will be executed without any validation.

In other words, CSRF is a **server-side** validation that is aimed to verify the origin of the request.



This practically applies to web-applications only.

Solution: every time we show the action form to the user, we could add some secret information (**CSRF token**) to that form. When executing a request, the browser will send that token as part of the request. This will help a backend side to validate that that specific request came from the original domain.

A **CSRF token** is a unique, secret, unpredictable value created by the server-side application and transmitted to the client for subsequent HTTP request made by the client application. The server-side application validates the CSRF token in the subsequence request, and rejects the request if the token is missing or invalid.

```
<form method="POST" action="/login">
   .....
   <input type="hidden" name="_csrf" value="d919ab77-0668-4a7b-a7ca-2de42c94a8de"/>
   .....
</form>
```

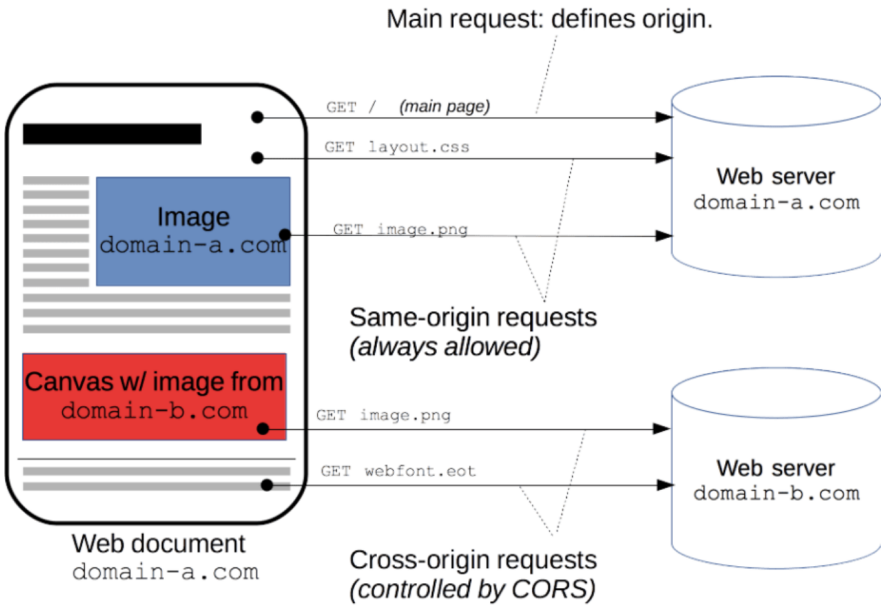
CORS

https://livebook.manning.com/book/cors-in-action/chapter-1/67

For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. This means that a web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers.

CORS is a **client-side** method to relax the same-origin policy. This is used to explicitly allow some cross-origin requests while rejecting others.

Your resources protected by CORS will still be available when requested by any tool or browser that doesn't respect CORS policy: server does no validation, it just enriches the request with a list of origin domains that are allowed to request that specific resource. Again, all checks are made on a client side.



CORS Headers:

```
Access-Control-Allow-Origin: https://domain-b.com
Access-Control-Allow-Methods: GET
```

CORS allows client code to make cross-origin requests to remote servers. CORS is necessary because the browser's same-origin policy traditionally disallows cross-origin requests, which makes it difficult to load data from other sites. Here are some benefits of CORS:

- Opens an API to a wider audience
- Puts servers in charge of how CORS behaves
- Allows flexible configuration options
- Makes it easy for client developers to use
- Reduces maintenance overhead for server developers

XSS

https://dev.to/maleta/cors-xss-and-csrf-with-examples-in-10-minutes-35k3

XSS stands for Cross Site Scripting and it is an injection type of attack.

There are 3 types of such attacks described below.

Stored XSS - Vulnerability coming from unprotected and not sanitized user inputs that are directly stored in the database and displayed to other users.

The attacker comes to your website and finds an unprotected input field such as comment field or user name field and enters a malicious script instead of expected value. After that, whenever that value should be displayed to other users it will execute malicious code. Malicious scripts can try to access your account on other websites (which can be prevented by configuring CSRF/CORS), can be involved in DDoS attacks or similar.

—

Reflected XSS - Vulnerability coming from unprotected and not sanitized values from URLs those are directly used in web pages. Reflected XSS involves the reflecting of a malicious script off of a web application, onto a user's browser. The script is embedded into a link, and is only activated once that link is clicked on.

—

DOM based XSS - Similar as reflected XSS, unprotected and not sanitized values from URLs used directly in web pages, with difference that DOM based XSS doesn't even go to server side.

Expected URL: https://mywebpage.com/search?q=javascript

malicious URL(reflected XSS): https://mywebpage.com/search?q=<script>alert('fortunately, this will not work!')</script>

malicious URL(DOM based XSS): https://mywebpage.com/search#q=<script>alert('fortunately, this will not work!')</script>

Difference is in the character **#** being used instead of **?**. The browsers do not send part of URL after # to the server so they pass it directly to your client code.