# TLS settings of Servers

## Protocol version

Services must support TLS version 1.2, and support TLS 1.3 as soon as it is possible. Modern browsers increasingly support TLS 1.3.

TLS version 1.1 may only be used in situations where the other endpoint does not support TLS version 1.2. However, an endpoint that would support 1.1 but not 1.2 are very rare. Older TLS or SSL versions (TLS 1.0, SSL) must not be supported.

Removing TLS 1.0 support will cause compatibility problems mostly with older mobile browsers, such as Android 4.4 WebView that only supports TLS 1.0, and Java 7 and older.

**Sources:**

* Compatibility tables –
  1. TLS 1.3 support <https://caniuse.com/#feat=tls1-3>
  2. TLS 1.2 support <https://caniuse.com/#feat=tls1-2>
* Ficora decree 72A/2018 on electronic authentication and trust services (in Finnish) requiring the use of TLS 1.2 https://www.viestintavirasto.fi/attachments/maaraykset/M72A\_2018\_M.pdf

## Certificates

For services that do not have a national security level classification, certificates must use RSA keys, in which case the minimum key length is 2048 bits, or ECDSA keys, in which case the minimum key length is 224 bits. Certificates must use SHA-256 as their hashing algorithm. These key lengths and algorithm choices need to be revised regularly to keep current with cryptanalytic progress. Systems designed for a long lifespan should allow for changes in used algorithms and key lengths.

For services which have a national security level classification, key lengths should follow the cryptographic strength requirements from Ficora (see Sources).

When certificates expire, new keys should be generated, and certificates should be created for those new keys. Certificates must not be wildcard certificates that would be accepted for all subdomains.

As new certificates are being issued, issuance is noted in Certificate Transparency logs. This allows external parties to follow which certificates have been issued to an organization. Additionally, if certificates are issued for internal service domains, this information may leak.

Certificate validity can be checked by the browser by using OCSP (Online Certificate Status Protocol). However, using OCSP causes the browser to send OCSP requests to the certificate issuer and this may leak information about the browsing activity to the certificate issuer. Therefore, TLS endpoints should enable OCSP Stapling, where the server retrieves OCSP responses and proactively provides them to browsers in the TLS handshake. This also speeds up the connection.

**Sources:**

* Ficora guideline 190/651/2015 (as current): Cryptographic strength requirements for confidentiality protection – national security levels (in Finnish) https://www.viestintavirasto.fi/attachments/tietoturva/Kryptografiset\_vahvuusvaatimukset\_-\_kansalliset\_suojaustasot.pdf
* <https://www.certificate-transparency.org/>
* Checking certificate settings, including OCSP Stapling**:** https://testssl.sh/

## Certificate issuers

Certificate issuers must have valid trust roots in all of the endpoints that connect to the service, such as most common browsers, operating systems and mobile devices. Users must be educated to understand security implications of certificate error messages, for example, not to accept warning messages on untrusted certificates.

## Encryption algorithms

### Key agreement

For services that do not have a national security level classification, key agreement must use either ECDHE or DHE algorithms. When using ECDHE, the minimum key length is 256 bits, and for DHE, 2048 bits. When using elliptic curve cryptography (ECC), all acceptable curves must be explicitly specified in the protocol configuration. These key lengths and algorithm choices need to be revised regularly to keep current with cryptanalytic progress. Systems designed for a long lifespan should allow for changes in used algorithms and key lengths.

For services which have a national security level classification, key lengths should follow the cryptographic strength guideline from Ficora (see Sources).

Recommended elliptic curves are BrainpoolP256r1, BrainpoolP384r1, BrainpoolP512r1, NIST Curve P-224, NIST Curve P-256, NIST Curve P-384 and NIST Curve P-521.

### Signatures

For services that do not have a national security level classification, signatures must use either ECDSA or RSA algorithms. The minimum key length for ECDSA is 256 bits, and for RSA, 2048 bits. These key lengths and algorithm choices need to be revised regularly to keep current with cryptanalytic progress. Systems designed for a long lifespan should allow for changes in used algorithms and key lengths.

For services which have a national security level classification, key lengths should follow the cryptographic strength guideline from Ficora (see Sources).

### Symmetric encryption

The encryption algorithm used must be AES in CBC or GCM mode, either with a 128 or 256 bit key. When selecting algorithm modes for implementation, CBC mode is preferred as it has less failure modes than GCM. These key lengths and algorithm choices need to be revised regularly to keep current with cryptanalytic progress. Systems designed for a long lifespan should allow for changes in used algorithms and key lengths.

For services which have a national security level classification, key lengths should follow the cryptographic strength guideline from Ficora (see Sources).

### Hash functions

Hash functions must be either SHA-256, SHA-384, SHA-512 or SHA-3. These algorithm choices need to be revised regularly to keep current with cryptanalytic progress. Systems designed for a long lifespan should allow for changes in used algorithms and key lengths.

For services which have a national security level classification, hash function choice should follow the cryptographic strength guideline from Ficora (see Sources).

### TLS cipher suites

For services that do not have a specified national security level, the following TLS 1.2 cipher suites fulfill the requirements above.

DHE-RSA-AES256-GCM-SHA384  
ECDHE-ECDSA-AES256-GCM-SHA384  
ECDHE-RSA-AES256-GCM-SHA384  
DHE-RSA-AES128-GCM-SHA256  
ECDHE-ECDSA-AES128-GCM-SHA256  
ECDHE-RSA-AES128-GCM-SHA256  
DHE-RSA-AES256-SHA256  
ECDHE-ECDSA-AES256-SHA384  
ECDHE-RSA-AES256-SHA384  
DHE-RSA-AES128-SHA256  
ECDHE-ECDSA-AES128-SHA256  
ECDHE-RSA-AES128-SHA256

**Sources:**

* Ficora guideline 190/651/2015 (as current): Cryptographic strength requirements for confidentiality protection – national security levels (in Finnish) https://www.viestintavirasto.fi/attachments/tietoturva/Kryptografiset\_vahvuusvaatimukset\_-\_kansalliset\_suojaustasot.pdf
* eIDAS - Cryptographic requirements for the Interoperability Framework
* Ficora decree 72A/2018 on electronic authentication and trust services
* <https://wiki.mozilla.org/Security/Server_Side_TLS>
* https://www.owasp.org/index.php/TLS\_Cipher\_String\_Cheat\_Sheet
* TLS settings generator: https://mozilla.github.io/server-side-tls/ssl-config-generator/
* Verifying TLS settings: https://testssl.sh/

# HTTP

HTTP based services must not depend on browser extensions (plugins), such as Java applets or Adobe Flash.

## Cookies

Cookies should be scoped as strictly as possible only for the services that rely on them. This can be done for all browsers using the Path parameter and Secure and HttpOnly attributes.

When naming cookies, the cookie name should additionally start with \_\_Secure- if it is set from a HTTPS source. If the cookie can be scoped to a specific host (subdomains do not require the cookie), it should start with \_\_Host- which also implies \_\_Secure-.

All cookies should be purged from the users’ browsers when the browser is closed, unless there is an explicit conflicting requirement that has been analysed for its security impact. This can be done by leaving out the Expires parameter of the cookie.

## HTTP headers towards browsers

In general, it is also recommended to set these headers for APIs that are not supposed to be called by browsers, especially if those APIs are visible to the Internet. For further API security considerations, please refer to this guideline’s API security section.

### Content Security Policy

There are two levels (1 and 2) of the Content Security Policy header. All modern browsers support CSP Level 2. Notably, Internet Explorer 11 does not support the standard CSP header.

A CSP header can be used to restrict the sources from where the browser may load images, scripts or style information.

It is recommended to set a CSP level 2 header that is as strict as possible, using a whitelisting approach. If support for Internet Explorer 11 is required, also the legacy headers (below) should be set.

#### Examples of CSP headers

The following header only allows content from the same source, and also restricts framing:

Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src 'self'; frame-src 'self'; worker-src ‘self’; frame-ancestors 'none'

The following header is the same as above, but also reports problems towards a reporting endpoint that may be useful for detecting configuration problems:

Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src 'self'; frame-src 'self'; worker-src ‘self’; frame-ancestors 'none'; report-uri /some-report-uri

The following header only reports non-compliant requests but does not actually restrict them. This is only meaningful for testing, and not for production.

Content-Security-Policy-Report-Only: default-src 'none'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src 'self'; frame-src 'self'; worker-src ‘self’; frame-ancestors 'none'; report-uri /some-report-uri

**Sources:**

* Browser compatibility tables for CSP header https://caniuse.com/#search=csp
* https://content-security-policy.com/
* <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>
* A commercial tool for following up on non-compliance reports https://report-uri.com/

### HTTP Strict Transport Security

The HTTP Strict Transport Security (HSTS) header forces the browser to use HTTPS for all subsequent requests towards the server. The HSTS header should be set for every response over HTTPS, and because everything should be served over TLS, this means HSTS should be set for every request.

The header has an expiration time, specified in seconds, after which unencrypted connections become possible again. A suitable starting point for a HSTS expiration is 6 months (15 552 000 seconds). For services that are visited less often (e.g., once a year), a longer expiration time may be useful. The header may also be specified to include all subdomains using the includeSubDomains parameter.

Using a HSTS header may cause a self-inflicted Denial of Service condition in the case that the service’s certificate expires, or a load balancer or a content delivery network is using the same address without a proper certificate.

For additional assurance of browsers using HTTPS by default, your site may be submitted to the browser vendors’ ‘preload’ list (see Sources). This requires the header to include the preload directive.

Strict-Transport-Security: max-age=15552000

**Sources:**

* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security>
* Submitting services on the preload list: https://hstspreload.org/

### Legacy headers

The following headers have been deprecated and mostly replaced with new browser features and the Content Security Policy header. However, for some browsers (given the TLS 1.2 requirement that also weeds out old browsers, most notably Internet Explorer 11), these would still be useful for some time.

#### X-Content-Type-Options

Forbids browsers from guessing the content type. Generally, all responses from the server should always define the content type using the Content-Type header, including the character set, if applicable.

X-Content-Type-Options: nosniff

**Sources:**

* https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Options

#### X-XSS-Protection

Instructs the browser to use its internal heuristics against Cross-Site Scripting (XSS) attacks. This is an old header and only applies to very old browsers. A suitably tight Content Security Policy header covers this area for modern browsers.

X-XSS-Protection: 1; mode=block

**Sources:**

* https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection

#### X-Frame-Options

The X-Frame-Options header prohibits browsers from embedding the page within another page using frames. This protects against a clickjacking attack, where the user would be clicking on a link that is behind an invisible link that actually gets activated.

For modern browsers, Content Security Policy level 2 frame-ancestors directive can be used for the same effect. The Content Security Policy frame-src directive controls the loading of frames within the current page, which is not covered by X-Frame-Options.

X-Frame-Options: DENY  
X-Frame-Options: SAMEORIGIN  
X-Frame-Options: ALLOW-FROM https://example.com/

**Sources:**

* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options>
* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors>

### Cache-Control and Expires

The Cache-Control header controls the functionality of the browser cache and caching proxies. The header can be used to specify an expiration time for content, restrict caching on intervening proxies, and provide information for cache refreshes.

From the security perspective, it may be useful to restrict caching of confidential data. The following example prohibits storing any information about the HTTP request or response and forces a new request to be made. These headers should likely not be set for static resources due to performance reasons.

Cache-Control: no-cache, no-store, must-revalidate

The Expires header specifies a time after which content will be marked stale in a cache. These examples prohibit caching by immediate expiry of content. This type of expiry should be used in conjunction with a Cache-Control header.

Expires: Thu, 01 Jan 1970 00:00:00 GMT  
Expires: 0

**Sources:**

* https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control
* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Expires>

### Public key pinning

Public key pinning means only accepting a specific public key (or certificate) for a server. This security feature was intended to reduce the risk of certificates being issued for attacker’s keys and then used in a spoofing attack.

However, support for the public key pinning headers is being phased out in browsers, as it failed to gather momentum and because it introduces a high risk for self-inflicted denial of service. It is still supported in some browsers, but notably was deprecated by Chrome, and never supported by Safari.

Public key pinning should still be used in mobile application development, where it is easier to specify the acceptable trust roots, and those can be updated in an emergency through an application update through the app store.

Public key pinning in browsers is implemented using a HPKP header, Public-Key-Pins.

Instead of HPKP, the organization should actively monitor Certificate Transparency logs and react if they notice unauthorized certificates being issued for their domains.

**Sources:**

* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Public_Key_Pinning>
* https://www.certificate-transparency.org/

## Subresource Integrity

Subresource integrity is a method to ensure the integrity of resources (for example, JavaScript libraries) that are loaded from a potentially untrusted source, such as a content delivery network. Subresource integrity can also be used to deter supply chain attacks in some cases; unauthorized changes to JavaScript dependencies would be rejected by subresource integrity unless the integrity checksums would also be changed.

Subresource integrity should be used for any JavaScript that is loaded from a location that is under a third-party control.

When using subresource integrity, the resources need to be fetched from a source which points to a known and immutable version of the resource. It is not possible to point to a changing ‘latest’ version of a resource when using subresource integrity.

As an example, embedding the version of jQuery 3.3.1 while ensuring its integrity using subresource integrity would look like this:

<script src="https://code.jquery.com/jquery-3.3.1.min.js"  
 integrity="sha256-47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU="  
 crossorigin="anonymous"></script>

The following command can be used to create a base64 encoded SHA-256 hash:

curl -s https://code.jquery.com/jquery-3.3.1.min.js | openssl dgst -sha256 -binary | openssl base64 -A

**Sources:**

* https://developer.mozilla.org/en-US/docs/Web/Security/Subresource\_Integrity

## Session identifiers

The HTTP protocol is stateless. The browser must therefore be uniquely identified using an application layer parameter, namely a session identifier. Secure ways of implementing session identifiers include cookies, POST-type form submission parameters, and custom HTTP headers added with JavaScript.

If a third-party, trustworthy application framework is available providing session identifiers, that must be used.

Session identifiers must be unique and cryptographically random. A recommended length for a random identifier is at least 256 bits.

In many contexts, UUIDs are available. In this case, the only acceptable version of UUIDs is UUIDv4, which are random. Even in this case, it needs to be ensured that the UUIDv4 is generated using a cryptographically strong random number generator. Usually this would be explicitly stated in the random number generator function documentation.

Session identifiers must be recreated every time there is a change in the authentication status or authorization level of the session (for example, an anonymous user logging in, or a user elevating themselves to an admin user, or a user logging out). The previous session identifier must expire when such a change occurs.

**Sources:**

* UUIDv4: <https://en.wikipedia.org/wiki/Universally_unique_identifier#Variants>
* https://www.owasp.org/index.php/Session\_Management\_Cheat\_Sheet

### Creating random identifiers in various programming languages

#### Java

import java.util.SecureRandom;  
SecureRandom random = new SecureRandom();  
byte bytes[] = new byte[32];  
random.nextBytes(bytes);

#### Python 3.6 and newer

import secrets  
secrets.token\_urlsafe()

#### Python before version 3.6

import random  
generator = random.SystemRandom()  
generator.getrandbits(256)

#### JavaScript

var array = new Uint32Array(8);  
window.crypto.getRandomValues(array).join("");

# Logging

Applications should produce audit log records that allow forensic analysis. Log data should be sent in real time to a central log aggregation and analysis system, which would usually be provided by the organization as a common service.

In order for log data to be useful for forensics, it is important that log items contain accurate timestamps in UTC. All servers must be synchronized to a common time standard, typically using NTP.

Additionally, especially in a microservice architecture, log items should have a ‘request ID’ that can be used to correlate log items of different microservices together. For details of this pattern, see below.

Services should log all incoming API requests. They should always log the IP address from where a request originated, even when the service would only be visible to an internal network. If the service is behind a load balancer or a proxy, the proxy must be configured to pass the external IP address towards the service in an X-Forwarded-For header.

Logs should usually consist of JSON objects. This allows the addition of new and custom data elements, and easy parsing of log items in any modern log analysis software.

Especially in a deployment that scales elastically under load, the service should put their unique instance identifier into the log items.

The actions that should be audit logged include:

* Reading, changing or deleting personal data. This log event should contain an application-internal identifier of the person whose personal data was targeted, but usually not the personal data itself
* Changes in session authentication or authorization state (log in, elevation of privilege to an admin, log out)
* Failure to authenticate or authorize an action

The following may generate too much log data, but they may be useful for auditing purposes:

* A failed request towards a back-end server (or another microservice), as this may be an indication of invalid input provided
* Failure to validate input (e.g., an API receives a malformed request)

## The request identifier pattern

For microservice architectures, a request identifier pattern may help correlating various log items together to reconstruct what happened as a result of an incoming request.

The request identifier should be a random identifier that is created at first opportunity when there is an incoming request (e.g., in a load balancer). It would be added to the HTTP request headers in a suitable custom header such as X-Request-Id. Note that externally supplied request identifier headers would need to be removed as these are under attacker control.

Every service (microservice) that receives an X-Request-Id would copy that value into all outgoing requests towards other services. The service would also store this identifier with all the audit log events that are generated as a result of this request.

Paired with timestamps, a request identifier allows forensic reconstruction of the complete microservice call tree, and its correlation with the external IP address from where the original request originated.

# Docker

Hardening Docker and orchestration tools (such as Kubernetes) is a more complicated topic. The Center for Internet Security (CIS) has published its own guidelines for hardening these systems. This guideline concentrates on architectural and design level aspects. The hardening of Docker and orchestration tool installation would also have to be done.

## Docker hosts

Servers who are running Docker containers (hosts) must be dedicated to hosting containers. These hosts must not be used for other purposes.

One server may only run containers that have an identical trust level. As an example, Docker containers supplying externally exposed services should be run on a different host than those running back-end services.

When using an orchestration system, the placement of containers on different VMs can sometimes be controlled. For example, Kubernetes has the possibility of node restrictions.

**Sources:**

* CIS Docker Community Edition Benchmark v1.1.0
* <https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#node-isolation-restriction>

## Docker containers

Any single Docker container must be built on a trusted base image, and any software built in have to originate from a trusted source. This trustworthiness can be based on verification of cryptographic signatures or the fact that the source is under the organization’s control.

The base images must be pinned to a specific version using a SHA-256 hash value, even if the base image would be internal. This means that using a latest base image is not possible.

An example of how a base image can be pinned to a specific version:

FROM alpine@sha256:3dcdb92d7432d56604d4545cbd324b14e647b313626d99b889d0626de158f73a

Building containers with Dockerfiles must ensure that any installed software is integrity checked. If the Dockerfile downloads software from an untrusted source, the same Dockerfile should not also obtain the integrity verification keys from a similarly untrusted source.

Dependencies should preferably be installed from a local repository. This allows the organization to control which version of the dependency is being used, to exert change control, and to build the Docker image even if the third-party repository would be unavailable.

Docker containers should always be run with read-only root filesystems unless there is a serious technical rationale of not to.

Docker containers must never be run using the --privileged option or provide access to the docker.sock management interface. Docker containers should be provided with as little host file system and network access as possible.

When using an orchestration system that provides an overlay network, the containers’ network traffic should be restricted to only those destinations that are necessary for the container to work. For example, in Kubernetes, this is known as NetworkPolicy.

When performing a security assessment for a system that uses Docker or an orchestration system, a review of Dockerfile and the orchestration system’s configuration files must always be included in the scope of the security assessment.

**Sources:**

* CIS Docker Community Edition Benchmark
* https://kubernetes.io/docs/concepts/services-networking/network-policies/

## Secrets management

Secrets, such as passwords, unencrypted private keys or API keys must never be stored in plaintext in version control systems or Dockerfiles. Long-term secrets storage should be done using secrets management software. These range from password managers for individual developers to various ‘vault’ services that can be deployed as cloud services. Secrets should be also backed up.

Delivering secrets to running containers is a development and delivery pipeline specific problem, and it is not easy to give a solution that would work in all cases. Secrets provisioning should preferably be done using a ‘vault’ type service, or a system provided by the orchestration tools. If these are not available, secrets can be provided through encrypted files made available on a storage location, and the decryption key can be provided to the container in an environmental variable. If using environmental variables, these should be zeroed after the value has been obtained, as in many failure modes, debugging output dumps the environment contents.

### Environment variables

Environment variables can be provided to the Docker containers on command line:

docker run -e SECRET=1234567890abcdef alpine

### Third party software

Third-party software exist for secrets management, such as HashiCorp Vault. Kubernetes and OpenShift have ‘secret’ objects. Infrastructure clouds have their own key and secrets storage systems. Using an orchestrator-provided secrets provisioning scheme is usually the best choice.

**Sources:**

* <https://kubernetes.io/docs/concepts/configuration/secret/>
* <https://docs.openshift.com/container-platform/3.11/dev_guide/secrets.html>
* <https://www.vaultproject.io/>

# Application Programming Interfaces (APIs)

All new web interfaces should be implemented as JSON. The JSON presentation is simple and readable, and support from software is good. The other common implementations, JSONP and XML, have distinct challenges:

* JSONP should not be used as the API responses are executed as program code. This removes the abstraction between an interface and the application using the interface. Interfaces should only provide data which is processed by the calling application.
* XML parsers have often had errors that have led to data being interpreted incorrectly. This may be a problem if different XML parsers interpret their input differently. XML also allows information to have multiple different structures that in the end are parsed in a similar way.

## API security

For each incoming request, the API should perform multiple checks, desxribed below. Depending on the architecture, the API implementation may trust an intervening API gateway or proxy or perform these checks itself.

Should one of these checks fail, the request should be immediately rejected. The API should not try to handle malformed requests. If there is a problem with authentication, authorization, or integrity of data, the content of the request should not be parsed. Instead, processing should end with minimal overhead.

* Session identifiers: Is the session valid? What is the level of authorization? (See the session identifier section in this guideline.)
* Integrity of the request and the data. Determine the integrity of the request. When using an API key or another shared secret between the caller and the API, integrity protection can build on that secret. Typically, authorization credentials are transferred using JSON Web Tokens (JWT) and complete API call integrity can be attained using AWS Signature v4 scheme.
* Is the incoming request well formed? Even when using JSON, the request should be validated against a schema. This protects the API against injection of arbitrary data fields. Without schema validation, there is a risk that arbitrary extra data fields may end up even in a database – a typical problem when using schema-less databases.
* Authorisation. Authorisation checks are twofold: Is the caller authorized to call this API, and is the caller authorized to receive the information.
* Does the API need to create audit log events for this call? (Please see the section on audit logging in this guideline.)
* Does the API set all required HTTP headers in the response, such as HSTS and Content-Type? (Please see the section on HTTP headers in this guideline.)

When using JWT for authorisation, its implementation must ensure that the secret used for HMAC-SHA-256 calculation is 256 bits in length. See the section on session tokens in this guideline for how this can be ensured.

**Sources:**

* JSON Web Tokens theme pages <https://jwt.io/>
* AWS Signature v4, which is not AWS specific https://docs.aws.amazon.com/general/latest/gr/sigv4\_signing.html

## Cross-Origin Resource Sharing (CORS)

CORS is a framework where the server informs the browser that it is allowed to bypass the normal same-origin rules and call an API on another domain. The browser performs a separate initial request, called a preflight request, to determine whether this type of cross-origin access can be allowed. It is important to note that enforcement of CORS happens on the browser side. An attacker can still call an API without a browser irrespective of what its CORS set-up is. Server-side access control must always be implemented using appropriate access credentials.

The target server provides their CORS policy using the Access-Control-Allow-Origin header. Access-Control-Allow-Origin specifies the sources of JavaScript code that may call that API (that is, where the JavaScript originated that can call the API).

For example, if an organization has a web application at app.company.com, and an API at api.cloud.example, and they want JavaScript from the app.company.com web page to be able to call api.cloud.example, the api.cloud.example API must be configured to return something like

Access-Control-Allow-Origin: app.company.com  
Access-Control-Allow-Methods: GET POST PUT DELETE

in their HTTP responses. If the API requires credentials (e.g., in cookies), it also has to explicitly grant the browser the possibility to deliver cross-origin credentials with

Access-Control-Allow-Credentials: true

If an API needs to be callable from whatever JavaScript code, irrespective from where it has been served, the header would read

Access-Control-Allow-Origin: \*

If the API is not supposed to be generally callable, existence of this header may indicate that the CORS policy is too lax.

**Sources:**

* https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
* https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin\_policy

## CSRF tokens

Cross Site Request Forgery (CSRF) is an attack where an attacker site performs a HTTP request towards another site using the user’s browser. If the browser has an active session on the target site, the attacker may perform actions on the user’s behalf.

CSRF can be mitigated by providing a cryptographically random and sufficiently long value, known as a CSRF token, with each HTTP request. The tokens are stored in the browser in such a way that the attacker site, or JavaScript that originates from the attacker site, cannot access them, and therefore cannot forge the requests.

CSRF is easily mitigated by using a proper framework that provides CSRF protection by default. In most cases, this would be preferable.

If this type of framework support is not available, a recommended way to mitigate is to use a “double submit cookie”. This method is useful because it is stateless, requiring no server-side storage, and as such it can be used when requests are routed to random servers through a load balancer. As with many other security features, naïve implementations of this may introduce vulnerabilities.

**Sources:**

* <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet#Double_Submit_Cookie>

## Confidential data in parameters

API calls must be formed in a way that personal data or other confidential data is not provided as a part of a URL. Parameters and path elements of a URL will easily get logged, stored in cache and browser history, or be stored by proxies.

All confidential data should be transferred in the body of a POST request.

The correct way could look for example like this:

POST /search/address HTTP/1.1  
  
SSN=010170-123F

The wrong way could look something like this:

GET /search/010170-123F/address HTTP/1.1

In these examples, 010170-123F is a Finnish personal identification number.

# Palvelimen TLS-asetukset

## Protokollaversio

Palveluiden tulee tukea TLS 1.2 -versiota ja tukea TLS 1.3:ta heti kun mahdollista. Nykyaikaiset selaimet tukevat lisääntyvässä määrin TLS 1.3 -versiota jo nyt.

TLS:n versiota 1.1 tulee käyttää ainoastaan, jos toinen osapuoli ei tue 1.2-versiota. On kuitenkin hyvin harvinaista, että osapuoli tukisi 1.1:tä mutta ei 1.2:ta. Vanhempia TLS- tai SSL-versioita (TLS 1.0, SSL) ei tule tukea.

Tuen poistaminen TLS 1.0 -protokollalta aiheuttaa yhteensopivuusongelmia lähinnä vanhempien mobiiliselainten kuten Android 4.4:n WebView’n, sekä Java 8:aa vanhempien versioiden kanssa, koska ne tukevat ainoastaan TLS 1.0:aa.

**Lähteet:**

1. Yhteensopivuustaulukot
   1. TLS 1.3-tuki <https://caniuse.com/#feat=tls1-3>
   2. TLS 1.2 -tuki <https://caniuse.com/#feat=tls1-2>
2. Viestintäviraston määräys 72A/2018 sähköisistä tunnistus- ja luottamuspalveluista, joka vaatii TLS 1.2:n käyttöä: https://www.viestintavirasto.fi/attachments/maaraykset/M72A\_2018\_M.pdf

## Varmenteet

Järjestelmissä, joilla ei ole kansallista suojaustasoa, varmenteiden tulee olla joko RSA-avaimia, jolloin tulee käyttää vähintään 2048-bittisiä avaimia, tai ECDSA-avaimia, jolloin tulee käyttää vähintään 224-bittisiä avaimia. Varmenteiden allekirjoitus tulee perustua SHA-256-tiivisteeseen. Nämä avaimenpituus- ja algoritmisuositukset tulee säännöllisesti tarkistaa, jotta edistysaskeleet kryptoanalyysissä otetaan huomioon. Pitkäikäisiksi tarkoitetuissa järjestelmissä pitää huomioida se, että käytettyjä algoritmeja ja avainpituuksia voidaan vaihtaa tarvittaessa.

Jos järjestelmälle on määritelty kansallinen suojaustaso, avainpituuksissa tulisi noudattaa Viestintäviraston kryptografisen vahvuuden ohjetta (ks. Lähteet).

Varmenteen vanhetessa tulisi luoda uusi avain ja tälle avaimelle hakea uusi varmenne. Varmenteet eivät saa olla wildcard-varmenteita, jotka ovat kelpoja kaikille alidomaineille.

Uusista myönnettävistä varmenteista tallennetaan tieto Certificate Transparency -lokeihin. Tämä mahdollistaa seuraamisen organisaatiolle myönnetyistä varmenteista. Tämä myös saattaa paljastaa sisäisesti käytettyjä palveluosoitteita, jos näille haetaan varmenne julkisesta lähteestä.

Varmenteiden voimassaolo voidaan tarkistaa OCSP-protokollalla (Online Certificate Status Protocol). OCSP:n käyttö aiheuttaa sen, että selain lähettää varmenteen myöntäjälle pyyntöjä ja nämä pyynnöt saattavat vuotaa tietoa selailusta varmenteen myöntäjälle. Tämän vuoksi palvelinten tulisi käyttää proaktiivista OCSP-vasteiden jakelua TLS-kättelyssä, joka tunnetaan nimellä OCSP Stapling. Tämä myös nopeuttaa yhteyksien luomista.

**Lähteet:**

* Viestintäviraston ohjeen 190/651/2015 ajantasainen versio: Kryptografiset vahvuusvaatimukset luottamuksellisuuden suojaamiseen – kansalliset suojaustasot https://www.viestintavirasto.fi/attachments/tietoturva/Kryptografiset\_vahvuusvaatimukset\_-\_kansalliset\_suojaustasot.pdf
* <https://www.certificate-transparency.org/>
* Varmenneasetusten, mm. OCSP Staplingin tarkastaminen**:** https://testssl.sh/

## Varmenteiden myöntäjät

Käytettävien juurivarmenteiden tulee olla kaikkien palvelua käyttävien tahojen laitteissa oletuksena luotettuna. Tämän tulisi sisältää ainakin käytetyimmät selaimet, käyttöjärjestelmät ja mobiililaitteet. Käyttäjiä tulee tukea ymmärtämään varmennevirheiden tietoturvavaikutukset, kuten ohjeistaa olemaan hyväksymättä selaimen antamaa varoitusviestiä epäkelvosta varmenteesta.

## Salausalgoritmit

### Avaintenvaihto

Jos järjestelmälle ei ole määritetty kansallista suojaustasoa, avaintenvaihdossa on käytettävä ECDHE- tai DHE-menetelmiä. Käytettäessä ECDHE-menetelmää tulee avaimen pituuden olla vähintään 256 bittiä, DHE-menetelmää käytettäessä 2048 bittiä. Nämä avaimenpituus- ja algoritmisuositukset tulee säännöllisesti tarkistaa, jotta edistysaskeleet kryptoanalyysissä otetaan huomioon. Pitkäikäisiksi tarkoitetuissa järjestelmissä pitää huomioida se, että käytettyjä algoritmeja ja avainpituuksia voidaan vaihtaa tarvittaessa.

Jos järjestelmälle on määritelty kansallinen suojaustaso, avainpituuksissa tulisi noudattaa Viestintäviraston kryptografisen vahvuuden ohjetta (ks. Lähteet).

Käytettäessä elliptisten käyrien menetelmiä (ECC) tulee kaikki käytettävät käyrät luetella konfiguraatiossa.

Suositeltuja käytettäviä käyriä ovat BrainpoolP256r1, BrainpoolP384r1, BrainpoolP512r1, NIST Curve P-224, NIST Curve P-256, NIST Curve P-384 ja NIST Curve P-521.

### Allekirjoitus

Jos järjestelmälle ei ole määritetty kansallista suojaustasoa, allekirjoitusalgoritmina voidaan käyttää joko ECDSA- tai RSA-algoritmia. ECDSA-avaimen pituus tulee olla vähintään 256 bittiä ja RSA-avaimen 2048 bittiä. Nämä avaimenpituus- ja algoritmisuositukset tulee säännöllisesti tarkistaa, jotta edistysaskeleet kryptoanalyysissä otetaan huomioon. Pitkäikäisiksi tarkoitetuissa järjestelmissä pitää huomioida se, että käytettyjä algoritmeja ja avainpituuksia voidaan vaihtaa tarvittaessa.

Jos järjestelmälle on määritelty kansallinen suojaustaso, avainpituuksissa tulisi noudattaa Viestintäviraston kryptografisen vahvuuden ohjetta (ks. Lähteet).

### Symmetrinen salaus

Salausalgoritmin tulee olla AES käytettäen joko 128-bittistä tai 256-bittistä avainta. Salausmoodin on oltava CBC tai GCM. Valittaessa algoritmien moodeja toteutustarkoituksissa, CBC-moodi on parempi, koska sen kanssa on GCM:ää vaikeampi tehdä virheitä. Nämä avaimenpituus- ja algoritmisuositukset tulee säännöllisesti tarkistaa, jotta edistysaskeleet kryptoanalyysissä otetaan huomioon. Pitkäikäisiksi tarkoitetuissa järjestelmissä pitää huomioida se, että käytettyjä algoritmeja ja avainpituuksia voidaan vaihtaa tarvittaessa.

Jos järjestelmälle on määritelty kansallinen suojaustaso, avainpituuksissa tulisi noudattaa Viestintäviraston kryptografisen vahvuuden ohjetta (ks. Lähteet).

### Tiivistefunktiot

Tiivistefunktion on oltava joko SHA-256, SHA-384, SHA-512 tai SHA-3. Nämä algoritmisuositukset tulee säännöllisesti tarkistaa, jotta edistysaskeleet kryptoanalyysissä otetaan huomioon. Pitkäikäisiksi tarkoitetuissa järjestelmissä pitää huomioida se, että käytettyjä algoritmeja ja avainpituuksia voidaan vaihtaa tarvittaessa.

Jos järjestelmälle on määritelty kansallinen suojaustaso, tiivistefunktion valinnassa tulisi noudattaa Viestintäviraston kryptografisen vahvuuden ohjetta (ks. Lähteet).

### TLS-salausalgoritmimääritykset

Järjestelmille, joille ei ole määritelty kansallista suojaustasoa, alla on listattu nämä ehdot täyttävät TLS 1.2-algoritmit:

DHE-RSA-AES256-GCM-SHA384  
ECDHE-ECDSA-AES256-GCM-SHA384  
ECDHE-RSA-AES256-GCM-SHA384  
DHE-RSA-AES128-GCM-SHA256  
ECDHE-ECDSA-AES128-GCM-SHA256  
ECDHE-RSA-AES128-GCM-SHA256  
DHE-RSA-AES256-SHA256  
ECDHE-ECDSA-AES256-SHA384  
ECDHE-RSA-AES256-SHA384  
DHE-RSA-AES128-SHA256  
ECDHE-ECDSA-AES128-SHA256  
ECDHE-RSA-AES128-SHA256

Lähteet:

* Viestintäviraston ohjeen 190/651/2015 ajantasainen versio: Kryptografiset vahvuusvaatimukset luottamuksellisuuden suojaamiseen – kansalliset suojaustasot https://www.viestintavirasto.fi/attachments/tietoturva/Kryptografiset\_vahvuusvaatimukset\_-\_kansalliset\_suojaustasot.pdf
* eIDAS - Cryptographic requirements for the Interoperability Framework
* Määräys sähköisistä tunnistus- ja luottamuspalveluista, Viestintävirasto 72A/2018 M
* https://wiki.mozilla.org/Security/Server\_Side\_TLS
* https://www.owasp.org/index.php/TLS\_Cipher\_String\_Cheat\_Sheet
* **Asetusgeneraattori:** https://mozilla.github.io/server-side-tls/ssl-config-generator/
* **Asetusten tarkastaminen:** https://testssl.sh/

# HTTP

HTTP-palvelut tulee kehittää ilman riippuvuuksia selaimeen asennettaviin lisäosiin (plugin) kuten esimerkiksi Javaan tai Flashiin.

## Evästeet

Evästeet (cookies) tulisi rajata mahdollisimman tiukasti ainoastaan niitä tarvitsevien palveluiden käyttöön. Mahdollisia kaikilla selaimilla toimivia rajauskeinoja ovat Path-parametri sekä Secure ja HttpOnly -attribuutit.

Evästeiden nimeämisessä kannattaa lisäksi käyttää \_\_Secure- etuliitettä evästeille, jotka on asetettu HTTPS-lähteestä. Jos eväste voidaan rajata vain tiettyyn palvelinnimeen (alidomainit eivät tarvitse evästettä), nimi kannattaa aloittaa \_\_Host-, joka sisältää myös \_\_Secure- -toiminnallisuuden.

Kaikkien käytettyjen evästeiden tulisi poistua käyttäjän selaimesta kun selain suljetaan, ellei vastakkaiselle ole erikseen määriteltyä vaatimusta, jonka tietoturvariski on arvioitu. Tämä saavutetaan jättämällä evästeestä pois Expires-arvo.

## HTTP-otsakkeet selaimille

Yleisesti on myös suositeltavaa asettaa nämä otsakkeet rajapinnoille, joita selaimien ei ole tarkoitus kutsua, erityisesti jos nämä rajapinnat ovat avoinna Internetiin. Lisää tietoa API-turvallisuudesta on tämän ohjeen API-turvallisuuskappaleessa.

### Content Security Policy

Content Security Policy (CSP) -otsakkeesta on määritelty tasot 1 ja 2. Kaikki modernit selaimet tukevat CSP taso 2:ta. Huomionarvoisesti Internet Explorer 11 ei tue standardia CSP-otsaketta.

CSP-otsakkeella voidaan rajata, mistä lähteistä selain saa ladata sivulle esimerkiksi kuvia, skriptejä tai tyylitiedostoja.

On suositeltavaa asettaa niin tiukka CSP-otsake kuin suinkin mahdollista, sallien ainoastaan nimetyt lähteet. Mikäli tuki Internet Explorer 11 -selaimelle on tarpeen, myös vanhemmat tietoturvaotsakkeet (alempana) pitäisi asettaa.

#### Esimerkkejä CSP-otsakkeista

Seuraava otsake sallii sisällön latauksen ainoastaan samasta lähteestä ja rajoittaa kehysten käyttöä:

Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src 'self'; child-src 'self'; frame-ancestors 'none'

Seuraava otsake on sama kuin edellinen, mutta lisäksi raportoi poikkeukset raportointirajapintaan, josta voi olla hyötyä konfiguraatio-ongelmien löytämisessä:

Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src 'self'; child-src 'self'; frame-ancestors 'none'; report-uri /some-report-uri

Seuraava otsake ainoastaan raportoi sääntöjä rikkovat pyynnöt, muttei rajoita niiden lataamista. Tätä vaihtoehtoa kannattaa käyttää vain palvelun kehitysvaiheessa, ei tuotannossa.

Content-Security-Policy-Report-Only: default-src 'none'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src 'self'; child-src 'self'; frame-ancestors 'none'; report-uri /some-report-uri

**Lähteet:**

* CSP-otsakkeen selaintukitaulukko https://caniuse.com/#search=csp
* https://content-security-policy.com/
* <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>
* Kaupallinen työkalu raporttien seurantaan https://report-uri.com/

### HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) -otsake pakottaa selaimen käyttämään kaikissa tulevissa yhteyksissä sivustolle HTTPS-protokollaa. HSTS-otsake tulisi asettaa kaikille HTTPS-yhteyden kautta annettaville vastauksille, ja koska kaikki palvelut tulisi tarjota TLS:llä, HSTS tulisi asettaa kaikille vastauksille.

Säännölle annetaan yläraja sekunteina, jonka jälkeen voidaan taas muodostaa salaamattomia yhteyksiä. Sopiva arvo HSTS-otsakkeelle on 6 kuukautta eli 15 552 000 sekuntia. Palveluille, joita käytetään samalta selaimelta harvemmin (esimerkiksi kerran vuodessa), pidempi ajanjakso voi olla tarpeen. Sääntö voidaan myös ulottaa käsittämään kaikki aliosoitteet includeSubDomains -parametrilla.

HSTS-otsake saattaa estää palvelun käytön, jos palvelimen varmenne on unohdettu uusia tai jos kuormantasaus tai CDN-palvelu käyttää käyttää samaa osoitetta ilman toimivaa varmennetta.

HTTPS:n oletusarvoisen käytön varmistamiseksi sivusto voidaan myös ilmoittaa etukäteen selainvalmistajien ’preload’ -listalle (ks. Lähteet). Tällöin otsakkeeseen on lisättävä preload -direktiivi.

Strict-Transport-Security: max-age=15552000

**Lähteet:**

* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security>
* Palvelun ilmoittaminen preload-listalle: https://hstspreload.org/

### Vanhat otsakkeet

Seuraavat otsakkeet ovat poistuvia otsakkeita, jotka ovat korvautuneet selainten uusilla ominaisuuksilla tai Content Policy -otsakkeella. Joidenkin selainten, kirjoitushetkellä erityisesti Internet Explorer 11:n, osalta nämä ovat silti edelleen tarpeen.

#### X-Content-Type-Options

Estää selaimia arvaamasta sisällön tyyppiä. Yleisesti ottaen kaikkien palvelimen vastausten tulisi aina määritellä sisällön tyyppi Content-Type -otsakkeella, jossa pitäisi myös määrittää käytetty merkistö, jos sisältötyyppi on tekstiä.

X-Content-Type-Options: nosniff

**Lähteet:**

1. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Options

#### X-XSS-Protection

Kertoo selaimelle ottaa käyttöön suojaukset XSS-hyökkäyksiä vastaan. Tämä on vanha otsake ja sisällytetty ainoastaan turvaamaan vanhoja selaimia. Oikein määritelty Content Security Policy kattaa tämän nykyaikaisten selainten osalta.

X-XSS-Protection: 1; mode=block

**Lähteet:**

* https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection

#### X-Frame-Options

X-Frame-Options -otsakkeella voidaan estää selaimia upottamasta sivua osaksi toista sivustoa. Tätä käytetään suojaamaan käyttäjiä clickjacking-hyökkäyksiltä, jossa käyttäjä houkutellaan painamaan selaimessa linkkiä, jonka "eteen" on asetettu näkymätön linkki, jota käyttäjä todellisuudessa tulee painaneeksi.

Nykyaikaisilla selaimilla Content Security Policy tason 2 frame-ancestors -direktiivillä saadaan aikaan sama efekti. Content Security Policyssä on lisäksi frame-src -direktiivi, jolla kontrolloidaan sivun itse lataamia kehyksiä, ja tätä X-Frame-Options ei kata.

X-Frame-Options: DENY  
X-Frame-Options: SAMEORIGIN  
X-Frame-Options: ALLOW-FROM <https://example.com/>

**Lähteet:**

* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options>
* https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors

### Cache-Control ja Expires

Cache-Control-otsakkeella ohjataan selaimen välimuistin ja mahdollisten välityspalvelimien toimintaa. Otsakkeella voidaan määrittää sisällölle vanhenemisaika, rajoittaa sisällön tallentamista välimuisteihin ja ohjeistaa välimuistia sisällön päivittämisestä.

Turvallisuusnäkökulmasta voi olla tarpeen estää luottamuksellisen tiedon tallennus välimuistiin. Seuraava otsake-esimerkki kieltää sisällön tallentamisen välimuistiin, kieltää kaiken HTTP-pyyntöön liittyvän tiedon tallentamisen ja pakottaa selaimen aina tekemään pyynnön uudestaan. Tätä otsaketta ei tulisi asettaa staattisten resurssien osalta suorituskykyvaikutusten vuoksi.

Cache-Control: no-cache, no-store, must-revalidate

Expires-otsake määrittää aikaleiman, jolloin kyseinen sisältö tulisi merkitä vanhentuneeksi. Nämä esimerkit määrittävät tiedon olevan jo valmiiksi vanhentunutta. Tämänkaltaista vanhenemisotsaketta tulisi käyttää yhdessä Cache-Control -otsakkeen kanssa.

Expires: Thu, 01 Jan 1970 00:00:00 GMT  
Expires: 0

**Lähteet:**

* https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control
* https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Expires

### Public key pinning

Julkisen avaimen sitominen (public key pinning) tarkoittaa, että palvelimelta hyväksytään vain ennakkoon määritelty julkinen avain tai varmenne. Tämän tietoturvaominaisuuden tarkoitus oli pienentää riskiä siitä, että hyökkääjän avaimelle myönnetään varmenne, joka mahdollistaa palvelimeksi tekeytymisen.

Tuki julkisen avaimen sitomiselle on kuitenkin poistumassa selaimista, koska siitä ei tullut kovin suosittua ja koska sitä käytettäessä on suuri riski itseaiheutetusta palvelunestotilasta. Jotkin selaimet tukevat sitä edelleen, mutta huomionarvioisesti Chrome poisti sen käytöstä ja Safari ei koskaan tukenutkaan sitä.

Julkisen avaimen sitomista tulisi edelleen käyttää mobiilikehityksessä, jossa mobiilisovellukselle on helpompaa määritellä luotettujen varmenteiden joukko, ja luotetut varmenteet voidaan hätätilassa päivittää ohjelmistopäivityksellä sovelluskauppojen kautta.

Julkisen avaimen sitominen selaimissa tehdään HPKP-otsakkeella, Public-Key-Pins.

HPKP:n sijaan organisaation tulisi aktiivisesti seurata Certificate Transparency -lokeja ja reagoida, jos heidän hallussaan oleville domaineille myönnetään varmenteita ilman lupaa.

**Lähteet:**

* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Public_Key_Pinning>
* https://www.certificate-transparency.org/

## Subresource Integrity

Subresource integrity mahdollistaa vähemmän luotetusta lähteestä kuten esimerkiksi CDN-palveluista ladattavan JavaScript-kirjastojen eheyden tarkistamisen selaimessa. Subresource integrityä voidaan käyttää myös joidenkin toimitusketjuhyökkäysten estämiseen: subresource integrity hylkää hyökkääjän muuttamat JavaScript-riippuvuudet, ellei hyökkäjä muuta myös eheyden tarkistukseen käytettäviä tiivisteitä.

Subresource integrityä tulisi käyttää kaikelle JavaScriptille, joka ladataan kolmannen osapuolen hallussa olevasta sijainnista.

Käytettäessä subresource integrity -tarkisteita resurssit on haettava lähteestä, joka osoittaa resurssin tunnettuun ja muuttumattomaan versioon. Subresource integrityä käyttäessä ei ole mahdollista osoittaa ajoittain muuttuvaan, viimeisimpään (’latest’) versioon.

Esimerkkinä jQuery-kirjaston version 3.3.1 sisällyttäminen sivulle ja sen eheyden tarkistus SHA-256-tiivisteen avulla:

<script src="https://code.jquery.com/jquery-3.3.1.min.js"  
 integrity="sha256-47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU="  
 crossorigin="anonymous"></script>

Kirjaston SHA-256-tiiviste Base64-muodossa voidaan laskea seuraavalla komennolla:

curl -s https://code.jquery.com/jquery-3.3.1.min.js | openssl dgst -sha256 -binary | openssl base64 -A

**Lähteet:**

* https://developer.mozilla.org/en-US/docs/Web/Security/Subresource\_Integrity

## Istuntotunnisteet

HTTP-protokolla on tilaton, jolloin käyttäjän käyttämä selain on yksilöitävä jollain sovellustason parametrilla, ts. istuntotunnisteella. Turvallisia tapoja istuntotunnisteen välittämiseksi ovat evästeet, lomakkeiden kohdalla POST-pyynnön parametrit, ja JavaScript-kutsuissa HTTP-otsakkeet.

Mikäli käytössä on valmis ja luotettavana pidetty istuntotunnisteet hoitava sovelluskehikko, sitä tulee käyttää.

Istuntotunnisteena käytettävän tunnisteen tulee olla yksilöllinen, riittävän pitkä, ja kryptografisesti satunnainen. Suositeltu ratkaisu on käyttää vähintään 256 bittiä pitkää satunnaista tunnistetta.

Monissa tilanteissa on saatavissa UUID-tunniste. Jos käytetään UUID-tunnisteita, tulee varmistua, että käytetään UUIDv4-muotoisia tunnisteita, jotka generoidaan satunnaisesti. Tällöinkin tulee varmistua, että UUIDv4-toteutuksessa käytetään kryptografisesti vahvaa satunnaislukugeneraattoria. Tyypillisesti tämä sanotaan selvästi satunnaislukugeneraattorin dokumentaatiossa.

Istuntotunniste on luotava uudelleen aina kun käyttäjän tunnistautumisen tila tai valtuutustaso muuttuu (esimerkiksi sisäänkirjautumisessa, pääkäyttäjäksi siirtyessä tai kirjautuessa ulos). Edellisen istuntotunnisteen on tämänkaltaisissa tilanteissa vanhennuttava.

**Lähteet:**

* UUIDv4: <https://en.wikipedia.org/wiki/Universally_unique_identifier#Variants>
* https://www.owasp.org/index.php/Session\_Management\_Cheat\_Sheet

### Satunnaisen tunnisteen luominen eri ohjelmointikielillä

#### Java

import java.util.SecureRandom;  
SecureRandom random = new SecureRandom();  
byte bytes[] = new byte[32];  
random.nextBytes(bytes);

#### Python 3.6 ja uudemmat

import secrets  
secrets.token\_urlsafe()

#### Python ennen versiota 3.6

import random  
generator = random.SystemRandom()  
generator.getrandbits(256)

#### JavaScript

var array = new Uint32Array(8);  
window.crypto.getRandomValues(array).join("");

# Lokitus

Sovellusten tulisi tuottaa auditointilokia, joka mahdollistaa forensisen analyysin. Lokitiedot tulisi lähettää reaaliaikaisesti keskitetylle lokien keräily- ja analyysijärjestelmälle, jollaisen organisaatio yleensä tuottaa yhteisenä palveluna.

Jotta lokitiedot olisivat käyttökelpoisia forensiikkaan, on tärkeää, että lokitiedoissa on tarkat UTC-aikaan sidotut aikaleimat. Kaikkien palvelinten on oltava synkronoituja yhteiseen aikastandardiin, yleensä NTP:llä.

Lisäksi erityisesti mikropalveluarkkitehtuureissa lokitietoihin tulisi tallentaa pyyntötunniste, jota voidaan käyttää korreloimaan lokitapahtumia eri mikropalveluiden välillä. Pyyntötunnisteesta on lisäkuvaus alempana.

Palveluiden tulisi lokittaa kaikki sisään tulevat rajapintapyynnöt. Lokiin tulisi merkitä myös pyynnön lähettänyt IP-osoite, vaikka palvelu olisikin auki vain sisäverkkoon. Mikäli palvelu on kuormantasaimen tai välityspalvelimen takana, välityspalvelin on konfiguroitava välittämään ulkoinen IP-osoite edelleen X-Forwarded-For -otsakkeessa.

Lokien tulisi yleensä olla JSON-objekteja. Tämä mahdollistaa uusien ja tapauskohtaisten tietoelementtien lisäämisen sekä helpon lokien jäsentämisen millä tahansa nykyaikaisella lokianalyysityökalulla.

Erityisesti ympäristössä, joka skaalautuu elastisesti kuorman mukaan, palvelujen tulisi tallentaa lokiin myös yksikäsitteinen instanssitunnisteensa.

Tapahtumat, jotka tulisi kirjoittaa lokimerkintä, ovat:

* Henkilötietojen lukeminen, muuttaminen ja poisto. Lokikirjauksen tulisi sisältää sovelluksen sisäinen tunniste siitä henkilöstä, kenen tietoihin kajottiin, mutta yleensä ei henkilötietoja itsessään
* Muutokset istuntojen todennuksen tai valtuutuksen tilassa (sisäänkirjautuminen, pääkäyttäjäksi siirtyminen, uloskirjautuminen)
* Epäonnistuneet todennus- tai valtuutusyritykset

Seuraavat tapahtumat voivat generoida liikaa lokidataa, mutta niitä voidaan myös harkita auditointitarkoituksiin:

* Epäonnistuneet kutsut taustapalvelimelle tai muihin mikropalveluihin, koska tämä saattaa olla merkki vääristä kutsuparametreista
* Epäonnistunut syötteen tarkastus (esimerkiksi rajapintakutsu, johon tulee vääränlainen syöte)

## Pyyntötunnisteet

Pyyntötunniste on erityisesti mikropalveluarkkitehtuureille hyödyllinen tapa korreloida lokitietoja. Niiden avulla voidaan jälkikäteen rakentaa kuva siitä, mitä sisään tulleen pyynnön seurauksena tapahtui.

Pyyntötunnisteen tulisi olla satunnainen tunniste, joka luodaan mahdollisimman aikaisessa vaiheessa sisään tulevia pyyntöjä käsitellessä (esimerkiksi kuormantasaimessa). Se lisätään HTTP-pyyntöön sopivassa otsakkeessa, kuten X-Request-Id. On huomattava, että ulkopuolelta syötetyt pyyntötunnisteet tulee poistaa, koska ne ovat hyökkääjän manipuloitavissa.

Jokainen palvelu (mikropalvelu), joka vastaanottaa X-Request-Id -arvon, kopioi vastaavan arvon kaikkiin muihin palveluihin lähettämiinsä pyyntöihin. Lisäksi palvelu tallentaa tämän arvon jokaiseen auditointilokitapahtumaan, jotka pyyntö aiheutti.

Yhdistettynä aikaleimoihin pyyntötunniste antaa mahdollisuuden muodostaa täydellisen kuvan mikropalveluiden kutsuhierarkiasta ja korreloida kaikki pyynnöt siihen ulkoiseen IP-osoitteeseen, josta alkuperäinen pyyntö tuli.

# Docker

Dockerin ja konttien orkestrointiin käytettävien ohjelmistojen (mm. Kubernetes) kovennus on laaja kokonaisuus, jota varten esimerkiksi Center for Internet Security (CIS) on luonut omat ohjeensa. Tässä ohjeessa keskitytään lähinnä arkkitehtuuri- ja ohjelmistosuunnittelutason aspekteihin. Docker- tai orkestrointityökalun varsinaisen asennuksen kovennus tulee toki myös varmistaa.

## Isäntäkone

Docker-kontteja ajavien isäntäkoneiden tulee olla vain tässä käytössä. Palvelimella ei saa ajaa muita palveluja.

Yhdellä palvelimella tulee ajaa ainoastaan saman turvatason kontteja: esimerkiksi ulkoisille käyttäjille näkyvät Docker-kontissa ajettavat palvelut tulee ajaa eri palvelimella kuin missä taustapalvelinkontteja ajetaan.

Orkestrointijärjestelmät voivat antaa mahdollisuuksia konttien asetteluun virtuaalikoneille. Esimerkiksi Kubernetes tukee ominaisuutta nimeltä ’node restriction’.

**Lähteet:**

* CIS Docker Community Edition Benchmark v1.1.0
* https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#node-isolation-restriction

## Docker-kontin asetukset

Yksittäisen Docker-kontin tulee pohjautua luotettavaan pohjakuvaan (base image), johon on kontin rakennusvaiheessa asennettu ohjelmistoja vain luotetuista lähteistä. Lähteiden luotettavuus voi perustua kryptografisten allekirjoitusten tarkastamiseen tai siihen, että ne ovat organisaation hallussa.

Docker-konttien pohjakuvaan tulee viitata SHA-256-tunnisteella, eikä esimerkiksi viitata uusimpaan versioon latest-määreellä. Tätä viittaustapaa on käytettävä myös silloin, kun pohjakuva on sisäisesti luotu.

Alla esimerkki hyväksytystä tavasta viitata julkisesta lähteestä saatavaan pohjaan:

FROM alpine@sha256:3dcdb92d7432d56604d4545cbd324b14e647b313626d99b889d0626de158f73a

Kun kontteja rakennetaan Dockerfilellä, kaikkien asennettavien ohjelmien eheydestä on varmistuttava. Jos Dockerfile hakee ohjelmiston ei-luotetusta lähteestä, saman Dockerfilen ei pitäisi hakea eheystarkistukseen tarvittavia avaimia niin ikään ei-luotetusta lähteestä.

Riippuvuudet tulisi mieluiten asentaa paikallisesta varastosta. Tämä mahdollistaa sen, että organisaatio voi kontrolloida käytettyjä riippuvuuksien versioita, pitää yllä muutoshallintaprosessia, ja rakentaa Docker-kontit vaikka kolmannen osapuolen varasto ei olisi saatavilla.

Docker-kontit tulee aina ajaa read-only -tiedostojärjestelmällä, ellei ole erittäin painavaa syytä päättää toisin.

Docker-konttia ei tule koskaan ajaa --privileged-asetuksella tai antaa kontille pääsyä docker.sock-hallintaliittymään. Lisäksi Docker-kontille tulee antaa pääsy mahdollisimman pieneen osaan isäntäkoneen tiedostojärjestelmää ja verkkoa.

Käytettäessä orkestrointijärjestelmää, joka tarjoaa nk. overlay-verkon, konttien verkkoliikenne tulisi rajata vain niihin kohteisiin, joiden kanssa niiden on pystyttävä kommunikoimaan. Esimerkiksi Kubernetes tuntee tämän nimellä NetworkPolicy.

Kun Dockeria tai orkestrointia käyttävälle järjestelmälle tehdään tietoturvatarkastus, Dockerfilen ja orkestrointijärjestelmän konfiguraation tarkastuksen on aina sisällyttävä tietoturvatarkastukseen.

**Lähteet:**

* CIS Docker Community Edition Benchmark
* https://kubernetes.io/docs/concepts/services-networking/network-policies/

## Salaisuuksienhallinta

Salaisuuksia, kuten salasanoja, salaamattomia yksityisiä avaimia tai rajapintatunnuksia (*API token*), ei tule koskaan tallentaa selväkielisenä esimerkiksi versionhallintaan tai Dockerfile-tiedostoon. Pitkäaikaiseen säilöön salaisuudet kannattaa tallettaa salaisuuksienhallintajöärjestelmällä. Näitä on erilaisia yksittäisen kehittäjän salasanamanagerista pilviasenteisiin holveihin (’vault’). Salaisuuksista tulisi myös olla varmuuskopio.

Salaisuuksien toimittaminen ajettaville konteille on ympäristökohtainen ongelmakenttä, eikä yleispäteviä ratkaisuja ole helppo antaa. Yleisesti voidaan sanoa, että salaisuuksien viennissä tuotantoon tulisi mieluiten käyttää holvityyppisiä (’vault’) palveluita tai orkestrointityökalun tarjoamia mahdollisuuksia. Mikäli näitä ei ole saatavilla, salaisuudet voidaan laittaa salattuun tiedostoon, joka tallennetaan sopivaan paikkaan, johon kontti pääsee käsiksi. Tarvittava salauksenpurkuavain voidaan toimittaa kontille ympäristömuuttujassa. Käytettäessä ympäristömuuttujia ne tulisi nollata niiden lukemisen jälkeen, koska monissa virhetilanteissa debug-tuloste sisältää ympäristön sisällön.

### Ympäristömuuttujat

Docker-kontille voi antaa ympäristömuuttujia komentorivillä, joka on suositeltu tapa salaisuuksien antamiseen kontille:

docker run -e API\_TOKEN=1234567890abcdef alpine

### Kolmannen osapuolen ohjelmistot

Salaisuuksien hallintaan on olemassa kolmansien osapuolien sovelluksia, kuten HashiCorp Vault. Kubernetesissa ja OpenShiftissä on ’secret’ -objekteja. Infrastruktuuripilvipalveluilla on kullakin oma avainten ja salaisuuksien hallintapalvelunsa. Orkestraattorin salaisuuksien hallinta on yleensä paras vaihtoehto.

**Lähteet:**

* <https://kubernetes.io/docs/concepts/configuration/secret/>
* <https://docs.openshift.com/container-platform/3.11/dev_guide/secrets.html>
* <https://www.vaultproject.io/>

# Ohjelmointirajapinnat (API)

Kaikki uudet web-ohjelmointirajapinnat tulisi toteuttaa käyttäen JSON-formaattia. Tätä puoltavat yksinkertainen esitystapa ja laaja ohjelmistotuki. Yleisesti käytetyissä JSONP- ja XML-formaateissa on molemmissa ongelmansa.

* JSONP-muotoisia rajapintoja ei tulisi käyttää, koska tällöin suoritetaan rajapinnan tarjoamaa dataa suoraan ohjelmakoodina. Tällöin erotus rajapinnan ja rajapintaa käyttävän palvelun välillä puuttuu täysin. Rajapinnan tulisi tarjota ainoastaan dataa, jota palvelu käsittelee.
* XML-jäsentimien (parser) on useasti osoitettu toimivan virheellisesti mahdollistaen datan tulkitsemisen usealla eri tavalla. Tämä muodostuu ongelmaksi, jos dataa tulkitaan useassa eri vaiheessa käyttäen eri jäsentimiä, jolloin dataa saatetaan tulkita useammalla eri tavalla. XML mahdollistaa myös tietosisällön esittämisen usealla eri esitystavalla, jonka jäsennin kuitenkin tulkitsee samalla tavalla.

## API-turvallisuus

Ohjelmistorajapintojen tulisi varmistua jokaisesta sisään tulevasta kutsusta alla mainituista asioista. Arkkitehtuurista riippuen API-toteutus saattaa luottaa välissä olevaan API-välityspalvelimeen tai tehdä nämä tarkistukset itse.

Mikäli jokin tarkistus ei mene läpi, kutsu kannattaa suoraan hylätä ilman, että rajapinnan toteuttava osapuoli lähtisi arvailemaan, mitä kutsuja on tarkoittanut. Mikäli todennus-, valtuutus- tai eheystarkistukset epäonnistuvat, kutsun sisältöä ei pitäisi lähteä lainkaan jäsentämään pidemmälle vaan prosessointi tulisi päättää mahdollisimman tehokkaasti.

* Istuntotunniste: onko istunto voimassa? Mikä on istunnon valtuutuksen taso? (Ks. tämän ohjeen istuntotunnisteisiin liittyvä erityisohje.)
* Kutsun ja tietojen eheys. Varmistutaan siitä, että kutsu on eheä. Jos käytetään API-avainta tai muuta kutsujan ja rajapinnan välistä salaisuutta, eheystarkistus voidaan sitoa siihen. Tyypillisiä toteutuksia ovat valtuutusten siirtoon JSON Web Tokens (JWT) ja kokonaisten rajapintakutsujen eheystarkistuksiin AWS Signature v4.
* Onko sisään tuleva kutsu oikein muotoiltu? Myös JSON-pohjaisten rajapintojen tapauksessa kutsu tulisi tarkistaa skeemaa vasten, jotta rajapintaan ei esimerkiksi pysty syöttämään ylimääräisiä tietokenttiä. Ellei skeematarkistusta tehdä, nämä ylimääräiset kentät päätyvät erityisesti NoSQL-tapauksissa usein tietokantaan asti.
* Valtuutus. Varmistutaan siitä, että kutsujalla on oikeus ensinnäkin tehdä kysely, ja toiseksi saada kyselyn perusteella tulevat tiedot.
* Onko kutsu tarpeen lokittaa? (Ks. tämän ohjeen kappale auditointilokituksesta.)
* Asetetaanko vastauksessa kaikki tarpeelliset HTTP-otsakkeet, kuten HSTS ja Content-Type? (Ks. tämän ohjeen kappale HTTP-otsakkeista.)

Mikäli valtuutuksien siirtoon käytetään JWT:tä, sen toteutuksessa on varmistettava, että sekä HMAC-SHA-256-laskennassa käytettävä salaisuus on 256-bittinen. Tämän ohjeen istuntotunnisteita käsittelevässä kappaleessa käsitellään salaisuuksien luontia.

**Lähteet:**

* JSON Web Tokens -teemasivusto <https://jwt.io/>
* AWS Signature v4, joka ei ole AWS-spesifinen https://docs.aws.amazon.com/general/latest/gr/sigv4\_signing.html

## Cross-Origin Resource Sharing (CORS)

CORS on tapa, jolla palvelin ilmoittaa selaimelle, että se saa ohittaa normaalin nk. same-origin -rajoituksen ja kutsua toisessa domainissa sijaitsevaa APIa. Selaimet tekevät CORS-tarkistusta varten erillisen nk. preflight-kutsun, jonka perusteella saadaan tieto CORS-otsakkeista. On tärkeää huomata, että hyökkääjä voi tietenkin aina kutsua rajapintaa ilman selainta riippumatta CORSista. Palvelinpään pääsynhallinta tulee aina toteuttaa käyttäen oikeita pääsynhallintamekanismeja.

Kohdepalvelin ilmoittaa CORS-politiikkansa käyttäen Access-Control-Allow-Origin -otsaketta. Access-Control-Allow-Origin määrittelee ne JavaScript-koodin lähteet, jotka saavat kyseistä rajapintaa kutsua.

Jos esimerkiksi organisaatiolla on web-sovellus osoitteessa app.company.com ja API osoitteessa api.cloud.example, ja he haluavat app.company.comilta toimitetun JavaScriptin pystyä kutsumaan api.cloud.examplea, api.cloud.examplen rajapinnan on palautettava jotakin tämänkaltaista HTTP-otsakkeissaan:

Access-Control-Allow-Origin: app.company.com  
Access-Control-Allow-Methods: GET POST PUT DELETE

Mikäli rajapinta tarvitsee valtuutuksen (esimerkiksi evästeissä), sen tulee myös antaa selaimelle erityislupa näiden toimitukseen:

Access-Control-Allow-Credentials: true

Jos APIa pitää pystyä kutsumaan mistä tahansa JavaScript-koodista riippumatta sen alkuperästä, otsakkeen tulee olla

Access-Control-Allow-Origin: \*

Mikäli tämä otsake on näkyvissä rajapinnan otsakkeissa, mutta rajapinta ei ole tarkoitettu yleisesti kutsuttavaksi, CORS-politiikka on todennäköisesti liian lepsu.

**Lähteet:**

* https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
* https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin\_policy

## CSRF-tunniste

Cross-Site Request Forgery (CSRF) -hyökkäyksessä hyökkäävä sivusto aiheuttaa selaimen kautta pyynnön jotakin toista sivustoa kohden. Jos selaimella on aktiivinen istunto kohdesivustolla, hyökkääjä voi hyökkääjä tehdä käyttäjänä erilaisia operaatioita.

CSRF estetään antamalla jokaisen HTTP-pyynnön mukana kryptografisesti vahva ja riittävän pitkä salaisuus. Nämä salaisuudet (nk. CSRF-tokenit) tallennetaan selaimessa niin, että hyökkääjän sivusto tai siltä peräisin oleva JavaScript ei pääse niihin käsiksi, eikä siten pysty tekemään tekaistuja pyyntöjä.

CSRF on helpointa estää käyttämällä sovelluskehikkoa, joka toteuttaa CSRF-suojauksen automaattisesti. Useimmiten tämä on paras vaihtoehto.

Mikäli tällaista kehikkotukea ei ole, suositeltava tapa on käyttää nk. ”double submit cookie” -tapaa. Tämä metodi on kätevä, koska se on tilaton eikä vaadi palvelinpään tallennustilaa, joten se sopii myös tilanteisiin, jossa pyynnöt selaimelta tulevat satunnaisesti eri palvelimille kuormantasaajan läpi. Kuten monissa muissakin tietoturvaominaisuuksissa, tämän naiivi toteutus saattaa olla haavoittuva.

**Sources:**

* <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet#Double_Submit_Cookie>

## Parametrit ja luottamuksellinen tieto

HTTP-rajapintakyselyt tulee muotoilla niin, ettei henkilötietoja tai muita luottamuksellisia tietoja välitetä URL-osoitteen osana. URL-osoitteen osana olevat tiedot (ns. GET*-*parametrit) saattavat jäädä talteen lokeihin, selaimen välimuistiin ja selaushistoriaan tai välityspalvelimen tietoihin.

Kaikki henkilötiedot tulee täten välittää HTTP POST -pyynnön parametreina.

Oikea tapa voisi näyttää esimerkiksi tältä:

POST /hae/osoite HTTP/1.1  
  
hetu=010170-123F

Väärä tapa voisin näyttää esimerkiksi tältä:

GET /hae/010170-123F/osoite HTTP/1.1