

**Collaboard Security**

OVERVIEW

|  |  |
| --- | --- |
| Title: | CollaBoard Security |
| Abstract: | This document details the specification for the security implemented in CollaBoard project |
| Status\* | Approved |
| Document Owner: | IBV |
| Document Author: | Gian Paolo Santopaolo |
| \*Draft, In progress, Reviewed, Authorized, Approved, Completed | |

|  |  |  |
| --- | --- | --- |
| **Document Control** | | |
| Document Distribution: | Internal: | Gian Paolo Santopaolo, Gianluca Centulani |
|  | External: | Whom of interest |
| Document Authorization: | Authorizer(s): |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Change History** | | | |
| Date | Ver | Changed By | Details |
| 06/11/2019 | 1.0 | GP Santopaolo | Creted the first version based on the NDA in depth security document |
| 15/2/2021 | 1.1 | D. Stellakis | Added information regarding OAuth and SAML external login support |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Index

[Introduction 3](#_Toc55572083)

[Purpose of the document 3](#_Toc55572084)

[An overview to Security in collaboard 4](#_Toc55572085)

[API Security 4](#_Toc55572086)

[Authenticating using an external login provider 5](#_Toc55572087)

[Two-Factor Authentication (2FA) with OTP 5](#_Toc55572088)

[Roles 5](#_Toc55572089)

[Security on the database 5](#_Toc55572090)

[File Security 6](#_Toc55572091)

[Security on Cloud 6](#_Toc55572092)

[File Security on-premises 7](#_Toc55572093)

[Vulnerability Scans 8](#_Toc55572094)

[Monitoring security health 8](#_Toc55572095)

[Security on Continuous Integration and Continuous Delivery 9](#_Toc55572096)

[Bibliography 11](#_Toc55572097)

# Introduction

## Purpose of the document

This document aims to give an overview of the security used in CollaBoard running on Azure and on-premises environments

Further questions can be answered by our engineers.

# An overview to Security in collaboard

## API Security

The client and server communicate with each other in two different ways:

* Via Standard Web APIs
* SignalR (web sockets)

All the communication is secured with the process described below.

Every method in the API is protected by means of authentication and authorization. In order for someone to use the API, they need to have acquired a valid authorization token, which they would include in the API method request as a Bearer token in the Authentication HTTP header.

The authorization token itself is a JSON Web Token (JWT). JWT is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. JSON Web Tokens are useful in the following scenaria:

* Authorization: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.
* Information Exchange: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

The API endpoints only work over HTTPs, so the communication between the client and the server is secure and encrypted. Each user, when he wants to access an API method, must first login by calling a special API endpoint, and passing their username and password. In return, the client gets back a JWT token, which it shall use in every subsequent API request. In this way, the user login credentials get send only once to the API, and from there after a signed JWT token is used. This token contains information about the logged-in user, but now his password. The token is also designed to be short-lived, so that even if an attacked somehow managed to get access to the token, it would expire shortly afterwards and it would be of no use to him.

The API, once it receives a request, checks if it contains a valid and not expired JWT token. If not, an error will be returned to the client. If the token is valid, the call can proceed and the API will then check if the user contained in the token has access to call the specified API endpoint (authorization), by checking if the user has the specific role needed by the endpoint in order to work.

If the user does not want to provide his own credentials to Collaboard, they can sign in using a third-party provider, such as Google or Microsoft. By doing that, the user authenticates in the external service and Collaboard does not get access to their password. The flow is the same, since after authentication the client is again going to get back a JWT token.

## Authenticating using an external login provider

Users can also authenticate with the app using third party authentication providers, such as Google or Microsoft. The authentication process involves two steps: at first, the user logins with the external login provider and the app receives an authorization code. The app sends this code to the web api, and the web api exchanges it for an access token. The web api then uses this access token to verify the user and access their details.

In order for the external provider authentication to work, each application that wishes to use it shall be assigned an app code, to uniquely identify the app and for the server to be able to load the configuration settings for this app. For instance, if the React web app wishes to use the external provider authentication functionality, we may assign it the code react-web and it shall pass it onto each web api call.

Besides the commonly known authentication providers, such as Google, Microsoft and Apple, our system supports integration with external authentication providers that support the OAuth+OpenID standard, as well as the SAML 2.0 standard. Regarding OAuth, we support the Authorization Code flow, in which a user signs in with their identity provider and gets redirected back to Collaboard with an authorization code, which is then exchanged by our server for an access token. Regarding SAML, we support the SP-initiated flow, which works in a similar manner. For both flows, all that is needed is to configure our system with the specifics of the OAuth or SAML identity provider respsctively, such as the redirect URLs, the client IDs etc.

## Two-Factor Authentication (2FA) with OTP

In case the user opts for two-factor authentication (2FA), the client app should prompt him to enter the one-time code (OTP) that he has obtained using a client authenticator app (such as Google Authenticator or Microsoft Authenticator), or that was sent to him via email from the web api.

## Roles

Users can have roles, and these roles determine their access levels for the operations and data of the system.

If a user does not have a role, it is assumed that he is a simple user.

## Security on the database

We use encryption to store hashed password in our database.

Our password are hashed and sotored into the database with the following steps:

* Create a salt of 16 bytes or longer.
* Feed the salt and the password into the hashing algorithm, which is PBKDF2.
* Use HMAC-SHA-256 as the core hash inside PBKDF2.
* Perform 80000 iterations
* Take the output from PBKDF2 as the final password hash.
* Store the iteration count, the salt and the final hash the database.

This process can be extended to use a different hashing algorithm, or more iterations, as computer systems become more powerful and can calculate hash values more quickly, in order to keep the time the final hash value is produced more or less constant.

# File Security

Our application can run both on the Cloud or on-premises.

Different security approaches are usd to achive the best security level using native envitonment features:

## Security on Cloud

On Azure, we grant limited access to Azure Storage resources using shared access signatures (SAS).

A shared access signature (SAS) provides secure delegated access to resources in your storage account without compromising the security of your data. With a SAS, you have granular control over how a client can access your data. You can control what resources the client may access, what permissions they have on those resources, and how long the SAS is valid, among other parameters.

A shared access signature is a signed URI that points to one or more storage resources and includes a token that contains a special set of query parameters. The token indicates how the resources may be accessed by the client. One of the query parameters, the signature, is constructed from the SAS parameters and signed with the key that was used to create the SAS. This signature is used by Azure Storage to authorize access to the storage resource.

We use SAS to provide secure access to resources in our storage account to any client who does not otherwise have permissions to those resources.

A common scenario where a SAS is useful is a service where users read and write their own data to your storage account. In a scenario where a storage account stores user data, we a design pattern shown in the diagram below.

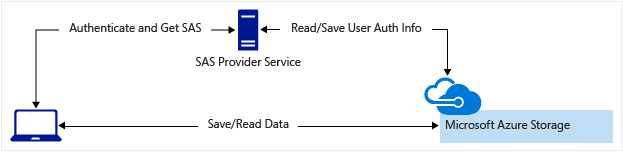


Figure 9 Security for files on cloud diagram

CollaBoard service authenticates the client and then generates a SAS for each project. That way the client even if authenticated it can only access the project it got authenticated for and not all our cloud storage where also all the other project are stored. Once the client application receives the SAS, they can access storage account resources directly with the permissions defined by the SAS and for the interval allowed by the SAS. The SAS mitigates the need for routing all data through the front-end proxy service.

After the CollaBoard client authenticated, typed the username and the password for the project it wants to access, it receives the SAS token that will only allow access to resources for the project it authenticated.

At this point the application, in background, starts do download/upload all the files on the resource it got access.

## File Security on-premises

Instead of what happened on the Cloud, in an on-premises environment, all the resources need to be routed from our Web API. Explicitly for this purpose, we developed a managed file transfer solution (for all the insight on the MFT solution please refer to the “MFT FDS.docx” document).

The authentication process used in Collaboard is also used in MFT (refer to chapter API security).

Once the user is guaranteed to log in the application and access the specified project it can then upload and download files.

The user does not have any rights on the network share where the files are stored it can only access those files throught the app.

No real user has direct access to the source network folder where files are stored, only the server part and its worker process can access them thanks to the AD credential the server processes are running.

Depending on each customer’s security rules and compliance the access to the network share where CollaBoard’s files are stored can be guaranteed only to MFT services or also to real humans. Usually our recommendation is to give access to the network share only to the CollaBoard/MFT processes.

The same applies when our software is delivered with containers running on Linux host OS.

# Vulnerability Scans

For the Azure envinroments we constanly monitor the security state of our Azure resources,

When the Security Center identifies potential security vulnerabilities, it creates recommendations that guide you through the process of configuring the needed controls. Recommendations apply to Azure resource types: virtual machines (VMs) and computers, applications, networking, SQL, and Identity and Access.

## Monitoring security health

We monitor the security state of your resources on the **Security Center – Overview** dashboard. The Resources section provides the number of issues identified and the security state for each resource type.

# Security on Continuous Integration and Continuous Delivery

We integrate the OWASP Dependency Check in to development process and in our CI/CD processes.

OWASP Dependency Check is a well known open-source tool which track dependencies in project and identify components with known published vulnerabilities.

OWASP dependency-check is an open source solution the OWASP Top 10 2013 entry: A9 - Using Components with Known Vulnerabilities. Dependency-check can currently be used to scan Java and .NET applications to identify the use of known vulnerable components. Experimental analyzers for Python, Ruby, PHP (composer), and Node.js applications; these are experimental due to the possible false positive and false negative rates. To use the experimental analyzers they must be specifically enabled via the appropriate experimental configuration. In addition, dependency-check has experimental analyzers that can be used to scan some C/C++ source code, including OpenSSL source code and projects that use Autoconf or CMake.

Dependency-check works by collecting information about the files it scans (using Analyzers). The information collected is called Evidence; there are three types of evidence collected: vendor, product, and version. For instance, the JarAnalyzer will collect information from the Manifest, pom.xml, and the package names within the JAR files scanned and it has heuristics to place the information from the various sources into one or more buckets of evidence.

These CPE entries are read “cpe:/[Entry Type]:[Vendor]:[Product]:[Version]:[Revision]:…”. The CPE data is collected and stored in a Lucene Index. Dependency-check then use the Evidence collected and attempt to match an entry from the Lucene CPE Index. If found, the CPEAnalyzer will add an Identifier to the Dependency and subsequently to the report. Once a CPE has been identified the associated CVE entries are added to the report.

One important point about the evidence is that it is rated using different confidence levels - low, medium, high, and highest. These confidence levels are applied to each item of evidence. When the CPE is determined it is given a confidence level that is equal to the lowest level confidence level of evidence used during identification. If only highest confidence evidence was used in determining the CPE then the CPE would have a highest confidence level.

Because of the way dependency-check works both false positives and false negatives may exist. Please read How to read the report to get a better understanding of sorting through the false positives and false negatives.

Dependency-check does not currently use file hashes for identification. If the dependency was built from source the hash likely will not match the “published” hash. While the evidence based mechanism currently used can also be unreliable the design decision was to avoid maintaining a hash database of known vulnerable libraries. A future enhancement may add some hash matching for very common well known libraries (Spring, Struts, etc.).

The OWASP Dependency Check Azure DevOps Extension enables the following features in an Azure Build Pipeline:

* Software composition analysis runs against package references during each build
* Export vulnerability data to HTML, JSON, XML, CSV, JUnit formatted reports
* Download vulnerability reports from the build's artifacts

Security checks performed against the following databases:

This report contains data retrieved from the National Vulnerability Database.

This report may contain data retrieved from the NPM Public Advisories.

This report may contain data retrieved from RetireJS.

This report may contain data retrieved from the Sonatype OSS Index.

For a detailed report of Dependency Check run as part of our CI/CD pipeline on CollaBoard Services please refer to the attached document “Collaboard Security - Dependency Check Report.html”

# Bibliography

Asymmetric Encryption, Symmetric Encryption, Asymmetric Encryption Vs Symmetric Encryption, SSL Encryption - Digicert https://www.digicert.com/ssl-cryptography.htm

Vulnerability Scans Microsoft https://docs.microsoft.com/en-us/azure/security-center/security-center-virtual-machine-protection

Mattew King - Password security best practices

Paul Ducklin - Serious Security: How to store your users’ passwords safely

The best way to store secrets in your app is not to store secrets in your app - Laurent Jalbert Simard

https://medium.com/poka-techblog/the-best-way-to-store-secrets-in-your-app-is-not-to-store-secrets-in-your-app-308a6807d3ed

Hardware security module – Wikipedia

https://en.wikipedia.org/wiki/Hardware\_security\_module

Where to store the private key? - softwareengineering.stackexchange.com

https://softwareengineering.stackexchange.com/questions/220950/where-to-store-the-private-key