

**Collaboard Security**

how the security is implemented in collaboard

|  |  |
| --- | --- |
| 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: | Collins Aerospace |
| Document Authorization: | Authorizer(s): |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Change History** | | | |
| Date | Ver | Changed By | Details |
| 01/07/2019 | 1.0 | GP Santopaolo | Initial Draft |
| 02/07/2019 | 1.1 | GP Santopaolo | Added server layer security chapter |
| 03/07/2019 | 1.2 | GP Santopaolo | Added client layer security chapter |
| 04/07/2019 | 1.3 | GP Santopaolo | Added vulnerability chapter and full review |
| 10/07/2019 | 1.3 | A. Gorog | Document review and approval |
| 02/03/2020 | 1.4 | GP Santopaolo | Added new authentication description |
| 27/10/2020 | 1.5 | GP Santopaolo | Added new chapter storing secrets |
| 15/2/2021 | 1.6 | D. Stellakis | Added chapters for OAuth and SAML |

Index

[Introduction 4](#_Toc55549567)

[Purpose of the document 4](#_Toc55549568)

[Key security concepts and algorithms 5](#_Toc55549569)

[Asymmetric Encryption 5](#_Toc55549570)

[Public-Key Encryption Algorithms 5](#_Toc55549571)

[RSA 5](#_Toc55549572)

[ECC 6](#_Toc55549573)

[Pre-Shared Key Encryption Algorithms 6](#_Toc55549574)

[Symmetric Encryption 6](#_Toc55549575)

[Asymmetric Encryption Vs. Symmetric Encryption 7](#_Toc55549576)

[Web API Security 8](#_Toc55549577)

[SSL Encryption for the communication layer 8](#_Toc55549578)

[Database security 8](#_Toc55549579)

[Users’ passwords should not be recoverable from the database. 9](#_Toc55549580)

[Identical, or even similar, passwords should have different hashes. 9](#_Toc55549581)

[The database should give no hints as to password lengths. 10](#_Toc55549582)

[Avoid brute force 10](#_Toc55549583)

[Recommendation for safe password storage storage 10](#_Toc55549584)

[Implementation in our code 10](#_Toc55549585)

[Our password encryption in a nutshell 12](#_Toc55549586)

[Storing secret keys 12](#_Toc55549587)

[The safest way of storing a secret 13](#_Toc55549588)

[Alternative way 14](#_Toc55549589)

[Credentials-less service access 14](#_Toc55549590)

[Current state and future plans 15](#_Toc55549591)

[API Security 16](#_Toc55549592)

[Authentication modes 17](#_Toc55549593)

[Update the user's profile 18](#_Toc55549594)

[Authenticating using local accounts 19](#_Toc55549595)

[User registration 19](#_Toc55549596)

[User verification 20](#_Toc55549597)

[Login 21](#_Toc55549598)

[Password reset 21](#_Toc55549599)

[Token refresh 22](#_Toc55549600)

[Authenticating using an external login provider 22](#_Toc55549601)

[External user authentication 23](#_Toc55549602)

[External user registration 24](#_Toc55549603)

[Token refresh 25](#_Toc55549604)

[Two-Factor Authentication (2FA) with OTP 25](#_Toc55549605)

[Changing Authentication Mode for a User 25](#_Toc55549606)

[Getting OTP codes without the need of an authenticator app 26](#_Toc55549607)

[Roles 26](#_Toc55549608)

[Integration with Web API projects 27](#_Toc55549609)

[DotNet Core 27](#_Toc55549610)

[SignalR Authenticaiton 29](#_Toc55549611)

[File Security 31](#_Toc55549612)

[Security on Cloud 32](#_Toc55549613)

[Vulnerability Scans 35](#_Toc55549614)

[Monitoring security health 35](#_Toc55549615)

[Unmonitored VMs and computers 36](#_Toc55549616)

[Recommendations 36](#_Toc55549617)

[VMs and computers 37](#_Toc55549618)

[App services 37](#_Toc55549619)

[Vulnerability Scans for CollaBoard Azure resources 40](#_Toc55549620)

[Continuous Security - OWASP Dependency-Check 42](#_Toc55549621)

[Bibliography 44](#_Toc55549622)

# Introduction

## Purpose of the document

This document details the security used in CollaBoard running on Azure and On-premise environments

All the aspect of security, from client to server, including database are listed in this document.

Further questions can be answered by our engineers.

## Key security concepts and algorithms

## Asymmetric Encryption

Asymmetric encryption (or public-key cryptography) uses a separate key for encryption and decryption. Anyone can use the encryption key (public key) to encrypt a message. However, decryption keys (private keys) are secret. This way, only the intended receiver can decrypt the message. The most common asymmetric encryption algorithm is RSA; however, we will discuss algorithms later in this article.

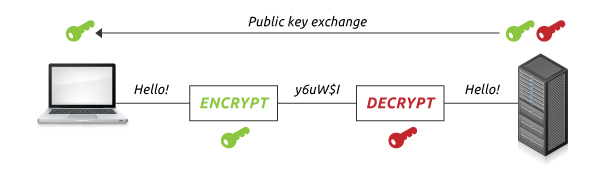


Figure 1 Asymmetric Encryption diagram

Asymmetric keys are typically 1024 or 2048 bits. However, keys smaller than 2048 bits are no longer considered safe to use. 2048-bit keys have enough unique encryption codes that we won’t write out the number here (it’s 617 digits). Though larger keys can be created, the increased computational burden is so significant that keys larger than 2048 bits are rarely used. To put it into perspective, it would take an average computer more than 14 billion years to crack a 2048-bit certificate

### Public-Key Encryption Algorithms

Public-key cryptography (asymmetric) uses encryption algorithms like RSA and Elliptic Curve Cryptography (ECC) to create the public and private keys. These algorithms are based on the intractability\* of some mathematical issues.

With asymmetric encryption, it is computationally easy to generate public and private keys, encrypt messages with the public key, and decrypt messages with the private key. However, it is extremely difficult (or impossible) for anyone to derive the private key based only on the public key.

### RSA

RSA is based on the presumed difficulty of factoring large integers (integer factorization). Full decryption of an RSA ciphertext is thought to be infeasible on the assumption that no efficient algorithm exists for integer factorization.

A user of RSA creates and then publishes the product of two large prime numbers, along with an auxiliary value, as their public key. The prime factors must be kept secret. Anyone can use the public key to encrypt a message, but only someone with knowledge of the prime factors can feasibly decode the message.

RSA stands for Ron Rivest, Adi Shamir, and Leonard Adleman— the men who first publicly described the algorithm in 1977.

### ECC

Elliptic curve cryptography (ECC) relies on the algebraic structure of elliptic curves over finite fields. It is assumed that discovering the discrete logarithm of a random elliptic curve element in connection to a publicly known base point is impractical.

The use of elliptic curves in cryptography was suggested by both Neal Koblitz and Victor S. Miller independently in 1985; ECC algorithms entered common use in 2004.

The advantage of the ECC algorithm over RSA is that the key can be smaller, resulting in improved speed and security. The disadvantage lies in the fact that not all services and applications are interoperable with ECC-based SSL Certificates.

### Pre-Shared Key Encryption Algorithms

Pre-shared key encryption (symmetric) uses algorithms like Twofish, AES, or Blowfish, to create keys—AES currently being the most popular. All of these encryption algorithms fall into two types: stream ciphers and block ciphers. Stream ciphers apply a cryptographic key and algorithm to each binary digit in a data stream, one bit at a time. Block ciphers apply a cryptographic key and algorithm to a block of data (for example, 64 sequential bits) as a group. Block ciphers are currently the most common symmetric encryption algorithm.

## Symmetric Encryption

Symmetric encryption (or pre-shared key encryption) uses a single key to both encrypt and decrypt data. Both the sender and the receiver need the same key to communicate.

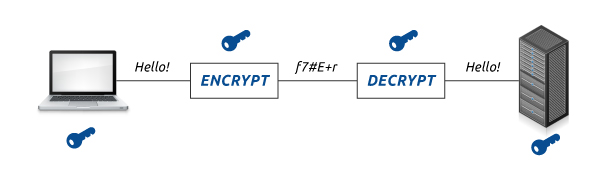


Figure 2 Symmetric Encryption diagram

Symmetric key sizes are typically 128 or 256 bits—the larger the key size, the harder the key is to crack. For example, a 128-bit key has 340,282,366,920,938,463,463,374,607,431,768,211,456 encryption code possibilities. As you can imagine, a ‘brute force’ attack (in which an attacker tries every possible key until they find the right one) would take quite a bit of time to break a 128-bit key.

Whether a 128-bit or 256-bit key is used depends on the encryption capabilities of both the server and the client software. SSL Certificates do not dictate what key size is used.

## Asymmetric Encryption Vs. Symmetric Encryption

Since asymmetric keys are bigger than symmetric keys, data that is encrypted asymmetrically is tougher to crack than data that is symmetrically encrypted. However, this does not mean that asymmetric keys are better. Rather than being compared by their size, these keys should be compared by the following properties: computational burden and ease of distribution.

Symmetric keys are smaller than asymmetric, so they require less computational burden. However, symmetric keys also have a major disadvantage—especially if you use them for securing data transfers. Because the same key is used for symmetric encryption and decryption, both you and the recipient need the key. If you can walk over and tell your recipient the key, this isn’t a huge deal. However, if you have to send the key to a user halfway around the world (a more likely scenario) you need to worry about data security.

Asymmetric encryption doesn’t have this problem. As long as you keep your private key secret, no one can decrypt your messages. You can distribute the corresponding public key without worrying who gets it. Anyone who has the public key can encrypt data, but only the person with the private key can decrypt it.

# Web API Security

All the CollaBoard Client/Service communication are performed through HTTPS which means SSL encryption through the TCP channel.

## SSL Encryption for the communication layer

SSL (Secure Sockets Layer) is a standard security technology for establishing an encrypted link between a server and a client—typically a web server (website) and a browser/client. It allows sensitive information such as credit card numbers, social security numbers, and login credentials to be transmitted securely. To establish this secure connection, the client/browser and the server need an SSL Certificate.

Public Key Infrastructure (PKI) is the set of hardware, software, people, policies, and procedures that are needed to create, manage, distribute, use, store, and revoke digital certificates. PKI is also what binds keys with user identities by means of a Certificate Authority (CA). PKI uses a hybrid cryptosystem and benefits from using both types of encryption. For example, in SSL communications, the server’s SSL Certificate contains an asymmetric public and private key pair. The session key that the server and the browser create during the SSL Handshake is symmetric. This is explained further in the diagram below.

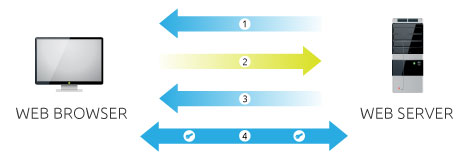


Figure 3 SSL Encryption diagram

1. **The server** sends a copy of its asymmetric public key.
2. **Client/Browser** creates a symmetric session key and encrypts it with the server's asymmetric public key. Then sends it to the server.
3. **The server** decrypts the encrypted session key using its asymmetric private key to get the symmetric session key.
4. **Server** and **Client/Browser** now encrypt and decrypt all transmitted data with the symmetric session key. This allows for a secure channel because only the browser and the server know the symmetric session key, and the session key is only used for that session. If the browser were to connect to the same server the next day, a new session key would be created.

# Database security

Database security is a hot topic these days, especially with all the new and seemingly never-ending security compliance requirements being imposed, such as GDPR.

To fulfill the GDPR recommendation and to have a high standard on sored password, we started from three initial requirements:

1. Users’ passwords should not be recoverable from the database.
2. Identical, or even similar, passwords should have different hashes.
3. The database should give no hints as to password lengths.
4. Avoid brute force

## Users’ passwords should not be recoverable from the database.

Hashing passwords is the best thing to do. Using a general-purpose hash function (such as MD5, SHA, etc.) is NOT. In the past, 'conventional wisdom' was that hashing+salting your passwords were the way to go. These days, if you use a general-purpose hash function, you're doomed. Modern GPUs can calculate hashes very, very fast. Anyone who wants to throw some GPUs at your password database will be able to brute force billions of passwords per second. All the salt in the world won't save you at this point - nobody bothers with rainbow tables anymore. Brute force will have you cracked in no time.

You should use a hash function that was DESIGNED FOR PASSWORD HASHING! The best examples are PBKDF, BCRYPT, and SCRYPT. These hash functions are designed to be slow, and they are designed so that they can be slowed down in the future (by increasing their 'work factor'), so that they keep pace with growing hardware power. As an example, one of these functions may take 100ms to hash a single password. Compare this to the general-purpose functions above, where millions or billions of passwords can be hashed in the same unit of time. Your users won't care if there's 100ms worth of overhead on the login process (which they only have to go through on an infrequent basis). Crackers who have compromised your password database WILL care when they are stuck trying to brute force 10 passwords per second, instead of the billions of passwords per second that they would otherwise be able to try when a general-purpose hash function was used

using PBKDF, as it has an implementation built in to the .NET base class library (Rfc2898DeriveBytes). This means that you can safely assume the IMPLEMENTATION of the algorithm is correct and secure, without having to audit it yourself. A lot of .NET developers use PBKDF for this reason, rather than having to constantly audit and verify third-party BCRYPT/SCRYPT implementations.

## Identical, or even similar, passwords should have different hashes.

Indeed, the text password will always come out as 5E884898DA28..EF721D1542D8, whenever anyone chooses it.

That means the crooks can pre-calculate a table of hashes for popular passwords – or even, given enough disk space, of all passwords up to a certain length – and thus crack any password already on their list with a single database lookup.

We can adapt the hash that comes out for each password by mixing in some additional data known as a **salt**, so-called because it “seasons” the hash output.

A salt is also known as a *nonce*, which is short for “number used once.”

Simply put, we generate a random string of bytes that we include in our hash calculation along with the actual password.

The easiest way is to put the salt in front of the password and hash the combined text string.

The salt is not an encryption key, so it can be stored in the password database along with the username – it serves merely to prevent two users with the same password getting the same hash.

For that to happen, they would need the same password and the same salt, so if we use 16 bytes or more of salt, the chance of that happening is small enough to be ignored.

## The database should give no hints as to password lengths.

Having chosen the way of hashing then the third requirement is implicitly satisfied because the hashes are all the same length, so we aren’t leaking any data about the size of the password.

Also, because we can predict in advance how much password data we will need to store for each password, there is now no excuse for needlessly limiting the length of a user’s password. (All SHA-256 values have 256 bits, or 32 bytes.)

## Avoid brute force

Although we have satisfied the three requirements (non-reversibility, no repeated hashes, and no hint of password length), the hash we have chosen – a single SHA-256 of salt+password – can be calculated very rapidly.

In fact, even hash-cracking servers that cost under $20,000 five years ago could already compute 100,000,000,000 or more SHA-256 hashes each second.

The nature of a cryptographic hash means that attackers can’t go backward, but with a bit of luck – and some poor password choices – they can often achieve the same result simply by trying to go forwards over and over again.

Indeed, if the crooks manage to steal your password database and can work offline, there is no limit other than CPU power to how fast they can guess passwords and see how they hash.

It, therefore, makes sense to slow down offline attacks by running our password hashing algorithm as a loop that requires thousands of individual hash calculations.

That won’t make it so slow to check an individual user’s password during login that the user will complain, or even notice.

But it will reduce the rate at which a crook can carry out an offline attack, in direct proportion to the number of iterations you choose.

We need to slow things down a bit to stymie the crackers, as of March 2019 the recommendation is 80.000 iterations

## Recommendation for safe password storage storage

In summary, here is our recommendation for safe storage of your users’ passwords:

* Use a strong random number generator to create a salt of 16 bytes or longer.
* Feed the salt and the password into the PBKDF2 algorithm.
* Use HMAC-SHA-256 as the core hash inside PBKDF2.
* Perform 80.000 iterations or more [March 2019].
* Take 32 bytes (256 bits) of output from PBKDF2 as the final password hash.
* Store the iteration count, the salt and the final hash in your password database.
* Increase your iteration count regularly to keep up with faster cracking tools.

## Implementation in our code

Store the following in our user database:

* Password salt
* Password hash

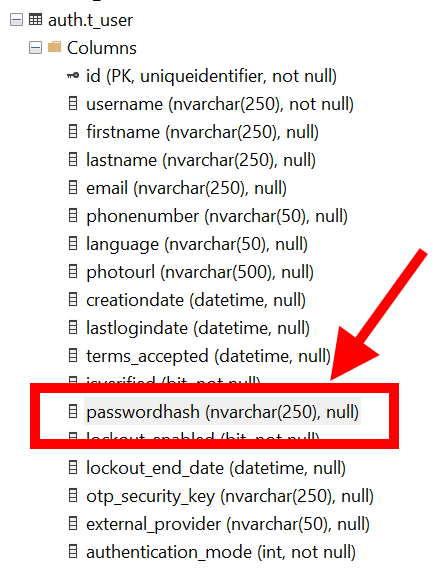


Figure 4 CollaBoard Database with the new salt and hash columns

Both password has and salt are stored together into the passowrdhash field serialized as byte arrays to increase a bit more the difficulty level in case of a breach.

The Iterations/work factor inside the configuration and they can be easly updated by need without the need of a new deployment.

When a user creates an account:

* Generate a new salt.
* Generate a hash using the generated salt and the provided password.
* Save the salt, hash, and work factor in the database.

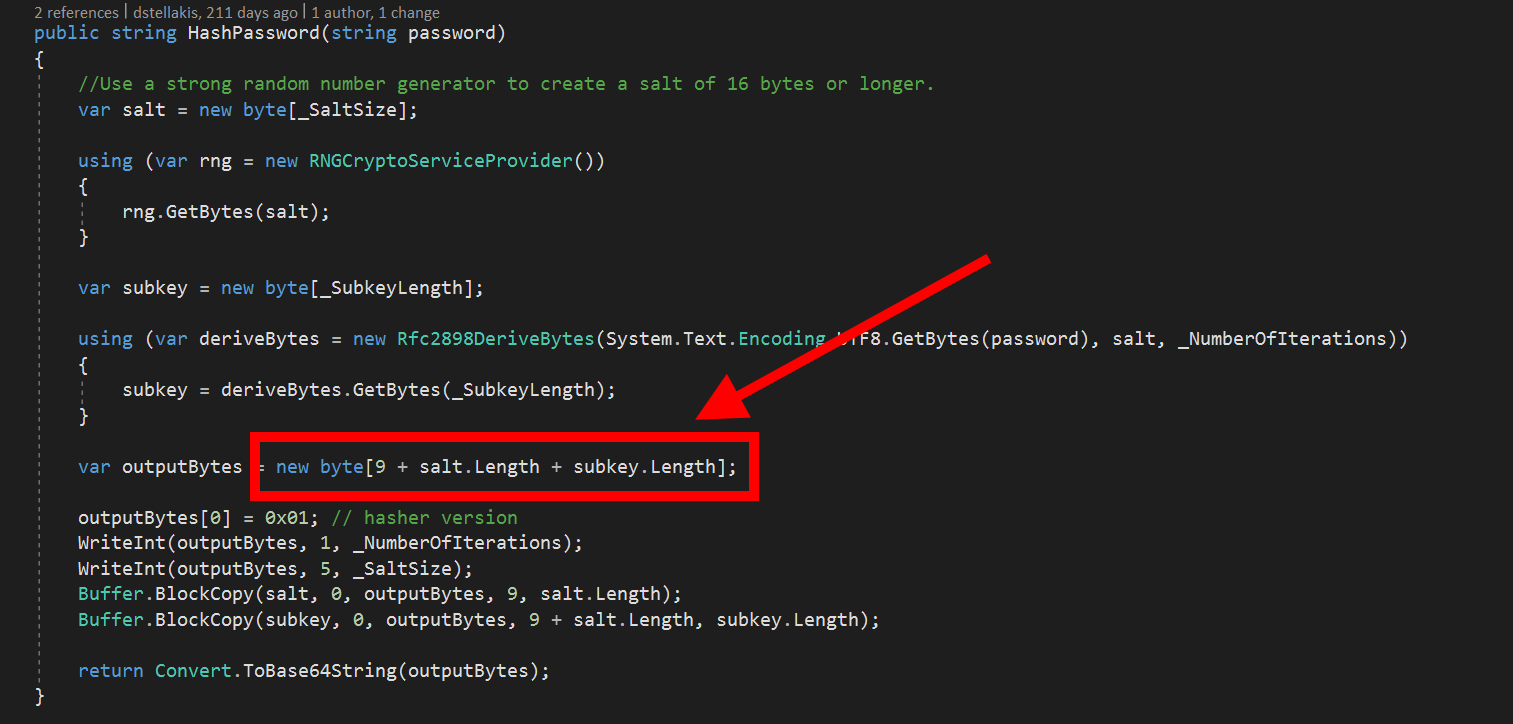


Figure 5 Source code listing for the new methods GenerateSalt and GenerateHash

When a user tries to log in:

* Generate a hash using the provided password and the stored salt and work factor.
* If the hash generated above matches the stored hash, the password was correct; otherwise, the password was incorrect!

If you want to increase the work factor at a later date, write a script that will run on user login to:

* Verify the user's password by comparing the hash generated using the provided password, the stored salt, and the stored work factor with the stored hash.
* Generate a new password hash using the provided password, the stored salt, and the new (increased) work factor, and store the new password hash and the new work factor in the database.

## Our password encryption in a nutshell

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.

## Storing secret keys

Several approaches and layers can be added to a system to raise the security level and avoid intrusions.

There are several ways of storing encryption keys. Each of them bearing a certain level of security and convenience.

The simplest forms are:

**Storing secrets in code**

This is the most convenient method. Developer hardcode the secret key in the code as a costant, then he pushes to the source control.

But, it’s also the least safe method. We just allowed a critical secret to be spread on every developer’s machine, source control provider’s storage, the CI provider’s server, etc. This highly increases the risk that this secret could leak.

**Storing secrets in the environment**

This is one I’ve used quite often. You’ll either need to manually set those environment variables on each server or use some kind of orchestrator to handle that for you, which is slightly less convenient than the previous method. Despite this, it has the benefit that only some trusted individuals have access to it.

There are still some problems: a simple misconfiguration (such as running a production server in debug mode) or a security bug could result in the leak of all the environment variables.

**Storing secrets in the database**

I’ll briefly address this one: you will still need to have your database credentials outside somewhere, therefore, defeating the purpose of putting your secrets there. Besides, having daily backups of all your services’ API keys also increases the risk of a leak at some point.

**Using a secret syncing service**

There are SaaS that offers to take care of your API keys and other secrets for you. They keep them safe and let you query them from their service as needed. We’re still facing the same issue as the previous method: you’ll need a very secret API key to get all your API keys.

**Storing secrets in your code but encrypted**

A modern version of the first method is to encrypt the secrets in your code, thus not exposing their values to your source control, other developers, and so on. However, to decrypt those secrets, the server still needs to manage a key. This comes back to the issue of distributing and maintaining one great secret that unlocks all the other secrets.

### The safest way of storing a secret

Is to use an HSM (hardware security module) a physical computing device that safeguards and manages digital keys, performs encryption and decryption functions for digital signatures, strong authentication and other cryptographic functions.

It is a tamper-proof server that contains a key in hardware. This key can then be used to create other keys used for encryption. The idea here is that the secret key never leaves the HSM.

Many HSMs use key cards from maintenance and admin of the features. A quorum of key cards (e.g. 5 of 9) must be physically put into the server to change a key. This raises the bar pretty high by only allowing a breach if quorums of super users collude.

There may be software solutions that provide similar features to an HSM, but we are not aware of what they are.

Some more research has to be done to understand if they exist.

This solution is compatible for both the on-premises installations where the customer can buy the hardware itself, and each Cloud offers its own dedicated service like the Azure Dedicated HSM <https://azure.microsoft.com/en-us/services/azure-dedicated-hsm/>

### Alternative way

There are a few options to store secrets that shall be evaluated by implementing the .Net Core **IDataProtectionProvider** and the **IDataProtector** Interface

The **IDataProtectionProvider:**

* Represents the root of the data protection system
* It Cannot directly be sued to protect our unprotected data

The **IDataProtector**

* Can directly be used to protect our unprotected data

The data protection architecture in .Net Core come with different settings

* Key management
* Key lifetime
* The algorithm used for data encryption and authenticity
* Key storage providers

Any of these settings can be easily configured by need.

The Keys Storage provider brings in an essential feature because it allows storing secret keys in different repositories:

* File System
* Azure and Redis
* Registry
* Entity Framework
* Custom Key Repository

As a Custom Key Repository, it shall be worthy of mention that a custom repository implementation could be developed to enable store and retrieval of secret keys from Docker secrets (<https://docs.docker.com/engine/swarm/secrets/>)

### Credentials-less service access

For some use cases, it might be convenient to look for an Identity and access management (IAM) Service; each public could have its native implementation offered as a service.

This approach does not cover all our scenarios to be compatible with both the Cloud and on-premises.

It also does not cover all the scenarios because it can be used for granting access to resources but not for generating encrypted artifacts.

It also couple the solution with the native cloud solution chosen.

### Current state and future plans

On the roadmap, we have the task of developing two new providers so that the data encryption and decryption shall be performed by an HSM. For compatibility of our solution, we need an implementation for HSM on-premises on one for Azure.

# API Security

The client and server communicate 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.

The Authentication Flow:

The flow of the authentication process is described in the following steps:

* User registers with the application using the Web API.
* User is requested to verify his account (an email is sent to the email address provided during the registration).
* Uses logins to the application using username/password, and the client calls the Web API to get an authorization token.
* If two-factor authentication (2FA) is enabled for the user, the application prompts the user to enter a one-time password (OTP). The client shall exchange the original authorization token along with the OTP code, with a new authorization token that has passed 2FA.
* The application sends the token with every request to the Web API.
* On each request, the Web API validates the token and executes the request, otherwise returns an Unauthorized status.

The authorization flow is also described in the following diagram:

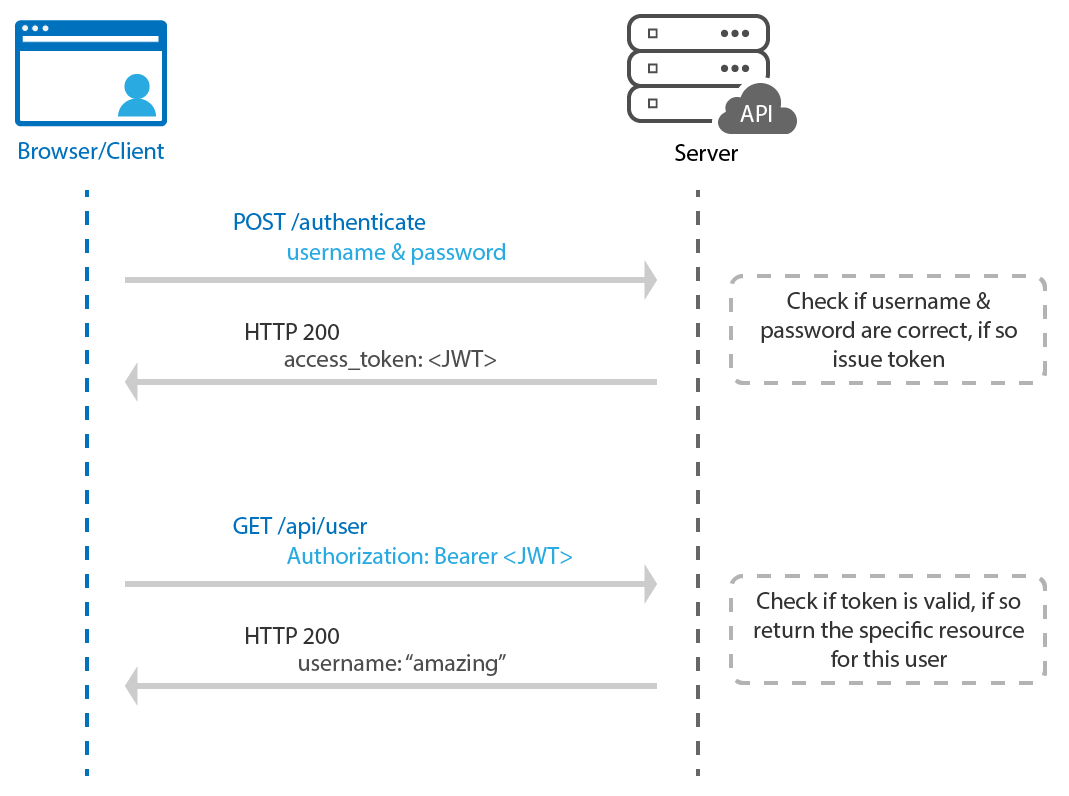


Figure 6 Source code listing for the new methods GenerateSalt and GenerateHash

## Authentication modes

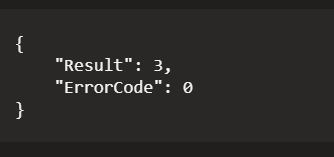
Currently, there are three authentication modes supported by the Web API:

|  |  |  |
| --- | --- | --- |
| ID | Mode | Description |
| 1 | Username & Password | The user must provide a username and password to login |
| 2 | Username & OTP | The user must provide a username and an OTP code to login |
| 3 | Username & Password with 2FA | The user must provide a username and password to login, and after a successful login, must provide an OTP code to be able to user the Web API methods, as described in the section ***Two-Factor Authentication (2FA) with OTP*** |
| 4 | Guest | The user is a guest user and cannot login by himself, an auth token will be provided by means of specific API methods |

Guest users can be disabled on the specific environment by a configuration parameter

These authentication modes can be used by both local users and external users. In case of an external account and if a password is required, the user shall be authenticated using with the external login provider.

In order to determine the authentication mode a user has selected, the client can call the get authentication mode api endpoint /api/Authorization/GetAuthenticationMode and provide the user name required. The server's response will be:



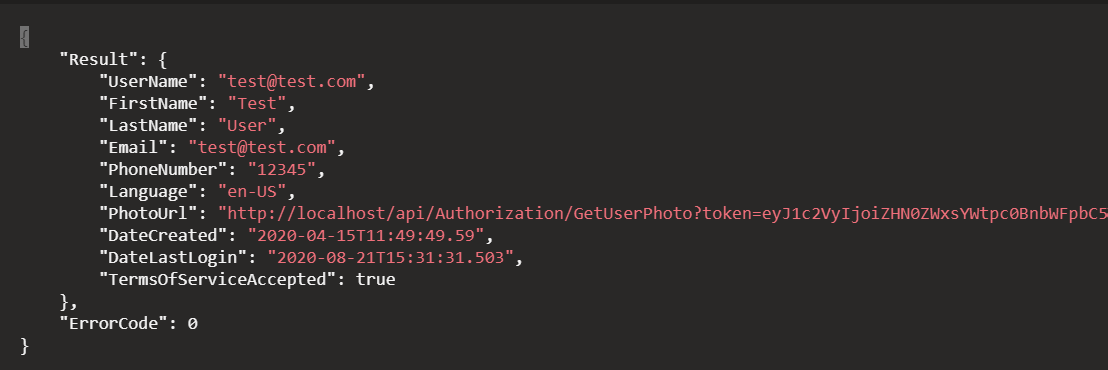
where the result property will contain the authentication mode selected by the user.

The client can use this information to display an appropriate UI to the user, depending on the authentication mode required.

Note: To prevent malicious usage, if the user is not found the api will return 1 as the result without an error code, so that an attacker will not actually know if the user is registered or not.

Authenticated user details

In order to get the details of the currently authenticated user, the client call the api/Authorization/GetAuthenticatedUser endpoint using HTTP GET and passing the authorization token in the Authorization header using Bearer authentication. The web api will return a JSON object that contains the authenticated user's details:



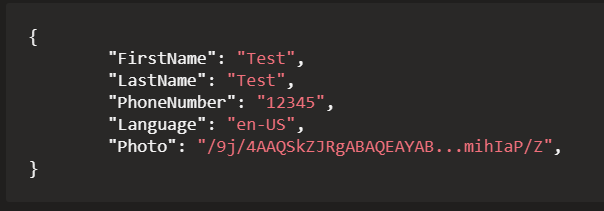
The user's profile image url is returned in field PhotoUrl.

The value for the Country property is a 2 letter ISO code of the country (https://en.wikipedia.org/wiki/List\_of\_ISO\_3166\_country\_codes )

DateCreated is the date that the user was first registered, DateLastLogin is the date that the user last logged in successfully.

### Update the user's profile

To update a user's profile, the app can call the update user profile api endpoint api/Authorization/UpdateUser using HTTP POST, and provide the user details that need to be updated:



The user that will be updated is found from the authorization bearer token, so there is no need to pass the username in this call.

The photo is passed as a base64 string, and must be maximum 1MB in size.

Please note that if any of the above properties is not included in the request payload or is null, they will not be updated. To clear a property, you need to pass it as an empty string value.

## Authenticating using local accounts

### User registration

Provides a way to register a user with the server. Usually a form will be shown by the app, requesting the user to enter a username, email, password and any other necessary fields.

In order to do this, the client shall call the register user api endpoint api/Authorization/RegisterUser using HTTP POST, and providing the user details in JSON format:



The value for the Country property is a 2 letter ISO code of the country (https://en.wikipedia.org/wiki/List\_of\_ISO\_3166\_country\_codes )

The MessageTheme property specifies the theme that the api will use to send the verification email to the user. If unspecified, the api will use the first available theme.

The server will validate the input and if successful it will create a new user account with the given details. Please note that in order for the registration to be successful, the user must have accepted the terms of service, i.e. the parameter TermsOfServiceAccepted must be true.

The account will have an unverified status, and an email will be sent to the user with instructions on the way to verify their account in order to be able to login to the system. The response of this api method will be a JSON object containing the details of the newly created user:



The AuthorizationToken returned by this method, allows the client to call the GetPendingUser and CreateUserProfile API methods immediately if needed. The CreateUserProfile method can be a method defined in an external service API, so that the client can continue the registration process to the external service if needed, before the user validates their account.

The email that will be sent to the user, will include a verification url that is composed of the property VerificationUrl of the request, appended with a token parameter that is a Base64-encoded JSON object containing the username and a verification code.

For instance, in the above example, the complete verification url would be:

https://localhost/verify-user?token=eyJVc2VyIjoidGVzdEB0ZXN0LmNvbSIsIkNvZGUiOiIzMDA0OTcifQ%3D%3D

and the decoded token would be:



If the user did not receive the verification email, or if he would like to receive the email again, the API can call the send registration verification code endpoint api/Authorization/SendRegistrationVerificationCode using HTTP POST and providing the user name of the user, and optionally a verification redirection url that will be included in the email.



The system will resend the registration verification email to the given user, if the user name matches that of a registered user. In case the user is not found, the api will return a success result code, because we do not want this to be used for malicious requests, trying to determine if a username exists or not.

### User verification

Once a user is registered, we need to verify them to make sure that the information entered is correct and that the user is who he claims to be. This is done by sending an email to the email address specified by the user containing a one-time password (OTP), which the user can use to verify their account. The client will prompt the user for the OTP code, and will call the verify user account api endpoint api/Authorization/VerifyUser using HTTP POST, and providing the verification details in JSON format:



### Login

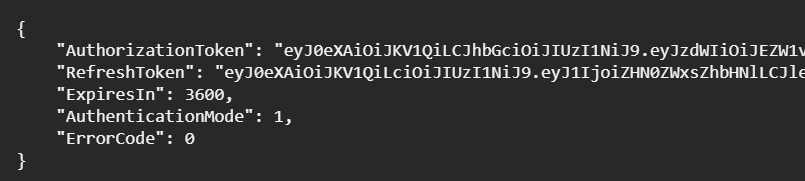
The login process consists of authenticating the user with the web server using their username and password, and obtaining a user-specific token, that the client app will then send with every https request to the Web API. If the token is valid, access to the resource will be granted, otherwise the client will get an HTTP Unauthorized status message.

The token format is used is JSON Web Tokens (JWT). More information about their format can be found on the following sites:

https://en.wikipedia.org/wiki/JSON\_Web\_Token

https://jwt.io/

The client will call the authentication api endpoint api/Authorization/Authenticate using HTTP GET, and will pass the username and password (or OTP code, depending on the user's authentication mode) of the user in the Authorization header using Basic authorization. The server will validate the credentials and, if valid, will return an authorization token AuthorizationToken, the number of seconds in which the token will expire ExpiresIn, and the user's authentication mode. Also included is a refresh token, that can be used to obtain a new authorization token when this one expires (see section Token Refresh):



The authorization token contains information about the current user, and shall be passed in the Authorization header of any subsequent API request, in the form of a Bearer token:

Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJEZW1vIiwidGZhIjpmYWxzZSwiZXhwIjoxNTc1MzY5MTQ2fQ.\_cfJnh7Jl5TMGUw9fyzyrSoJwfLZXUDkcCESQ93M11U

The server will validate the authorization token and proceed to return the results of the API method call, or HTTP Unauthorized status. Because the Web API is stateless, the authorization token must be provided with every request.

If the login call did not succeed, the server will return HTTP Unauthorized status, and the response will contain the error that occurred. Some of there are:

|  |  |
| --- | --- |
| Error | Meaning |
| 127 | The username or password provided was not correct |
| 130 | The user is not yet verified |

NOTE: If the user has opted for two-factor authentication, the refresh token will be null. The user will need to complete the authentication process in order to get a valid refresh token.

### Password reset

The user should be able to recover his password in case he has forgotten it. In such case, the client app should prompt the user to enter the email address he used during his registration, and if the system finds a corresponding user in the database, an email will be sent to him, containing a one-time password (OTP) which the user can use to reset his password.

In order for this, the client shall call the send password reset token endpoint api/Authorization/SendUserPasswordResetToken using HTTP POST, passing a JSON object that contains the username of the user that has requested the password reset, along with an optional reset url that will be included in the email. The url will be appended with a token parameter, in the same way as the registration url in the User registration section.



The API will send an password reset OTP token to the registered email of the user, which the user can then use to reset his password from the UI provided by the client.

Note: To prevent malicious usage, the service will always return success, even if the user is not registered, so than an attacker will not be able to know if a user account exists or not in the system.

After that, the client shall call the update user password api endpoint api/Authorization/UpdateUserPassword using HTTP POST and providing the username, the new password and the password reset token retrieved from the previous step:



### Token refresh

When the client authenticates with the Web API, the number of seconds until the token expires is returned in the property ExpiresIn of the JSON response.

If the client needs to extend the authorization time window, it should issue an HTTP GET request in the refresh authorization token endpoint api/Authorization/RefreshToken, providing in the Authorization header the already obtained refresh token from the Login step:

The authorization token contains information about the current user, and shall be passed in the Authorization header of any subsequent API request, in the form of a Bearer token:

*Bearer eyJ0eXAiOiJKV1QiLciOiJIUzI1NiJ9.eyJ1IjoiZHN0ZWxsZhbHNlLCJleHAiOjE1ODYwMTQ3ODV9.8ZFtW8jd4O2yNXJm9AaHVj2pwjYjjAU*

If the refresh token is valid, the response will contain the new authorization and refresh tokens, along with the extra details for the expiration time and 2FA, similar to the login process.

NOTE: The refresh token must be kept a secret and stored in a secure location, because it is in fact a way of obtaining an access token for a user without the need to login. It is the client app's responsibility to ensure that these safety standards are met. If there is no way of securely storing the refresh token, it would be best if the token is only kept in-memory (although this would require the user to login each time they open the client app).

## 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.

Please note: It might be possible for different app versions or deployments to have different external provider configuration settings. In that case, the calling application shall be responsible of providing the web api with the correct app code to use.

### External providers supported

Our authentication system can be integrated with external identity providers (IdP) for Single-Sign-On (SSO). Currently the following protocols are supported:

#### OAuth

We support the OAuth 2.0 protocol for SSO. Currently, we have integration with Google, Microsoft, Apple and Active Directory (ADFS) SSO identity providers.

An external login provider needs to support OAuth2 + OpenID to be used for external user authentication. Our service API is using the Authorization Code flow (https://developer.okta.com/blog/2018/04/10/oauth-authorization-code-grant-type)

The external login service must provide at least an authorize endpoint and a token endpoint, to be used by the above flow, and must have a client configured with the provided redirect url, and provide the client id and client secret. The required scopes are openid and email. The profile scope is recommended as well, as it will provide more information for the authenticated user.

In the id\_token returned by the service, the claims needed are:

|  |  |  |
| --- | --- | --- |
| Claim | Description | Required |
| unique\_name or preferred\_username | The user name | Yes |
| email | The user's email | Yes |
| given\_name | The user's first name | No |
| family\_name | The user's last name | No |
| name | The user's full name | No |
| picture | The user's profile photo url | No |

#### SAML

SAML stands for Security Assertion Markup Language. It is an XML-based open-standard for transferring identity data between two parties: an identity provider (IdP) and a service provider (SP). We are compatible with identity providers supporting the SAML 2.0 HTTP Redirect / POST Binding

Identity Provider — Performs authentication and passes the user's identity and authorization level to the service provider.

Service Provider — Trusts the identity provider and authorizes the given user to access the requested resource

**Flows**

SAML supports two different types of flows: those initiated by the service provider and those initiated by the identity provider. In our case, we cover the SP-initiated flow. In SP-initiated flows, you start out at the service provider (IBV), are redirected to the identity provider to authenticate, and are then redirected back to the service provider. This flow is usually initiated when a user clicks on “Login with SSO” button or something similar.

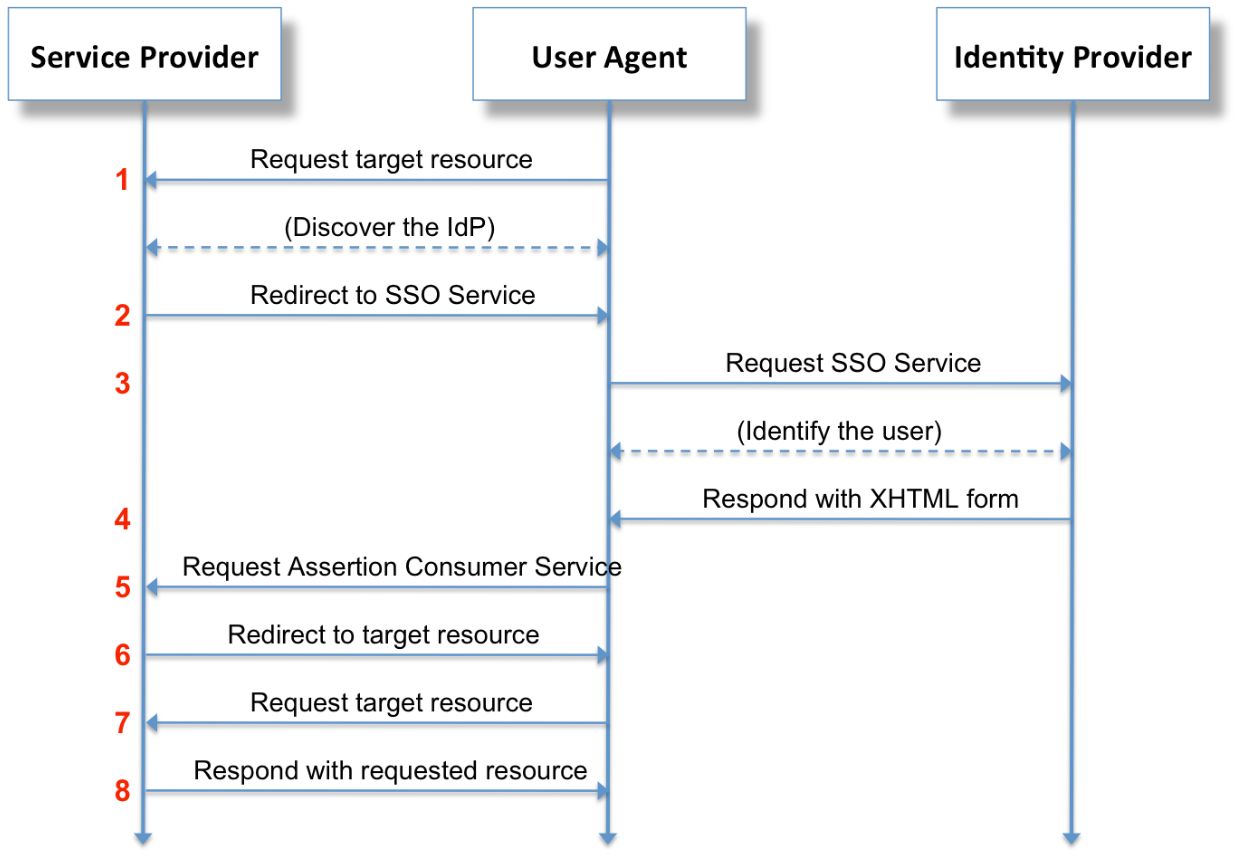
**Bindings**

Bindings are the format in which data is transferred between service providers and identity providers. The two most popular are HTTP Redirect Binding and HTTP POST Binding. HTTP Redirect Bindings transfer data using HTTP redirects and query parameters; this type of binding is typically used in authentication requests. HTTP POST Binding transfer data using HTTP POST forms, this type of binding is typically used in authentication responses.

**Assertions**

Assertions are statements made by the identity provider about the principal. For example, the principal’s email address and/or groups/roles the principal may be associated with. Assertions are used by the service provider to create and configure sessions for a principal.

A typical sign-in flow using SAML redirect binding is:



To integrate with an identity provider using the SAML 2.0 protocol, we will need the XML Metadata of the provider. A sample IdP XML metadata file would look like this:



Alternatively, we would need the following information from the identity provider:

IdP EntityID

IdP Redirect URL

IdP Logout URL

IdP public certificate

NameId Format

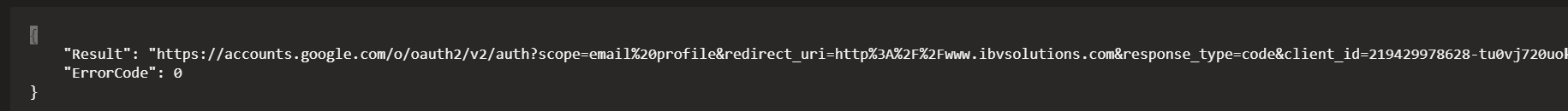
User-specific attributes returned by the authentication response

### External user authentication

In this case, the user will first choose a provider to authenticate with, and the app will contact the web api to get the login url for this provider by issuing an HTTP GET request to the /api/Authorization/GetExternalLoginUrl endpoint, and passing as the provider parameter the app code and the name of the chosen provider. The valid values for the provider parameter are Google and Microsoft:

*/api/Authorization/GetExternalLoginUrl?app=react-web&provider=Google*

The result will be a JSON response, containing the url of the provider's login page:

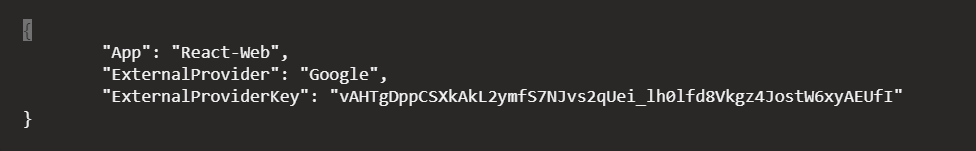


The app will then present the user with a web browser window of this login url. The user will enter his credentials for that provider and if the authentication succeeds, the browser will redirect back to the registered app url with a one-time authorization code that the app must use to complete the login process. The app will get the authorization code from the query string of the redirected url:

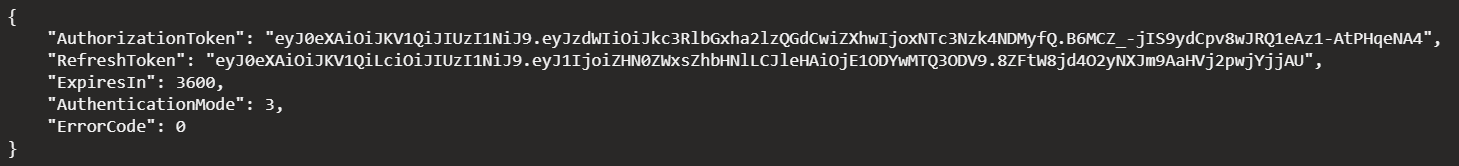
*https://ibvsolutions.com/en/home/?code=4%2FvAHTgDppCSXkAkL2ymfS7NJvs2qUei\_lh0lfd8Vkgz4JostW6xyAEUfI&scope=email%20profile%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fuserinfo.profile%20openid%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fuserinfo.email&authuser=0&session\_state=42fb79922199f8629b50c3fd5c7681c69facb939..7ba0&prompt=none#*

In the above example, Google redirects the browser to the IBV home page, and passes along a code parameter that is the authorization code we will use for the next step. The value of the code (after url decoding) will be 4/vAHTgDppCSXkAkL2ymfS7NJvs2qUei\_lh0lfd8Vkgz4JostW6xyAEUfI

Once the app has the authorization code, it will issue an HTTP POST request to the external authentication Web API endpoint api/Authorization/AuthenticateExternal, providing the login provider used for the authentication and the authorization code returned by the provider:



The Web API will verify the validity of the given key with the external login provider using the provider's secret key, and will return an authorization token, similar to the Login process:

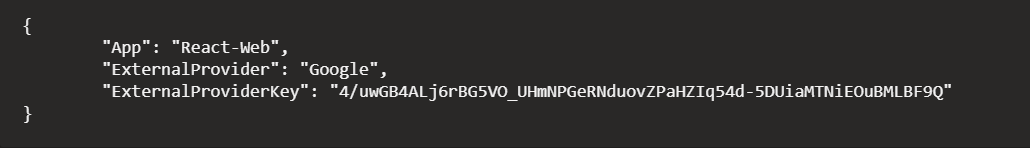


The client app can then use this authorization token to call the Web API methods, in the same way as if the user was a local user.

Note: In case the user has not accepted the terms of service, the api will return an ErrorCode 134 (UserNotAcceptedTermsOfService), and the AuthorizationToken will only be valid for the CompleteExternalUserRegistration api endpoint, as described in the next section. This will allow the user to update any information needed and accept the terms of service, in order to proceed.

### External user registration

Similar to the external user authentication, for a new user registration using an external login provider, the user will first login to the external provider, get an authentication code and the app will issue an HTTP POST request to the external registration Web API endpoint api/Authorization/RegisterExternalUser, providing the login provider used for the authentication and the authorization code returned by the provider during the authentication step:



The Web API will verify the validity of the given key with the external login provider, and if the user is not yet registered, it will create a new user account linked to the external login, populated with the user details returned by the external login provider. No password information will be stored in the database. This newly created account will also have a verified status.

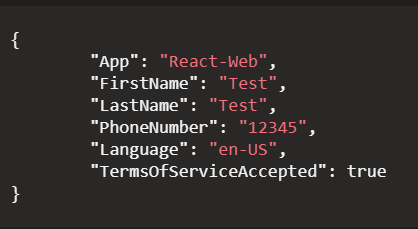
The api will return the newly registered user's details, along with an authentication token:



To get the user details of the partially registered user, the client app can call the api/Authorization/GetPendingUser endpoint. This endpoint returns the same payload as the

GetAuthenticatedUser , but it can only be invoked with the token returned from the above call.

To complete the registration, the client will then show a UI with the details of the new user, prompting him to change anything that he would like (except the username) and to accept terms of services. The client will then call the /api/Authorization/CompleteExternalUserRegistration using HTTP POST and providing the user details, using as the authentication token the token it got from the previous step:

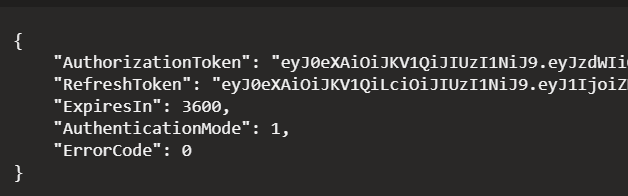


The value for the Country property is a 2 letter ISO code of the country (https://en.wikipedia.org/wiki/List\_of\_ISO\_3166\_country\_codes )

Note: This authentication token is only valid for the GetPendingUser, CompleteExternalUserRegistration

and CreateUserProfile endpoints, the app cannot use it to invoke other api endpoints. Also, the GetPendingUser and CompleteExternalUserRegistration endpoints can only be invoked by this token type. The CreateUserProfile can be a method defined in an external service API, so that the client can continue the registration process to the external service if needed, before the user validates their account.

The API will update the user's details and return a normal authentication token, similar to the login process:



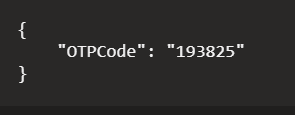
### Token refresh

Same as the token refresh functionality for local user accounts.

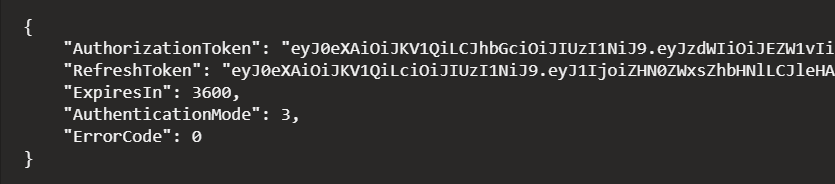
## Two-Factor Authentication (2FA) with OTP

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.

After the Login process, and if the server responds that the user has enabled 2FA, the client should prompt the user to enter an OTP code, and call the 2FA verification api endpoint api/Authorization/ValidateUser2FA using HTTP POST, passing the authorization token received by the login process, and a JSON object with the OTP Code:

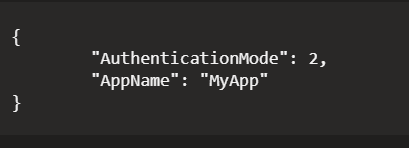


The server will validate the OTP code provided, and it will return new authorization & refresh tokens that are now 2FA-verified. The client shall use the new authorization token for every subsequent API request. If the OTP code cannot be verified, the server will return HTTP Unauthorized status.

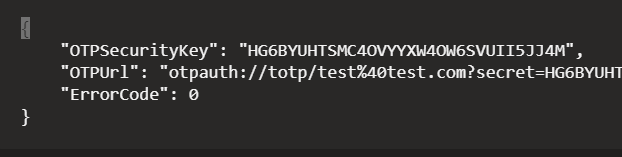


### Changing Authentication Mode for a User

The client can request to change the authentication mode for a given user by calling the update user authentication mode api endpoint /api/Authorization/UpdateUserAuthenticationMode. It shall provide the authorization token, along with a JSON object indicating the new authentication mode (as per the authentication modes table provided above):



If the authentication mode selected needs an OTP code (as is the case with Username & OTP and Username & Password with two-factor authentication), the caller must also provide the AppName parameter. The server will respond with an OTP security key, otherwise it will respond with a null key:



If OTPs are enabled, this security key shall be displayed to the user as a string or a QR Code in the app's UI. The app can use an existing library to generate the QR code from the URL, depending on the app's platform. The user will scan the QR code (or use the security key directly) in an authenticator app of his preference (for example, Google Authenticator), in order to register MyApp in the app and be able to generate OTP codes.

For testing purposes, one can use the following tool to generate QR codes:

https://stefansundin.github.io/2fa-qr/

Note: Each time the client calls this method, a new OTP security key is generated, and the user must update his authenticator app accordingly.

### Getting OTP codes without the need of an authenticator app

In case the user is using OTPs in his authentication mode, and he does not have access to an authenticator app, the system can generate OTP codes for him. To do so, the client shall call the send otp token api endpoint /api/Authorization/SendUserOTPToken, and provide the username of the user and the messaging platform that will be used to send the message (Email or SMS):



If the messaging platform is omitted, the OTP code will be sent to the user via email.

The MessageTheme property specifies the theme that the api will use to send the verification email to the user. If unspecified, the api will use the first available theme.

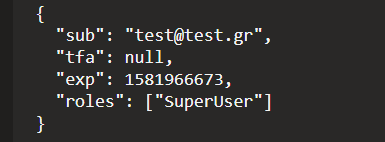
If the user is registered & validated in the system and he has enabled OTP codes, the web api will send to his registered email address an OTP code.

### 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.

If needed, the roles of a logged-in user can be derived from the authorization token, from the payload section after JWT decoding, in the property roles that is a string array including all the different roles of the user:



*Payload section of JWT authorization token*

Each time a web api method is called, the server will check if the user has the necessary role to invoke it, as defined in the method's signature. If not, an HTTP 401 Status will be returned.

## Integration with Web API projects

### DotNet Core

To use the authentication in your .NET Core Web API project, you need to reference the IBV.Auth project.

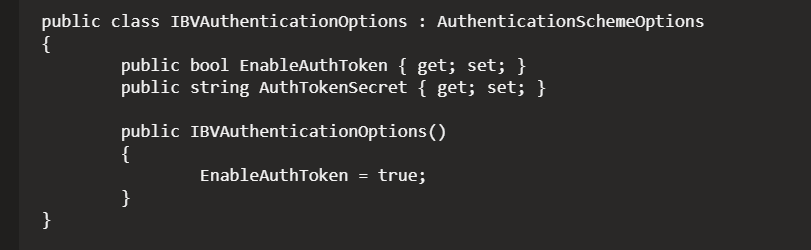
The IBV authentication handler is asp.net core middleware that handles request authentication by inheriting from the asp.net core AuthenticationHandler base class and overriding the HandleAuthenticateAsync() method.

The authentication logic is implemented in the HandleAuthenticateAsync() method by verifying the token present in the HTTP Authorization header, verification is done by calling AuthHelper.GetPrincipal(token, tokenSecret). On successful authentication the method returns AuthenticateResult.Success(ticket) which makes the request authenticated and sets the HttpContext.User to the currently logged in user.

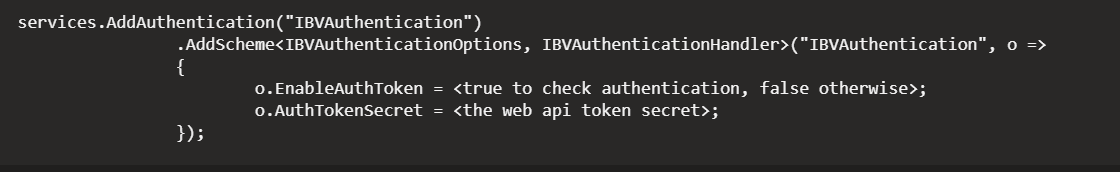
The IBV authentication middleware is configured in the application inside the ConfigureServices(IServiceCollection services) method in the application Startup file below.



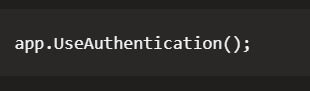
And



Then, in the startup class, you need to register the IBV authentication handler the request pipeline. You can do this by adding the following code inside the ConfigureServices(IServiceCollection services) method:



and also add the following line of code in the Configure(IApplicationBuilder app, IWebHostEnvironment env) method, above the app.UseAuthorization() call



Please note that we have used the ConfigurationManager API in order to get the necessary properties from the IBV configuration storage.

Finally, we can use the [Authorize] attribute in our controller class and/or methods, in order to specify that this controller / method should use authorization:



If needed, we can use the [AllowAnonymous] attribute to exclude a method from requiring authentication.

We can also restrict access on a controller or method based on the user's role, for example to only alow users with the SysAdmin role to call a method, we could decorate it as:

[Authorize(Roles = "SysAdmin")]

Once authenticated, a controller action can use the User.Identity property, to get information about the authenticated user.

### SignalR Authenticaiton

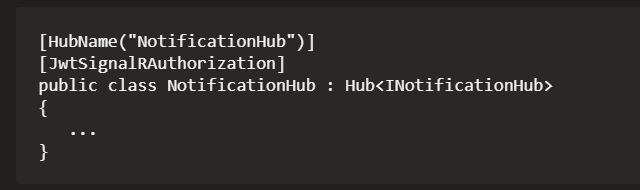
To use the authentication in your .NET Framework SignalR project, you need to reference the IBV.Auth project.

We need to create a custom AuthorizeAttribute that will apply the authorization logic:



The above code will try to get the authentication token from the Authorization header of the request or from the access\_token query string parameter.

We can then apply the authorization attribute to our notification hub class:



This will check for a valid authentication token on every hub connection or method invocation request, and will return an error if no valid authentication token is found. Also, the Context.User.Identity property will be populated with the calling user and become available to the hub's methods.

# File Security

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.

In the picture below you can see that the application pool running the MFT APIs is running with a custom identity that has the permission to access for reading and write on the network share where files are stored

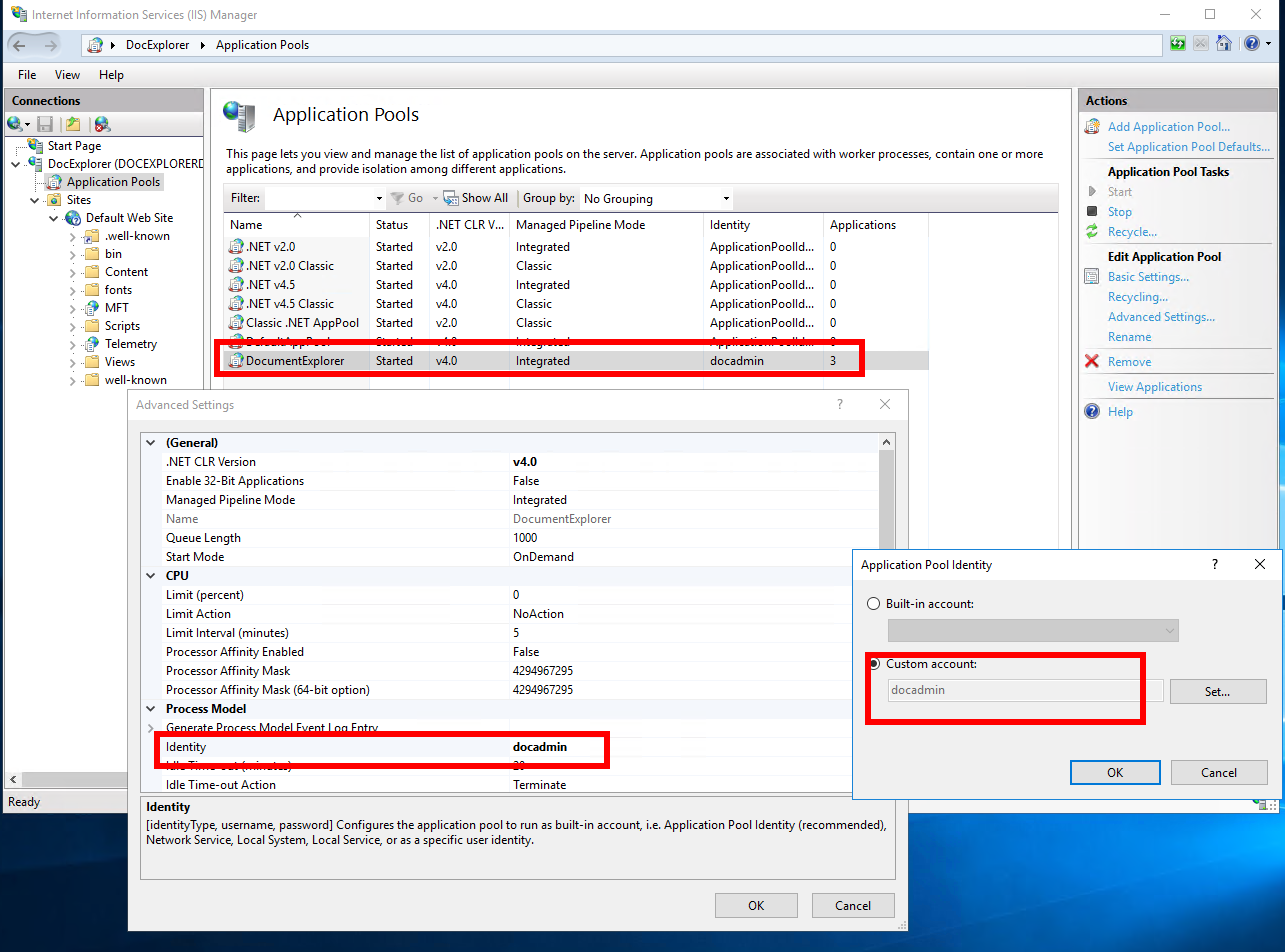


Figure 7 MFT APIs application pool identity

All the MFT state machines are also running, as Windows Services, with a custom identity (see picture below), that way they can access the network file share where CollaBoard’s files are stored

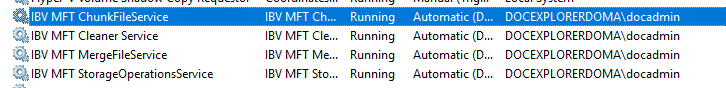


Figure 8 MFT services running on a custom AD identity

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.

## 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.

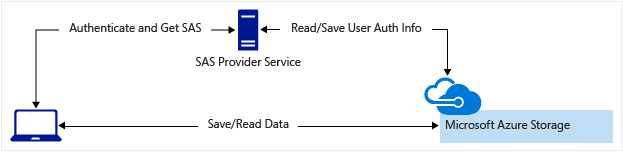


Figure 9 Security for files on cloud diagram

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.

In our code, once the user is authenticated, can call an API to create a new project (for detailed documentation on the CollaBoard Service Web APIs please refer to the document CollaBoard Service APIs).

Once the business logic validates and decides the request, if it is legit, then a new container (under the IBV’s CollaBoard Storage is created) (step 1 on the source code below). A new access policy is created for the container (step 2 on the source code below). The container is created (step 3 on the source code below) and the access policies are created (see code method ManageSharedAccessContainerPermission below). The container token is extracted to send back to the client (step 5 on the source code below). The container Uri is extracted to send back to the client (step 6 on the source code below). The container token is added to the return object (step 7 on the source code below).

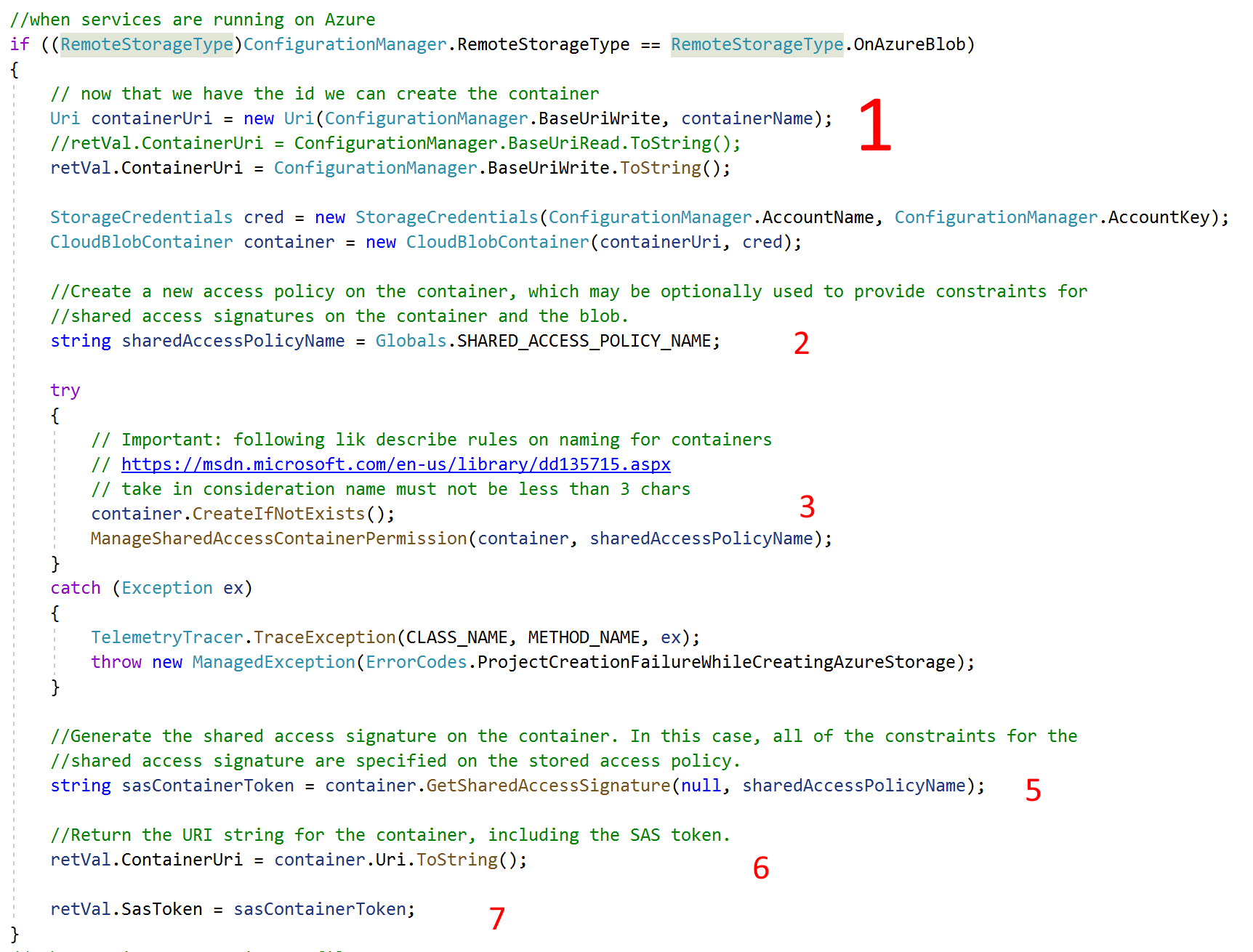


Figure 10 Code listing for the creation of the container, the access token

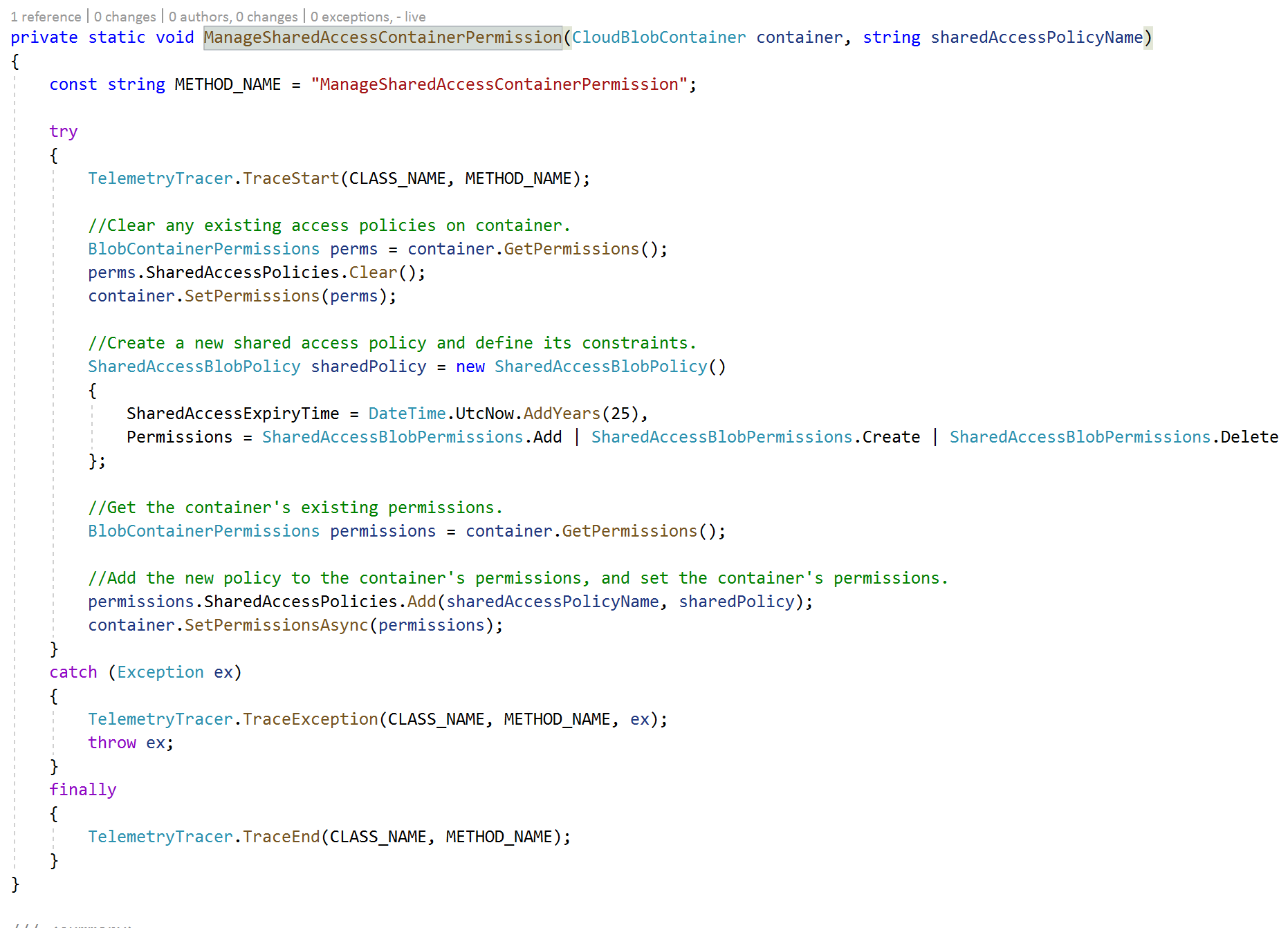


Figure 11 Code listing method ManageSharedAccessContainerPermission

# Vulnerability Scans

For our Azure envinroment 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

You can 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.

You can view a list of all issues by selecting Recommendations. For more information about how to apply recommendations, see Implementing security recommendations in Azure Security Center.

To continue, select **Compute & apps** under **Resources** or the Security Center's main menu.

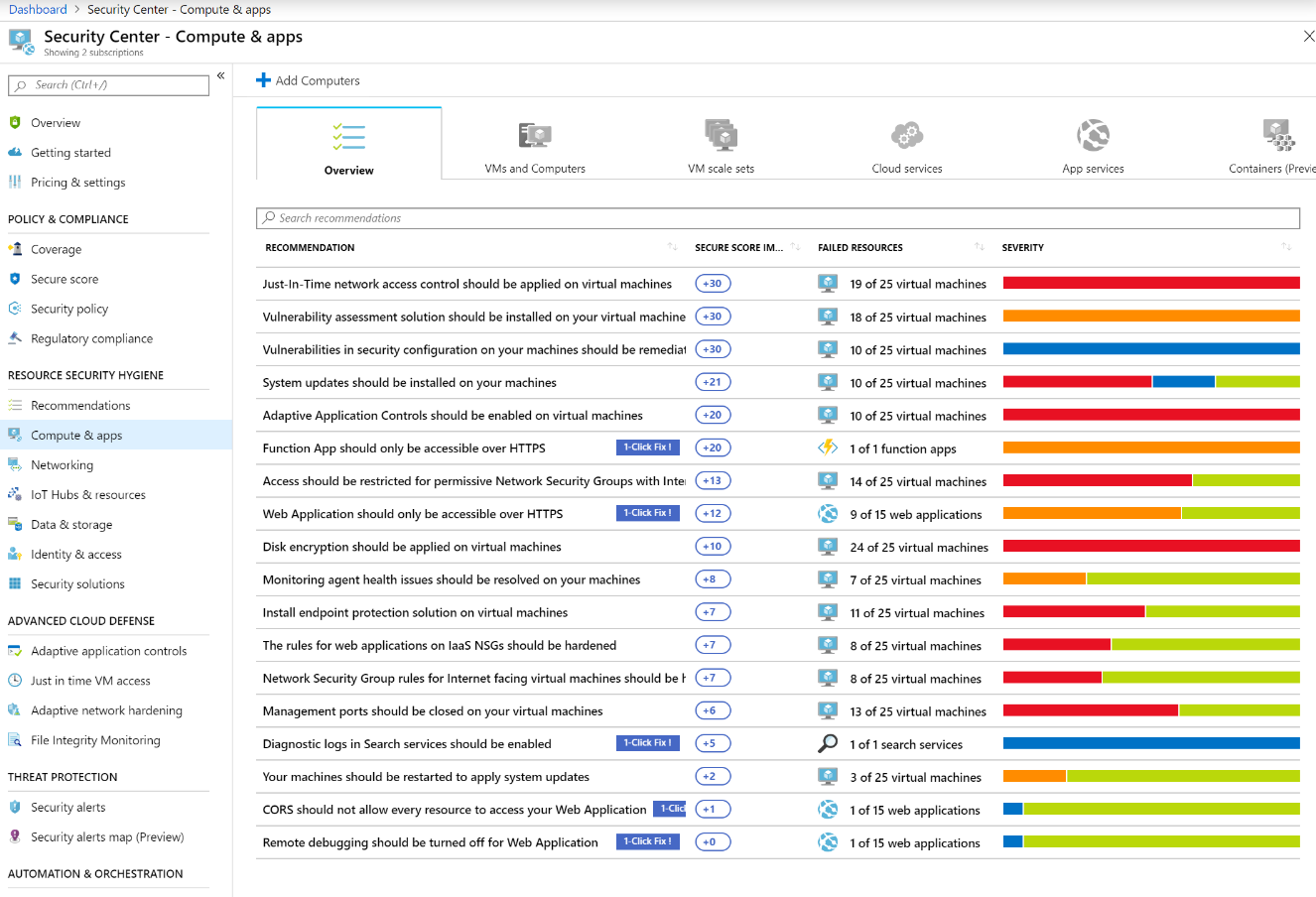


Figure 12 Compute & Apps overview

Monitor compute and app services

Under **Compute & apps**, there are the following tabs:

* **Overview**: monitoring and recommendations identified by Security Center.
* **VMs and computers**: list of your VMs, computers, and current security state of each.
* **Cloud Services**: list of your web and worker roles monitored by Security Center.
* **App services**: list of your App service environments and current security state of each.
* **Containers (Preview)**: list of your containers hosted on IaaS Linux machines and security assessment of their Docker configurations.
* **Compute resources (Preview)**: list of recommendations for your compute resources, such as Service Fabric clusters and Event hubs.

To continue, select Compute & apps under Resource security hygiene.

In each tab you can have multiple sections, and in each section, you can select an individual option to see more details about the recommended steps to address that particular issue.

## Unmonitored VMs and computers

A VM or computer is unmonitored by Security Center if the machine is not running the Microsoft Monitoring Agent extension. A machine may have a local agent already installed, for example, the OMS direct agent or the System Center Operations Manager agent. Machines with these agents are identified as unmonitored because these agents are not fully supported in Security Center. To fully benefit from all of Security Center’s capabilities, the Microsoft Monitoring Agent extension is required.

You can install the extension on the unmonitored VM or computer in addition to the already installed local agent. Configure both agents the same, connecting them to the same workspace. This enables Security Center to interact with the Microsoft Monitoring Agent extension and collect data. See Enable the VM extension for instructions on how to install the Microsoft Monitoring Agent extension.

## Recommendations

This section has a set of recommendations for each VM and computer, web and worker roles, Azure App Service Web Apps, and Azure App Service Environment that Security Center monitors. The first column lists the recommendation. The second column shows the total number of resources that are affected by that recommendation. The third column shows the severity of the issue.

Each recommendation has a set of actions that you can perform after you select it. For example, if you select Missing system updates, the number of VMs and computers that are missing patches, and the severity of the missing update appears.

Apply system updates has a summary of critical updates in a graph format, one for Windows, and one for Linux. The second part has a table with the following information:

* NAME: Name of the missing update.
* NO. OF VMs & COMPUTERS: Total number of VMs and computers that are missing this update.
* UPDATE SEVERITY: Describes the severity of that particular recommendation:
  + Critical: A vulnerability exists with a meaningful resource (application, virtual machine, or network security group) and requires attention.
  + Important: Non-critical or additional steps are required to complete a process or eliminate a vulnerability.
  + Moderate: A vulnerability should be addressed but does not require immediate attention. (By default, low recommendations are not presented, but you can filter on low recommendations if you want to view them.)
* STATE: The current state of the recommendation:
  + Open: The recommendation has not been addressed yet.
  + In Progress: The recommendation is currently being applied to those resources, and no action is required by you.
  + Resolved: The recommendation was already finished. (When the issue has been resolved, the entry is dimmed).

To view the recommendation details, click the name of the missing update from the list.

### VMs and computers

The VMs and computers section gives you an overview of all VM and computer recommendations. Each column represents one set of recommendations.

There are four types of icons represented in this list:

* Non-Azure computer Non-Azure computer.
* Azure Resource Manager VM Azure Resource Manager VM.
* Azure Classic VM Azure Classic VM.

VMs that are identified only from the workspace that is part of the viewed subscription. This includes VMs from other subscriptions that report to the workspace in this subscription, and VMs that were installed with Operations Manager direct agent, and have no resource ID.

The icon that appears under each recommendation helps you to quickly identify the VM and computer that needs attention, and the type of recommendation. You can also use the filters to search the list by Resource type and by Severity.

To drill down into the security recommendations for each VM, click on the VM.

### App services

To view the App Service information, you must enable App Service in your subscription. For instructions on enabling this feature, see Protect App Service with Azure Security Center. [!NOTE]

Monitoring App Service is in preview and available only on the Standard tier of Security Center.

Under App services, you find a list of your App service environments and the health summary based on the assessment Security Center performed.

There are three types of icons represented in this list:

* App services environment App services environment.
* Web application Web application.
* Function application Function application.

1. Select a web application. A summary view opens with three tabs:

* Recommendations: based on assessments performed by Security Center that failed.
* Passed assessments: list of assessments performed by Security Center that passed.
* Unavailable assessments: list of assessments that failed to run due to an error or the recommendation is not relevant for the specific App service

Under Recommendations is a list of the recommendations for the selected web application and severity of each recommendation.

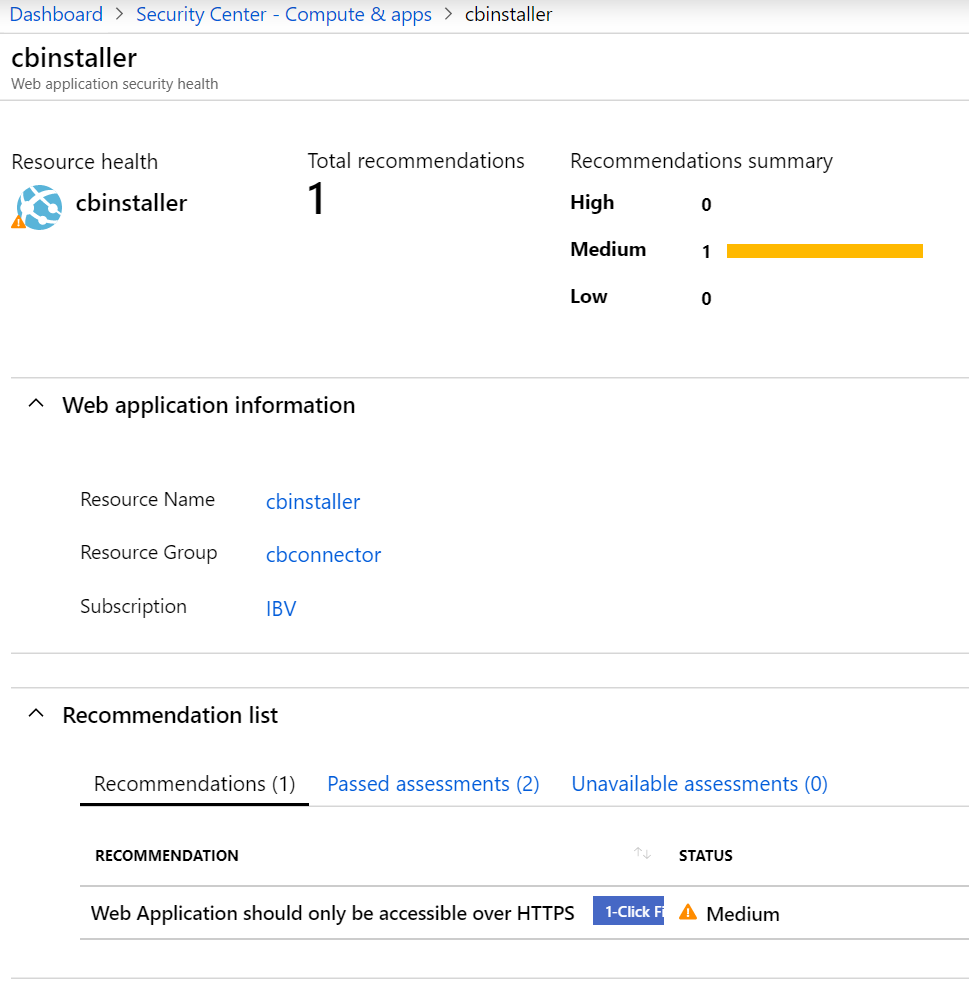


Figure 13 List of recommendations

1. Select a recommendation to see a description of the recommendation and a list of unhealthy resources, healthy resources, and unscanned resources.

* Under the Passed assessments column is a list of passed assessments. The severity of these assessments is always green.
* Select a passed assessment from the list for a description of the assessment, a list of unhealthy and healthy resources, and a list of unscanned resources. There is a tab for unhealthy resources but that list is always empty since the assessment passed

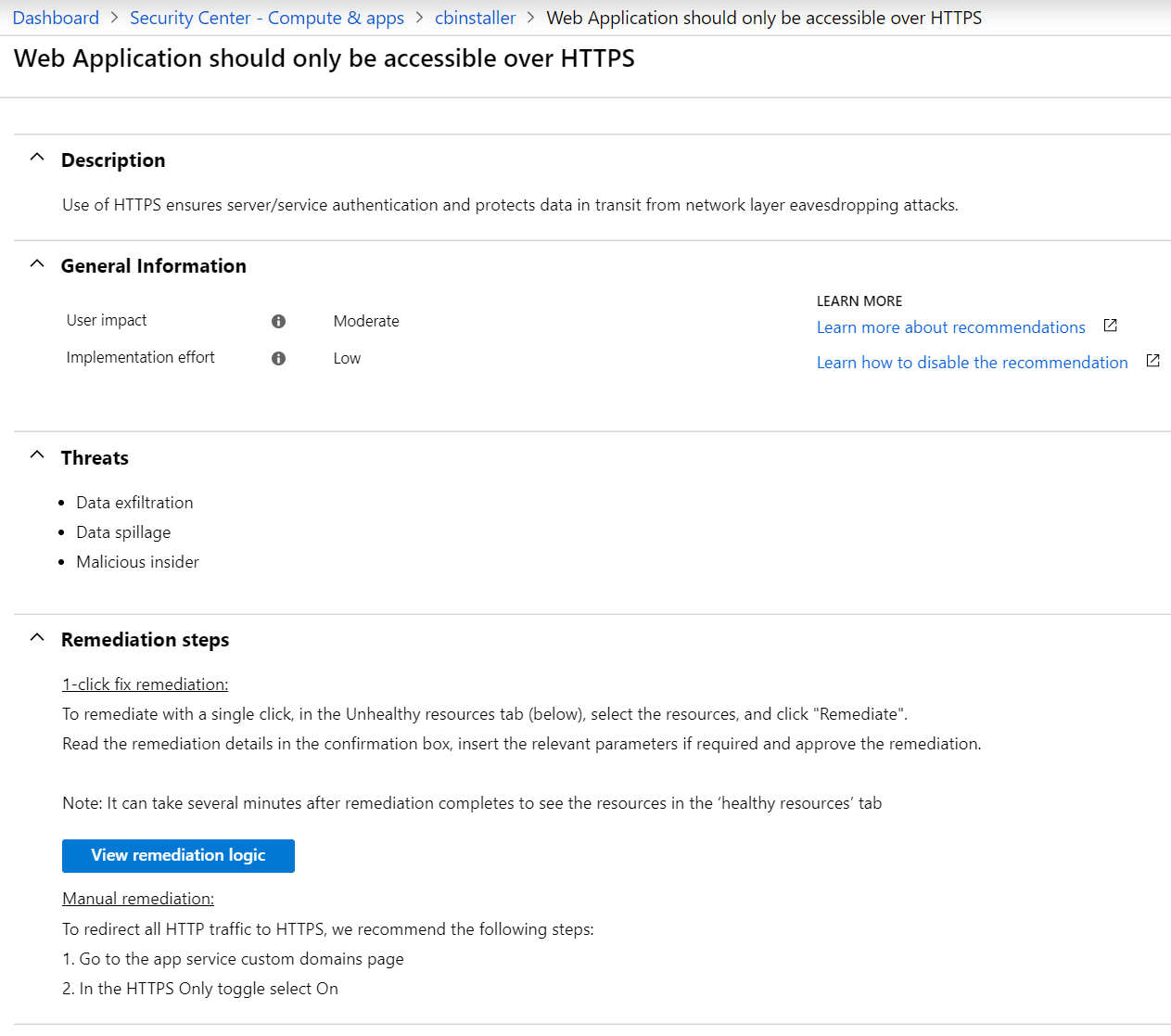


Figure 14 Recommendation detail

## Vulnerability Scans for CollaBoard Azure resources

There are no currently actions to be taken as shown in the picture below.

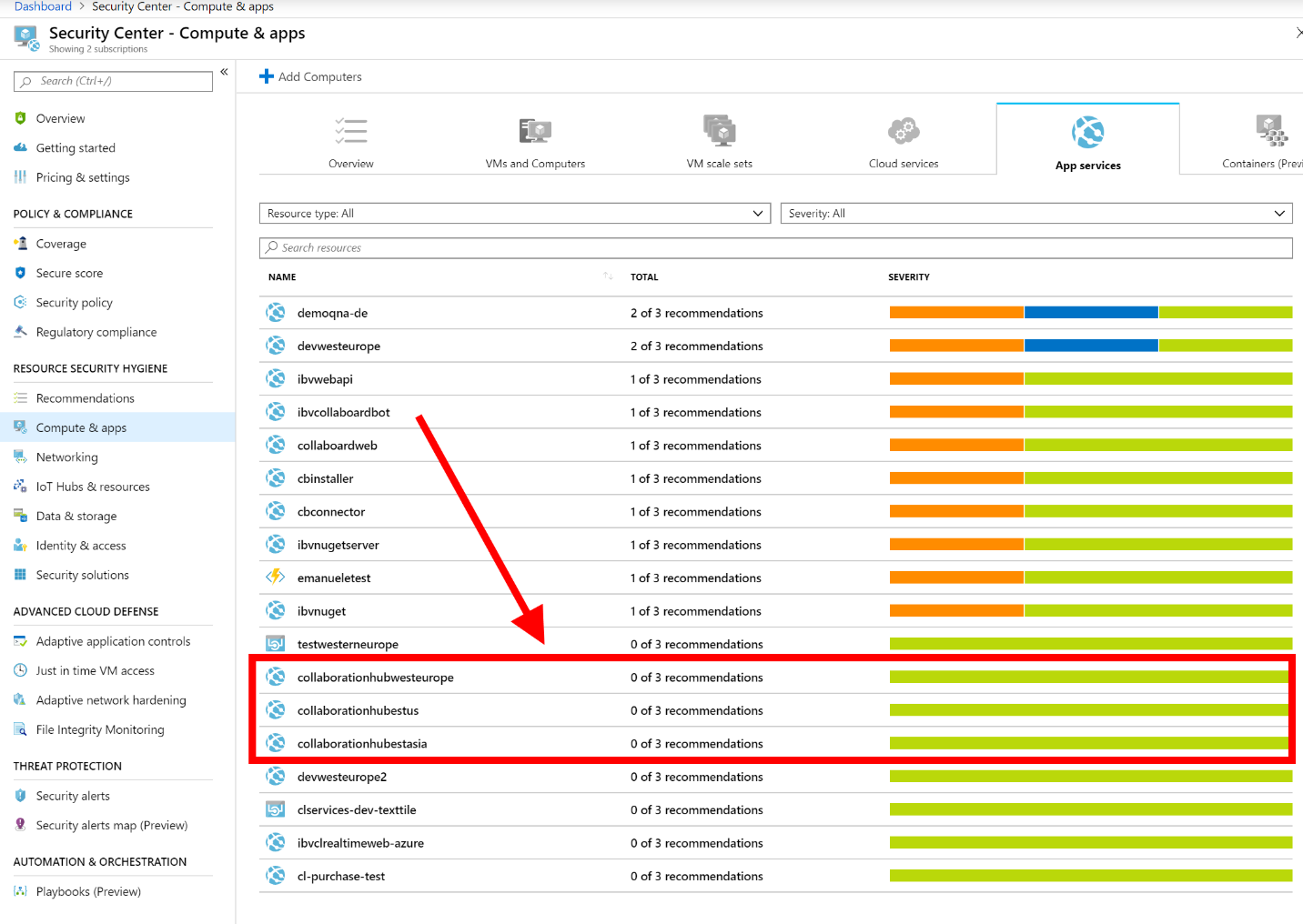


Figure 15 Recommendation for CollaBoard production App Service

For our Azure storage we have one recommendation to set the access only through https, even though this is a good recommendation, we are accessing our production storage only through the application, and we programmatically define the https access

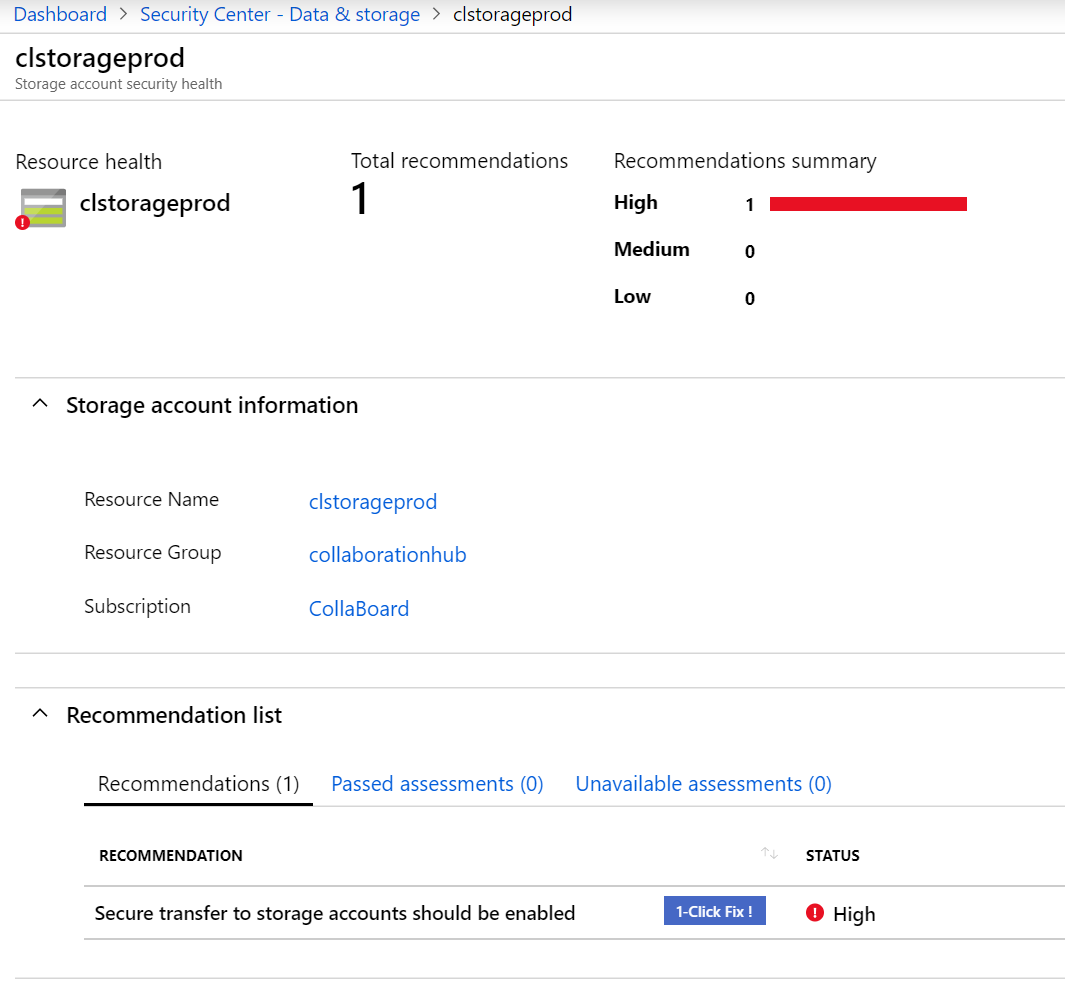
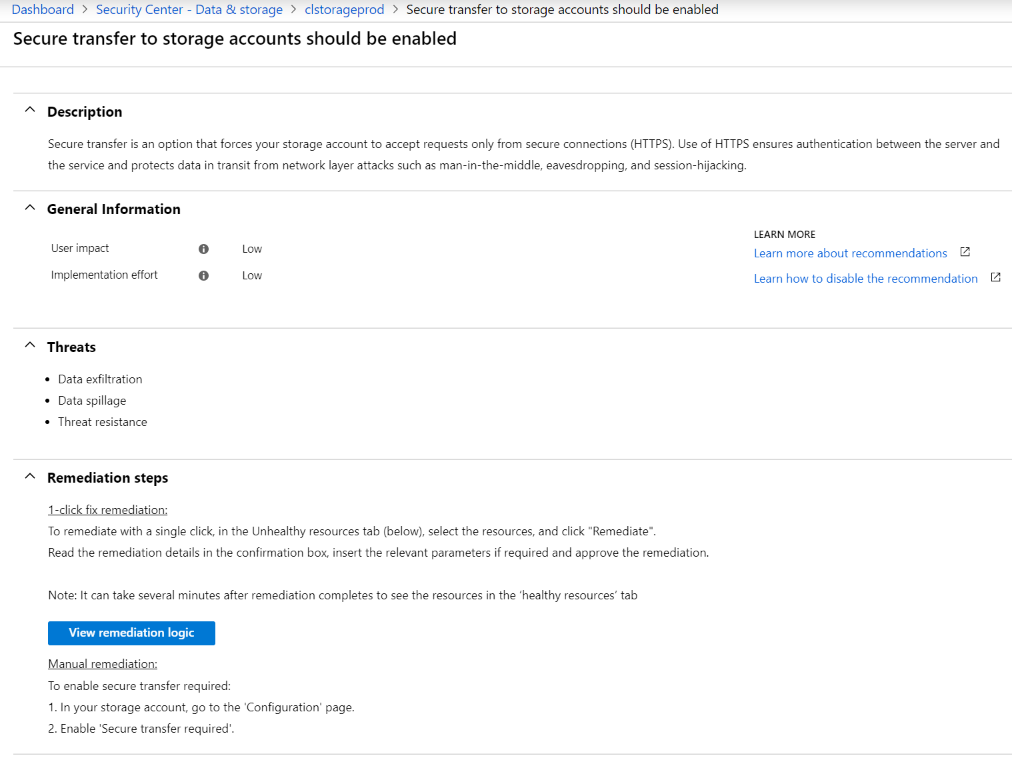


Figure 16 Recommendation for CollaBoard production storage



*Services and Client, how the security is implemented*

# Continuous Security - OWASP Dependency-Check

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