# Excalibur
# Whitepaper

Last edit: 22.5.2019
mailto: xclbr@xclbr.com
https://getexcalibur.com

# Table of contents

# Introduction

Excalibur utilizes the user's smartphone to act as a secure hardware token for any and all authentication and authorization needs. The ultimate goal is to move all forms of authentication and authorization away from passwords, replace them seamlessly with smartphone-based strong but user friendly multi-factor authentication. Excalibur unique value is in providing backward compatibility with all the applications, Operating Systems (OS) and services used today thus creating a bridge between the password-based present and password-free future.

One of the core innovations of Excalibur is its ability to defeat all attacks on credentials as Excalibur is able to automatically change a password on each login. In the Excalibur user flow – the password is no longer entered by the user – the user never even knows the password, it is just a random string used in the background, seamlessly injected into the login process by Excalibur. The user instead just interacts with the smartphone – using it to provide various authentication factors as required by the defined security policy.

Clearly for such scheme to work - compatibility is the decisive factor. Excalibur supports all major platforms and authentication protocols – yet there always might be some legacy apps that require custom integration. To deal with such situations and allow for a gradual rollout - Excalibur supports on-demand displaying of a randomly generated password after successful authentication. This means even if there are systems not supported by Excalibur, the user will just need to retype a OTP password which will be changed after use.

There are several ways how Excalibur is able to handle on-boarding, either via sending a registration QR via email or if password based authentication is still used self-registration can be performed.

During self-registration, the user is asked to provide the current password. This password is used for a purpose of authenticating the user – at the moment when initialization is performed – the user is still using password based authentication. Password is verified – and changed automatically.

# Excalibur Design

Excalibur was designed to overcome limitations of existing authentication systems, core design elements can be summarized as:

- **Straightforward user experience** with no room for error or the possibility for the user to delegate access – eliminating human element weaknesses and exploitation attacks such as social-engineering or phishing.

- **Elimination of passwords** as a mean of User authentication – thus no more passwords to remember.

- **Replacing static passwords with a distributed PKI scheme** – passwords are still used on background to provide legacy compatibility with all the password-based systems used today but can be dynamically changed on each use thus degrading their security value to one-time passwords, thus no more passwords to steal / reuse.

- **Physical security and peer-verification** – by “freeing” authentication from passwords, unique novel authentication / authorization flows are possible – colleagues / managers are able to verify directly from their mobile phone that you are really you instead of having to wait for IT Security to react in case of any security incident / policy violation - thus allowing for stricter policies, yet dynamically adjusting to real life needs.

- **Avoiding single point of failure** – the system must be fail-safe from both reliability and security perspectives even in worst case situations.

- **Reliability perspective** – User must always be able to login in any online or offline situation even if any / all Excalibur components fail.

- **Security perspective** – even if any single Excalibur component is fully compromised, the system function must not be affected, this is accomplished by distributed storage of credentials (an XOR random scheme, protecting data at rest) and a distributed PKI scheme providing means to verify authenticity of each action and event in a distributed manner, where every action has to be signed and verified by multiple separate components - meaning that even if Excalibur Server would be totally compromised attacker would not be able to gain access to any Client or impersonate any Token.

# Excalibur Components

1. **CA** - Excalibur Certificate Authority issues certificates to other Excalibur components, for security reasons - Excalibur provides CA as a cloud service, in specific cases can be deployed on-premise.

2. **Server** - provides a persistent network & storage central point, must be reachable by all components, also provides the Management Interface - Dashboard.

3. **Token** - Excalibur uses the smartphone as a security token, that is why we call the smartphone with Excalibur application the Excalibur Token. It is used for interaction with the User - entering authentication factors, showing session history and providing capability to remotely lock / terminate active sessions. Excalibur Token utilizes phone-based biometry and hardware-backed secure element whenever possible.

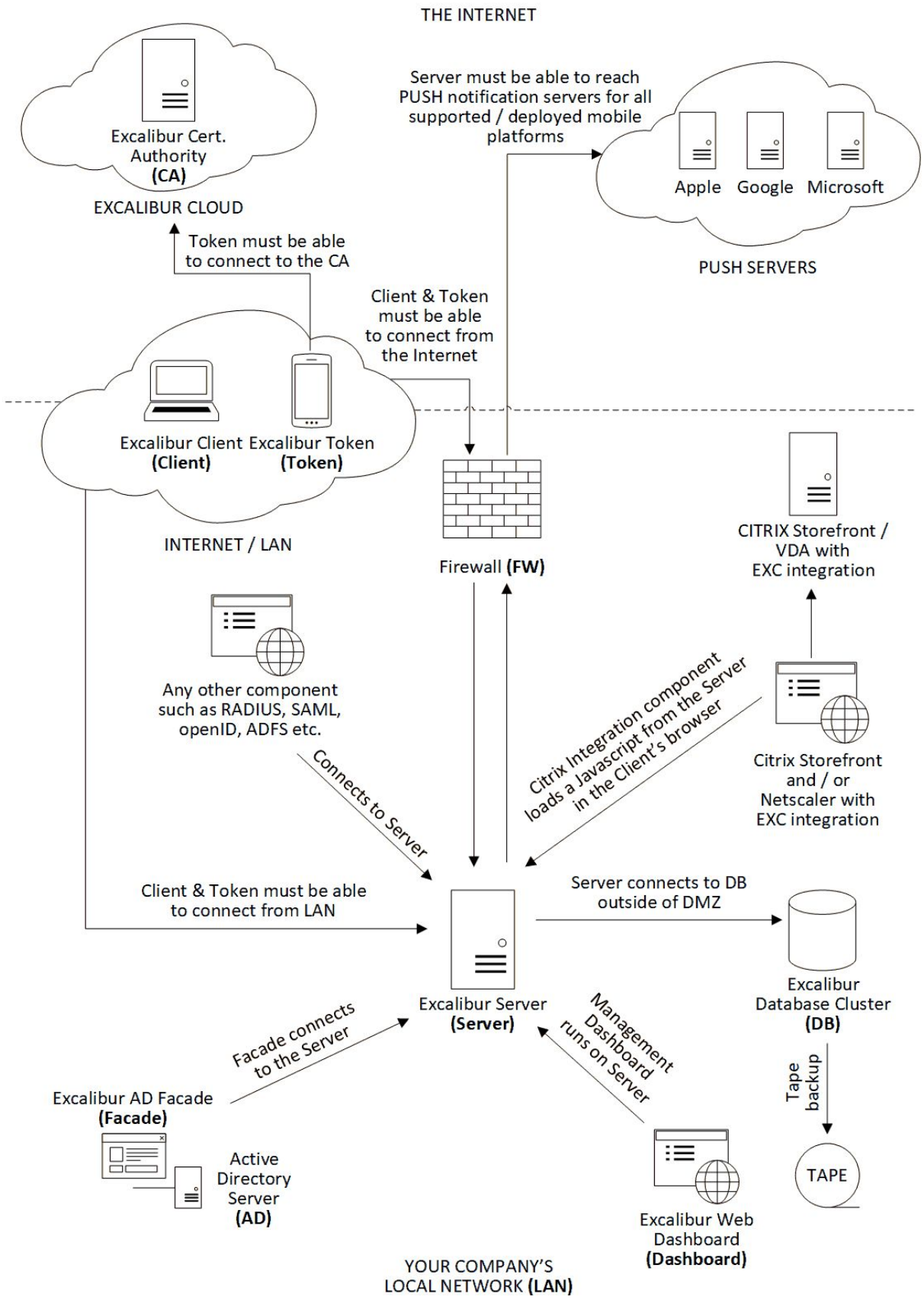
4. **Client** - in the context of Excalibur - Client is any end-point where the User logs into utilizing the Token.

5. **Facade** - Active Directory integration component, must be installed on at least one Active Directory server, runs as a system service and integrates with AD via Directory Replication Service (DRS) Remote Protocol with fallback to LDAP.

6. **Company** - means organization in which all the components (Server, Token, Client, Facade) are deployed.

7. **User** - is the end-user which is using Excalibur to Authenticate, Authorize or Verify.

# Excalibur Topology



# Deployment considerations

Excalibur utilizes TLS over TCP. **Server** listens on the following ports:

1) **Application server port - 6632/TCP** - must be reachable by all clients and tokens, thus ideally reachable from public Internet.

2) **Facade port - 65321/TCP** - must be reachable only from the Active Directory Server on which **Facade** component is installed. **Facade** connects to the Excalibur **Server** on this port.

3) **Dashboard port - 443/TCP** - web (HTTPS) management dashboard must be reachable from management LAN only, used only by Administrators.

4) **Static web content port - 8443/TCP** - used to serve static components necessary for integration to web-based systems such as ADFS, Citrix etc, the visibility of this port should match the visibility of the systems that will load the static content, thus if for example, ADFS is reachable from public Internet, this port must be reachable from the Internet too.

Specified port numbers are default port numbers and can be changed.

**Excalibur connects to push notification servers** to send push notifications to registered Tokens. Firewall must be configured to allow this Excalibur Server outgoing communication.

Excalibur can use built-in MySQL database or it can use any of the following database engines:

1) MS SQL
2) Oracle SQL
3) MySQL

**MySQL is the preferred DB** option. Firewall must be configured to allow this Excalibur Server outgoing communication.

Excalibur Server runs on **Node.js**, always on the latest stable Node.js release at the time of deployment. Excalibur ships with static build of Node.js and with all the required packages pre-packaged - as all packages used are inspected by hand.

The preferred **Excalibur Server Operating System** is the latest **Ubuntu Server LTS** at the time of deployment. Excalibur can ship as a virtual machine, container or it can be deployed from sources on a pre-configured machine. Other Linux distributions can also be used with prior notice before deployment for compatibility testing.

**Pilot deployment HW requirements for the Excalibur Server** are:

- 4 CPU cores
- 4 GB of RAM
- 20 GB of disk for OS / application / internal DB use.

**Facade** must be installed on an Active Directory Server in the domain where the pilot users are and pilot machines are domain-joined. **Facade** does perform cryptographic operations for which the AD Server should have sufficient CPU capacity to accommodate. The AD Server on which **Facade** is installed on can be any **Windows Server 2008R2 and newer**. **Facade** is 64 bit only.

Windows is the preferred Client OS,

**Client OS can be any Windows 7 / Windows 2008R2 and newer**. Both 32 and 64 bit OS versions are supported.

**Token support considerations:**

1) Android 5.0 is the oldest supported version
2) Android 6 and newer are preferred due to native fingerprint support
3) iOS 8 is the oldest supported iOS version
4) iOS 9 and newer are preferred due to Secure Enclave support
5) Hardware biometric sensor is strongly preferred on all devices

# Certificates

1. **Server certificate**
    a. issued by **CA**, included in the installation package
    b. different for each company
    c. privateKey is known only to **Server**
    d. used for mutually authenticated TLS and for ***Signing the DB entries***
    e. publicKey can be obtained by any authenticated component from **Server**

2. **Facade certificate**
    a. issued by **CA,** included in the installation package
    b. different for each company
    c. privateKey is known only to **Facade**
    d. used for mutually authenticated TLS and for ***Signing the DB entries***
    e. publicKey can be obtained by any authenticated component from **Server**

3. **Token certificate**
    a. built-in - used for **first** TLS connection to **Server**
    b. issued - issued by **Server**, used for all consecutive TLS connections to **Server**
    c. both built-in and issued certificate private keys are known only to **Token**

4. **Client certificate**
    a. built-in - used for **first** TLS connection to **Server**
    b. issued - issued by **Server**, used for all consecutive TLS connections to **Server**
    c. Used for encrypting / decrypting Token cryptopart
    d. publicKey can be obtained by any authenticated component from **Server**
    e. both built-in and issued certificate privateKeys are known only to **Client**

5. **User certificates**
    a. **userCompany** certificate
        - issued by **CA** during registration
        - privateKey is known only to **Token**
        - used for ***Signing the DB entries***
        - publicKey can be obtained by any authenticated component from **Server**

    b. **userClient** certificate
        - issued by **CA** during registration
        - privateKey is known only to **Token**
        - used for ***Signing the DB entries*** and ***User Intents***
        - **publicKey is only shared with paired clients**

# Cryptographic operations

**Signing the DB entry** means converting the DB entry to JSON format (sorting keys alphabetically), generating SHA512 hash from the JSON string and then RSA encrypting it with private key.

**Signature** = ***RSA_Private_Encrypt***(*privateKey*, ***SHA512***(***JSONsort***(DB *entry*)))

Signatures can be chained creating signature chain so each entry can be signed by multiple parties.

**Verification of the DB entry signature** means converting the DB entry to JSON format (sorting keys alphabetically), generating SHA512 hash from the JSON string and then comparing it to the RSA decrypted signature with known public key of the signatory for each of the signatures from the signature chain. Each certificate used is verified against CA chain.

**Encryption / Decryption using RSA** - in Excalibur context, data being encrypted is never longer than the key, thus RSA can be used to encrypt the whole data. **Optimal Asymmetric Encryption Padding** (**OAEP**) is used as the padding scheme. By default, all RSA operations use 2048 bit key length.

All certificates are issued utilizing **Certificate Signing Requests** (CSR).

**AES** is utilized with 256 bits key size.

# Token TLS certificate initialization

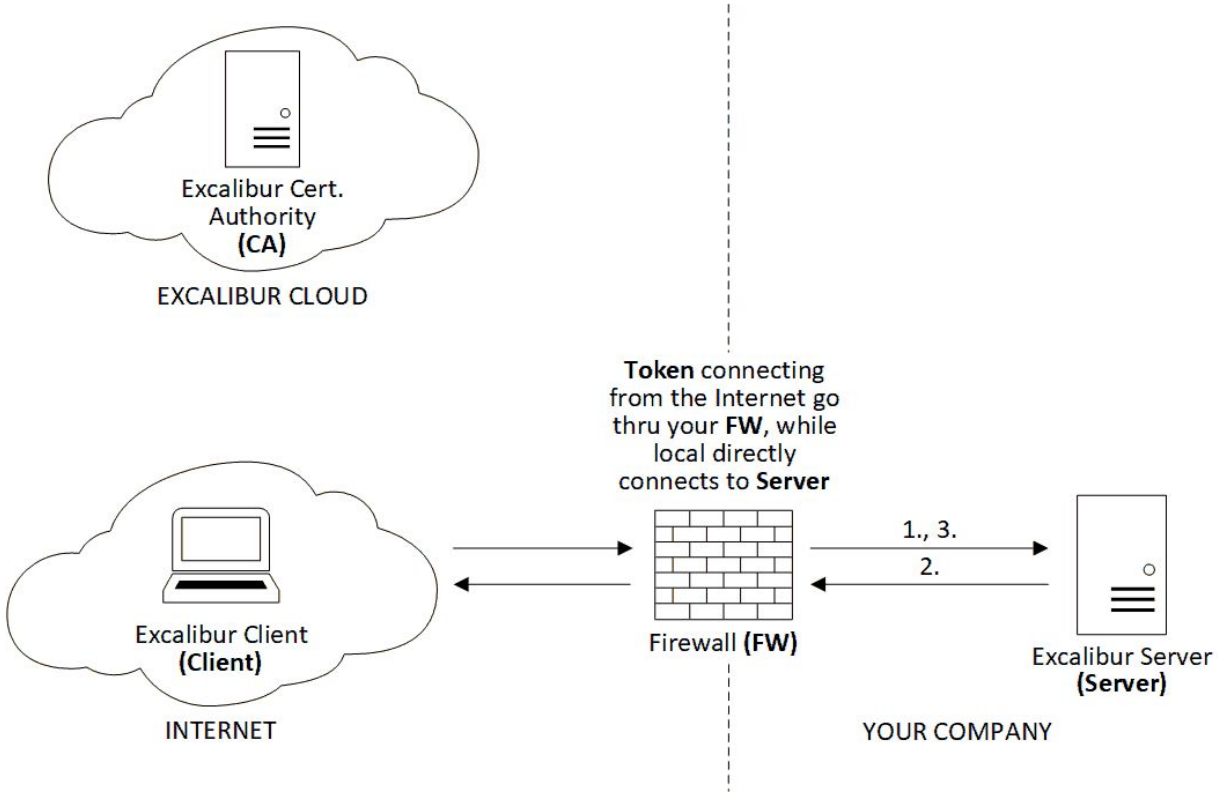
1. **Token** establishes mutually authenticated TLS connection (verified by built-in CA chain) to the **Server** using built-in certificate issued by **CA**.

2. **Server** verifies the Client based on defined policy (for example Client must be connecting to the Server from Internal network).

3. **Server** issues certificate for the token containing L=*companyID*, OU=token, CN=*tokenID*, using **serverPrivateKey** and sends it back to the **Token**.

| **Token DB Entry** | | |
|---|---|---|
| **tokenID** | uint64 | *random id generated by Server* |
| **uid** | string | *unique identifier* |
| **name** | string | *token name* |
| **platform** | string | *Android, iOS, WindowsPhone* |
| **certificate** | string | *tokenPublicKey certificate used for TLS connection* |
| **active** | uint | *active status ( 0 / 1 )* |
| **signature** | string | *Server Signature of the Token DB entry* |

4. **Token** reconnects using newly issued certificate and **Server** authenticates it.



## Client TLS certificate initialization

1. **Client** establishes mutually authenticated TLS connection (verified by built-in CA chain) to the **Server** using built-in certificate issued by **CA** and sends its *name, uid, domainGUID, objectSID* which are read from AD using LDAP thus proving that the Client is indeed domain-joined, further verifications are performed by inspecting Kerberos tickets and system registry to verify the identity of the Client.

2. **Server** verifies the **Client** based on defined policy (for example Client must be connecting to the Server from Internal network).

3. **Server** forwards the received data to the **Facade**.

4. **Facade** verifies that such client (*name, uid, domainGUID, objectSID)* is known to ActiveDirectory domain, generates *Client DB entry* to be stored in **DB** and ***signs the DB entry*** with **facadePrivateKey** and sends it to the **Server**.

| **Client DB Entry** | | |
|---|---|---|
| **clientID** | uint64 | *random id generated by Facade* |
| **uid** | string | *unique identifier* |
| **name** | string | *client name* |
| **type** | string | *client type ( workstation / web / ... )* |
| **data** | string | *additional json encoded data ( domain, objectGUID, ... )* |
| **certificate** | string | *clientPublicKey certificate used for TLS connection* |
| **active** | uint | *active status ( 0 / 1 )* |
| **signature** | string | *Facade Signature of the Client DB entry* |

5. **Server** verifies the **Client DB entry** signature using **facadePublicKey**, stores the entry in **DB** and issues certificate for the client containing L=*companyID*, OU=client, CN=*clientID*, using **serverPrivateKey** and sends it back to the **Client**.

6. **Client** reconnects using the issued client certificate and **Server** authenticates it.

# Policies and Factors

Policy is a set of rules specified for an action performed by Excalibur User which needs to be fulfilled to allow the action. Policy can specify which factors need to be provided by the user, allows the action to be performed just on some subset of clients, inside specified set of geofences, and / or at the right time and day of a week.

Policies can contain any of the following rules and their combinations:

1. Factors
    a. Fingerprint
    b. PIN
    c. Face recognition
2. Geofences - subset of geofences where user must be physically located to perform the action
3. Clients - subset of clients on which the action is allowed to be performed
4. Time of day
5. Day of week
6. IP address of the client
7. IP address for the token
8. Additional verification by manager / admin / support center

When registering, **Token** generates **privateKey**, **publicKey** pairs for each factor it supports using HW-backed secure enclave, **publicKey** is then signed by **Token** with **userCompanyPrivateKey** and sent to the **Server** to be stored for **Factor Verification**.

Fingerprint **private-public Key** pair is generated in such a way that every future signing of the data with private key requires fingerprint to be provided to unlock it.

PIN **privateKey** is only accessible after entering the correct PIN, rate limit is applied both locally and on the **Server**.

Location **privateKey** is used to sign the location sent to the **Server**.

## Factor Verification

When verifying the factor for some intent (see **Excalibur Actions**: User Intent) the **factorPrivateKey** is first retrieved from the secure enclave by providing fingerprint or correct PIN, which is then used to sign intent encoded in JSON format (keys sorted alphabetically). Factor signature is then appended to the intent and sent to the **Server** which can verify that signature with **factorPublicKey** for the given user and factor**.**

When verifying location {latitude, longitude, accuracy} JSON is signed with **locationPrivateKey** and sent to the **Server** to prove that location was indeed reported from the **Token.**

Every Excalibur action is represented by intent DB entry which is signed by the **User** with **userCompanyPrivateKey** and sent to the **Server**. **Server** verifies the intent's signature and user's right to perform the requested action, then asks the **User** to provide all necessary factors according to the matching security policy.

| **User Intent** | | |
|---|---|---|
| **tokenID** | uint64 | *tokenID generated by the Server* |
| **userID** | uint64 | *userID from userCompany certificate, verificator userID* |
| **accountID** | uint64 | *accountID to use while performing action, 0 for verification* |
| **action** | string | *"registration" / "authentication" / "authorization" / "verification"* |
| **actionToken** | uint64 | *actionToken* |
| **targetID** | uint64 | *clientID for authentication, userID of initiator for verification* |
| **targetSignature** | string | *userCompany Signature of the target entry ( client, user )* |
| **timestamp** | uint64 | *current timestamp* |
| **signature** | string | *userCompany Signature of the user intent* |

Intent authentication factor checking is performed by the intent being signed on the **Token** using the **privateKey** of the Authentication factor used to confirm it. Multiple factors and their combinations are supported which is represented by multiple signatures of the intent. **Server** verifies the intent signature(s), verifies it's matching the intent being confirmed, signs the intent using **serverPrivateKey**, stores the intent in DB and forwards it to the target component which verifies the intent and policy before executing performed action.

Policy together with the signed intent and factor signatures of the intent can be then easily verified by each component participating in the processed action, and if just one component fails to verify all the signatures in the signature chain, the action is immediately stopped.

# Excalibur Actions

1. **Registration**
   User needs to register first to perform any other action using Excalibur. During first registration, private-public pairs for factors are created and set for the **Token**.

2. **Authentication**
   Every Excalibur Authentication is intent-based, where the User intent is performed by scanning the dynamically changing (by default every 15 seconds, with 90 seconds validity) QR code from the display of the device where he wishes to login to. The QR code contains *companyID*, *clientID*, *authenticationToken* and *pairingToken* and by its dynamic nature provides a proximity feature - the user must be physically at the device to be able to scan it as it has limited time validity. After authentication factors are successfully verified - password is securely reconstructed at the **Client** and injected into the login process.

3. **Authorization**
   Every Excalibur Authorization is push notification based, for example, VPN-login where username needs to be entered will trigger a push notification to the **Token** of the User specified by the username. On the **Token**, the User is asked to confirm the given action with the exact specification of what is being confirmed and authentication factors are requested. RADIUS is the primary use case of Excalibur Authorization. In these cases, no password is reconstructed as no password is required.

4. **Verification**
   Verification is a special action where the User (Verificator) is confirming another User's' (Initiator) identity directly from his mobile phone, either by scanning a QR code of the initiator phone or by receiving a push notification or using the Management Dashboard. Verification can be configured as a required action for every policy violation, it can be used for PIN reset, registration or login with policy violation. The Verificator confirms the action using his authentication factors and signs the action with his signature. Every policy change is also understood as verification of the change by an Admin.

5. **Factors Reset**
   Registered Users have an option to reset their authentication factors on their Tokens. Firstly, authentication factors are verified based on the security policy and if it succeeds, User proceeds to set new factors.

# Registration

## Self Registration using password

1. **User** begins self-registration by entering his credentials on any **Client** connected to the **Server**. **Client** verifies the credentials using system methods, on success sends the registration request containing the *domain*, *username* and *password* encrypted by **facadePublicKey** to the **Server**.

2. **Server** forwards the received data to the **Facade**.

3. **Facade** decrypts the password using its **facadePrivateKey**, verifies the credentials for the account against Active Directory, generates the **Account DB Entry**, **Registration DB Entry**, signs them with **facadePrivateKey** and sends them back to the **Server**.

| **Account DB Entry** | | |
|---|---|---|
| **accountID** | uint64 | *random id generated by Facade* |
| **uid** | string | *unique identifier* |
| **clientID** | uint64 | *0 for AD account, clientID for local account* |
| **type** | string | *account type ( active directory / local / ... )* |
| **data** | string | *additional json encoded data ( username, domain, objectGUID, ... )* |
| **username** | string | *username / domain\username / email / ...* |
| **active** | uint | *active status ( 0 / 1 )* |
| **signature** | string | *Facade Signature of the Account DB entry* |

| **Registration DB Entry** | | |
|---|---|---|
| **registrationID** | uint64 | *random id generated by Facade* |
| **userID** | string | *0 for any user, userID for a specific user only* |
| **accountID** | uint64 | *accountID* |
| **creator** | string | *registration origin - client / dashboard / ...* |
| **validUntil** | uint64 | *validity timestamp - max 24h from creation* |
| **active** | uint | *active status ( 0 / 1 )* |
| **signature** | string | *Facade Signature of the Registration DB entry* |

4. **Server** verifies the signatures, stores the **Account DB entry** and **Registration DB entry** and generates **Registration QR Code** containing encrypted payload consisting of *companyID*, *registrationID* and *server IP address* which is then sent back to the **Client**.
5. **Client** renders the **Registration QR code**.

6. **User** using his **Token** scans the generated **Registration QR code,** decrypts it and proceeds with setting / verifying his factors (*face, fingerprint, pin, location*). After successful factor verification / setup - registrationID is sent to the **Server**.

7. **Server** retrieves stored **Registration DB Entry** using *registrationID*, verifies its signature and generates *registrationToken* = {userID: *userID*, timestamp: *currentTimestamp, accountID: accountID*}) which is signed with **serverPrivateKey** and sent back to the **Token** together with **serverPrivateKey** signed **Registration DB Entry**.

8. Signed **Registration DB Entry** together with *RegistrationToken* are forwarded to the **CA** by the **Token**.

9. **CA** verifies signatures of **Registration DB entry** proving that both **Server** and **Facade** signed it, on success verifies signature of the *registrationToken* with **serverPublicKey**, verifies that it matches data in Registration DB entry proving its authenticity, checks *timestamp* and issues **userCompany** and **userClient** certificates containing L=*companyID*, OU=user, CN=*userID*. **userCompany** and **userClient** certificates are then AES encrypted with random key *R*, *R* is then encrypted with **serverPublicKey** and both are sent back to the **Token**.

10. **Token** forwards random key *R* encrypted with **serverPublicKey** to the **Server**.

11. **Server** decrypts random key *R* using its **serverPrivateKey** and sends it back to the **Token**.

12. **Token** decrypts **userCompany** and **userClient** certificates using the decrypted random key *R* received from **Server**, decrypted certificates are stored securely at the **Token** using system methods, **Registration Intent** and **UserAccount DB Entry** is created and signed with **userCompanyPrivateKey**. The signed **Registration Intent, UserAccount DB Entry** together with **userCompanyPublic Certificate** are sent to **Server.**

| **Registration Intent** | | |
|---|---|---|
| **tokenID** | uint64 | *tokenID generated by the Server* |
| **userID** | uint64 | *userID from userCompany certificate* |
| **accountID** | uint64 | *accountID of the registered account* |
| **action** | string | *“registration”* |
| **actionToken** | uint64 | *registrationID* |
| **targetID** | uint64 | *registrationID* |
| **targetSignature** | string | *userCompany Signature of the Registration DB entry* |
| **timestamp** | uint64 | *current timestamp* |
| **signature** | string | *userCompany Signature of the registration intent* |

| UserAccount DB Entry | | |
|---|---|---|
| **userID** | uint64 | *userID from certificate* |
| **accountID** | uint64 | *accountID from registration* |
| **active** | uint | *active status ( 0 / 1 )* |
| **signature** | string | *UserCompany Signature of the UserAccount DB entry* |

13. **Server** verifies that the **userCompanyPublic** certificate was issued by the **CA**, on success uses it to verify the signature of **User Account DB** entry, verifies that information in the issued certificate match Registration / User Account DB entries then forwards **userCompanyPublic Certificate** and **UserAccount DB Entry** to the **Facade**.

14. **Facade** verifies that the **userCompanyPublic** certificate was issued by the **CA**, on success uses it to verify the *signature* of **User Account DB,** verifies that information in the issued certificate match Registration / User Account DB entries, signs the *signature of* **User Account DB** entry with its **facadePrivateKey** and sends the updated *signature* back to the **Server**.

15. **Server** may perform additional signing of the **UserAccount DB Entry** based on security policies (for example additional verification by Administrator, Manager or Service Desk operator may follow with each party further signing the signature with its **userCompanyPrivateKey**). Then, it stores the updated **UserAccount DB Entry** to the DB and informs the **Token** about successful registration.

| UserAccount DB Entry | | |
|---|---|---|
| **userID** | uint64 | *userID from certificate* |
| **accountID** | uint64 | *accountID from registration* |
| **active** | uint | *active status ( 0 / 1 )* |
| **signature** | string | *Facade Signature of UserCompany Signature of the UserAccount DB entry* |

## Registration by Admin, Manager or Service Desk Operator

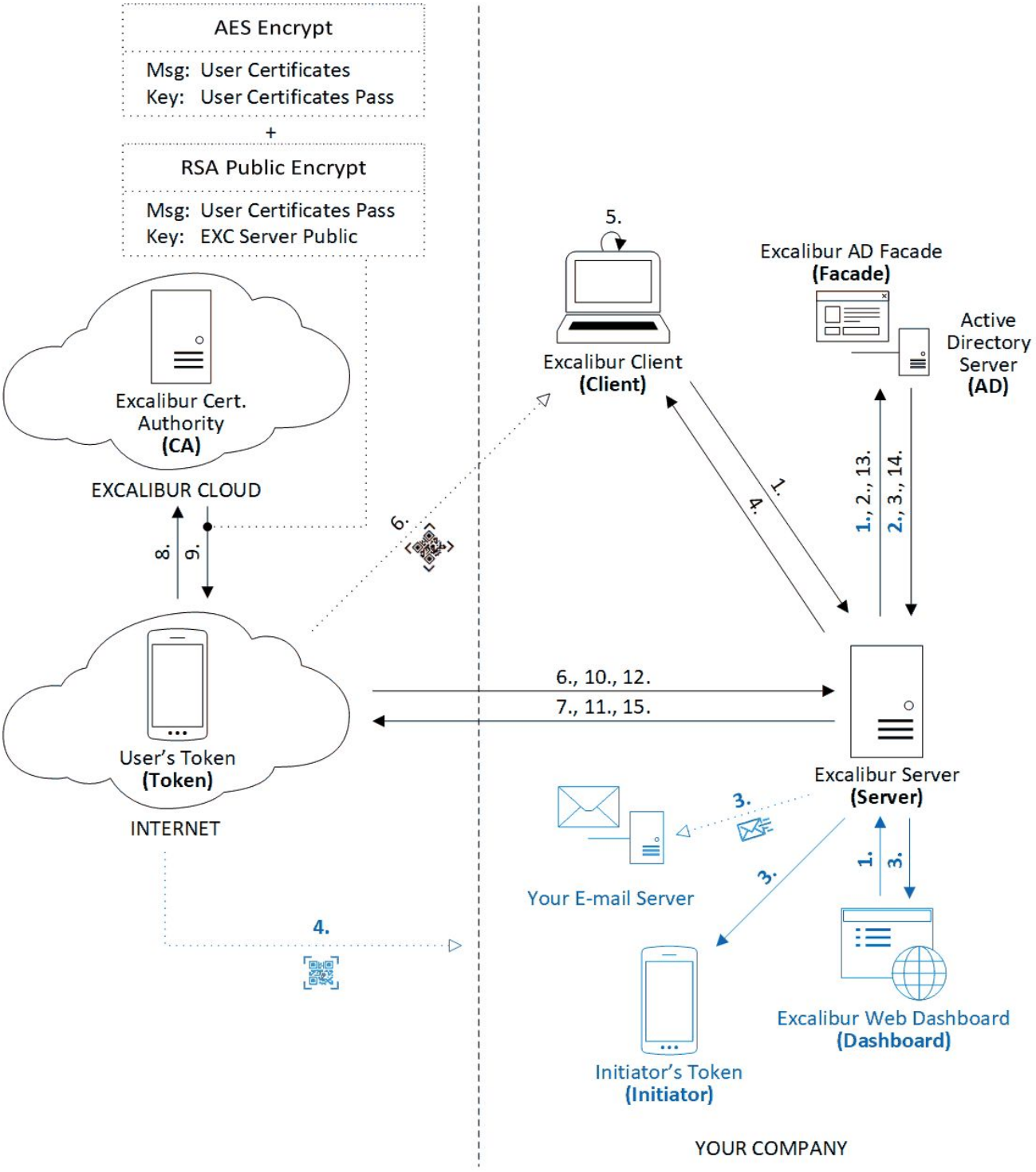
1. **Administrator**, **Manager** or **Service Desk Operator** (*initiator*) begins registration by selecting the user via Excalibur Dashboard. **Server** verifies the permissions of the *initiator* to perform the registration and then sends the registration request containing the *domain*, *username* and *initiator's UserAccount* entry signed by **serverPrivateKey** to the **Facade**.

2. **Facade** verifies signature of the *initiator's UserAccount* entry after that verifies permissions to perform the registration for the user. On success generates the **Account DB Entry**, **Registration DB Entry** with *creator* equal to *initiator* (see step 3

of Self Registration using password) and sends them back to the **Server**.

3. **Server** verifies the signatures and stores the **Account DB entry** and **Registration DB entry**, generates **Registration QR Code** containing encrypted payload consisting of *companyID*, *registrationID* and *server IP address* which is then sent to user's email address or rendered in Excalibur Dashboard / on *Initiator's token*.

4. **User** using his **Token** scans the **Registration QR Code** and continues in step 6 of **Self Registration using password**.



# Authentication

1. Each **Client** generates new **Authentication QR** containing *companyID*, *clientID*, *authenticationToken* and *pairingToken* by default every 15 seconds, with 90 seconds validity. *AuthenticationToken* and *pairingToken* are random 64-bit unsigned integers.

2. **User** begins the authentication procedure by scanning the generated **Authentication QR** with his registered **Token**. **Token** then sends authentication request containing *clientID* to the **Server**.

3. **Server** then retrieves each linked accounts that can be used to perform authentication at the **Client** (*clientID*) for the user from DB and sends them back to the **Token**.

4. **User** then selects which account (*accountID*) will be used for the authentication procedure (if there is only one account it is automatically selected) and replies to the **Server** with **userClientPrivateKey** signed *authentication intent* for selected account and token cryptopart *T* encrypted with **facadePublicKey**.

5. **Server** verifies the *authentication intent*, performs **factors verification** against matching authentication policy and if successful, sends server cryptopart *S* fetched from DB, token cryptopart *T* encrypted with **facadePublicKey**, the client DB entry and the *authentication intent* to the **Facade**.

6. **Facade** verifies the *authentication intent* - signature, and data, on success verifies the signature of the client DB entry, on success decrypts token cryptopart *T* using **facadePrivateKey**, performs XOR operation on cryptoparts *S* and *T* to get *P* ($P = S \oplus T$), verifies against Active Directory that *P* for the given user is correct, on success generates random passphrase *R*, and sends AESencrypt(*R*, *P*) together with *R* encrypted using **clientPublicKey** fetched from client DB entry to the **Server**.

| **Authentication Intent** | | |
|---|---|---|
| **tokenID** | uint64 | *tokenID generated by the Server* |
| **userID** | uint64 | *userID from userCompany certificate* |
| **accountID** | uint64 | *selected accountID* |
| **action** | string | *“authentication”* |
| **actionToken** | uint64 | *authenticationToken* |
| **targetID** | uint64 | *clientID* |
| **targetSignature** | string | *userCompany Signature of the Client DB entry* |
| **timestamp** | uint64 | *current timestamp* |
| **signature** | string | *userCompany Signature of the authentication intent* |

7. **Server** forwards AESencrypt(*R*, *P*), Authentication Intent and *R* encrypted with **clientPublicKey** to the **Client**.

8. **Client** checks if it has got the **userClientPublicKey** and **userCompanyPublicKey** stored in its secure storage. If not, it proceeds with the **Pairing procedure**:

   **8.1.** **Client** sends pairing request containing *authenticationToken* and *tokenID* to the **Server**.

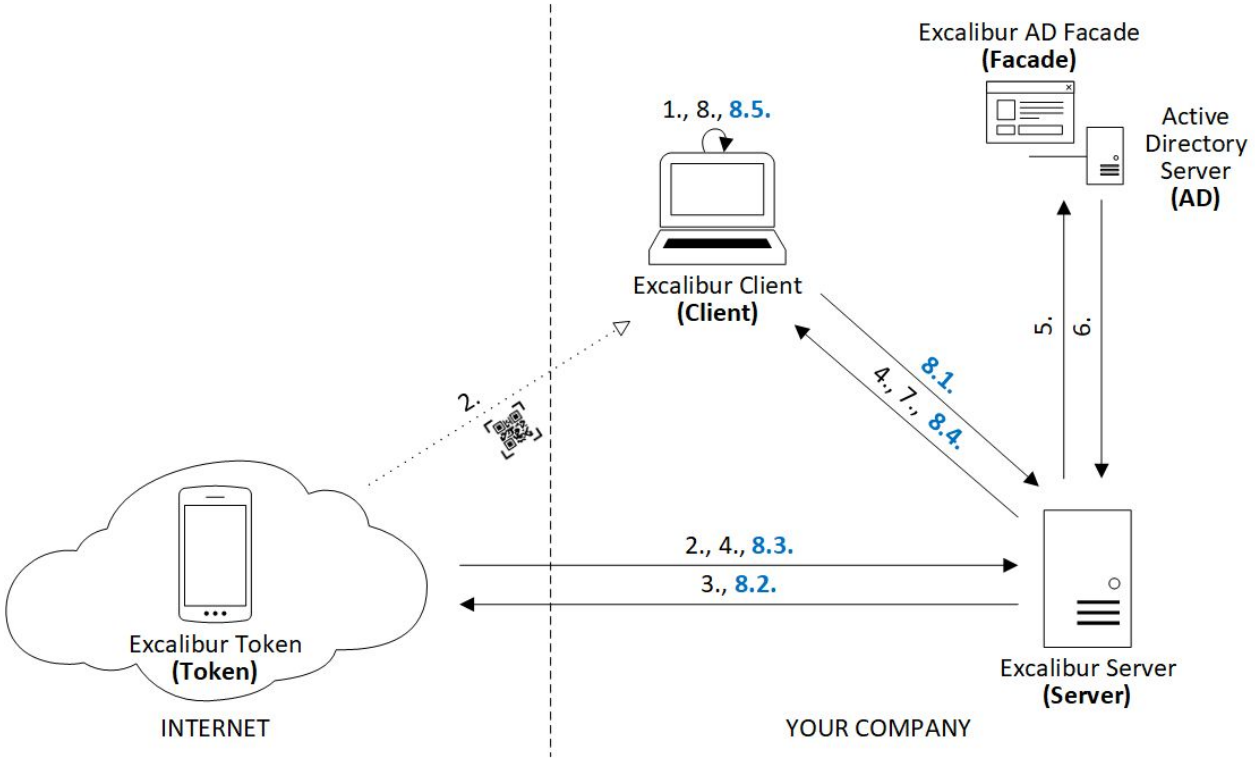
   **8.2.** **Server** forwards the pairing request to the **Token** (*tokenID*).

   **8.3.** **Token** retrieves the *pairingToken* corresponding to the *authenticationToken* and sends ***AES Encrypted*** **userCompanyPublicKey** and **userClientPublicKey** with *pairingToken* as passphrase back to the **Server**.

   **8.4.** As *pairingToken* is only known to the **Token** and the **Client**, **Server** is unable to decrypt the certificates, it just forwards them ***AES Encrypted*** to the **Client**.

   **8.5.** **Client** then using corresponding *pairingToken* as passphrase ***AESDecrypts*** the **userCompanyPublicKey** and **userClientPublicKey,** and securely stores them.

Using the **userCompanyPublicKey Client** verifies the *authentication intent*, then verifies the **Account DB Entry** and using the **clientPrivateKey** decrypts passphrase *R,* AES Decrypts using *R* and gets password *P*. *P* is then used to log into the account.



## Getting/Setting Credentials

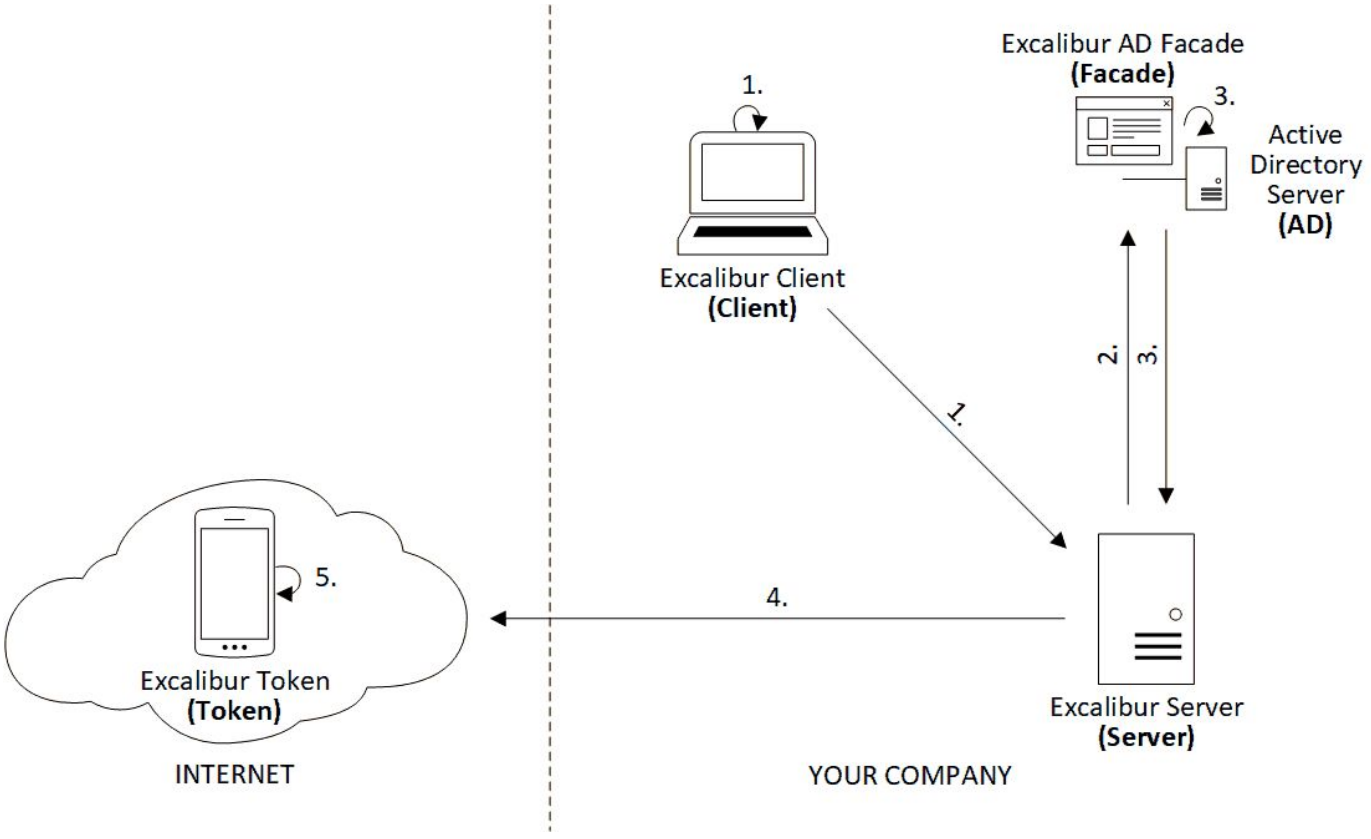
1. If the password is not correct, it's expired or there were no cryptoparts *S, T* present in the **Authentication** request, **Client** proceeds by asking the **User** for the current password for the account (and new one in case of expired password). **Client** then verifies the current password submitted by the **User** (and sets the new password if it was expired). *Domain*, *username* and *password* encrypted by **facadePublicKey** are then sent to the **Server** and **User** is being logged in.

2. **Server** forwards the received data to the **Facade**.

3. **Facade** decrypts the password using its **facadePrivateKey**, verifies the credentials for the account against Active Directory and generates random *S* which is then XORed with the password *P* so we get *T* ($P = S \oplus T$). **Facade** then encrypts the token cryptopart *T* with **userCompanyPublicKey** and sends it together with *S* to the **Server**.

4. **Server** then stores cryptopart *S* for the account in **DB** and forwards encrypted token cryptopart *T* to the **Token**.

5. **Token** then decrypts encrypted *T* using its **userCompanyPrivateKey** and stores it securely.



## Dynamic Passwords

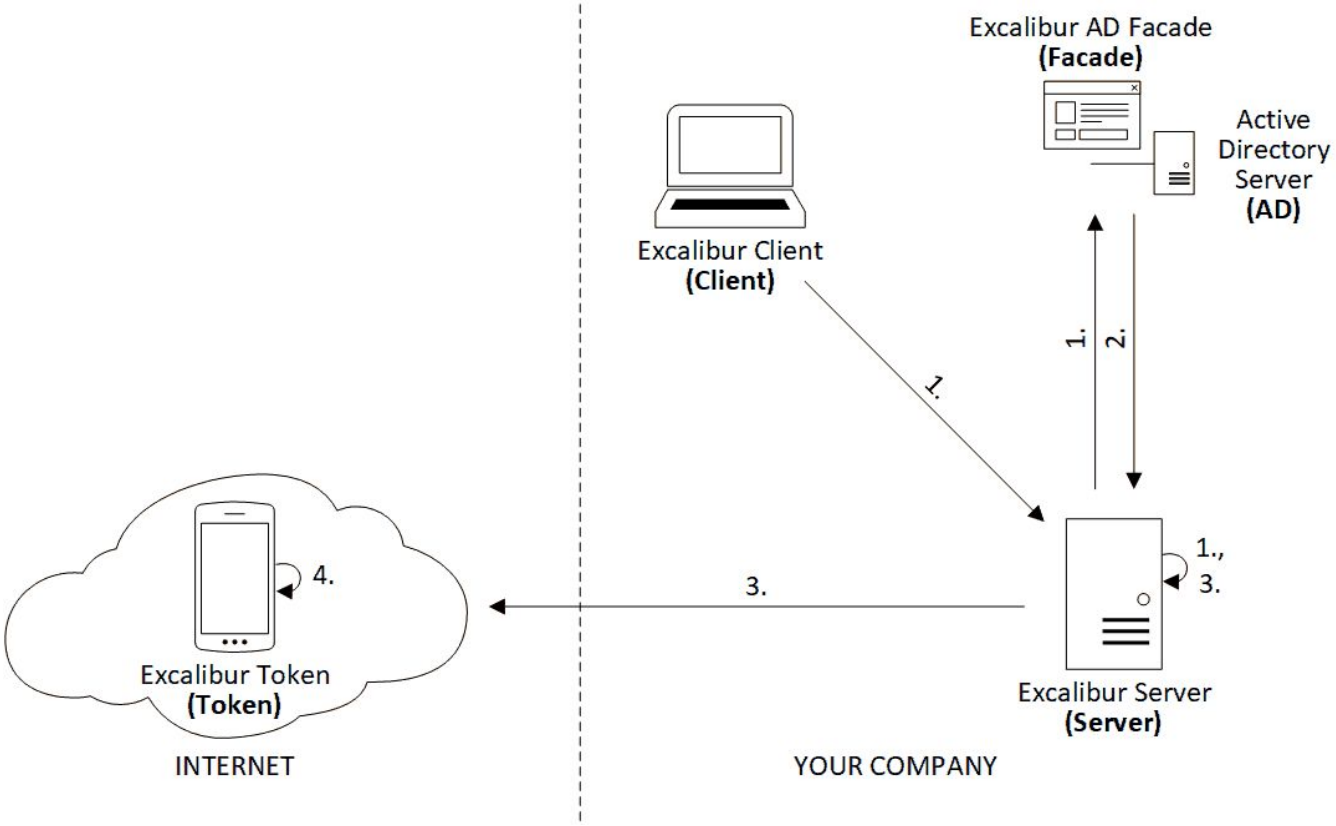
When Dynamic passwords are enabled, the user passwords are changed automatically after the last active session is terminated, effectively creating a system using random One-Time Passwords (OTPs). While there are parallel sessions of a single user on different devices, the password for that user cannot be changed to avoid SSO-related complications.

1. On each session, termination **Client** notifies the **Server** and if there are no more active sessions for **User's** account, **Server** verifies **User's** access to the account - verifying the signature of **UserAccount DB Entry**, and forwards it to the **Facade**.

2. **Facade** again verifies the **UserAccount DB Entry** and generates new random password *P* for the account. Password *P* is then XORed with another randomly generated *S* so we get *T* (*P* = *S* ⊕ *T*). Password for the account is then changed by the **Facade** to *P*, token cryptopart *T* encrypted with **userCompanyPublicKey** and server cryptopart *S* are then sent to the **Server**.

3. **Server** stores cryptopart *S* into **DB**, and if possible send encrypted token cryptopart *T* to the **Token** - otherwise, it is cached in the **DB** until **Token** reconnects.

4. **Token** receives the encrypted cryptopart *T*, decrypts it with its **userCompanyPrivateKey** and stores it securely in its Secure Storage.



# Authorization

1. **Authorization** starts by any *supported component* (for example RADIUS integration component) requesting access on behalf of **User's** account. **Facade** receives the authorization request, generates *authorizationToken* and forwards it together with *account* and *component info* to the **Server**.

2. **Server** fetches *accountID* for the account, *componentID* for the component, verifies authorization request of the component, compares it against policy and sends *authorizationToken*, *accountID* and *component entry* to the **Token** via Push message or via TLS connection if the **Token** is currently connected.

3. **Token** receives the authorization request and displays Authorization dialog (containing Account and Component information) for the **User** with Cancel and Confirm options. After confirming **Token** generates Authorization Intent, sings it with **userCompanyPrivateKey** and sends it to the **Server**.

| **Authorization Intent** | | |
|---|---|---|
| **tokenID** | uint64 | *tokenID generated by the Server* |
| **userID** | uint64 | *userID from userCompany certificate* |
| **accountID** | uint64 | *selected accountID* |
| **action** | string | *"authorization"* |
| **actionToken** | uint64 | *authorizationToken* |
| **targetID** | uint64 | *componentID* |
| **targetSignature** | string | *userCompany Signature of the Component entry* |
| **timestamp** | uint64 | *current timestamp* |
| **signature** | string | *userCompany Signature of the authentication intent* |

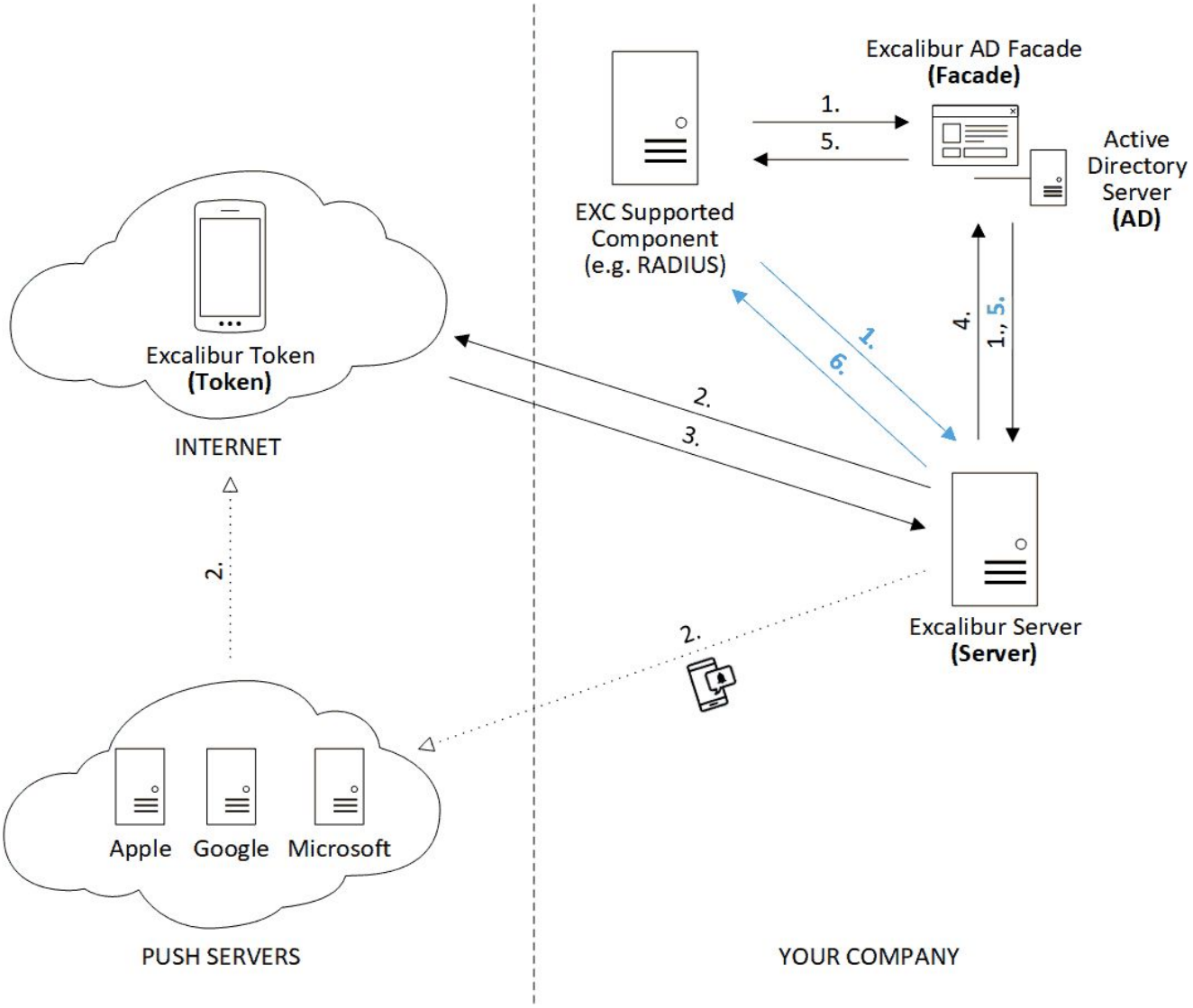
4. **Server** then verifies the *Authorization Intent*, performs **factors verification** against matching authorization policy and if successful forwards verified Authorization Intent to the **Facade**.

5. **Facade** again verifies the intent and then sends Access-Granted reply to the *Component*.

*Supported component* can also communicate with the **Server** instead of **Facade**. In that case, some steps in the flow change as follows:

1. **Authorization** starts by any *supported component* (for example RADIUS integration component) requesting access on behalf of **User's** account. **Server** receives the authorization request, generates *authorizationToken*, associates it with received *account* and *component info*, and proceeds with the regular step 2.
5. **Facade** again verifies the intent and then sends Access-Granted reply to the **Server**.

6. **Server** forwards Access-Granted to the *Component*.



# Verification

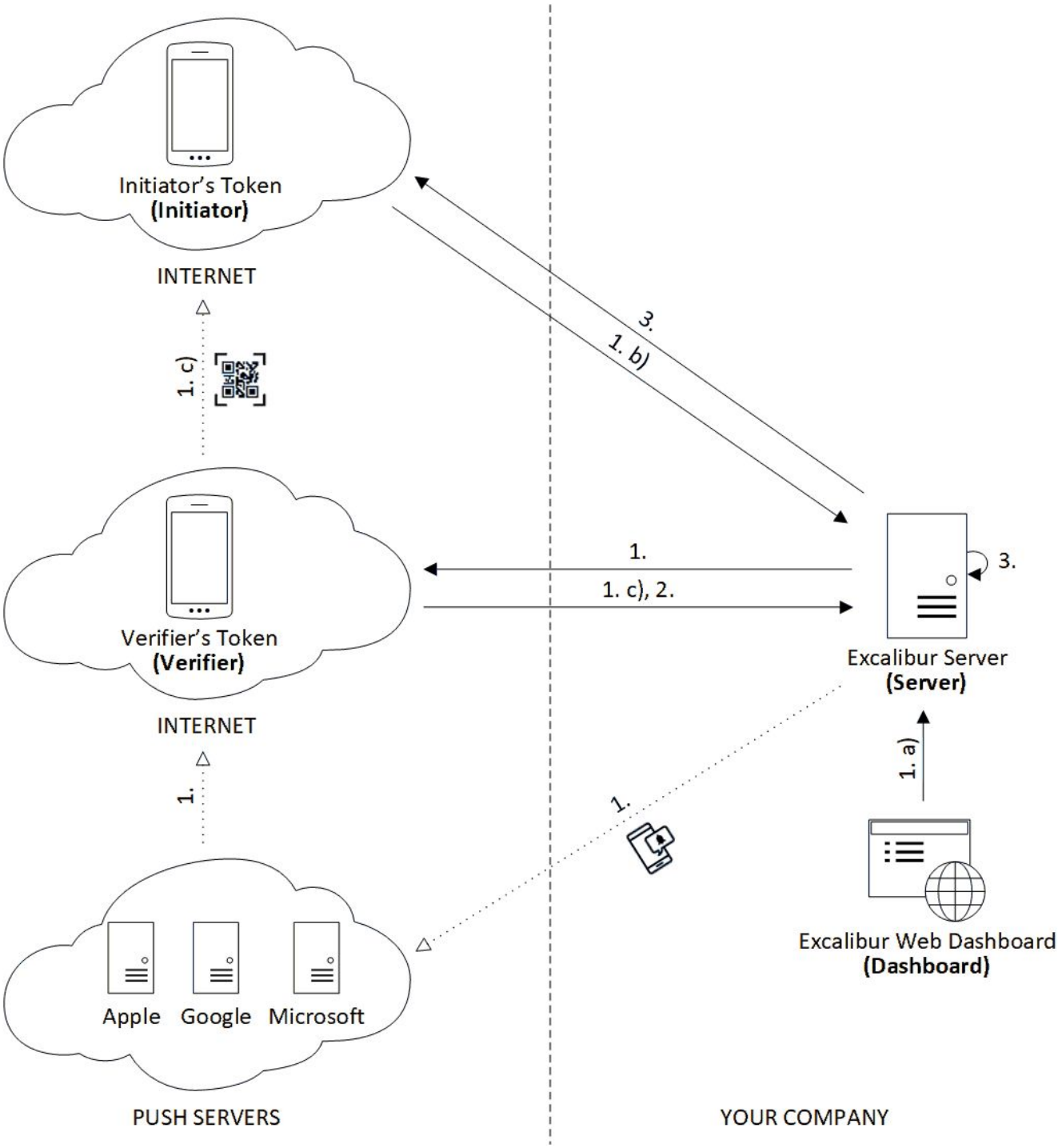
1. **Verification** starts by a) making any relevant changes via Excalibur Dashboard (setting new policies, creating geofences, allowing access for **User**) or b) by some **User** performing action (on *Initiator's token*) which is not allowed without further verification (Verification request might be sent to manager or Service desk operator via push message or TLS connection) or c) by scanning verification QR by verifier. **Server** creates Verification DB entry containing the data which is to be verified (policyID, userID - initiator) and sends it with signed related entries (User entry, Policy Entry) to *Verifier's token*.

| **Verification DB Entry** | | |
|---|---|---|
| **verificationID** | uint64 | *verificationID generated by the Server* |
| **...** | | *Each entryID related to verification:*<br><br>*policyID,*<br>*userID,*<br>*violation,*<br>*etc...* |
| **timestamp** | uint64 | *current timestamp* |
| **signature** | string | *Server Signature of the Verification DB Entry* |

2. *Verifier's token* verifies (signature of) each related entry and then displays the content of verification (information about user, details regarding the policy and action which is to be performed) and asks for user's confirmation. When confirmed, *Verifier's token* generates verification intent, signs it and any updated DB entry with its **userCompanyPrivateKey**, and sends it to the **Server**.

| **Verification Intent** | | |
|---|---|---|
| **tokenID** | uint64 | *tokenID generated by the Server* |
| **userID** | uint64 | *userID from userCompany certificate* |
| **accountID** | uint64 | *0* |
| **action** | string | *"verification"* |
| **actionToken** | uint64 | *verificationToken* |
| **targetID** | uint64 | *verificationID* |
| **targetSignature** | string | *userCompany Signature of the Verification DB entry* |
| **timestamp** | uint64 | *current timestamp* |
| **signature** | string | *userCompany Signature of the authentication intent* |

3. **Server** then verifies the *verification intent*, performs **factors verification** against matching authentication policy and it verifies the action of the Initiator or updates the DB entry.


Initiator's Token
(Initiator)
INTERNET
1. c)
3.
1. b)
Verifier's Token
(Verifier)
1.
1. c), 2.
3.
Excalibur Server
(Server)
INTERNET
1.
1. a)
1.
Excalibur Web Dashboard
(Dashboard)
Apple
Google
Microsoft
PUSH SERVERS
YOUR COMPANY

# Factor reset

1. **User** begins the factor reset procedure by clicking “Factor reset” button in his registered **Token**. **Token** then generates *Factor Reset Intent* and sends it together with *factorType* to the **Server**.

| **Factor reset Intent** | | |
|---|---|---|
| **tokenID** | uint64 | *tokenID generated by the Server* |
| **userID** | uint64 | *userID from userCompany certificate* |
| **accountID** | uint64 | *0* |
| **action** | string | *“reset”* |
| **actionToken** | uint64 | *flowID* |
| **targetID** | uint64 | *tokenID* |
| **targetSignature** | string | *userCompany Signature of the Token DB entry* |
| **timestamp** | uint64 | *current timestamp* |
| **signature** | string | *userCompany Signature of the factor reset intent* |

2. **Server** verifies the *factor reset intent*, performs **factors verification** against matching factor reset policy and if successful, asks token to set up new factor.

3. After user provides new factor (PIN,Fingerprint) **Token** generates **privateKey**, **publicKey** pairs for the factor using HW-backed secure enclave. **PublicKey** is then signed by **Token** with **userCompanyPrivateKey** and sent to the **Server**.

4. **Server** verifies the signature and stores the **publicKey** for updated factor.