Devolutions

CRYPTOGRAPHIC DESIGN FOR

Password Hub Business

TABLE OF CONTENT

INTRODUCTION 4

SECURITY PRINCIPLES 5

ZERO KNOWLEDGE 5

CLIENT-SIDE ENCRYPTION 5

STRONG DATA ENCRYPTION 5

OVERALL DESIGN 6

LUCID COMPONENT 7

DEVOLUTIONS PASSWORD 8

HUB COMPONENT 8

THE CLIENT 8

TECHNICAL DETAILS 9

INTRODUCTION 9

RANDOM NUMBER GENERATION (RNG) 9

DEVOLUTIONSCRYPTO 9

ELLIPTIC CURVE INTEGRATED ENCRYPTION SCHEME (ECIES) 10

USER REGISTRATION 12

AUTHENTICATION 13

PASSWORD CHANGE 14

INITIAL VAULT SETUP 15

ACCOUNT RECOVERY 15

USER INVITES 16

ACCOUNT PROVISIONING 16

INTER-CLIENT COMMUNICATION 16

USER PROVISIONING VIA EMAIL 17

CLIENT LIMITATIONS 17

USER REVOCATION 18

NO PER-VAULT DATA KEY 18

INTRODUCTION

Devolutions Inc. is committed to provide the safest products and due diligence care in handling customer information. This commitment enforces the implementation of various security measures aimed at preventing and mitigating security threats that may have a potential negative impact on our customers' data or on the ability to provide our services. Effectiveness of these measures are tracked and monitored, but is it enough? Transparency, being a core value of the organization, is mandatory to provide an ongoing trust relationship with our partners and customers. This document is an effort towards that goal.

This whitepaper describes the cryptographic design that Devolutions Password Hub (the “System”) is based on to protect customer data from internal and external threats. It details the selection of and the rationale behind the use of security principles, conceptual decisions, technical implementation details and remaining security considerations and risks.

If you have any considerations or comments to share about this document, please contact us at security@devolutions.net. Our dedicated security personnel will strive to satisfy your needs regarding the security of our products and services.

SECURITY PRINCIPLES

Security principles are concepts that govern all design and implementation decisions for the System. Any failure to comply with those principles would result in the violation of Devolutions' own commitment to provide secure and reliable products and services.

ZERO KNOWLEDGE

The System is designed to prevent Devolutions nor any third party to access customer-owned data. This principle can be enforced only when relying on a strong client-side encryption scheme to prevent anyone, except the data owner, to access the encryption keys. The implementation of the design does not leak any information about encryption keys and protected data at any point in the system or outside the system unless authorized by its owner. Zero knowledge objectives are achieved using Client-Side Encryption.

CLIENT-SIDE ENCRYPTION

Sensitive cryptographic operations must be performed on the client to avoid unnecessary exposure of sensitive information such as encryption keys and confidential data to unauthorized parties. At no point in time, the protected information nor its encryption keys shall be exposed outside the boundaries of the client software or device to maintain its confidentiality. Any information leaving the boundaries of the client system shall be encrypted securely in such a way that it won't be possible to fully nor partially recover its original form. Strong data encryption is required to maintain the security level of this principle.

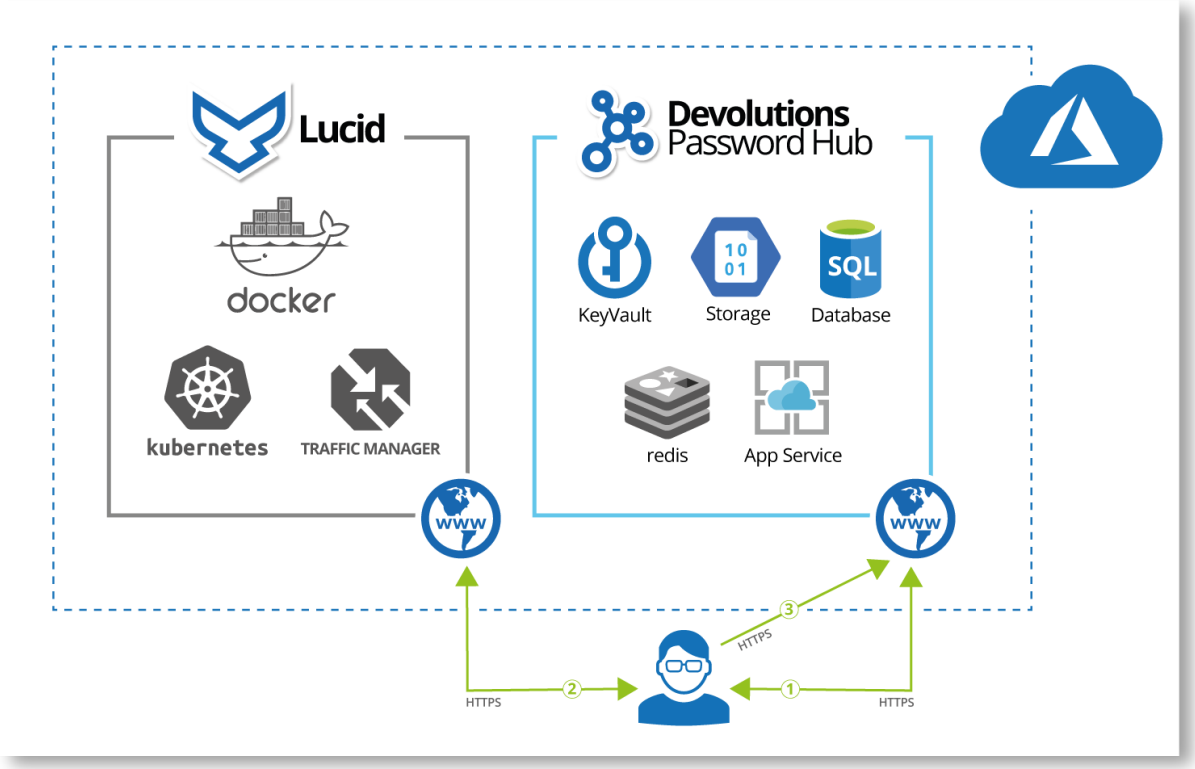
STRONG DATA ENCRYPTION

All data is encrypted using industry-approved algorithms, protocols and settings. Keys are generated, managed, and destroyed securely to avoid any threat that could expose confidential information to compromise by unauthorized parties.

OVERALL DESIGN

The following content presents a high-level view of the technologies and techniques used to apply and maintain all principles from the previous section to the System. Devolutions Password Hub consists of three distinct components that interact with each other. The cryptographic design had to take into consideration all those components and address any potential constraint or inherent risk they induced.

- Lucid, the authentication component.
- Password Hub, the authorization and application platform.
- Client, the web application running on customer device or browser.



As pictured in the diagram above, the relationship between components works the following way. Customers initially contact the Password Hub service over TLS and are redirected to Lucid for authentication (1). Once authenticated, they are redirected again (2) to the initial resource requested (3) : Password Hub.

Devolutions Password Hub is available in two editions; Business and Personal. These editions of the same service address different customer requirements in terms of usage, security and size. They both use the same protocols and cryptographic algorithms for authentication, authorization and privacy. However, they differ lightly on the usage side.

A customer creates a Company that is an isolated private environment for him to use. The first user is called an Administrator and has administrative privileges within the whole Company. This administrator can then create Vaults which can be used to store a collection of Entries such as passwords and documents. Users can be invited, individually or by bulk, and be assigned to Vaults by Administrators.

A customer is assigned an isolated private environment for its use. The first and only user has full access to the whole and only one Vault. The Vault is exactly the same as the Business edition, a collection of Entries such as passwords and documents.

LUCID COMPONENT

This authentication component is located at https://login.devolutions.net and used to validate user identity before granting an access token which is authorized by the Password Hub component. It provides a Single Sign-On (SSO) experience across Devolutions service portfolio over OpenID standard. Lucid is also responsible for user key management. For each user, a corresponding cryptographic key pair (public/private) is stored on the component's storage. The user private key is stored encrypted using a derived value from the user's password. The public key is stored on the component's storage.

With Password Hub for Business, only an Administrator of a Company can get access to the public keys of his own authorized users. The private key transmitted, when authorized after a successful authentication process, to the client application. Since the private key is encrypted with a derived value from the user's password only the client code running on the user's computer can have access to that key. In the Password Hub Personal edition, only one user is available with its own cryptographic key pair. These keys are stored and used just like the Business edition.

DEVOLUTIONS PASSWORD HUB COMPONENT

This component manages user authorization, access control, security, storage and many other services related to Company, Vault, Entry and User management. A unique secret key is mapped to each Company for the Business edition and to each Vault in the Personal edition. This key is used to encrypt the Entry collections stored in Vaults. To provide accessibility to multiple users, the secret key is duplicated and stored encrypted for each authorized user. Each key is stored encrypted using the user public key managed by Lucid. Only authorized users can access their own encrypted key. This model ensures that:

- Password Hub component does not have sufficient information to leak any information about the secret even if access to Lucid information is obtained (public key).

- Password Hub users cannot access data outside their authorized boundary (Company/Vault) since a valid key is required for each of those repositories and each of those keys are unique.

THE CLIENT

Devolutions Password Hub supports a variety of web clients and companion tools. This is where customer data is encrypted and decrypted by the secret presented in the Password Hub Component section. To allow access to the secret key, it must first be decrypted by a derived value from the user's password. More specifically, as detailed in section User Registration, the password hash is used to decrypt a private key, which is then used to decrypt the secret key.

Since the password, its derived value and decrypted keys never leave the Client boundaries, no other component of the solution can obtain information about them. This enforces the requirement that only the Client can access unencrypted Entries stored in Vaults. Information exchanged between the Client and the other two components are also encrypted in transit to provide an additional layer of security. Finally, no other component can grant unauthorized user access to Vaults or Companies without access to the secret key.

TECHNICAL DETAILS

INTRODUCTION

In this section, we explain in detail how the cryptographic system is implemented. A technical background and prior knowledge in cryptography is highly recommended to fully understand this section.

RANDOM NUMBER GENERATION (RNG)

Every time we generate a random value to use for cryptographic purposes, we use a Cryptographically Secure Pseudo Random Generator (CSPRNG) that is industry-approved and well accepted. More specifically, we use the OsRng module from Rust's rand crate available here: https://github.com/rust-random/rand

Under the hood, this implementation uses, on Windows platforms, RtlGenRandom from WIN32: https://docs.microsoft.com/en-us/windows/win32/api/ntsecapi/nf-ntsecapi-rtlgenrandom

When used in a web browser, the getRandomValues function from the WebCrypto API is used to generate initial entropy:
https://developer.mozilla.org/en-US/docs/Web/API/Crypto/getRandomValues

Both underlying implementations are considered secure and safe to use.

DEVOLUTIONSCRYPTO

We implemented our cryptographic schemes in our open source library that can be found here: https://github.com/Devolutions/devolutions-crypto. This source code provides a consistent implementation of those schemes across multiple programming languages and is designed to be misuse-resistant and be safely used without extensive cryptography knowledge. It is important to note that we didn't implement any cryptographic primitives and we use cryptographic modules that either already have been formally reviewed or are considered the standard implementation in the Rust programming language.
The primitives used are;

- XChaCha20Poly1305 with random nonces for symmetric encryption.
    - https://github.com/RustCrypto/AEADs/tree/master/chacha20poly1305

- Curve25519 and x25519 for asymmetric cryptography and key exchange.
    - https://github.com/dalek-cryptography/x25519-dalek

- Scrypt for password hashing used for user authentication.
    - https://github.com/RustCrypto/password-hashes/tree/master/scrypt

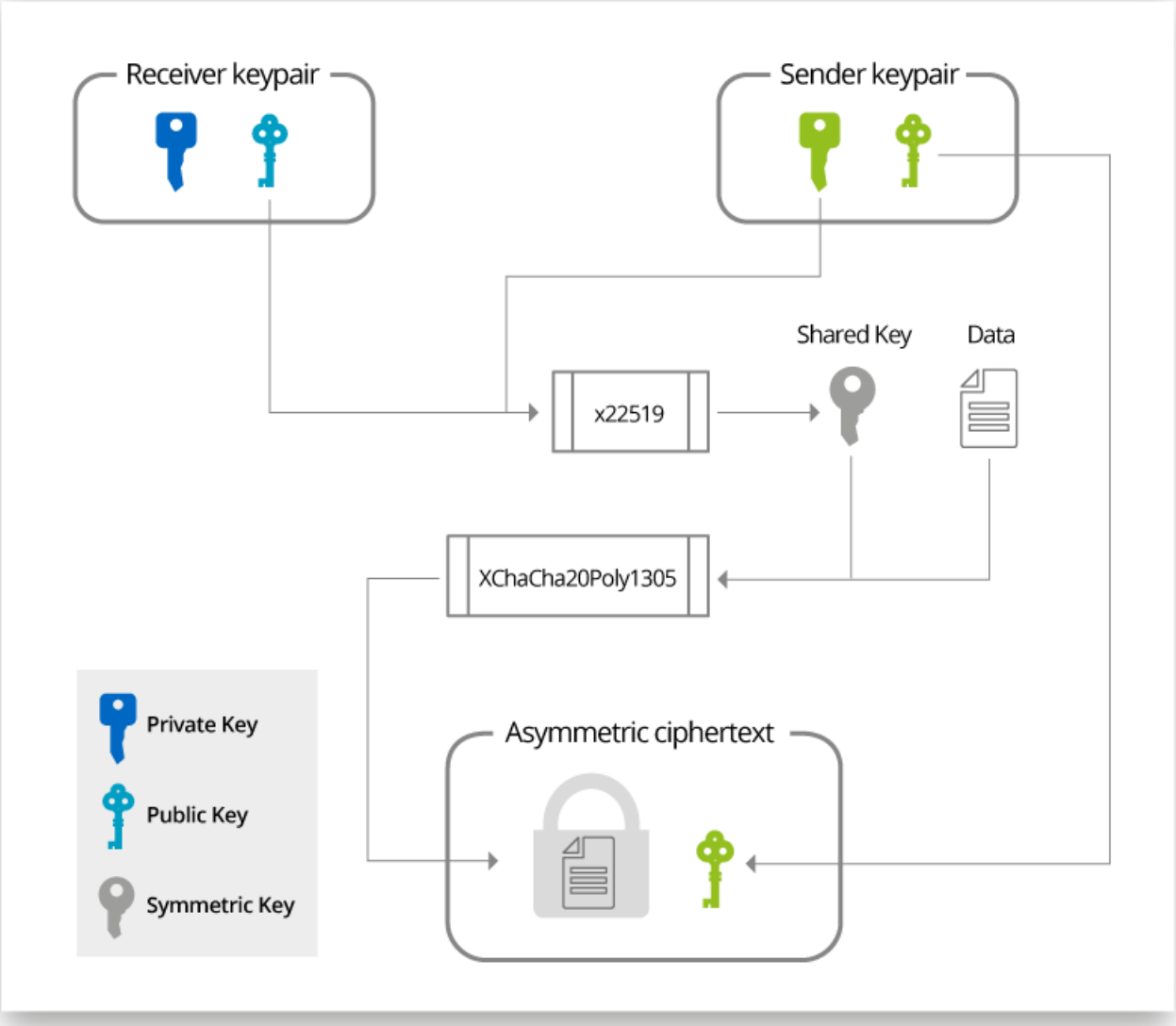
- Argon2id v1.3 to derive the password into a key used for cryptography.
    - https://github.com/sru-systems/rust-argon2

ELLIPTIC CURVE INTEGRATED ENCRYPTION SCHEME (ECIES)

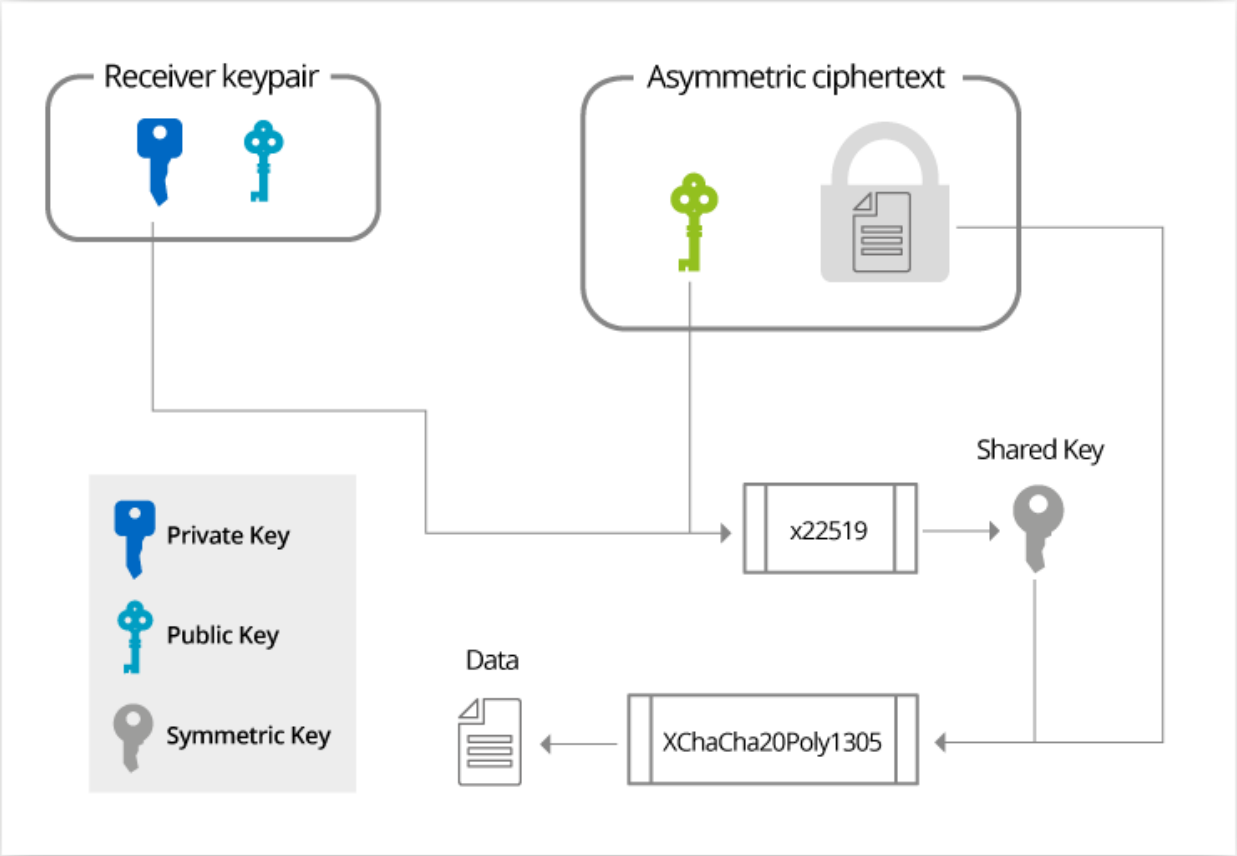
For our asymmetric cryptographic operations, we decided to use Elliptic Curve Cryptography (ECC) with Curve25519 because there are a lot of known issues with discrete logarithm-based primitives like RSA. However, the downfall is that it cannot directly encrypt and decrypt data like we can do with RSA. To circumvent this, we decided to abstract this operation by using the Elliptic Curve Integrated Encryption Scheme (ECIES).
Here's how that works in our case:

1. The receiving party generates a random Curve25519 key pair and distribute his public key, so the sending party has access to it.

2. The sending party can encrypt the data using the public key. Under the hood, what happens is that he generates another random Curve25519 key pair that is used for this specific ciphertext only. He then uses his newly generated private key and the receiving party public key to generate a symmetric key using x25519 (Diffie-Hellman over Curve25519). He then encrypts the data with that symmetric key using XChaCha20Poly1305, a symmetric authenticated cipher. The public key the sending party generated earlier is appended to the resulting ciphertext. This results in what can be simplified as the asymmetric ciphertext.



3. The receiving party can then decrypt this asymmetric ciphertext using his private key. Under the hood, what happens is that he uses his private key and the public key stored in the ciphertext and, using x25519, regenerates the same symmetric key that was used for encryption. He can then use that key to decrypt the ciphertext using XChaCha20Poly1305. If the key is invalid or the ciphertext is corrupted, the decryption will fail.



This process can be found in our cryptographic library under the *ciphertext::encrypt_asymmetric()* and *ciphertext::Ciphertext::decrypt_asymmetric() methods.*

USER REGISTRATION

This is a two-part process. First part creates secrets used to access Password Hub component data while the second part creates secrets to authenticate with Lucid. When the user is created on Lucid, the first part of the user registration process creates a random key pair generated using a secure pseudo-random generator for the specific user on the Client. The user then derives his password into a 256-bit symmetric key with Argon2id v1.3 key derivation function and encrypts his own private key using XChaCha20Poly1305. At the time of writing, the following parameters are used for this process:

1. A random 128-bit salt generated using a Cryptographically Secure Pseudo Random Number Generator (CSPRNG).

2. A degree of parallelism of 1 since the operation is performed within a browser most of the time and, therefore, cannot be executed in parallel.

3. The use of 4096 KiB (4 MiB) of RAM is configured to compute the hash. This parameter considerably augments the cost of GPU-based attacks.

4. Two (2) iterations are configured for this parameter. It makes it harder to compute the hash and therefore harder to brute force since the core algorithm used for hashing must be run twice sequentially instead of a single time.

The parameters were determined according to the methodology recommended by the argon2 specification (https://password-hashing.net/argon2-specs.pdf, Section 9). According to our internal benchmarks, this is what we consider as the best time/memory tradeoff in the specific context of this application.

All parameters used to derive the key and encrypt the private key are then transmitted to and stored by Lucid along with the public key and the encrypted private key. Those parameters and encrypted keys are only shared to authenticated and authorized users to prevent pre-computation attacks.

The second part of the process consists of creating a Scrypt hash of the user's password on the Client. This hash is then transmitted to Lucid over secure HTTPS (TLS). Upon reception, Lucid component hashes again the value using Scrypt before storing the final value. Details of parameters used by both components for hashing are included in the following Authentication section.

AUTHENTICATION

Before a user can get access to his parameters and encrypted private key, he must first authenticate with Lucid component. The user must provide his password on the Client which will then be hashed using Scrypt with the following parameters:

- $\log_n = 10$
- r = 8
- p = 1
- Salt = Randomly generated.

Then, the resulting hash is transmitted over secure HTTPS (TLS) to Lucid component. This step ensures that Lucid, and any unauthorized eavesdropper on the communication channel, ever have access to the user password. To further protect this value at rest, the value is hashed again using Scrypt with the following distinct parameters:

- $\log_n = 15$
- r = 8
- p = 1
- Salt = Randomly generated.

Note that both salts are randomly generated using a Cryptographically Secure Pseudo Random Generator (CSPRNG) for each user. The first salt is public and the second one is private and only available to the Lucid server. Both final hashes and parameters are stored on Lucid's side.

Lucid can authenticate a user by comparing the stored value of the final hash with the one shared by the user. If successful, Lucid generates and sign a valid JSON Web Token (JWT) and transmit it back to the user over secure channel (HTTPS/TLS). This token is then used to authenticate the user for further requests.

PASSWORD CHANGE

When a user changes his password, he needs to provide both his new password and his old one to the Client software. The following steps are then performed:

1. The client authenticates itself to Lucid as described in the previous Authentication section.

2. The client retrieves his encrypted private key and related parameters.

3. The client derives his old password using Argon2id with the parameters.

4. The client uses his old password hash as a key to decrypt his private key using XChaCha20Poly1305.

5. The client then generates new salt and set of parameters and uses them to derive his new password using Argon2id.

6. The client uses his new password hash as a key to encrypt his private key using XChaCha20Poly1305.

7. The client sends his new encrypted private key and his new parameters for storage over secure HTTPS (TLS) to Password Hub.

The client also creates and sends a new authentication password hash the same way that is explained in the "Authentication" section. He first hashes his password using Scrypt with the following parameters:

- $\log_n = 10$
- r = 8
- p = 1

- Salt = Randomly generated.

He then sends those parameters and the password hash to Lucid server. On reception, Lucid hashes it again using Scrypt and the following parameters:

- $\log_n = 15$
- r = 8
- p = 1
- Salt = Randomly generated.

Lucid server stores all those parameters and the final hash to authenticate the user when he wants to log in.

INITIAL VAULT SETUP

Upon the creation of the very first Vault for Password Hub for Business or Personal, two 256-bit random keys are generated by the owner's Client using a Cryptographically Secure Pseudo Random Generator (CSPRNG): The encryption key and the emergency key. The encryption key is used to encrypt all user data using XChaCha20Poly1305 for the whole lifetime of the Vault or Company (the latter for Business only). This key is then encrypted once with the Administrator's public key and again separately with the emergency key.

Those two encrypted keys are then transmitted to and stored by Password Hub component.

Transmission occurs over secure HTTPS (TLS). The emergency key is finally encoded in Base64 format and prompted to the user for safe keeping on the Client. If the Administrator forgets his password, the emergency key can be used to decrypt the encryption key and re-encrypt it with the Administrator's new private key.

ACCOUNT RECOVERY

If the user signals that he forgot his password, we send a confirmation email to the address he used to register in order to reset his account. He can then proceed to set up a new password. From there, the process is the same as the initial user registration and a new key pair is generated for the user.

After the password is reset on Lucid, the user immediately loses access to his Password Hub Vaults. For Password Hub for Business, an Administrator can re-invite the user to give him back access to his Vaults. If an administrator reset his password or if the data is stored in a Personal Hub instance, the user will need to enter his emergency key when login back in.

USER INVITES

When a Password Hub Business Company and first Vault are created, only the first Administrator has access to the encryption key. He can share access to the key to other users by inviting them via email.

To invite a user, the Administrator requests a specific user public key from Lucid. If the account does not exist, the Administrator's Client then proceeds to the "Account Provisioning" section below. If the account exists, the Administrator uses that public key to encrypt the encryption key and store it on the Password Hub component for that user. When the invited user logs on later on, he can decrypt the encryption key with his own private key.

ACCOUNT PROVISIONING

If a user invited by an Administrator does not have a Devolutions account on Lucid yet, it is instantly auto-created for him. A random password is generated for the user and used to generate and encrypt a key pair as described in the User Registration section. This password consists of 12 characters randomly chosen among all uppercase letters, lowercase letters, numbers and a set of special characters ("!#$%&\'()*+,-./:;<=>?[]^_{|}~"). When the user logs in, he will be required to change this password.

The password can then be transmitted by the Administrator's own means or sent by Devolutions' email services. Note that the latter has been added for simplicity but cannot be used without granting Devolutions access to your data (see the User Provisioning via Email section under Security Considerations). It is therefore not recommended to send the password with that feature and should be considered as breaking the zero-knowledge principle of this current design.

INTER-CLIENT COMMUNICATION

Before the authentication process, when a user attempts access to Devolutions Password Hub, the Client generates a random key pair that is used to protect the user private key from the authentication server. He then stores his key pair locally and redirects to the authentication server with the generated public key as a URL parameter.

Once the authentication phase is done and the authentication client has the user's private key, it is encrypted with the public key that comes from the Devolutions Password Hub client and sent to the authentication server. The authentication server then puts that encrypted key in the JSON Web Token (JWT) that it uses to maintain the user session. The browser is then redirected to the Devolutions Password Hub application with the JWT as a URL parameter, where the client can fetch the encrypted user private key. The user's private key can then be decrypted using the private key that the Devolutions Password Hub client generated before the login.

SECURITY CONSIDERATIONS

USER PROVISIONING VIA EMAIL

When an Administrator invites a user that does not have a Devolutions account, a temporary password is generated for that user on the Client. For the sake of simplicity and enhanced user experience, the Administrator is offered the option to send this password to the user via email. It should be noted that when using this option, the zero-knowledge property of this design breaks since the password is transmitted over Devolutions-owned infrastructure and the Internet to deliver the email to its destination. While the user is required to change its password when he first logs in, this does not regenerate a new key pair. The following mitigations are in place to reduce this risk.

- There is a warning prompted to the Administrator before usage of this feature to inform and prevent unintentional exposure.
- All Devolutions-owned systems within the Password Hub solution's boundaries are transmitting information over encrypted channels (TLS).
- Devolutions do not log or store this email content at any point within its infrastructure.
- The password sent can only be used once and expires after one month.

CLIENT LIMITATIONS

The Client component, being web-based, is limited to the same security boundaries of browsers, web views and HTTP protocols. This comes with a few limitations regarding the zero-knowledge and cryptographic properties of the design. More specifically, Password Hub and Lucid frontends are constantly serving the client code on every application access. Devolutions could technically obtain access to encryption keys by modifying the client code without customer notice. Also, the JavaScript ecosystem rely on immutable strings which cannot be zeroed out in memory. Even without the use of immutable strings, it would not be possible to execute code on "exit" and remove sensitive information from memory. However, we have mitigations in place to reduce these risks.

- Access and changes to production are controlled as validated in our SOC2 Type-II report.
- The encryption keys are kept only in RAM and are not stored in the local or session storage of the applications. This means that the keys have a shorter lifespan and are more likely to be overwritten by other data after being discarded. This also means that these keys will not survive a reboot.
- The sensitive data is zeroed out in the WebAssembly module since memory can be manually managed there.

USER REVOCATION

There is currently no way to avoid encryption key retrieval on the client prior to access removal for a user. An access being revoked to a user would not have any effect on the encryption key in his possession. This means that once a user has access to the data, there is no cryptographic way to deny his access other than server-side access controls.

NO PER-VAULT DATA KEY

The cryptographic design outlined in this document is laid out to enforce the zero-knowledge principle. While this principle effectively restricts Devolutions and unauthorized users to access customer data, it does not fully apply segregation of duty for Vaults within a Company. Devolutions Password Hub for Business, as outlined in Section Initial Vault Setup under Technical Details, uses the same encryption key for all Vaults within a Company. If a Vault is compromised, a user having access to any other Vault within the same Company could decrypt its content. Of course, one must obtain the data encryption key for the Company AND access to targeted customer Vault. Controls are in place to reduce such risk and are audited by external auditors as reported in our SOC2 Type-II report available on our main web site.