Security White Paper

2023-07-10

Contents

Overview

Imagine having the power to seamlessly transfer files and text between devices with a simple scan of a QR code or click of a link, all underpinned by cutting-edge encryption technology ensuring your data is completely secure during transmission. Welcome to the world of QRClip.

QRClip is not just a web application - it's a revolution in secure data sharing. This innovative solution empowers you with full control over your data, encrypting all information with a unique key that only you possess. Not even we, the creators of QRClip, can access your encrypted data. Furthermore, QRClip places you in control of your data's lifespan, allowing you to dictate when it self-destructs.

We believe in not just providing a service, but in ensuring complete transparency, too. This whitepaper delves into the heart of QRClip, guiding you through the various security measures baked into the application to guarantee the utmost safety of your data. Our goal is not just for you to use QRClip - we want you to understand its inner workings, too.

With QRClip, rest assured knowing your data's privacy and security is our number one priority.

Data Privacy

All files and text transmitted via QRClip are encrypted with XChaCha20-Poly1305, using a random 256-bit key generated on the user's device. This key is never transmitted to our servers, but instead shared solely by the user via a link or QR code. This implies that only the user possesses the ability to decrypt their data. The data we receive on our servers remains encrypted with a key inaccessible to us.

Attempting to brute force XChaCha20-Poly1305 encryption with a random 256-bit key, as in our setup, is essentially impossible. As it stands, XChaCha20-Poly1305 is considered to be quantum computing-resistant.If someone acquires a user's encrypted data, deciphering it without the decryption key would be impossible.

User Privacy

At QRClip, we highly prioritize the privacy and security of our users' data. To create an account, users are only required to supply an email address and a password. To add an extra layer of security, we hash users' email addresses using SHA256 and a custom salt, rendering even us at QRClip unable to access these email addresses. However, we maintain access to our email delivery service and the associated logs for a duration of 7 days, after which the data is permanently removed.

Regarding password security, we employ the industry-standard bcrypt algorithm for secure storage of our users' passwords. Since no other sensitive information is stored in this database, a potential breach wouldn't compromise the user's data. In the unlikely event of a database compromise, an attacker could only verify the existence of an email address, and to do so, they would also need our custom salt which is exclusively stored on our backend servers. Coupled with a robust, random password, it becomes exceedingly difficult to decipher the hashed password, assuring our users of their security.

As previously stated, all data is privately encrypted and the only metadata we retain is the data size and the number of files shared. We do not maintain logs, and a user's connection to a QRClip is only upheld for the duration of its existence. This information is confidential and is not disseminated externally. Once a QRClip is deleted, it is irrevocably lost.

What is a QRClip ?

A QRClip, retrieved from our backend, necessitates two unique identifiers: an ID and a SubID. The ID serves as the object ID for the QRClip entry in our MongoDB database. The SubID, on the other hand, is a 32-character random string generated during the creation process. This string, comprising a mixture of numbers and upper and lower case letters, yields a total of 60^32 possible combinations.

Access to these two identifiers only allows one to access the encrypted QRClip data. To decrypt this data in its entirety, the encryption key—generated on the user's device and never transmitted to our backend—is essential.

The encryption key is securely embedded within the link, which is accessible both as a QR code and a URL. The key is present in the fragment part of the URL, a segment that is never sent to the server. The QR code encapsulates the full link and can be scanned using the QRClip app or any other QR code scanning application. For safety, we recommend using the QRClip app to scan QRClips to ensure the link is secure and poses no threat to the user's device. Here's an example of the URL also encoded in the QR code:

*https://app.qrclip.io/receive/open?id=**63dd9373338f88eb2a5eb573**&subId=**r8x1D5mqAZE4ftYAgswLj5ZhzElYOkEK#w2NZoJd7WjYDAdKxn1AphK-TuODs05xsr7h4bgSJMT4***

And in general form:

https://app.qrclip.io/receive/open?id=<id>&subId=<subid>#<encryption-key>

To enhance security further, users can set time and download limits, restricting the period of data access. Users can also set a password and delete their data at any time, and optionally grant deletion rights to other users accessing the QRClip.

Encryption

All data within QRClip is encrypted using the XChaCha20-Poly1305 algorithm via the libsodium library (Web Version). The encryption key, common for all data in a QRClip, is coupled with varying Initialization Vectors (IVs) for each part - the text, each file name, and every file chunk. These IVs are generated using the SubID as a basis.

The primary purpose of the IV is not secrecy but to ensure that the same plaintext encrypted with the same key produces different ciphertexts. This characteristic prevents easy detection of patterns by potential attackers. While a simple counter would suffice as an IV, we chose to generate IVs for each part based on the SubID - unrelated to the encryption key. Each part of the QRClip has an IV index, which generates a unique IV.

Part	**IV Index**
Text Message	0
FileName 0	1
FileName 1	2
File 0 Chunk 0	3
File 0 Chunk 1	4
File 1 Chunk 0	5

We first generate a unique 24-character string containing only unique characters from the SubID. Using factorials, we calculate the permutation of the characters of this string at each index. To prevent overflowing the 64 integers, we limit the permutation to a 20 characters string. The index, padded with four zeros, is appended to reach 24 characters. We then replace the left zeros with letters from the permutation itself. This implementation can be examined in our open-source CLI. To ensure distinctness between adjacent indexes, we divide all the available permutations by 2500. For instance, with the following SubID "IjjQwUqHbqqOliBz9EXcKOjCf5iGanW4" you get the following IVs:

IV Index	Generated IV
0	fCKcXE9zBilObHqUwQjIzBi0
1	9zlQKEicbHfwOCqBXUjIlQK1
2	fOUwXzK9iqCHEclQBbjICHE2
50	fCKXEilbHqUwQj9BczOIli50
1003	CKwjXlQHBqcbfE9zUiIO1003

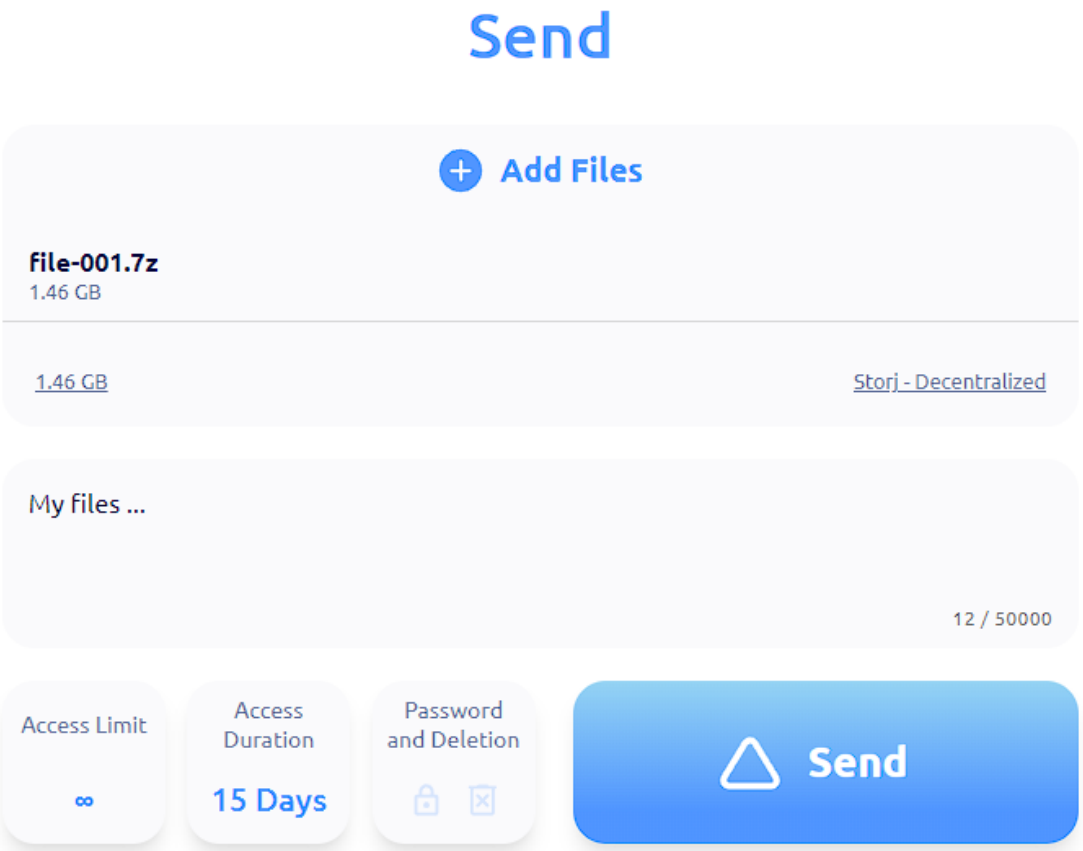
Each file is split into 50MB chunks, each encrypted with the same key but different IVs. The authentication tag is stored alongside the encrypted chunk, but the IV isn't included. If an attacker only has access to the encrypted chunks, the IV for each chunk remains unknown. In theory, this expands the key space beyond 256 bits and amplifies the difficulty of a brute force attack, although a 256-bit key space (offering 2^256 possible combinations) is already near-impenetrable.

To illustrate, the number of permutations for a 256-bit key space is a 77-digit number, approximately 1.157920892373x10^77. The enormity of this number renders brute force attacks practically impossible, bolstering the system's security. For context, the estimated number of atoms in the known universe lies between 10^78 and 10^82.

Generating IVs in this manner was intended for storage space efficiency and to prevent file chunks from being reordered, not primarily for security enhancement. The SubID, used as the foundation for the IVs, bears no relation to the encrypted data or encryption key, and the IVs do not repeat. Compared to a simple counter, this approach neither weakens or strengthens the security of the data.

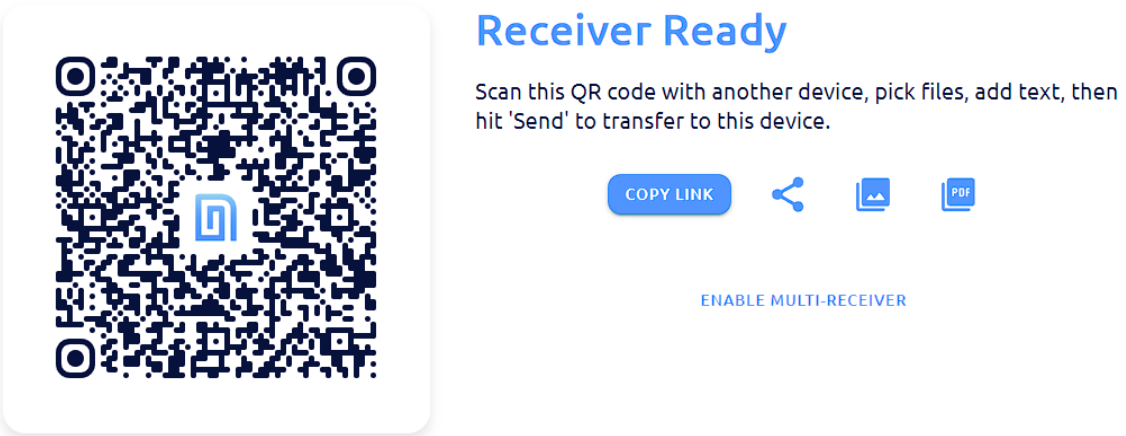
The system's implementation uses two different programming languages: GO for the CLI and Typescript for the client application, each leveraging different encryption libraries. The interoperability of these two components, coupled with mutual testing, fortifies the system's overall security. Currently, only the CLI is open-source and serves as a testament to the system working as described.

How it works



Sending Data

1. The user inputs text, adds files, and selects parameters.
2. Upon clicking "Send", the client sends a request to the backend to create a QRClip.
3. The backend creates a QRClip and sends back the ID and SubID to the client.
4. The client generates a random encryption key, encrypts the data, and uploads it.
5. Upon successful upload, the client generates a link that embeds the ID, SubID, and encryption key.



Sending to Receiver

1. On a PC, the user navigates to "Receive".
2. The PC client requests the backend to create a QRClip.
3. The backend responds by creating a QRClip and sending back the ID and SubID.
4. The PC client generates an encryption key and creates a QR code containing the ID, SubID, and the encryption key.
5. The user scans the QR code with a different device (e.g., a phone).
6. The phone client now obtains the ID, SubID, and encryption key from the scanned QR code.
7. On the "Send Page", the user inputs text, adds files, and selects parameters, then taps "Send".
8. The phone client encrypts the data and uploads it.
9. Once the phone client completes the upload, the backend notifies the PC client.
10. Upon receiving the notification, the PC client automatically fetches the QRClip. As it was the PC client that created the key, it is capable of decrypting the data.

Feature Implementations

QRClip Password Protection

QRClip utilizes password protection to enhance the security of your data through various mechanisms:

Access Restriction: Password protection introduces an additional layer of security that restricts data access exclusively to those who possess the correct password. Even if an individual intercepts the link or QR code, without the password, your data remains inaccessible.

Key Derivation: The application employs the Argon2Id algorithm to derive the necessary keys for QRClip data retrieval and decryption. When a user sets a password, the data is encrypted not solely with the randomly generated key.

To fetch data from the backend, an access key—derived from the user's password—is necessary. We use the first 16 characters of the subID as salt and employ lib Sodium's crypto_pwhash function with 64MB of memLimit and 3 opsLimit to derive a 64-byte length key, known as the Password Key.

The Access Key consists of the bytes from 32 to 48 of the Password Key, which is necessary for data retrieval from the backend. To derive the actual QRClip encryption key, we use the same function and parameters, combining the first 32 bytes of the Password Key with the actual QRClip random encryption key present on the link. We employ the last 16 bytes of the Password Key as the salt. With this derived key, we can decrypt the message and files. Without the password, fetching the encrypted data and decrypting it becomes impossible.

Access Limit Setting: QRClip allows users to establish an access limit, enhancing security further. Every failed password attempt counts against this limit. For instance, if you set an access limit of five attempts, an attacker would only have five opportunities to guess the password before the QRClip is permanently deleted. This limit significantly complicates an attacker's ability to brute-force the correct password.

In summary, the use of a password significantly increases the security of your QRClip.

Multi-Receiver Functionality

QRClip offers a unique feature, Multi-Receiver, allowing users to gather multiple QRClips, containing both text and files, via a single QR code. This streamlined process eliminates the need for email or message exchanges.

Operation

Users activate this feature by navigating to the 'Receive' section and clicking on "Enable Multi Receiver". This function changes the QRClip into multi-receiver mode and generates a unique key pair using the libsodium crypto_box_keypair. This key pair includes a public key, used for encrypting each QRClip's random key for backend transmission, and a private key for decryption. The private key is securely stored on the user's device, ensuring end-to-end encryption.

Afterwards, the user receives a unique link, including the QRClip ID, sub ID, and both keys. The link is user-specific, meaning only the creator can access the QRClips sent to it.

```
https://app.qrclip.io/receive/multi?id=64a2d2b91c14ff22338c81ce&su
bId=smePaz1JXdkrTfuJDmlNnT7edJy0QMz7#zSgFJdlqypWhQ7MN102je-wC5d4v3
_PPXDT7bju81xs&_6-AJOhHyMhsKT57y-O0hXLXmy_SUGKXsiXZVS3ukd4
```

Sharing via QR Code or Link

On the multi-receiver page, a QR code is displayed, containing the QRClip ID, subID, and public key. Users can also share a link with the same information. To send data, recipients scan the QR code or open the link, navigate to the 'Send' page, write a message, and choose files for transmission.

However, there are differences in sending data to a multi-receiver compared to a regular QRClip:

- The QRClip's random key is encrypted with the public key before being sent to the backend. In contrast, a regular QRClip never sends its encryption key to the backend.
- The initial 60 characters, or the first line of the message, is encrypted and sent to the backend to provide a preview in the list. This step is not necessary with a regular QRClip.

Once the QRClip is transmitted, the backend notifies the multi-receiver user to update their QRClips.

Ensuring End-to-End Encryption

If unauthorized individuals gain access to the QRClip database, they cannot retrieve the QRClips sent to the multi-receiver. The keys stored on the backend are encrypted with the public key, which only allows for encryption, not decryption. The decryption requires the private key, which is securely stored on the user's device. Therefore, end-to-end encryption is preserved because the private key is never sent to the backend.

Secret Vault

QRClip introduces a unique feature for enhanced privacy - the Secret Vault. This secure storage area is designed to protect sensitive data, such as encryption keys for various portals. The vault employs end-to-end encryption and utilizes a password separate from the main account password, providing an additional layer of security.

Vault Security

While we consistently verify password discrepancies, including when you change your account password, it remains possible to bypass this verification. This is due to the vault password verification being handled on the client side. Nonetheless, if a user can bypass this, it usually signifies they have substantial technical expertise. When accessing the Vault, users must enter a password that is never transmitted directly to the backend. The application employs the Argon2id13 algorithm (via libsodium crypto_pwhash) to derive the keys necessary for unlocking the Vault. While a user's login session can last several days, the Vault session is designed to be considerably shorter, requiring users to re-enter their password each time they access the Vault. For users seeking a more seamless experience, we offer a 'Remember Me' feature, allowing them to bypass the need to re-enter their password for every session. However, this feature is optional and should be used with careful consideration of its potential security implications.

Vault Creation

The process of creating a Vault involves several steps. Initially, the user is asked to provide their account password, which is compared with the proposed Vault password to ensure they're not identical. The user then enters the Vault password and selects a security level ranging from 2 to 20. The selected security level modifies the memory limit (memLimit) and operation limit (opsLimit) parameters of the Argon2i algorithm. The last step is the derivation of the necessary keys, the generation of a random cipher key (vault master key) and the encryption of this master key.

Vault Security Level

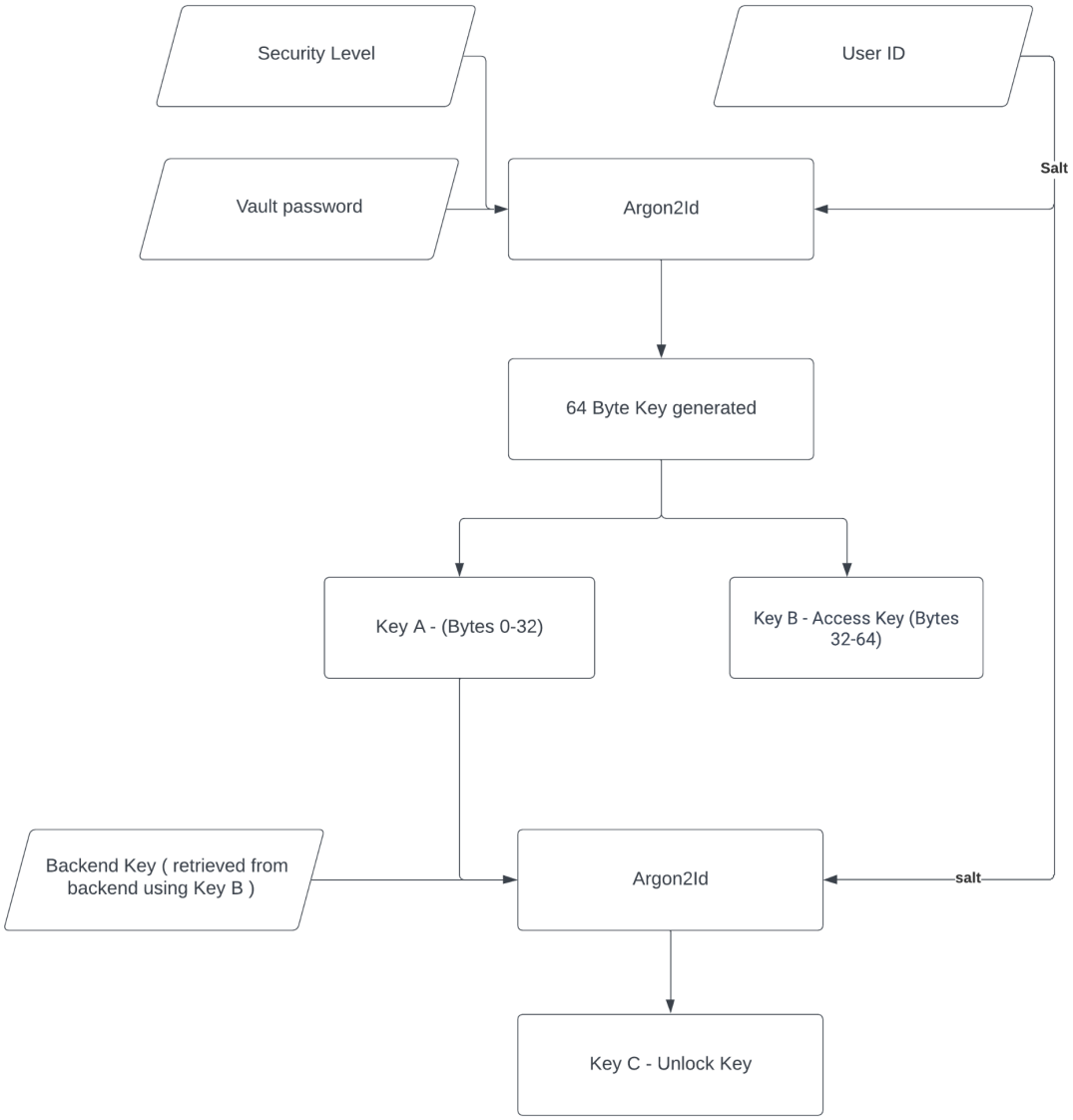
QRClip's Secret Vault provides users with the ability to customize the level of security applied to their vaults, enabling a personalized balance between security and performance. The user selects a security level on a scale of 2 to 20 during vault creation. This level determines the parameters utilized by the Argon2i key derivation algorithm, which provides the secure foundation for the vault's encryption.

The chosen security level directly influences two main parameters of Argon2i: memLimit and opsLimit. The security level multiplies the base memory limit of 16MB (memLimit) and the operation limit of 2 (opsLimit). For instance, a security level of 2 would result in a memLimit of 32MB and an opsLimit of 4. Conversely, a security level of 20 would result in a memLimit of 320MB and an opsLimit of 40. In practical terms, this means that the higher the security level, the longer it takes to open the vault. A security level of 2 might only require a second to open the vault, whereas a security level of 20 could take more than a minute. This flexibility allows you to balance convenience and security according to your personal needs and patience.

This approach ensures that even at the minimum level of 2, the parameter settings already exceed the minimum recommendations set by the Open Web Application Security Project (OWASP). Therefore, regardless of the chosen security level, users can be confident that their Secret Vault meets and surpasses industry-recognized security standards. However, higher security levels offer even more robust protection, albeit with increased computational demand. This flexibility allows users to adjust the level of security to their particular needs and device capabilities, enhancing the user experience without compromising on security.

Key Derivation

The password is processed through the Argon2i algorithm to derive keys. Initially, two keys, known as KeyA and KeyB, are generated using the user's ID as the salt and the chosen security level. KeyB serves as the access key that retrieves vault data from the backend. After the access key is retrieved, the Vault metadata, which includes an encrypted key and a random 32-byte key (Backend Key) generated at the time of Vault creation, can be fetched from the backend. This Backend Key is crucial for deriving the final Vault key. A third key, KeyC, is derived from KeyA and the Backend Key using the Argon2i algorithm. Known as the unlock key, KeyC is used to decrypt the Vault's encryption key received from the backend. This encryption is carried out using the XChacha20-Poly1305 algorithm.



Password Change and Data Storage

The system operates using a master encryption key, making password changes relatively simple. This process involves deriving new keys, updating the Vault's access key, and re-encrypting the master key with the new key. All data within the Vault is encrypted with the master key. Although there's no direct way to rotate this key if it's compromised, the user can export the Vault data, delete the Vault, create a new one, and then re-import the data.

Import and Export

The Vault provides users with the ability to import and export data in both encrypted and decrypted formats. For encrypted exports, users are prompted to provide a password. The system then generates a random 'salt' (additional data used when encrypting a password) which is used in the key derivation process. This random salt is placed at the beginning of the file, followed by a semicolon (;), and then the encrypted Vault data in Base64 format.

Vault Recovery

During Vault creation, users are provided with a recovery code, comprising the access key (KeyB) and the master key (KeyD). This code can be used to unlock the Vault in case the Vault password is forgotten, allowing the user to change the password.

Deleting a Vault

In cases where a user wishes to delete their Vault, due to a forgotten password or a lost recovery code, the application prompts the user for their email. If the provided email matches the hashed version stored in the system, a random code is emailed to the user, which must be entered into the application to confirm the deletion.

Remember Me

For user convenience, the Vault offers a 'Remember Me' feature. Though this feature potentially reduces security when implemented in a browser, it can be securely used on personal, trusted devices. Upon activation, the application generates a random encryption key, encrypts the access key and the master key, and sends a request to the backend to create a 'Remember Me' instance. This feature includes an expiration time, adding an additional layer of security. The 'Remember Me' ID and encrypted keys are stored locally on the device. Upon subsequent launches of the program, the application checks for a stored 'Remember Me' ID and fetches relevant information from the backend. If valid, the encryption key is sent, allowing the app to decrypt the access and master keys and open the Vault without further user intervention.

In the event that the user utilizes the 'Logout of All Sessions' feature, the encryption key is retained on the backend, adding another layer of security. This precaution ensures that 'Remember Me' data cannot be accessed using a JWT token that was created prior to the widespread logout.

Portals

QRClip introduces the Portals feature, a secure platform that provides end-to-end encryption for data transmission. Users can generate unique links or QR codes for each portal, and each transmitted QRClip can be accessed on a dedicated dashboard. This security measure prevents unauthorized access or viewing of the transmitted text or files.

Portal Creation

Creating a portal involves naming the portal and choosing to use end-to-end encryption for customizing it. Although this option slightly increases load times and URL sizes, it enhances data privacy. During the portal creation process, a key pair is generated on the user's device. The public key is stored on the backend, while the private key is securely stored in the user's Secret Vault, ensuring only the user can access it. Additionally, all sensitive information within the portal, such as client and user names, are encrypted with a unique portal encryption key. This key is encrypted using the public key with libsodium's crypto_box_seal, and only this encrypted version is sent to the backend.

Customizing a Portal

Once a portal is created, users can customize it by changing the color scheme, adding an icon, logo, or background image, and creating a unique message that appears on the send page. Users can also set limits on QRClip parameters like file size and expiration time. For added security, users can set a password for portal operation.

By default, only the portal owner or users with the appropriate permissions can open QRClips sent through the portal. However, these restrictions can be lifted, allowing the sender to distribute the QRClip as a standard QRClip.

Managing Portal Users

Only the portal owner has permission to edit access and open QRClips. However, owners can add other users and assign permissions to them, enabling collaboration.

Portal Clients

Users must create a client to send a link via a portal. This helps users identify the sender of the file. All client names are end-to-end encrypted for added security, and each client has a unique key necessary for sending files and messages.

The portal client URL, incorporating the client ID and key, appears as follows:
https://portal.qrclip.io/649d515b0ada410b08ecd561#4KQgKKEIWBHJtKGnMuABsNGmI2Shi0ZfXKmMuS0zqTc3

This URL allows the app to fetch the public key and portal customization data, which are used to encrypt each QRClip's random encryption key for secure backend storage. The portal's private key then decrypts this information, enabling access to the sent QRClips.

End-to-End Encrypted Customization

Users who prioritize advanced privacy and data protection can use end-to-end encrypted customization. While this feature does extend the portal link's length to incorporate the personalization key and increase the portal's load time due to the additional decryption required for images, it ensures that images remain secure and unseen by anyone without the link, including us. This not only bolsters security but also provides an exclusive viewing experience, reinforcing the private nature of the portal.

Please note that before the portal can fully load, the application must first load the essential encryption libraries to decrypt the images.

That's a broad summary of how the Portals feature of QRClip operates, providing users with a secure and customizable platform for data transmission. From creating a portal to managing its users and customizing its appearance, users have the power to control how their data is shared and viewed.

Architecture

Databases

Our system utilizes two distinct MongoDB clusters. One cluster is dedicated to storing all user-centric data, including associated components such as sessions and refresh tokens. The second cluster manages QRClip-specific data, encompassing the encrypted text and encrypted file names. Additionally, the system incorporates a Redis in-memory data store to facilitate backend communications.

Backend

The system supports the operation of multiple backend servers, each running a single container on an independent machine. To manage and distribute incoming requests, a load balancer is implemented, performing its role without decrypting the traffic – a process that occurs within the backend server.

Frontend

Mirroring the backend infrastructure, the frontend is designed to operate multiple instances concurrently. These instances are managed by a dedicated load balancer, serving as the sole point of external connection. The client application utilizes NGINX containers, each deployed on separate machines, mirroring the backend setup.

In our continued efforts for optimization, we are currently experimenting with incorporating Bunny.net CDN into specific sections of our system, such as the Outlook plugin and portal features. This move is aimed at enhancing speed and reliability, thus providing a seamless user experience.

Update

System control and updates are executed through Ansible playbooks. Ansible is entrusted with the responsibility of building, deploying, and launching Docker containers on each server. The Ansible playbooks, alongside all requisite SSH keys for system updates, reside on a server (the control server), which remains powered off when not in use. The data volume of this server is encrypted and requires manual decryption via a passphrase each time the server initiates. The control server's SSH port and the load balancers are the only system components exposed to the internet.

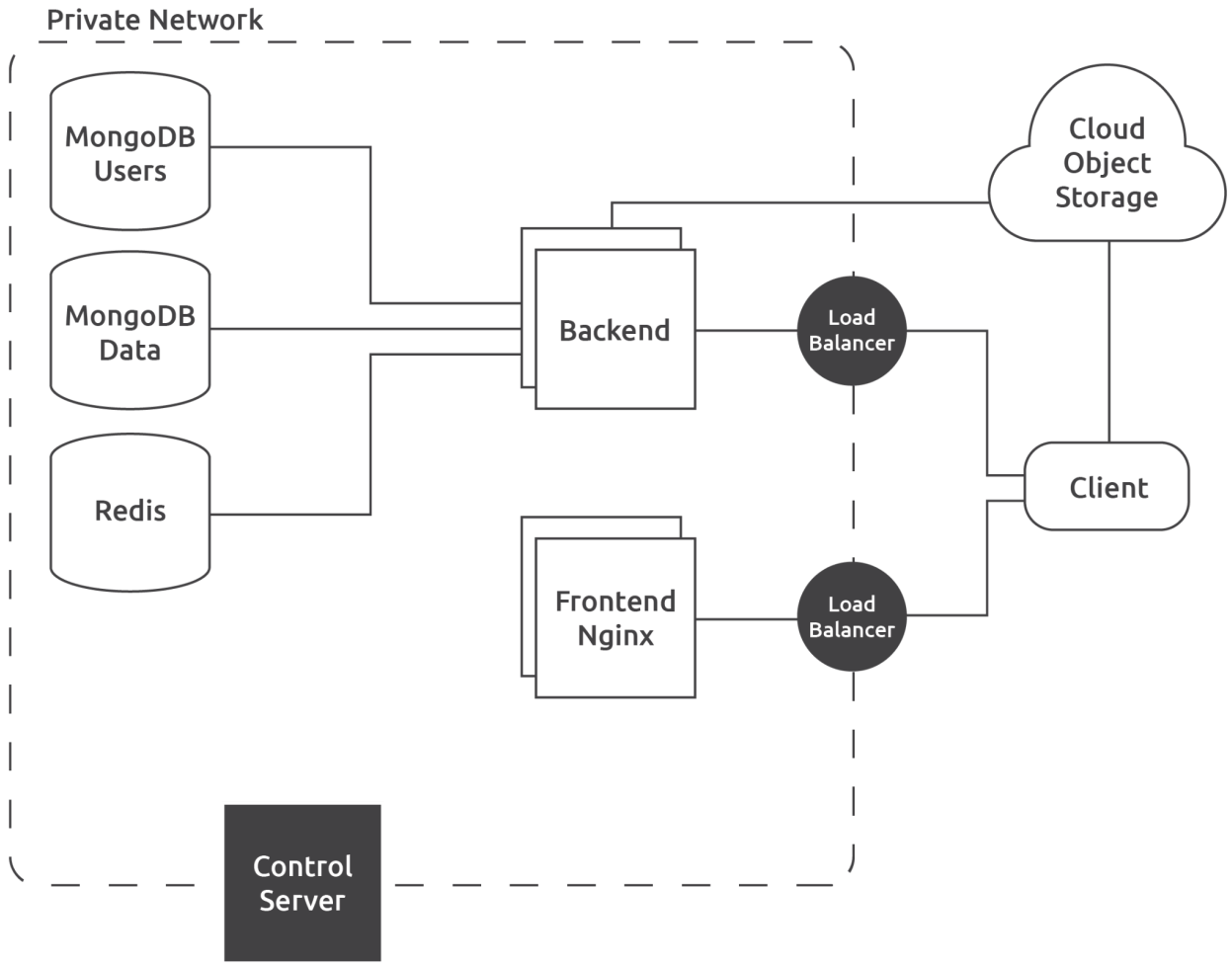
Network

Even though the entire system operates within a private network, all intra-network communication employs SSL/TLS. There are no unencrypted communications, extending to the Redis service which also utilizes SSL/TLS. This ensures that no information can be intercepted, even if the network security is breached. The entire infrastructure operates on the Hetzner Cloud. It's shielded by the Hetzner Cloud Firewall. The system's only points of

connection with the external world are through the load balancers and the control server, which engages for system updates using SSH. Each server within the system is equipped with a dedicated firewall with minimal open ports and necessitates a private SSH key for access.

Infrastructure

Each container operates on its own machine, maintaining a one-container-per-machine policy. All machines possess placement groups, a feature designed to avoid, for instance, the simultaneous allocation of all backend servers on identical hardware, ensuring both security and redundancy. The only data stored outside the private network are the encrypted files. They are stored either in DigitalOcean Spaces or Storj. These files are contained within private buckets and undergo upload/download processes via signed URLs.



Potential Threat Scenarios

In the context of digital security, it is a universally accepted premise that no system can guarantee absolute invulnerability. Novel vulnerabilities are perpetually being discovered, and any system can potentially be compromised. In recognition of this, let's examine hypothetical situations where certain components of our system are breached, and explore what the attacker could and could not achieve under such circumstances.

Total Control Over User Database

The user database houses all user-specific information and session data, including hashed emails, hashed passwords, user credits, account type, portals, secret vaults, along with session-related data.

Under this scenario, an attacker can:

- Attempt to crack user passwords (note: this is an arduous task given our stringent encryption techniques).
- Corrupt data, or outright delete it.
- Illegitimately manipulate credits within user accounts.
- Gain access to a user's vault if, and only if, the user has enabled the 'Remember Me' feature and the attacker has access to the user's computer to retrieve the encrypted keys. Given this, the attacker could theoretically decrypt these keys using the encryption key stored on the backend.

However, the attacker cannot:

- Retrieve users' emails. These are hashed and the crucial salt element is absent, as it is exclusively stored within the backend.
- Establish connections to other servers within the system.

Total Control Over QRClip Database

All the data within this database is encrypted. Each QRClip is safeguarded with its own randomly generated encryption key, which remains out of our possession. This significantly restricts what an attacker could obtain from this database.

Under this scenario, an attacker can:

- Engage in data sabotage by altering the encrypted data, hence making its successful decryption impossible.
- Erase user data.
- Gather enough information to download encrypted files only.

However, the attacker cannot:

- Decrypt the data, whether it is text or files.
- Establish connections to other servers within the system.

Total Control Over REDIS Database

REDIS houses a sequence of MongoDB object IDs, making it the least desirable target for a potential attacker.

In this scenario, an attacker can:

- Disrupt non-critical backend functions.

However, the attacker cannot:

- Establish connections to other servers within the system.

Total Control Over Backend Server

In an unlikely event of a security vulnerability allowing an attacker to gain full control over the backend, they would not command all requests due to the system's multiple backend servers. For argument's sake, let's assume the attacker gains control over all backend servers.

In this scenario, an attacker can:

- Modify the backend code to capture user passwords and emails during login (reinforcing why unique passwords for each account are essential).
- Destroy or corrupt data.
- Retrieve encrypted user data (note: crucial decryption information remains inaccessible).
- Connect to the databases, potentially compromising their integrity or availability. However, it's important to note that even with database access, encrypted data stored within cannot be decrypted without the necessary keys.
- Access the email service using the API key and retrieve email logs (only the last 7 days).
- Access Stripe using the API key (this is a restricted key with very limited permissions).

However, the attacker cannot:

- Decrypt user data.
- Modify the code to retrieve the user vault password. Since the vault password is never transmitted to the backend, and only a derived key is used, obtaining the original password would be impossible.

Total Control Over Frontend

In terms of threat potentials, the frontend remains critical. Without frontend control, attackers cannot gather sufficient information to decrypt user data since encryption keys are generated solely on the user's device.

In this scenario, an attacker can:

- Modify the code to transmit the encryption key and all necessary information to a separate server, allowing them to acquire and decrypt data.
- Change the code to capture the user's password upon login.
- Bypass encryption and transmit data without it.

However, the attacker cannot:

- Establish connections to other servers within the system.

Frontend (Javascript exploit)

Another conceivable method for compromising the frontend is through a third-party JavaScript library that we employ. There are two external libraries we load that could potentially be corrupted. The first, a PDF kit hosted on our proprietary CDN, undergoes an integrity check upon loading, making a compromise highly unlikely. The second is Stripe's payment script. While it lacks an integrity check, we trust Stripe's robust security measures. Furthermore, this script is only loaded when a payment is required and is unloaded upon completion through a page refresh. Additionally, we use various libraries for development that might be vulnerable. It falls upon us to routinely update these packages and ensure there are no issues prior to each release.

Control Server (Complete Control)

In the unlikely event that an attacker gains complete control of the control server, they would theoretically have control over everything, assuming they could decrypt the disk housing all private SSH keys, Ansible scripts, and so forth. As a reminder, this machine is consistently powered down and its encrypted volume requires manual decryption to become accessible.

In this scenario, an attacker can:

- Exercise total control over the system.
- Connect to other servers within the system.

Hetzner Cloud Console Access

In the scenario where an attacker gains access to the Hetzner cloud console, they would need not only the login password but also a physical key, barring any security vulnerabilities at Hetzner's end. In this case, a variety of actions become feasible, though direct server breaches are not among them. However, that becomes irrelevant as it would be simple to create a comparable frontend server with manipulated code and replace the original.

In this scenario, an attacker can:

- Create new servers, download the front-end code from the original server (which is publicly accessible), modify the code, initiate new virtual machines to run the tampered code, and substitute the original servers with the corrupted ones.
- Alter the DNS to point towards different servers. While it would be excessively complicated to implement the API and replace the backend, replacing only the frontend is relatively straightforward.
- Delete everything.

However, the attacker cannot:

- Establish connections to the existing servers within the system.

Physical Access to Servers

This situation parallels the previous point. The most straightforward path for an attacker with physical access to the servers would be to replace the genuine front-end servers with new ones carrying malicious code. Mitigating this risk is somewhat out of our hands. We must trust our service provider to prevent this scenario, though we can certainly take preemptive measures.

Domain Name Configuration Access

While we utilize Hetzner's DNS service, our domain is registered with Namecheap. If an attacker were to access the console, they could effortlessly reroute traffic. In this regard, we employ a secure password and a hardware key.

Note on Potential Threats

Despite the low likelihood of any of these scenarios materializing, they still remain possible. Risks arise primarily from the discovery of vulnerabilities in third-party services, which would not only impact our system but many others as well.

Preventive Measures

Our paramount objective is the protection of user data. To this end, the client application, acting as the front end, plays a pivotal role by encrypting user data using a key generated on the user's device. This key is never shared with any third parties unless the user explicitly authorizes it or triggers the action. Additionally, detecting any code modifications is relatively straightforward as the frontend operates on a web server, where the content is publicly accessible.

We conduct routine checks to verify the code's integrity, although the specifics of our monitoring process will remain confidential.

Our utmost priority is to ensure that the encryption key remains exclusive to the user and inaccessible to third parties. By doing this, we enable our users to maintain control over their data's decryption process.

Summary

At QRClip, safeguarding user data is our prime directive. We provide a secure and versatile file sharing solution that allows users to share data confidently. Advanced security measures are in place to uphold privacy and security, and users can choose to share their data anonymously (between sender and recipient) via QR codes.

For further information, please contact us at info@qrclip.io