MEGA

MEGA Security White Paper

Third Edition - June 2022

Content

1. Introduction

More than 250 million registered users rely on MEGA as their secure file storage and collaboration platform. MEGA pioneered user-controlled end-to-end encryption through a web browser in 2013. In this whitepaper, we'll describe in detail the security principles on which MEGA is built, its technical implementation and the philosophy behind it.

1.1 What is MEGA?

MEGA is a secure cloud storage and communication platform with user-controlled end-to-end encryption (E2EE). End-to-end encryption means that no intermediary - not even MEGA - has access to the user's encryption keys and therefore the stored data. However, users have the option to share data (individual files or entire folders), plus the associated encryption keys, with others.

MEGA is currently the only major cloud storage provider supporting browser access to end-to-end-encrypted cloud storage. This lowers the barrier to entry and supports the mass adoption of encryption.

In addition to its sophisticated web interface, MEGA provides a sync application for all major operating systems (Windows, macOS and Linux) to synchronise data in local folders with the user's cloud in real time. MEGA also provides mobile apps featuring file access, camera uploads and communication while on-the-go.

MEGA's communication suite supports text chat and audio/video calls with individual users or groups and is tightly integrated with all cloud storage features, which provides a unique intersect between the two where it's very easy to share and reference any cloud-stored data securely within any chat.

1.2 End-to-end encryption

Unlike most other cloud storage providers, only the user controls who has access to their data. From the outset, MEGA has been designed around user controlled end-to-end encryption. This means that files, messages and audio/video content is encrypted on the user's client machine before it gets transferred to the MEGA platform. Only the user holds the encryption keys to their data, and not even MEGA is able to access it. If a user wishes to share data with another user, the process encrypts the required encryption keys with the recipient's public key before transmitting them. To ensure the identity of the recipient, their key fingerprints can be verified through an independent channel. Any change will trigger an alarm, eliminating the risk of impersonation through a post-verification man-in-the-middle attack.

1.3 Source code transparency

All cryptographic operations relevant to the security of the users' data take place exclusively on their own devices. MEGA provides transparency of the actual implementation by publishing the full and up-to-date source code of all its client applications. This enables all interested third parties to independently verify whether MEGA's claims are true, whether the implementation is correct and that there are no backdoors or unintended vulnerabilities. Users are allowed to use the source code for research purposes and to build the client apps directly from the source. See also:

https://mega.nz/sourcecode

https://github.com/meganz/

1.4 Privacy

By properly applying E2EE, MEGA achieves actual privacy by design in comparison to most of its competitors who only provide privacy by policy. While all communication and files are user-encrypted and inaccessible by MEGA (or any other third party, unless the key is voluntarily shared), MEGA does store various other types of transaction and metadata, such as the user's email address and IP address, to which privacy by policy applies. The storage of this data is strictly for operational and compliance purposes. MEGA does not carry out any other processing activities on such data. For full information, see also:

https://mega.nz/privacy

1.5 Vulnerability Management

To ensure MEGA's security on an ongoing basis, MEGA offers rewards to anyone reporting a previously unknown security-relevant bug or design flaw. Qualifying categories are:

- Remote server-side code execution (including SQL injection);
- Remote client-side code execution (e.g. in a browser, through XSS);
- Any issue that breaks the cryptographic security model, allowing unauthorized remote access to or manipulation of keys or data;
- Any issue that bypasses access control, allowing the unauthorized overwriting/destruction of keys or user data;
- Any issue that jeopardizes the confidentiality of an account's data in case the associated email address is compromised.

MEGA offers up to EUR 10,000 per vulnerability, depending on its severity.

MEGA has paid out a total of EUR 17,000 for some minor vulnerabilities that were reported between 2013 and 2018. However, MEGA's brute-force challenge, open since February 2013, hasn't been claimed and probably never will be.

You can submit your findings to security@mega.nz
For more information see also: https://mega.io/security/bug-bounty

1.6 Data redundancy

MEGA's end-to-end encryption sets it apart from the cloud storage mainstream, but data protection is not enough. Reliability of the infrastructure and resilience against hardware failures and human error are also important considerations.

MEGA owns and controls its server infrastructure directly and does not rely on any third-party VPS providers, which is beneficial in the age of CPU side-channel attacks ("Spectre"). All hardware is hosted in secure facilities in Europe or in countries (such as New Zealand and Canada) that the European Commission has determined to have an adequate level of protection under Article 45 of the GDPR. No user files are stored in, or made available from, the United States of America.

MEGA has various classes of infrastructure, each with their own redundancy and data integrity protection mechanisms:

Primary backend infrastructure: (API, database clusters, etc). These servers hold all the encrypted keys, the encrypted file metadata, and the personal user data. Any operation in the user's account is managed through the API and stored in one of MEGA's database clusters.

- All database clusters have real-time and delayed slave replication across various geographic locations and have various off-site backup mechanisms. The primary infrastructure is stored in a state-of-the-art Tier IV data centre in Luxembourg.
- Code updates to the API infrastructure (which allows MEGA clients to "talk" to the database clusters) are done through rigorous procedures with various levels of code review by different key MEGA staff members to protect the integrity of the database clusters.

File storage infrastructure: These servers store all the encrypted user data. MEGA has more than 200 petabytes of storage capacity and currently has stored more than 68 billion files.

- All storage servers use standard RAID level 6. Multiple hard drives can fail without affecting the availability of the data. Failed hard drives are swiftly replaced to minimize the risk of multiple concurrent failures.
- In addition, MEGA also stores user data spread over multiple geographic locations to protect against the more improbable disasters (such as multiple hard drives failing at the same time, or the destruction of an entire server or data centre). We call this arrangement CloudRAID, for "Redundant Array of Independent Datacentres". User data will remain available even in the event of an entire data-centre becoming unavailable.

File metadata infrastructure: These servers store all the user-generated and encrypted attributes for the files stored on MEGA, such as thumbnails of images, videos and documents. These servers are replicated in real time across multiple geographic locations.

1.7 Compliance

MEGA was designed, and is operated, to ensure that it achieves the highest levels of compliance with regulatory requirements.

MEGA's service is governed by New Zealand law and users submit exclusively to New Zealand arbitral dispute resolution. MEGA has sought extensive legal advice on its service from lawyers in New Zealand and various other jurisdictions, including the United States, to minimise the risk of non-compliance with regulatory requirements in the main jurisdictions in which it supplies services.

MEGA maintains market leading processes for dealing with users who upload and share copyright infringing material or breach any other legal requirements. MEGA cannot view or determine the contents of files stored, as files are encrypted by the user before the files reach MEGA. However, if a user voluntarily shares a link to a file they have stored (with its decryption key), then anyone with that link can decrypt and view the file contents. MEGA's Terms of Service provide that copyright holders who become aware of public links to their copyright material can contact MEGA to have the offending links disabled.

When implementing its takedown notice policy and processes, MEGA initiated discussions with New Zealand law enforcement authorities. MEGA has adopted policies and processes which it has been advised are consistent with their requirements.

The US Digital Millennium Copyright Act (DMCA) process, the European Union Directive 2000/31/EC and New Zealand's Copyright Act s92B provide MEGA with a safe harbour, shielding MEGA from liability for the material that its users upload and share using MEGA's services. MEGA complies with the conditions on which those safe harbours are made available by allowing any person to submit a notice that their copyright material is being incorrectly shared through the MEGA service. When MEGA receives such notices it promptly removes or disables access to the offending file or files, depending on the type of request, consistent with the Terms of Service agreed to by every registered user. The number of files which have been subject to such takedown notices continues to be very small, indicative of a user base which appreciates the speed and flexibility of MEGA's system for fully legal business and personal use.

The DMCA requires links to be taken down expeditiously. Most cloud providers target takedown within 24 hours. MEGA targets takedown, within a maximum of 4 hours, with takedowns, frequently being actioned much quicker than the 4-hour target.

MEGA periodically publishes statistics on its compliance activities to provide public confidence that the service it provides is lawful and legitimate and is operated in a manner that protects the rights of users.

The most recent transparency report can be found here: https://transparency.mega.io/

2. Client application security

2.1 Cryptography in the browser

When MEGA introduced browser-based cryptography to the general public in 2013, some controversy ensued. The three major points of criticism were:

1. The transport path - the famed cryptographer Moxie Marlinspike told Forbes that "if you can break SSL, you can break MEGA";

2. Loading the code afresh upon every site visit, enabling the delivery of targeted backdoors;

3. Concerns about the browser environment, such as the lack of cryptographically strong pseudo-random number generators, side channel attacks and the risk of remote code execution through cross-site scripting.

While these concerns could all be valid, the traditional alternative is by no means better. Downloading the binary of a cryptographic application depends on the security of the transport path just as much - "if you can break SSL, you can insert a backdoor into pgp.exe". Targeted backdoors can similarly be delivered when users visit a vendor site to download and install an application binary, or when they use the built-in auto-update feature. Few, if any, users are able to perform a comprehensive code review before trusting a complex piece of code - irrespective of whether it is delivered as source code, a precompiled binary or as JavaScript. Modern browsers feature crypto APIs that are as strong as anything native code could achieve. Remote code execution has become virtually impossible thanks to the introduction of the content security policy. And, apart from the cryptographic aspect, the browser acting as a sandbox (assuming no known unpatched vulnerabilities) safely shields the victim's system against rogue code, confining the damage to the origin domain. That's far better than the potential damage resulting from inadvertently running a backdoored binary!

Users who fear the on-the-fly delivery of targeted backdoors when they load https://mega.nz/ have the option of installing MEGA's browser extension which runs the code locally and supports cryptographically signed updates (although the user must perform a full code review for this to make a real difference). See below for details.

It is undisputed that the barrier to entry is lower when users can gain access to an application by visiting a web URL as opposed to having to download and install a binary first. It is therefore unsurprising that MEGA has many more active users than PGP despite being more than 20 years younger.

2.2 Browser extensions

MEGA builds the desktop web client into three separate browser extensions for Firefox, Chrome/Chromium and Edge. The extensions contain all the JavaScript, HTML and CSS files which means that executable client code runs directly from the user's local machine, rather than being loaded from MEGA through TLS at runtime (more on that below in 8.3). The extensions can be downloaded from https://mega.nz/extensions.

Chrome extension updates are cryptographically signed by Google upon loading the extension update onto the Chrome Webstore. The Google account used for loading the extension onto the store has a strong unique password and two-factor authentication. It is axiomatic that the user does need to trust Google, when using the Chrome extension from their store. Similarly users need to trust Microsoft when using the extension from their store.

For power users not wanting to trust web stores, the unsigned Firefox extension and Chrome/Chromium extension can also be downloaded directly from MEGA's main web server and loaded manually into the browser. The links are https://mega.nz/meganz.xpi for Firefox and https://mega.nz/chrome-extension.zip for Chrome/Chromium.

Power users who wish to use the MEGA webclient from their local machine without having to rely on browser extensions can do so by cloning the webclient repository and running it from a local webserver. The repository and instructions can be found at https://github.com/meganz/webclient

2.3 Secure boot (webclient at runtime through https://mega.nz/)

The main core webclient files (index.html and secureboot.js) are delivered via SSL from the root web cluster, which is the axiom of the webclient's security when loaded through MEGA's primary domain (https://mega.nz/).

Users loading the webclient through MEGA's primary domain need to trust the webclient codebase as served at runtime through SSL. This is a convenient option for users accessing MEGA through the browser without having to install anything (particularly convenient for first-time users) but does provide a reduced security level compared to using the browser extensions or native apps due to the inherent reliance on SSL at runtime.

To minimize the potential attack vectors under the above described reduced security level, the integrity of executable files delivered by the static servers is verified so that only the root web cluster (https://mega.nz) has to be trusted. MEGA operates multiple static clusters in various geographic locations to serve JavaScript and other client files to ensure that the webclient loads as fast as feasible for the user. JavaScript, HTML and CSS files loaded through https://mega.nz/secureboot.js are hashed with SHA-256 and this is appended to the filename in hex at the time of the webclient release's build. The secureboot.js file contains a list of all the valid static files and their hash digests in the corresponding file name. Each file is loaded from the static cluster, hashed and then the computed hash digest is compared against the valid hash digest which was preloaded in secureboot.js. Any failure means the website will fail to load and an error will be shown to the user. This process predates the Subresource Integrity (SRI) web API but is very similar in practice. Fonts and images are not currently included in this integrity checking process, due to the fact that using images or fonts as a plausible attack vector is highly theoretical (and would still require an attacker to actually breach any of our static clusters).

The same applies to users using the MEGA webclient in a mobile browser, but mobile browsers do not allow for extensions and ought to use MEGA's mobile apps for enhanced security.

2.4 Android

MEGA's Android application package (APK) can be downloaded and installed from the Google Play Store. The installation file and updates are cryptographically signed both by MEGA and by Google. It is axiomatic that the user does need to trust the Android operating system and the Google Play Store platform when installing the APK from the Google Play Store.

Users who wish to use the MEGA Android app without using the Google Play Store can do so by cloning the MEGA Android repository and building their own MEGA Android client directly from the source code:

https://github.com/meganz/android

2.5 iOS

MEGA's iOS application archive (IPA) can be downloaded and installed from the Apple App Store. The installation file and updates are cryptographically signed both by MEGA and by Apple. It is axiomatic that the user does need to trust the iOS operating system and the Apple App Store platform when installing the IPA from the Apple App Store.

Users who wish to use the MEGA iOS app without relying on the Apple App Store can do so by cloning the MEGA iOS repository and build their own MEGA iOS client directly from the source code:

https://github.com/meganz/ios

This does, however, require an Apple developer certificate or a jailbroken iOS version. In case of the latter, please do make sure iOS jailbreaking is allowed in your jurisdiction:

https://en.wikipedia.org/wiki/IOS_jailbreaking#Legal_status

2.6 MEGAsync

For Windows and macOS users we offer an initial installation binary served through MEGA's secure root servers encrypted and authenticated with SSL. The installer for Windows is signed using an extended validation (EV) code signing certificate from GlobalSign. The installer for macOS is signed with a certificate issued by Apple.

Subsequent updates are installed automatically through a secure cryptographic update mechanism using SHA-256 and a 4096-bit RSA key controlled by MEGA. These automatic updates can be turned off by the user (although it is recommended to always use the latest version for more stability and security).

For Linux, we offer repositories that hold our binaries and offer updates. The repositories also provide a public key that can be used to check the authenticity and integrity of the offered packages. In order to do that, MEGA Linux repositories also include standardized metadata that describes the contents of the repository and file fingerprints. Thus, users can rely on the integrity of MEGA packages. Updates are performed by system's package managers, which validate that the contents in the repository are correctly signed by MEGA and they have not been tampered with. Users can configure our repositories manually by adding our public key to their trusted list, or automatically upon the installation of a MEGAsync package. We offer both the public key and installation packages secured via SSL to ensure their authenticity.

Users who wish to build MEGAsync directly from the source code can do so by cloning and following instructions in this repository:

https://github.com/meganz/megasync

2.7 MEGAcmd

For Windows and macOS users we offer an initial installation binary served through MEGA's secure root servers encrypted and authenticated with SSL.

On Linux, the primary target platform of MEGAcmd, we share the repositories mentioned in 8.6 MEGAsync. Hence, the same security mechanisms apply.

Currently, we don't offer automatic MEGAcmd updates for either Windows or macOS. Updates for these platforms must be applied manually by reinstalling new binaries downloaded from MEGA's SSL protected root servers.

Users who wish to build MEGAcmd directly from the source code can do so by cloning and following instructions in this repository:

https://github.com/meganz/megacmd

2.8 Endpoint security

While MEGA's E2EE paradigm does enhance the overall security with privacy by design compared to many of its competitors who only provide privacy by policy, it is not a silver bullet solution to all potential security threats. It is axiomatic that the endpoint devices are deemed secure in the E2EE paradigm. Any breach on the endpoint device breaks the E2EE chain. For example, a backdoor or virus on the user's endpoint device's operating system would potentially allow a rogue attacker to intercept unencrypted data, log keystrokes or directly capture audio/video from the device's microphone/camera. It is therefore important that the user does not solely rely on MEGA, but also ensures that the endpoint device (on which the user runs the MEGA client software) is secure. MEGA does not protect the endpoint device directly. MEGA recommends the use of device-level encryption and general best practices against malicious software and physical device security.

3. User registration & login process

Notation and symbols

|| = concatenation
⊕= Exclusive OR bitwise operation

3.1 Registration process

The security of the registration and login process is closely based on the research paper “Method to Protect Passwords in Databases for Web Applications” [1].

It is important to note that all communications with the API are done via TLS. Where possible, the API server’s TLS public key is pinned in the clients.

The registration process is:

- At registration time, the user enters their **Email**, **Password** and the **Password** again as a confirmation that they are the same. The password entries must match exactly, or the user cannot proceed.

- The email address is limited to a maximum of 1190 characters in the clients and on the API side (the reason for this is explained later).

- The client will allow registrations with passwords of a minimum length of 8 characters and a minimum score of 1 as per the current ZXCVBN password strength estimator library (v4.4.2), which scores the password between 0 - 4.

- The password strength checker library is not activated on the password until the minimum 8 characters have been entered. If 8 characters have not been entered yet, the application says “Too short” in red text next to the field as it is being typed. This eliminates a password like “€cc” being accepted which is too short but still scored as 1 (Weak) by the library. In summary:

  Length < 8 Too short
  Score 0 and Length >= 8 Too weak
  Score 1 and Length >= 8 Weak
  Score 2 and Length >= 8 Medium
  Score 3 and Length >= 8 Good
  Score 4 and Length >= 8 Strong

1. Refer to p49

- For the Password Processing Function (PPF), we use the established PBKDF2 standard. While this is not the latest, best or state of the art, it is well known and has wide language support, especially in the WebCrypto API, where it importantly performs at native speed. For the web client, there is currently no reliable WebCrypto API or asmCrypto library support for Argon2 (preferred), Scrypt etc. Also, if they did have a JavaScript library equivalent, its performance would be so slow that we would likely have to reduce the processing cost factor (number of iterations) accordingly to still be performant on low-mid range mobile devices, which would adversely affect security.

- The client generates a random **Master Key** of 128 bits (16 Bytes) using the client's native CSPRNG. For example, the web client uses the WebCrypto API's Crypto.getRandomValues() function.

- For the PPF, the client will also generate a **Client Random Value** of 128 bits.

- For the Salt, the client will compute:

  Salt = SHA-256("**mega.nz**" || "**Padding**" || **Client Random Value**)

  The combined length of the strings "**mega.nz**" and "**Padding**" together will be exactly 200 characters. The padding character will be the letter "P" (in uppercase), which is repeated until the combined string is 200 characters in length. This aids in making all the salt calculations take the same computation time to reduce timing attacks.

  The research paper "Method to Protect Passwords in Databases for Web Applications" recommends including the user identifier (i.e. **Email** in our case) as well in the **Salt** calculation. However, for MEGA, the email address is subject to change by the user (in which case we would need to re-do the PPF calculation and re-encrypt the **Master Key** every time they changed their **Email**). Also, emails need to be able to be changed by helpdesk at the request of users but this would lock the user out of their account if the Salt calculation and PPF is based off an old **Email** which had been overwritten. This would be detrimental to usability.

- For the PPF, the number of **Iterations** will be set at 100,000. This number of iterations is set for reasonable performance on low-mid range mobile devices because MEGA has iOS, Android, Windows Phone apps and a mobile web client.

- For the PPF, the **Length** of the derived key will be set at 256 bits (32 Bytes).

- The client will create a **Derived Key** (256 bits) by computing:

  Derived Key = PBKDF2-HMAC-SHA-512(**Password** , **Salt** , **Iterations** , **Length**)

- The first half (from left) of the **Derived Key** (128 bits) is called the **Derived Encryption Key**, and it encrypts the **Master Key** using the current method:

  Encrypted Master Key = AES-ECB(**Derived Encryption Key** , **Master Key**)

- The second half (128 bits) of the **Derived Key** is used as the **Derived Authentication Key**, which is used with the API to authenticate the user's login before more sensitive data (such as encrypted keys, Session ID etc) are sent.

- The client will now compute:

  Hashed Authentication Key = SHA-256(**Derived Authentication Key**)

  The **Hashed Authentication Key** only contains the first 128 bits of the SHA-256 output, so it reduces the amount of storage space on the API.

- The client will register the account with the API by sending:

  First Name
  Last Name
  Email
  Client Random Value (128 bits / 16 Bytes)
  Encrypted Master Key (128 bits / 16 Bytes)
  Hashed Authentication Key (128 bits / 16 Bytes)

  The API stores all this information as-is. Note that the ***Salt*** is not sent to the API, as this is always computed from the ***Client Random Value*** for reasons explained later.

- The API then generates a random ***Email Confirmation Token*** (128 bit) using its native CSPRNG and sends a confirmation link:

  Confirm Link = "**https://mega.nz/#confirm**" || Base64UrlEncode("**ConfirmCodeV2**" || **Email Confirmation Token** || **Email**)

  The Base64 URL encoding replaces two non-alphanumeric characters so it works in URLs (see https://github.com/meganz/webclient/blob/master/nodedec.js#L491 for an example).

- Upon clicking the **Confirm Link** in the email, the client will send the Base64 URL encoded data after the #confirm token back to the API which can decode it. If that API request succeeds, the client will be taken to the Login page to log in once more where the ***Email*** field can be pre-populated, but the manual entry of the password is required. The user needs to log in once more in case they opened the link in another browser.

- After the user logs in for the first time, they go straight to the key generation step where the following keys are generated:

    - An RSA key pair, 2048 bits (used for sharing folders/files).
    - An Ed25519 key pair, 256 bits (used as the trust root for user fingerprint verification and signing of other keys. This key pair are referred to as the Signature Keys).
    - A Curve25519 key pair, 256 bits (used for MEGAchat).

    The private keys are encrypted by the Master Key using AES-ECB and stored by the API. MEGA has no access to plaintext private keys by any means.

3.2 Login process

The login process is:

- The user must enter their **Email** and **Password** into the local client interface.
- The **Email** is sent to the API. If the email belongs to a valid user account, the API will send back:

  Salt (computed by SHA-256("**mega.nz**" || "**Padding**" || **Client Random Value**)

  The combined length of the strings "***mega.nz***" and "***Padding***" will be exactly 200 characters. The padding character will be the letter "P" in uppercase which is repeated until the resulting string is 200 characters in length.

- If the **Email** is not found in the database, the API will send back:

  Salt (computed by SHA-256(**Email** || "**mega.nz**" || "**Padding**" || **Server Random Value**)

  The length of the strings **Email**, "**mega.nz**" and "**Padding**" together will be exactly 200 characters in total. The padding character will be the letter "P" in uppercase which is repeated until the combined string is 200 characters in length. The email should be a maximum of 1190 characters in length or it will be rejected. Because the strings for a valid email and for an invalid email are both 200 characters in length, then the SHA-256 computation takes the same amount of time and this prevents a timing attack.

  The **Server Random Value** (128 bits) is static and stored API side for all these requests. It is randomly generated by CSPRNG and stored by the API. This will be used whenever an email is not in the database. This prevents email enumeration from a spammer to find valid emails and is changed yearly. Because the Salt is calculated by using a hash function for valid logins and invalid logins, this prevents a timing attack.

  There is also a random delay introduced by the API for this response, between 0 and 50 ms. 50 ms is the longest time to search through all database records and not find a result. This aids in not revealing whether an email was valid or not to an attacker by measuring the delay in the response from the database.

  The **Email** here received from the user is restricted to a valid email and maximum size of 1190 characters or the request is rejected. This also prevents the API hashing a large amount of data and prevents a DOS attack. This request is also rate limited.

- The client can now compute the Derived Key using:

  Derived Key = PBKDF2-HMAC-SHA-512(**Password** , **Salt** , **Iterations** , **Length**)

- The **Derived Encryption Key** is the first half (128) bits of the **Derived Key.**

- Using the **Authentication Key**, which is the second half (128 bits) of the **Derived Key**, the client will send to the API the:

  Email
  Authentication Key

- The API will compute the following:

  Hashed Authentication Key = SHA-256(**Authentication Key**)

  The API checks the first 128 bits of the **Hashed Authentication Key** against the one stored in the database for the user. If it matches, the user is considered successfully authenticated, so the API sends back their **Encrypted Master Key, Encrypted Private RSA Key, Encrypted Session ID** etc and the client can proceed as normal.

  The API does not store the unhashed **Authentication Key** sent by the user. It only stores the **Hashed Authentication Key** to prevent "pass-the-hash" attacks (where in the scenario of a leaked database, an attacker would just pass the **Hashed Authentication Key** to get authenticated and carry out actions as the real user). The server always hashes the **Authentication Key** received from the client, which prevents this attack vector.

  If the **Hashed Authentication Key** does not match the result in the database, the API responds with a negative response to indicate failure. The API side avoids timing attacks here by using Double HMAC Verification (https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2011/february/double-hmac-verification/).

- When the login is correct, the server will respond with the **Encrypted Master Key**, the **Encrypted RSA Private Key** and **Encrypted Session Identifier**.

- The client will decrypt the **Encrypted Master Key** using the **Derived Encryption Key**. Then the **Master Key** will be used to decrypt the **RSA Private Key**. Then the **RSA Private Key** will be used to decrypt the **Session Identifier**. The **Session Identifier** is a random string (token) generated by the API per login session and it is encrypted to the user's **RSA Public Key**.

- On all subsequent API requests, the user must send their unencrypted **Session Identifier** via the TLS connection to the API. If the user did not send the correct **Session Identifier** or it was not decrypted properly they will not be able to download any user account data such as contact information, encrypted files and file attributes.

3.3 Ephemeral accounts

Ephemeral accounts are trial accounts that have not been fully registered. They are limited in that they can only store files on the cloud with a temporary session. This is to provide a lower barrier to entry for first-time users and allow them to try MEGA without having to create a full account first. They cannot make contact relationships, share files, create public links or chat with other users. An ephemeral account is created when a user, who has not logged in to a registered account, uploads a file or folder into the homepage of the website or imports a file from a link. A Master Key is created at this time. The user can later upgrade to a full account by completing the full registration process.

3.4 Account recovery

As the user's password is effectively the root of all client-side encryption in a user's account, forgetting or losing it results in the inability to decrypt the user's data, which is highly destructive. For this reason, MEGA allows and highly recommends users to export their "Recovery Key" (which is technically their Master Key).

MEGA clients detect when a user has not entered their password for a lengthy period of time (for example due to enabling the "remember me" checkbox while logging in) and reminds users of the importance of their password. This reminder dialog prompts the user to test their password and/or export their Recovery Key.

MEGA has a convenient recovery interface where novice users are guided, based on their circumstances, in case of password loss: https://mega.nz/recovery

MEGA has found that users who forget or lose their password are often still logged in on another client (e.g. a mobile app or MEGAsync). For this reason, MEGA allows users with an active session to change their password in that client without first proving knowledge of the current password.

If the user has no other accessible active sessions, the user can use the Recovery Key (which is in effect the Master Key) to reset the password of the account. Technically, the user would re-encrypt the Master Key with a new password. Such a procedure requires email confirmation, so access to the Recovery Key alone is not sufficient to breach a MEGA account.

3.5 Two factor authentication

MEGA has implemented Two Factor Authentication (2FA) using the industry standard Time Based One Time Password (TOTP) method.
https://en.wikipedia.org/wiki/Time-based_One-time_Password_algorithm

The shared secret generation uses 32 random bytes. This is converted to base 32 and displayed to the user as a QR code for easy addition to Authenticator apps such as Google Authenticator. Before the user can activate 2FA on their account they need to provide a valid 6-digit code based on the shared secret. At this point 2FA will be enabled and required for authentication.

For the purposes of our 2FA implementation, we accept a valid code as being the current one, +30 seconds or -90 seconds to allow for clock drift on the device or the user taking time to input it.

When logging in, we first check if their existing login credentials match. If these fail we do not allow the user to proceed and the clients advise the user that their email & password combination is incorrect. Upon successfully validating these, the user will then be prompted to enter their 6-digit code if 2FA is enabled on their account. If the user successfully enters a valid 6-digit code, we record the last code used on the back-end API server to prevent replay attacks against the account. Failed logins or 2FA attempts are both tracked, and when the total number of failures of either type hits a threshold, we lock the account from accessing the MEGA service for 1 hour to prevent brute forcing of the 2FA.

Additionally, a number of activities in the account, such as turning off 2FA or changing the account email, are protected behind a 2FA challenge and will reject the user if they have not provided a valid and timely code. Failures here also count towards the same tracking and lockout protection system as the login flow.

As 2FA is used for authentication only, not for encrypting the data, the user's existing Recovery Key can still be used to regain access to the account in the case of losing access to the 2FA application where the seed was loaded. When the user enables 2FA, if they have not previously exported their Recovery Key, they are required to do so. Otherwise, we remind the user of the importance of keeping it.

2FA can be disabled by MEGA staff in the case of a user losing their Recovery Key, if the user provides sufficient evidence that they are the account's true owner. However, this is strictly controlled and requires authorisation from a senior staff member.

3.6 Remote session destruction

The MEGA client apps have the user's full session history with IP and client info, and the ability to remotely log out of specific sessions or all other sessions. For example, if a MEGA user loses a device, the user can remotely destroy the session so that the physical device loss does not compromise the security of the user's MEGA account. The session history can be viewed in the browser by accessing: https://mega.nz/fm/account/history

Note that if a lost device is found by a very sophisticated attacker, the attacker could, in theory, obtain any locally cached data from the device, such as the user's Master Key. As the user's password is never cached directly and the Master Key alone is not sufficient to start a new user session, a remote session destruction is still useful in this scenario. The attacker would not gain access to any data from the MEGA service without the password, so even with the Master Key the attacker still would not have access to the encrypted data unless they could also gain access to the user's email on the device and reset the password using the Master Key.

For enhanced protection for the physical device loss scenario, MEGA recommends the use of a strong screen passphrase and device-level full-disk encryption (FDE) which is now enabled by default on new Android/iOS phones.

4. Cloud drive encryption

4.1 File upload encryption

Each node (file or folder) has its own encryption key. Folders are not encrypted as they contain no data, just the folder attributes (i.e. the folder name) are encrypted. All nodes are stored in the same flat database structure, where the files and folders have a parent handle which references which folder they belong in. The folders/files in the root level have the main cloud drive or rubbish bin as their parent handle.

For encrypting files, a *File Key* is created of 128 random bits and a *nonce* of 64 random bits. Files are split into chunks, then each chunk is encrypted with AES-CCM. The 64-bit nonce is incremented for each chunk being encrypted.

After all the chunks have been encrypted, a *Condensed MAC* is calculated from all the previous chunk MACs. This works by initialising a 128-bit array to 0, then XOR that with a block MAC, then encrypt the result with AES-ECB. Then continue that with each subsequent block MAC until a final MAC is produced.

The **File Key** is then obfuscated as follows:

Obfuscated File Key = [
File Key[0] ⊕ IV[0],
File Key[1] ⊕ IV[1],
File Key[2] ⊕ Condensed MAC[0] ⊕ Condensed MAC[1],
File Key[3] ⊕ Condensed MAC[2] ⊕ Condensed MAC[3],
IV[0],
IV[1],
Condensed MAC[0] ⊕ Condensed MAC[1],
Condensed MAC[2] ⊕ Condensed MAC[3]
];

The **Obfuscated File Key** is then encrypted with the **Master Key** using:

Encrypted File Key = AES-ECB(**Master Key, Obfuscated File Key**)

Then it is uploaded to the API.

4.2 File attribute, preview and thumbnail encryption

When the file is ready to be sent, the file attributes (e.g. the file name, thumbnail, preview) also need to be encrypted. These are encrypted with:

AES-CBC(**File Key, File Attribute Data**)

5. Secure public links

5.1 Public file links

When a public file link is shared publicly, the following is embedded into the public link:

https://mega.nz/#! || Base64(**File Handle**) || ! || Base64(**Obfuscated File Key**)

This is enough information to find the file identified by its **File Handle** on the server, then download it, verify the **Condensed MAC** of the overall file, unobfuscate the **File Key**, then decrypt the file using the **File Key** and **IV**.

It should be noted that everything after an anchor hash (#) in the URL is not sent to the MEGA servers and is kept locally in the client's browser.

5.2 Public folder links

When a user creates a shared folder, a random Share Key is generated. This Share Key encrypts all the file nodes using:

AES-ECB(**Share Key, Obfuscated File Key**)

When a public folder link is shared publicly, the following is embedded into a public link:

https://mega.nz/#F! || Base64(**Folder Share Handle**) || ! || Base64(**Share Key**)

This is enough information to find the files in the shared folder identified by its **Folder Share Handle** on the server, then download the folder and file nodes under it, verify their **Condensed MACs**, unobfuscate the **File Keys**, then decrypt the files using the **File Key** and **IV**.

5.3 Chat links

When a user creates a chat link, the Unified Chat Key is embedded into the public link:

[https://mega.nz/chat/ || Base64(**Chat Handle**) || # || Base64(**Unified Chat Key**)]

This is enough information for MEGA clients to open and decrypt the specific chat that was linked. For more information about chat links see section 7.7.

5.4 Password protected links

This feature lets a user encrypt the regular file or folder link key with a password. Then it can be sent over an insecure channel such as email or shared on a public website without compromising the confidentiality of the data. How the password is shared securely with another user is up to them, however, if they know the other user well they might be able to construct a shared secret between them based on common history they share.

The implementation uses PBKDF2-HMAC-SHA512 with 100,000 rounds and a 256-bit random salt and the user's password to obtain a 512-bit **Derived Key**.

For folder links the key is 128 bits in length and for file links the actual key is 256 bits in length. The first 128 or 256 bits of the derived key will be used as the **Encryption Key** to encrypt the actual folder/file key using a simple XOR for encryption. The last 256 bits of the derived key will be used as the **MAC Key**. Using the Encrypt then MAC principle, the MAC will be calculated using HMAC-SHA256. In constructing the protected link, the format is as follows:

Algorithm || **Type** || **Public Handle** || **Salt** || **Encrypted Key** || **MAC Tag**

- **Algorithm** = 1 byte - A byte to identify which algorithm was used (for future upgradability), initially is set to 0
- **Type** = 1 byte - A byte to identify if the link is a folder or filelink (0 = folder, 1 = file)
- **Public Handle** = 6 bytes - The public folder/file handle
- **Salt** = 32 bytes - A randomly generated salt
- **Encrypted Key** = 16 or 32 bytes - The encrypted actual folder or file key
- **MAC Tag** = 32 bytes - The MAC of all the previous data to ensure the integrity of the link i.e. calculated as

  HMAC-SHA256(**MAC Key** , (**Algorithm** || **Type** || **Public Handle** || **Salt** || **Encrypted Key**))

The link data is Base64 encoded and then modified to substitute incompatible characters, for example

```
https://mega.nz/#P!AAA5TWTcs7YZg_hVxF0JTKxOZQ_s2d...
```

In receiving a protected link, the program will decode the Base64 portion of the URL after the #P! identifier, then get the first byte to check which algorithm was used to encrypt the data (this is useful if algorithm changes are made in future).

Then it will use the password to derive the same key using the same algorithm, provided salt and password. Then a MAC of the data can be calculated. If it's a match, then the link has not been tampered with or corrupted and the real folder/file key can be decrypted and the original link reconstructed. If it doesn't match, then an error will be shown which could indicate tampering or more likely that the user entered an incorrect password.

5.5 Time expiring public links

Users can add an expiry time to their public links if they want the folder/file contents to become unavailable after a certain date. Access to the link and file contents is barred by the API logic after the expiry time has passed.

5.6 Importing files from public links

When importing a file to the user's Cloud Drive from a public link, the **Obfuscated File Key** is simply encrypted by the user's **Master Key** using:

Encrypted File Key = AES-ECB(**Master Key**, **Obfuscated File Key**)

This encrypted key is then stored on the API for the user with the same public file handle. A new copy of the file is not downloaded, re-encrypted with a new key and re-uploaded. Therefore, if the original file link is taken down due to Type 3 copyright infringement or other Terms of Service violation, the users who have imported the file from a public link will also be unable to access the original file anymore.

6. Collaboration and Shared Folders

6.1 Making Contact Relationships

MEGA allows a user to make contact relations with another user by sending them a contact request. If the MEGA user knows the MEGA account's email address of another user, then this user can send them a contact request. Contact requests by email require user approval.

MEGA users are also able to share their contact links or establish a contact relationship by scanning their QR code. Contact establishments through QR or contact links are by default automatically approved, but this can be turned off optionally.

6.2 Key exchanges and verification

During application start-up, the following are fetched from the API for each of the user's contacts:

- **Ed25519 Public Key**
- **Ed25519 Public Key Fingerprint**
- **RSA Public Key**
- **RSA Public Key Signature**
- **Curve25519 Public Key**
- **Curve25519 Public Key Signature**

When a new contact is added by one of the user's other devices, this information is sent by the API in an action packet to update the local device's state. An action packet is the response to a long-running open connection to the API. At any time when all the user's open devices need to be updated with the latest state (i.e. new files, contacts, permission changes etc.), the API will respond via the open connection.

The **Ed25519 Public Key** is considered to be the trust root for the other keys. When a user account is created, each user signs their **RSA Public Key** and **Curve25519 Public Key** with their **Ed25519 Private Key** to create signatures for each. Private keys are encrypted first by the user's **Master Key** before being uploaded to the API. The public keys and Ed25519 signatures are uploaded to the API as-is. When a user fetches their contact's key information, it verifies their RSA and Curve25519 signatures. If the signature calculation does not match, an error is shown and no file sharing or communication with that contact is allowed, to prevent a man-in-the-middle (MITM) attack.

The first time a contact's **Ed25519 Public Key** is fetched by the user, a hash digest of this is calculated using:

Fingerprint = SHA-256(**Ed25519 Public Key**)

Only the first 160 bits of the 256-bit hash digest is kept as the fingerprint. This is encrypted for the user using:

Encrypted Fingerprint = AES-GCM(**Master Key**, **Fingerprint** || **Verification Type**)

Then the fingerprint is stored as a private user attribute on the API so that only the user can retrieve it. The initial **Verification Type** of 0x00 (seen) means that by default contacts will accept the first seen public key for a user. If it changes after that, the User Interface (UI) will throw a warning that the key has changed and that they need to verify it, or they cannot interact with the contact to share files or chat. This can happen in a legitimate circumstance, such as the user parking their account and starting a new one under the same email address, or it can also be a MITM attack.

MEGA supports fingerprint verification on contacts via the UI. This provides a stronger security guarantee that the user is really communicating with the expected contact. This rules out the MEGA service serving fake public keys to the user's contacts and performing a MITM attack. Users can verify fingerprints in person or via any existing secure channel e.g. PGP email, secure chat etc. If the fingerprint is confirmed, the **Verification Type** is updated to 0x01 (verified) and the UI shows a special green check icon on the contact.

6.3 Folder Sharing

When a folder is shared, a random 128-bit **Share Key** is created. All the File Keys in the folder are encrypted with the **Share Key** using:

Encrypted File Key = AES-CBC(**Share Key** , **Obfuscated File Key**)

The **Share Key** is then encrypted for the contact:

Encrypted Share Key = RSA(**Public RSA Key**, **Share Key**)

The **Encrypted File Keys** and **Encrypted Share Key** are then sent to the API. The contact will then download the **Encrypted Share Key** and use their **Private RSA Key** to decrypt it. Then they will be able to download and decrypt the folder share's **File Keys** which can be used to decrypt the files.

6.4 Sharing to non-contacts and unregistered users

When sharing to non-contacts, the share is in a pending state. Once the contact relationship is established, the sharing user can fetch the contact's key information and encrypt the **Share Key** and **File Keys** to them. The catch here is that the user needs to be online to encrypt the keys for the contact. If they are not currently online, the share sits in a pending state until they come online.

For unregistered users, they are a sent an invite link via email. Once they sign up, login and accept the pending contact request, they will be sent the encrypted **Share Key** and **File Keys**, so they can access the contents of the folder share.

6.5 Business User Key Exchange

A business on Mega is made up of three types of accounts. The business account, which is never logged into but is used to hold keys and attributes for the business as a whole; the master user, which has management permissions over the business; and sub-users, who have their master keys encrypted to the business account, thus enabling the master user to have cryptographic access to all sub-user accounts.

A business is created by converting an individual user account (new ephemeral or registered) into a master user account. The master user then creates a new set of keys (master key and RSA pair) for the business account. The business master key is encrypted with the master user's master key and stored by the API. Sub-users encrypt their keys to the business rather than directly to the master to support the future possibility of multiple master user accounts per business.

Encrypted Business Master Key = AES-ECB(**Master User Master Key**, **Business Master Key**)

The master user creates sub-users by sending an invitation email with a unique URL. Loading this URL signals the API to link the newly created user into the business. The sub-user creation process is the same as an individual user with the additional step of encrypting its own master key with the RSA public key of business user and sending it to the API with the other keys during registration. As the API allows the master user to request the file tree of the sub-user, then the master user will always have access to any files uploaded by the sub-user even if they leave the business.

Encrypted Sub-user Master Key = RSA(Business Public RSA Key, Sub-user Master Key)

7. MEGAchat text messaging

MEGAchat is intended to protect the privacy and confidentiality of content exchanged. MEGAchat supports exchanging of text messages, files and MEGA contacts. All messages and documentation exchanged by MEGAchat are protected with end-to-end (user controlled) encryption providing essential safety assurances:

Confidentiality
Only the author and intended recipients are able to decipher and read the content;

Authenticity
The system ensures that the data received is from the sender displayed, and its content has not been manipulated during transit.

7.1 Cryptographic Primitives

MEGA provides a cloud-based platform enjoying a large popularity. Therefore, server-side scalability is of importance as well as the feasibility for the messaging concept and the client implementation. Even though server scalability tends to be orthogonal to messaging encryption protocols, they have shown to add a significant overhead on the server infrastructure (due to the number of “blind” protocol bootstrapping messages and increased message size), so that fewer users can be served per server provided.

Out of experience, most of the clients will be using the messenger on the MEGA platform through a Web or mobile client with limited computational capabilities (e.g. with end-to-end cryptography implemented in JavaScript). To maximise user experience through a fast response time (less lag) and reduced load on the executing endpoint hardware (which often are mobile devices), it is desirable to avoid frequent “heavy” computing operations (e.g. frequent exponentiation of big integers). The cryptographic primitives have generally been chosen to match a general security level of 128 bit of entropy on symmetric ciphers such as AES. This is equivalent to 256-bit key strength on elliptic curve public-key ciphers and 3072-bit key strength on discrete logarithm problem based public-key ciphers (e.g. RSA, DSA). However, 3072-bit key strength is considered to be too expensive in many cases (computationally as well as with its demand on the entropy source). For security reasons, NIST standardised ECC curves have been avoided.

7.2 Message Encryption

7.2.1 Encryption Key

Every sender is responsible for generating their own symmetric encryption key, ensuring user-controlled keys for any encrypted content sent. Therefore, each sender generates their own symmetric encryption keys used for encrypting the message payload. These encryption keys then need to be exchanged with all other participants within a chat (pair-wise). The encryption keys as well the message payload content then need to be encrypted for message transport and storage.

The encryption key is randomly generated unique 128 bit long for use with AES in CTR mode.

Encryption key ID is 32 bit long, as it needs to be unique per sender per chat.

Encryption keys are encrypted using AES in CBC mode, using an initialisation vector (IV) along with the ECDH shared secret with each participant.

To derive the shared secret, one's own private (S_{own}) and the other participant's public key (P_{other}) is used. Through Curve25519 Diffie-Hellman scalar multiplication (ECDH) and subsequent application of a key derivation function (KDF, specifically [HKDF]-SHA256) the key is derived. It is trimmed to the required key size (128 most significant bits).

$$K_{DH,\,dest} = KDF(ECDH(S_{own}\,,\,P_{other}))$$

To derive an IV (initialisation vector) for a recipient, the MEGA user handle of the recipient is base64 URL decoded, yielding an 8-byte (64 bit) sequence (u). From these bytes then a keyed-hash message authentication code (HMAC, specifically HMAC-SHA-256) is computed using the message's master nonce (n) as a key, and subsequently trimmed to the required IV size (128 most significant bits).

$$IV_{dest} = HMAC(n_{master}\,,\,u_{other})$$

7.2.2 Message Encryption

Message payloads are encrypted using AES in CTR mode, using the sender key and message nonce derived via computing an HMAC using the message's master nonce as a key and the byte sequence "payload" as a value. The message nonce will be trimmed to use the 96 most significant bits (12 bytes) only, leaving 32 bits for the counter.

$$n_{message} = HMAC(n_{master},\,\text{"payload"})$$

7.2.3 Message Signatures

A version number, a message type number and the 128-bit encryption key together with the assembled message content are signed with the user's cryptographic signature, signing the entire following (encoded) binary message content. All message signatures are computed using the sender's Ed25519 identity key.

The content to sign is computed as follows:

$$(magic\ number || content\ to\ sign)$$

Here, **magic number** is a fixed string to distinguish the authenticator from any other content (for now it is the byte sequence "strongvelopesig").

7.2.4 Message Encoding

All fields in the messages exchanged are encoded as TLV (type, length, value) records. The entire message is prepended by a single byte indicating the protocol version (in case of future changes). Currently the protocol version is 0x03.

TLV records do not need to be in order according to the TLV type numbers, as long as it is assured that the SIGNATURE record is preceding all others. Individual records may be missing or repeated multiple times.

7.3 Encryption Key Rotation

Whenever a new sender key is required, the sender will generate one and send it (encrypted to all participants) along with the new key ID to all participating recipients. It is desirable to periodically refresh a sender key (preventing extensively long use), or when the composition of the group chat has changed (added and/or removed participants). Upon changes in the group composition, the first message a client sends to the group chat must be a message stating a new encryption key.

For convenience (e. g. when loading the chat history in reverse order), the previously used key with its key ID is re-sent to previous participants in the group as well. The client must not send the previous key to newly joined participants and must not send a new key to departed participants.

A chatroom can be created in an open mode. In contrast to the closed mode, the key is never rotated and every message is encrypted to the same key, the known as unified key. There is no way to rotate the unified key and it does not change when new users join or leave the chat. To extend this information please refer to the section 7.7.1 Encryption in open mode vs. rotation of keys in closed mode.

7.4 Message Decryption

Message decryption performs the opposite process of message encryption.

- Once receiver receives the assembled content and encrypted key from the server, it checks the version number and message type, then parses the assembled content.
- A pair of ECC point multiplications on Curve25519 is done for key agreement, and the derived key is generated to decrypt the encrypted 128-bit key.
- Signature is verified by the receiver to make sure the received content is not compromised.
- A 96-bit nonce is derived from the nonce parsed from the assembled content by applying HMAC-SHA256.
- The payload is then decrypted by with 128-bit encryption key and 96-bit nonce by using AES-CTR.

7.5 Message Order Protection

Even if the content of the message is protected from any man-in-the-middle attack, an attack by manipulating the message order can still compromise the conversations of chat systems.

MEGAchat uses the following strategy to detect whether the message order is tampered in a conversation and raises an error to users if it detects any.

1. When a message is generated from the client, it generates a unique Id for the message.

2. When a message is sent out, it iterates the previously received messages and randomly selects some. Then it appends the message Ids of these messages as references to the message.

3. Once the receiver receives the message, it checks whether all the references of the message appear before the message, if not, then it means the message order has been compromised and the receiver will receive an error from MEGAchat.

The references of the messages are randomly selected, so it has a good coverage on the conversation and makes it almost impossible to change the order without being noticed.

7.6 Rich links in MEGAchat

When users share URLs through MEGAchat, they can optionally turn on rich links support. Rich links improve usability as both the sender and the recipient(s) can preview metadata of the relevant URL (such as the title, description and an image). This metadata does, however, need to be requested through MEGA's API, which exposes the URL in plaintext to MEGA (hence the feature requires explicit user consent). This metadata is always requested by the sender of a URL and is subsequently encrypted in the MEGAchat message payload. URLs for which such metadata is requested through the API get their request cached for a maximum of 24 hours in a secure database (in order to avoid overloading URLs which may have very high frequency through MEGAchat). This data is only stored for operational purposes and privacy by policy applies as opposed to privacy by design.

7.7 Chat links in MEGAchat

Chat links allow people to easily preview and join a group chatroom without requiring to invite each person individually. Any user with a valid chat link can preview the chat history, the participants and the title, even to users who are not yet registered on MEGA. Registered users are entitled to join the chat with standard read/write permissions. The preview mode of an open group chat is allowed as long as the chat link is valid.

Chat links can be created only for chats in open mode (encryption key does not rotate) and it requires the moderator privileges. The resulting chat link includes the encryption key as part of the link, plus a public handle created for the chatroom. The moderator can invalidate the public handle on demand, which invalidates the chat link. Note that chat links are also invalidated when the last moderator leaves the room.

7.7.1 Encryption in open mode vs. rotation of keys in closed mode

A chat can be created directly in open mode, where all messages are encrypted to the same key, known as the unified key, rather than a scheme of rotating keys. There is no way to rotate the unified key, and it does not change when users join or leave the room.

Any moderator of a group chat in open mode can switch to closed mode in order to enable key rotation (the opposite operation is not possible).

One-on-one chats by default always have key rotation enabled, as these never need a static key. Group chats have key rotation by default turned off, so that operators have the ability to link to a group chat after it was created.

When an operator creates a link to a closed group chat, a management message will be shown to all participants informing them that this chat has been linked.

7.7.2 How the unified key is created and distributed to the participants

For every participant joining the group, the inviting user needs to provide the unified key encrypted to the inviting user's private Cu25519 key and the invitee's public Cu25519 key. In consequence, the new participant will be able to decrypt the unified key by using the inviting user's public key and its own private key.

For previewers, the unified key is included as part of the chat link in plain text (using B64Url encoding), see also section 5.3.

7.7.3 Preview mode and auto-join option

A user clicking on a valid chat link is able to preview the history of the chat (with read only permissions), including the current list of participants, even if they are not logged into MEGA. If somebody provides a chat link that is no longer valid, they will not be able to fetch a preview, even if they have the correct unified key.

Users who preview a chat link will have access to the whole history of messages while the chat was in open mode.

Any MEGA account in possession of a valid chat link can (auto) join the chat with the standard privilege (read/write permissions). They do not need additional approval to join, although any moderator can remove them from the chat at any time. If the operator wants to permanently keep them out, in order to prevent a new auto join from the user, the moderator would need to invalidate the chat link.

Users who ever participated in an open chat but already left the room (ex-participants) cannot do a preview any longer, so they are entitled to access to the history only until the moment they left. However, if the user has a valid chat link, it is still possible to re-join the chat again with standard read/write permissions and regain access to the whole history.

If a chat link is invalidated, any previewer using it will be immediately expelled from the chat. Opened previews are not persisted, so if the client is closed, the chat will disappear from the list (unless the previewer auto-joined, becoming a participant).

When a non-registered user, while previewing a chat-link, attempts to auto-join the chat, the user will be requested to login or create an account first. Once logged-in, the user will be able to complete the join.

7.7.4 Title encryption

The title of chat in open mode is encrypted to the unified key, since any previewer and participant needs to be able to decrypt it. In consequence, the title doesn't need to be updated when a user joins or leave the room, because the unified key does not change.

When a chat in open mode is switched to closed mode, the title is encrypted to a new key, different than the unified key, and updated with every new participant or when a participant leaves.

7.7.5 Switch to closed mode

A chat in open mode can be switched to closed mode by a moderator at any time. If a chat link exists, it will become invalid at this point and the participants in the group will start to rotate and distribute new keys. It is not possible for a chat in closed mode to be converted back into a chat in open mode.

Anybody who was previewing a chat at the time it became into closed mode will be no longer able to join, since the chat-link is automatically invalidated. In consequence, the history becomes unavailable for anyone but participants.

8. MEGAchat audio and video calling

MEGA users can make audio/video calls between each other.

Setting up a call involves three aspects: Signalling, media transport and encryption.

8.1 Call signalling

Call signalling is an exchange of messages between clients, through a server component, via a websocket link over HTTPS. Its purpose is to:

1. Set the call up by signalling the call request, answer/hangup and subsequent joins of new clients during a group call.

2. Establish webRTC media sessions between clients, by negotiating their common capabilities via SDP messages, and exchange information needed to set up a media transport path between them.

3. During the call, convey real-time information about the status of the participating clients and signals their eventual departure from the call. Also in case of a webRTC media connection loss, the connection is re-negotiated and established via the signalling channel

8.2 Media transport

During the webRTC session setup, endpoints exchange information (ICE candidates) that enables them to determine a transport path for the media they exchange. Whenever possible, the endpoints negotiate direct media flow between them.

8.3 Encryption

The call media stream is exchanged between endpoints in an SRTP-encrypted media session. To initiate the session, the SRTP (Secure Real-time Transport Protocol) encryption algorithm, keys, and parameters are negotiated through a DTLS (Datagram Transport Layer Security) handshake. The authenticity of the clients is also verified during the handshake, by authenticating the exchanged SDP (Session Description Protocol) descriptors, which, among other things, contain a fingerprint of the sender's SRTP encryption key. The SDP authentication is done by bundling the sent SDP with a MAC hash of it. The MAC hash is keyed with a peer-provided random key, which is encrypted with the key receiver's public elliptic curve (Cu25519) chat key.

8.4 WebRTC

MEGAchat is fully compliant with WebRTC and existing IETF standards. MEGAchat native endpoints can also securely exchange media with any WebRTC compliant web browser such as Google Chrome or Mozilla Firefox.

8.5 Group calling

Group calling is implemented with a mesh topology. Every client sends its media stream individually encrypted to every other client in the call. Everything is negotiated and exchanged on a peer-to-peer basis, and there are no shared keys or streams. The only common channel is the signalling channel.

9. Theoretical vulnerabilities

In this section we outline an assessment of the known theoretical vulnerabilities and shortcomings in the design of the system especially in regards to the end-to-end encryption model. These will ideally be improved upon in future versions of the system.

9.1 Vulnerability disclosure key separation, key integrity issues

Refer to issues found in vulnerability disclosure whitepaper[2]. The core issue, an "RSA oracle" that would allow an attacker, controlling the MEGA API or the user's TLS connection to the MEGA API, to decrypt the account of a user logging in at least 512 times was fixed on 22 June 2022 by way of a client software update.

9.2 Filesystem Integrity

An important question to address when designing an E2EE storage provider is whether to cryptographically protect the user's filesystem structure or not. MEGA opted against doing so, for the following reasons:

- The storage provider needs to be able to move or remove files on behalf of the user, e.g. when responding to helpdesk or compliance requests, expiring files in the rubbish bin, or purging excess older file versions.
- Files can be added to and removed from folders that the user is sharing with other users.
- Files can be added to a MEGAdrop folder by anyone.

Cryptographically protecting the integrity of public folder links is especially difficult because they can contain (dynamic) data from a (dynamic) set of user accounts: Folder links can point to shared folders, and any user with write access can make changes. Proper authentication from the point of view of a user accessing the folder link would require all data and all updates to be signed with a trusted public key of the respective acting user. To be trusted, independent confirmation of each signing key's authenticity has to take place (on the fly if it is a new one), and past experience indicates that virtually all real-world users would simply ignore that requirement.

2. Refer to p49

9.3 Server side authring deletion and rollback

In the webclient code, specifically around https://github.com/meganz/webclient/blob/v4.13.2/js/authring.js#L269, recreates the authentication keyring (authring) if it could not find it on the server. Each authring stored for a user's account contains the seen and verified fingerprints of their contacts. These are key/value pairs of user handles and fingerprints (SHA-256 hashes of the each contact's Ed25519 public key). It also stores fingerprints of the RSA and Curve25519 keys to prevent extra computation rechecking the signatures each time. This is a problem because if a user has spent time individually verifying the cryptographic fingerprints of their contacts and there is a network connectivity issue at the time of fetching this request (or MEGA arbitrarily decides to prevent this attribute from being downloaded), the client will automatically create a new authring and upload it to the server, thus overwriting their previous authring that was saved on the server. In the webclient UI, the verification status is displayed subtly, so a user may not notice that all their contacts were now no longer verified. If the user now initiated a file/folder sharing action, or new chat conversation with a contact, MEGA could be in a position to send different public keys and perform a MITM attack on the data or conversation.

This recreation is now unnecessary and will be removed. In the near future, failure to load the authring will trigger a prominent warning to the user.

9.4 Registration protocol issues

At present, if a user signed up with account using the old, lightweight login protocol (prior to 2019), there is no way to easily switch to the new login protocol. Old protocol accounts are substantially more vulnerable to dictionary and brute-force cracking, but they are perfectly safe if the user has set a password with sufficient entropy as recommended, but not enforced, by MEGA.

In the near future, a login or password change will automatically upgrade the account.

9.5 Pending contact shares

Users are able to share to people who are not contacts yet. This is called a pending contact share, and it involves one user sharing a file or folder with someone else by entering their email address. The other person then receives an email invite to join MEGA. Once that person has joined MEGA by registering their account, they are established as contacts. If both contacts are online at the same time, the sharing operation is able to be initiated, and this happens automatically in the background. At this point, because the contact relationship has just been established, the contacts would not have had time to verify each other's public key fingerprints first and the sharing process completes automatically.

MEGA could arbitrarily decide to intercept and MITM this process, thus being able to read the data being shared. They may also be able to read anything else being shared or communicated after that because the system trusts the first keys it sees and stores that as 'seen' in the authring. It is only until both contacts have verified fingerprints with each other that they may they discover that they have been MITM-attacked. For this reason, while it is fast and convenient to share files with someone who is not currently signed up to MEGA, this process should not be used to share anything sensitive.

9.6 Post-quantum security

Recent news articles suggest that commercially available general purpose quantum computers that can break public key cryptography used by MEGA may be between 5 and 15 years away. Shor's algorithm requires 6n logical qubits to break an elliptic curve key of size n. For RSA it requires 2n + 3 logical qubits. In terms of breaking MEGA's Ed25519 and Curve25519 256 bit keys this would require 1536 logical qubits. For MEGA's 2048 bit RSA keys this would require 4099 logical qubits to break. Once the logical qubit requirements have been met and using Shor's algorithm it can take seconds to break the encryption. It stands to reason that the elliptic curve cryptography with shorter keys will be the first to fall.

Google is well on the way to building a quantum computer in 2029 with 1000 logical qubits. A few years later, we might expect elliptic curve cryptography to be broken (although some other company like IBM could beat them to that first). Some nation states could already have a functioning quantum computer capable of breaking real-world public keys.

One of the main problems is that some nation states are storing all the encrypted data that can get their hands on today as it flows around the Internet, in the hopes that they can break it in the future with advances in computing power/quantum computers. Storing data with MEGA today could mean that the data is copied to a government datacentre and get stored until they have the ability to break it.

It also takes time to transition away from using vulnerable cryptographic algorithms and migrate user accounts. One problem is that the cryptographic community has not reached consensus on the security of any post-quantum cryptographic algorithms. There is an ongoing competition being run by NIST, but some may not decide to take up their recommendations. One popular theory is to use hybrid encryption (e.g. Curve25519 plus the post-quantum algorithm) in case the post-quantum algorithm turns out to be not as strong as it was thought to be against classical and quantum attacks. Then the system will still have the security of at least the classical algorithm.

9.7 Metadata sent back to MEGA with diagnostics enabled

Users sometimes run into issues when syncing and they are able to voluntarily report back diagnostics to the MEGA technical team for analysis. This process is described at https://help.mega.io/installs-apps/desktop-syncing/debug-diagnostics. Diagnostic files may contain metadata that would otherwise be protected by MEGA's end-to-end encryption. Diagnostic files are uploaded and shared with our senior engineers securely via end-to-end encryption, and our team undertakes to only use these diagnostics for technical analysis and to destroy the data once the inspection has concluded. Some log messages (e.g. those that report preview/thumbnail generation failures) could contain the file name that a failure occurred for. This would be useful for debugging, however, it could be a serious privacy concern for the user. In each case, it is important that the user reviews the log data before sending to make sure it does not contain information they do not want shared.

9.8 Code signing, authenticity

Apps

The MEGA apps in the Google Play and Apple stores are signed with keys from MEGA developers. However, the source code for the Play Store app in Android phones and the whole OS for Apple phones are closed source. In theory, Apple or Google may be able to tamper with the signature verification methods and allow the user to download backdoored software. There is no way for users to easily verify whether the app about to be installed has actually been tampered with. Leaked documents showed the NSA and its allies had the plans and methods in place to exploit mobile app stores since 2011.

Browser extensions

MEGA builds the desktop web client into three separate browser extensions for Chrome/Chromium, Firefox and Edge. Chrome extension updates are cryptographically signed by Google upon loading the extension update onto the Chrome Webstore. Addons Mozilla also has a signing process to say that the extension has passed their automated security checks and is considered authentic by Mozilla. However with these app stores, there is no method for MEGA to sign the contents of the extension to prove that this is the authentic source code written by MEGA. As these extensions are served by Google, Mozilla or Microsoft, they could insert their own backdoor into the code (either under their own volition or from a court order). The user needs to implicitly trust Google, Mozilla or Microsoft when using the extensions from their store.

MEGAsync / MEGAcmd

For MEGAsync and MEGAcmd, the application must be downloaded directly from the MEGA servers. To validate the Linux release files, the user needs to visit https://mega.nz/linux/repo/. Choosing their distribution directory e.g. https://mega.nz/linux/repo/Debian_11/, they can download Release.key and import it into their GPG keychain with gpg --import Release.key. Now they can download the Release and Release.gpg files. Then they can do gpg --verify Release.gpg Release to check the signature of the Release text file. In the Release text file is a list of hash sums. The user then downloads the Sources file and hashes it e.g. sha512sum Sources. This hash should match what is in the Release text file. The user then downloads the actual MEGAsync code e.g. megasync_4.6.5.orig.tar.gz and hashes it e.g. sha256sum megasync_4.6.5.orig.tar.gz, and this should match the hash in the Sources text file.

Through this process, the user is able to verify that the MEGAsync code is signed by the key that is found in the same directory. However, there is nothing preventing a hacker from hacking the web server and replacing the MEGAsync code, the key file, subsequent signatures and hashes to produce an authentic looking backdoored version. Users must verify that the key file they downloaded is indeed the MEGA key file by verifying the fingerprints of this key from at least one other source.

Source code

Source code links for the various client side projects is listed on https://mega.nz/sourcecode. Privacy-conscious users can technically build and run the projects from source code for use on their own devices if they wish, which would give better peace of mind and security against potential backdoors that MEGA could have included in the build/release process, if they are able to read and understand the entire source code, as backdoors could have been inserted anywhere. Although the restrictive Code Licence mentions that use of the code is only permitted for review purposes, nobody would actually know if users were building and running the code that they built themselves for general use purposes.

In most repositories, the MEGA developers contributing code sign all their commits and release tags with GPG, which can help users verify that the source code is legitimate. However, there is no central lookup directory of valid developers for each project, or their GPG keys and corresponding fingerprints, which could help with verification.

9.9 Supply chain attacks

Webclient

The webclient code includes some NPM (Node Package Manager) libraries for use in building and developing the React based MEGAchat and the webclient. There are 23 main dependencies and 32 dev dependencies listed in the project's package.json. This translates to 1137 packages installed into the node_modules which is approximately 217 MB of code. Implicit trust has been given into the open source development processes from these various package authors and the npm installation process.

NPM libraries have been consistently vulnerable to supply chain attacks in the past with incidents such as the left-pad, eslint-scope, event-stream, is-promise and node-ipc incidents. The webclient code takes some methods to mitigate issues such as locking the exact versions in the package.json file, to prevent inadvertent automatic updates to minor or patch versions of libraries. The generated JavaScript code built from the React code is also output to a js/chat/bundle.js file which is committed into Git like regular code. This means the generated code is reviewed by a senior developer alongside the Pull/Merge Request containing the regular React code and compared to make sure it looks plausible. This somewhat mitigates random backdoors being introduced from rogue npm packages and the building process. Automated processes also run on GitHub to alert about any packages with recent found vulnerabilities. However there still exists a remote possibility that something subtle could slip into the built webclient code that compromises the security of the product.

Other products

The native apps rely on a number of components provided by third parties - cryptopp, libcurl, chromium-WebRTC, BoringSSL, FreeImage, to name just a few -, which are updated periodically from a trustworthy source. However, it is impossible to fully audit all changes for backdoors every time.

9.10 File attribute MACs

File attributes (thumbnails) are not integrity protected. An adversary controlling the file attribute/API infrastructure or the user's TLS connection to the same could alter thumbnails for files with known keys.

9.11 Legacy chat key exchange

The RSA-based legacy chat key exchange is obsolete, creates a potential security vulnerability, and will therefore be removed in the near future.

9.12 Processing of RSA node keys

MEGA clients support RSA-encrypted node keys. No cryptographic sender authentication is taking place. To conserve CPU cycles, RSA-encrypted keys are transformed into AES-encrypted keys when encountered. An adversary controlling the API infrastructure could place files in a victim's account that appear to have been uploaded by the victim, even without knowledge of the master key. MEGA is planning to protect clients against this vector by cryptographically marking subfolders where RSA keys are legitimately required, such as MEGAdrop folders.

9.13 Partial downloads are not MAC protected

MEGA protects the integrity of regular downloads by way of a cryptographic checksum. It can only be computed and verified once the file has been downloaded in its entirety. Any action taken on partial data, such as the progressive rendering of images, PDFs or video, is therefore not afforded the protection of the file's MAC. It is trivial for an adversary who controls the MEGA infrastructure or the user's TLS connection to the storage nodes to flip specific bits of a file being accessed. If the file's content is known, it can trivially be replaced with a different file of the same length without having access to its specific encryption key.

9.14 MEGA S4, MEGA hosted buckets, control of keys

MEGA offers an S3-compatible storage service ("S4") that proxies (plaintext) S3 traffic to its E2EE storage back-end. The S4 proxy can be hosted by the customer on-prem, in which case cryptographic keys remain under the customer's control with the E2EE paradigm intact (up to the proxy). Alternatively, however, the customer can choose to outsource hosting of the proxy to MEGA. This inevitably breaks end-to-end encryption, as the proxy necessarily requires plaintext access to the subtree of the user's account storing the S4 buckets and their contents. To achieve this, the customer has to consciously communicate the bucket subtree's encryption key to MEGA.

10. References

(1) S. Contini. Method to Protect Passwords in Databases for Web Applications. Cryptology ePrint Archive, Report 2015/387, 2015. https://eprint.iacr.org/2015/387

(2) M. Backendal, M Haller, and K. Paterson. MEGA Vulnerability Disclosure. Department of Computer Science, ETH Zurich, 2022.