A Year and a Half of End-to-End Encryption at Misakey

Cédric Van Rompay (cedric.vanrompay@misakey.com)
September 8, 2020

A journey through some of the reasonings and technical challenges that I had so far as a software developer at Misakey specialized in cryptography and security.

Visit www.misakey.com Discuss with the author

How I Got Here

I was recruited by Misakey shortly after its first fundraising (February 2019, €1M). The mission of Misakey was, and still is as of today, to provide an easy-to-use and highly secure way to connect people to the numerous accounts they have on other websites, as well as to connect people between each other. One key element of the solution was to encrypt user data in an end-to-end fashion, meaning that, while the data exchanged by users and websites would flow through our servers, it would be encrypted with a key that Misakey does not have. Doing so greatly increases the security of user's data, but it adds a lot of technical challenges.

“Do not roll your own crypto”: this adage is a reminder to software developers that cryptography is a very tricky discipline. Trying to build your own cryptography without a high degree of knowledge in this field is a sure way to introduce a security vulnerability in your product. Instead, you should rely entirely on third-party tools and services when it comes to cryptography, and you should avoid using them in a “creative” way.

At Misakey, we try to follow this principle as much as possible. For instance, we do TLS[1] in the most standard, boring, uncreative way. But end-to-end encryption is still quite a “bleeding-edge” technology, and as a result you are not sure to find a tool that perfectly fits your needs. In

[1] TLS is the protocol that lets you communicate securely with websites. Websites using TLS have their address starting with `https://`

this situation, there are two sane things to do: giving up, or investing massively in cryptographic expertise.

The founders of Misakey went for the second option. Unfortunately, professional software developers with a high expertise in cryptography are pretty rare. So they went for the opposite approach: they started looking for an expert in cryptography that would have decent skills in software development.

At that time, I had recently finished my PhD on cryptography and secure protocols after graduating as an engineer from Télécom Paris and EURECOM. Although I had never worked as a professional software developer, I had been programming as a hobbyist since the age of 15, and the engineering schools I graduated from are quite specialized in I.T. After a few discussions on the phone with the founders, I was hired. The deal was that I would progressively become a professional software developer in his own right by programming with the rest of the team, while using my knowledge of cryptography to design and implement the protocols Misakey needs. I also had some training in general cyber security ("hacking", sort of), so I would be quite active on this topic as well[2].

End-to-End Encryption

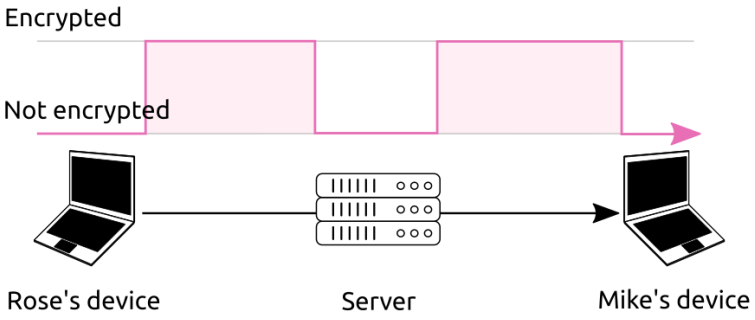
End-to-end encryption is really booming these years. This is sometimes taking the form of what is called "client-side encryption" in some cloud-based services, like what Cozy Cloud is doing for instance, but the biggest trend is with "encrypted chat applications". WhatsApp conversations use end-to-end encryption by default since 2016, and Facebook Messenger offers end-to-end encrypted conversations since more or less the same time. Telegram is another popular chat application that offers end-to-end encryption, and there are a few other applications having a smaller market share, like Signal and Element (formerly known as "Riot").

Before the rise of end-to-end encryption, messages were already encrypted in chat applications, but only with the TLS protocol[3] that provides encryption from a device (a computer or a phone) to a server. In TLS, the device and the server negotiate an encryption key with each

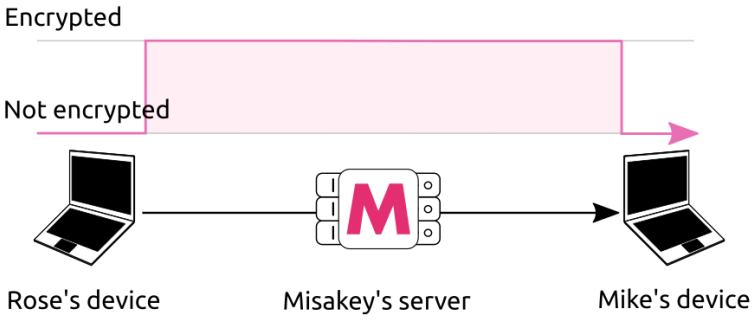
[2] Note however that, when it comes to cyber security in general, it is important that *everyone* in the company is involved to some degree, without exceptions.

[3] see here for the technical specification of its latest version, TLS 1.3

other so that they communicate securely, but because this key is known to the server, the server “sees“ the data. Most of the time this is perfectly fine because the data is actually intended for the server. For instance when you change your username in Misakey, the new user name is only encrypted with TLS. For most websites, TLS is (almost) all the cryptography that's required to operate the service securely, and nowadays it's quite simple for anyone to enable TLS on her website in a secure way thanks to Let's Encrypt and CertBot.



TLS only



TLS + end-to-end encryption

The situation is different in chat applications and in services like Misakey that simply act as intermediaries between users: the server is not one of the “ends” of the conversation anymore, it simply forwards data between users having a conversation. As a result, you cannot claim to provide “end-to-end encryption” simply because you are using

TLS.

It doesn't mean that a chat application using only TLS is “less secure“ than a usual website. It means that we could aim at a higher level of security: if the server doesn't need to see the data, we have an opportunity to protect the data from a hack of the server, and we do this by making the server unable to read the data.

It is tempting to ask: why not just use TLS from device to device, then? One reason is that TLS simply does not work from device to device: in TLS, servers do most of the work, so it is not trivial to recreate the same protocol with just devices. A second reason is that TLS will not give us some properties that we want for Misakey, like conversations between more than two devices and/or users. Of course, we will regularly see how things are done in the TLS protocol to better understand how to build our end-to-end encryption protocol, but it cannot be as simple as “using TLS from device to device”.

Existing Protocols and Why We Are Not Using Them

The end-to-end encryption protocol used by WhatsApp, Facebook Messenger and Signal is the “Signal Protocol” (because it was first developed by Signal, who then helped WhatsApp and Facebook integrate it in their own app) whose specification and implementation are free open-source software. This means that in theory we could use it to build Misakey, but in practice the Signal protocol is not really meant to be used as a stable platform for building other end-to-end encrypted products.

Element is a bit different from the other encrypted chat applications. The main goal of the people behind Element is to promote an entire messaging protocol, called the Matrix Protocol, whose original purpose was to “replace email”. Element is simply a client for this protocol, but the developers want anyone to be able to implement their own client, just like there are various programs to surf the Web or manage emails.

As a result, the Matrix protocol and the various parts of the Element application are designed to be usable as building blocks for other implementations and usage. In particular, the Matrix JS SDK gives you a high-level interface to use the Matrix Protocol, including the end-to-end encryption part, without having to worry too much about the technical details of it: you enable end-to-end encryption in a chat room by calling `matrixClient.setRoomEncryption` with the ID of the chat room,

and now all the messages sent to this room will be end-to-end encrypted, even if there are many people in it, each one using several devices. The Matrix team also provides a server for the Matrix Protocol, Synapse, which is very easy to use.

At the beginning of Misakey, the idea was to use the Matrix protocol as a communication platform to build our product. This way we did not have to implement end-to-end encryption ourselves, and we could enjoy a mature protocol and implementation which we would not have to maintain ourselves.

The Matrix protocol, and its end-to-end encryption protocol named "olm", are quite easy to use and to integrate in your own application ... as long as the application you are trying to build is close enough to Element. Now Misakey is not exactly a "chat" application, its goal is mainly directed towards automated management of people's accounts and data. As a result, we had some feature requirements that were quite far from what Matrix was designed for, like sending data to people that don't have an account yet, or seamless device-to-device synchronization. Implementing them with Matrix would have required to somehow "bend" the protocol, using it in ways it was not designed for, and this seemed overly complicated.

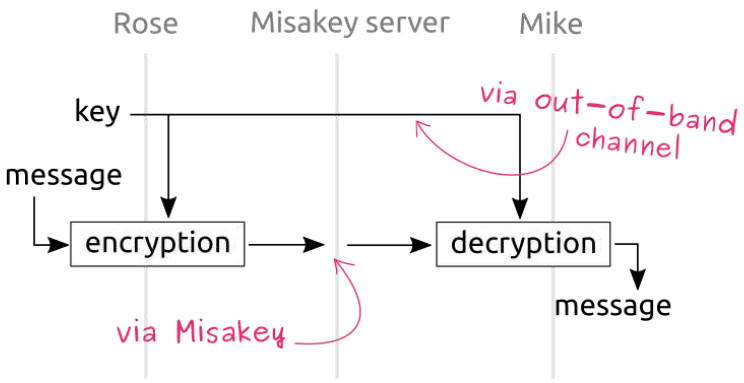
There were also a few features we were missing from the Matrix JS SDK, mainly regarding key management in the olm protocol. We had no idea if we could push for their integration, and it seemed too much of a hassle to implement them in our own fork of Matrix. At some time we tried to contribute to the Matrix JS SDK, but again this did not give us the speed we needed.

Maybe one day the Matrix protocol will become more versatile and we will be able to use it as a base for Misakey, but for now it seems faster to implement our own protocol, and to use Matrix as a source of inspiration. This lets us move faster, even if it is a great responsibility to implement end-to-end encryption from scratch. As we said, end-to-end encryption is a technology that is still quite "young".

The Most Trivial End-to-End Encryption Protocol

It's time to start building things. The quickest way of deploying end-to-end encryption between two users is to do the following: first, make the application of one user generate an encryption key. Then, tell this user to send the key to the other user through some other

communication channel, typically email or some other chat application,or in person. This “other communication channel” must be secure enough. When the application of the other user receives the key, the applications of both users can use this key to encrypt data for each other.



Note that the key must not be sent through Misakey itself, otherwise it is not “end-to-end encryption” any more. This is why I speak of another communication channel. This is called an out-of-band channel in end-to-end encryption.

Of course it is not ideal to have to assume that users already have this out-of-band channel to send cryptographic keys to each other in a secure manner. One could say, why wouldn't the users simply use this out-of-band channel for all their communications instead of using Misakey? But maybe it doesn't have the same features as Misakey, so it can still be interesting to use the out-of-band channel only to initiate end-to-end encryption in Misakey, and then switch to Misakey. In practice, all secure systems rely to some degree on the security properties of another secure channel (TLS, for instance, relies on the “root certificates” installed in your browser or your operating system). You cannot avoid it, you can only manage it, and later we will have to consider how this out-of-band channel can bring threats and how we can mitigate them.

Choosing an Encryption Algorithm

Now that applications can send encryption keys to each other, we can start encrypting, but what encryption algorithm are we going to use? Remember the rule: do not roll your own crypto. We are rolling our own end-to-end encryption protocol because there aren't any tools for it that are mature and versatile enough. But right now we are just looking for an encryption algorithm (a.k.a "cipher"), something that takes a key and a message as input and produces an encrypted message. This is not something new at all[4] and, consequently, there are mature tools we can use.

To me (and to a lot of people, see below), the best tool for this task is the NaCl specification. It provides high-level encryption mechanisms where most of the difficult choices are already made for you, it is hard to misuse it, and it has many high-quality implementations for more or less any language. Since Misakey is a Web application, we need a JavaScript implementation of NaCl, and we went for TweetNaCl.js because it has a wide adoption, it was audited by Cure53, and it seemed easy to use. I was tempted to use the official JS build of the famous libsodium library, but the project still looked quite in "beta version" and less trivial to use.

If you need to be convinced on how hard it is to get things right when picking cryptographic primitives yourself and combining them to get high-level encryption algorithms, you can read the famous article "Cryptographic Right Answers" by Latacora. And guess what they say in section "Asymmetric encryption"?

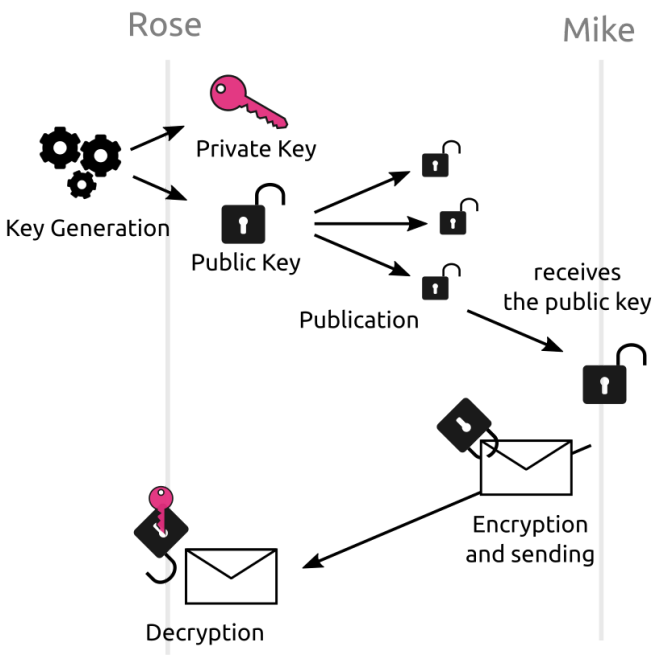
Use Nacl/libsodium [...] Your language will have bindings (or, in the case of Go, its own library implementation) to NaCl/libsodium; use them. Don't try to assemble this yourself.

Symmetric or Asymmetric Encryption?

Even if we use NaCl, there are a few decisions we have to make ourselves. NaCl provides both symmetric and asymmetric encryption. In symmetric encryption, the same key is used for both encrypting and decrypting data. On the other hand, in asymmetric encryption, a.k.a. "public key encryption", you don't generate one key but a pair of keys: one is called the public key and can only be used for encrypting data.

[4] the cipher we are using for instance (NaCl) was released in 2008, and that's one of the "recent" ones

The other key, called private key or secret key, is the one that decrypts data that was encrypted with the public key.



In most asymmetric encryption algorithms, the public key can also be easily re-computed from the secret one, but the converse is not true. This means that anything you can build with symmetric encryption, you can also build it using asymmetric encryption: you can always just use the asymmetric secret key as a symmetric key, and when you need to encrypt you first re-compute the public key (which, remember, is the one that encrypts).

Asymmetric encryption tends to be much slower than symmetric encryption however (see lines box and secretbox in this table), but this won't be a problem until we have a large number of messages to handle in a short amount of time. So it still looks like asymmetric encryption is an interesting alternative.

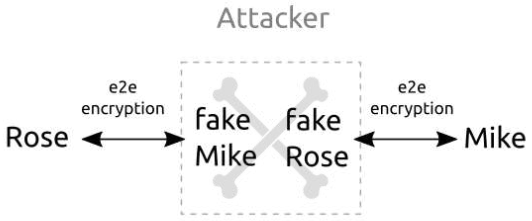
What's more, asymmetric encryption allows us to do the following: each user generates her own asymmetric key pair, uploads her public key to Misakey and keeps the secret key for her. Now it seems that we don't need an ”out-of-band” channel any more: When Rose wants to send

encrypted data to Mike[5], all she does is retrieve Mike's public key from Misakey's servers, and use it to encrypt the data. Even though Misakey's servers know Mike's public key, they don't know the corresponding secret key, and as a result they are unable to decrypt Rose's message.

Man-in-the-Middle Attacks and Why They Are Not a Priority for Us

Why did I say that “it seems” we don't need an out-of-band channel? Because there is a trick: an attacker controlling Misakey's server could perform one of the most famous attacks in secure protocols: man-in-the-middle.

In a man-in-the-middle attack, the attacker manages to change the key that Rose receives when she asks for Mike's public key: instead, she receives a public key from a pair that was generated by the attacker. As a result, when Rose encrypts data for Mike and sends it, the data can be decrypted by the attacker because Rose used the wrong public key. The attacker can then read and/or modify the data, and then send it to Mike after encrypting it with Mike's actual public key. Rose and Mike won't notice a thing, each thinking he/she's having an encrypted conversation with the other, while in reality each is having an encrypted conversation with the man-in-the-middle. To protect yourself against a man-in-the-middle attack, you need a mechanism to check the authenticity of the public key you are about to use.



TLS protects against man-in-the-middle by making servers go through a certification process, at the end of which the public key of the server is

[5] these are the Misakey version of the “Alice and Bob” names traditionally used in cryptography

signed by a Certification Authority (CA). As to the public keys of the CAs, they are installed in your operating system or in your Web browser. Such a system would not be applicable for Misakey or any chat application.

Existing end-to-end chat applications try to prevent man-in-the-middle attacks with a feature called “security code” (the name varies depending on the application, sometimes it's “safety numbers” for instance) : Rose and Mike must compare the security code generated by their respective applications for the chat room, and if they have the same code then no man-in-the-middle happened. However for Rose and Mike to compare their security code they should use ... an out-of-band channel. In practice nobody uses this feature, and even if you do it once, you are supposed to redo it again every time you see that “someone's security code changed”.

For now we don't have a “security code” mechanism at Misakey. We may deploy it one day, but I hope we will find something more practical. It turns out, it's not that bad regarding security. Man-in-the-middle is mostly an attack performed at the network level. The typical scenario is an attacker setting up a malicious WiFi access point: when a victim connects to this access point to get “free Internet”, every single packet sent between the victim's computer and websites the victim visits will go through the attacker's computer, so the attacker can perform a man-in-the-middle.

This scenario cannot happen when you use Misakey thanks to TLS that is protecting messages sent on the network between your device and Misakey's server. A man-in-the-middle attack against Misakey would require the user to take a complete control of Misakey's server itself to change the public keys that are sent to users, as well as the encrypted messages. This is much more difficult to do than simply setting up a malicious “free WiFi” somewhere. Complete takeover of servers is extremely rare in practice, and we put a lot of effort making sure this does not happen.

Encrypting Files

The ability for users to send files was a requirement from the beginning of Misakey: in the first version of Misakey you could not send text messages, only files.

If Rose sends a bunch of files to Mike, Mike will likely want to see a list of the files that were sent, with some information such as the size, the file name etc, before he decides to download some of them. It would be

very inefficient if Mike had to download all the (encrypted) files before he is able to see the list, especially if he doesn't end up downloading most of them right now.

As a result, when Rose sends a file through Misakey, there are two encrypted payloads that are sent: one is the file encrypted with a key called the file key that was generated just for this file, and the second is a small JSON object that contains this file key and some additional information like the file name. It is this second object, the JSON, that is encrypted with Mike's public key.

Now Mike can have the list of files sent by Rose (with their names) in a very efficient way by just downloading the encrypted JSON objects. When he decides there is a file he wants to download, he downloads the encrypted file itself, and he can decrypt it because he got the corresponding file key in the encrypted JSON. Recall, each file has a different file key.

The fact of generating a fresh symmetric key to encrypt data and then using asymmetric encryption to encrypt the key is called hybrid encryption and it is a wonderful trick to have both the performance of symmetric encryption and the features of asymmetric encryption. Actually the asymmetric encryption utility of NaCl already uses hybrid encryption under the hood.

Having two separate encrypted payloads has the additional benefit that we can use different storage technologies for them: the encrypted JSON goes in a SQL database so that we can run queries like “get all encrypted files sent by Rose to Mike from this date to that date”, while the encrypted file goes in a object storage database that is simpler, cheaper and faster for storing large objects.

Backing Up User Secrets in Our Servers

People have many devices nowadays, at least one phone and one computer, and they expect to be able to use an application in one device and then switch to another device without any problem. Many applications like your calendar, contact book, agenda etc. are able to offer this kind of cross-device synchronization by storing a copy of the data on a server. With end-to-end encrypted applications like Misakey, this strategy poses a problem: we cannot store the decryption keys in the server, or it would not be “end-to-end encryption” any more since end-to-end encryption mandates that the server does not have the decryption keys.

This problem is difficult enough to solve that WhatsApp simply gave up on it: your WhatsApp account and decryption keys are tied to your phone. You can use WhatsApp on your computer through WhatsApp Web but you are still going to need your phone with you and it must stay connected.

For Misakey it was essential that users had a better multi-device experience and did not depend on a phone. To this end, we put all the cryptographic keys of the user in a data structure which we call the user's secret backup, and the secret backup is stored at Misakey's server, but the Misakey Web app encrypts it before sending it to the server. The key used to encrypt (and decrypt) the secret backup is called the backup key, and because it is not known to the server, this is still end-to-end encryption.

Now we did not solve the problem, we just moved it: instead of having many secrets that we could not store at the server, we now have a single one, the backup key. The situation is a bit better because it is just one secret, and it should not change too often, whereas the list of decryption keys will evolve at least every time the user creates a new chat room on Misakey.

Actually we could just tell the user to save her backup key somewhere safe, and use it whenever she wants to start syncing a new device: the device would download the secret backup, decrypt it with the given backup key, and then it would have access to the decryption keys for all of the user's discussions. The device would also be able to create new chat rooms and upload a new secret backup that includes this key: it just has to use the backup key to encrypt the new version of the backup.

This is more or less the strategy that the end-to-end encrypted chat application Element (formerly Riot) has chosen: it asks you to choose a “passphrase” so that it can store an encrypted backup of your decryption keys at the server (this is called “Secret Storage” in the Element source code).

Deriving the Backup Key from the User's Password

Element gives you this weird instruction: the passphrase for your key backup should not be the same as your password. Why is that so? It is very tempting indeed for the user to just use the password she uses to log in to her Element account as her “secret backup passphrase”. But if she does so, then Element is technically not end-to-end encrypted

anymore.

Why? Because Element's server knows the password you use to log in Element (see this comment in the Element project and the few ones after it). If you use this same password as a passphrase for Element to generate your backup key, then Element's server, or any hacker getting control of it, can compute your backup key as well, then decrypt your secret backup, and then it has your decryption keys. And you just lost the benefits of end-to-end encryption.

People often forget that servers know the password you are using to log into them. The password you use at a website is not your secret, it is a shared secret between you and the website. Even if the password is only stored in hashed form on the server's disk, the server still sees the plaintext version of it every time you log in. This is the reason why it is so important to have one different password for each website (see my post on Misakey's blog on the topic of passwords).

Considering that users forget their password pretty regularly, it is far from ideal to ask them to have yet another secret to remember (or to write down somewhere, as we suggested earlier). Fortunately, there is a neat trick you can use to allow the user to use a single password without compromising end-to-end encryption: client-side password hashing.

Password hashing is a transformation the server applies to passwords before storing them on disk. This way, if an attacker manages to steal the database of the server, she does not get the passwords but just their “hash”. The hashing algorithm must be one-way, that is, infeasible to invert, but it must also be quite costly to execute in the “normal” way, so that it is hard for attackers to just hash every possible password and see which one results in a given hash (called “brute-force attack” or “dictionary attack”). The state-of-the-art algorithm for this task is Argon2, which is the one we use at Misakey, but older algorithms such as BCrypt, SCrypt and PBKDF2 are still widely used.

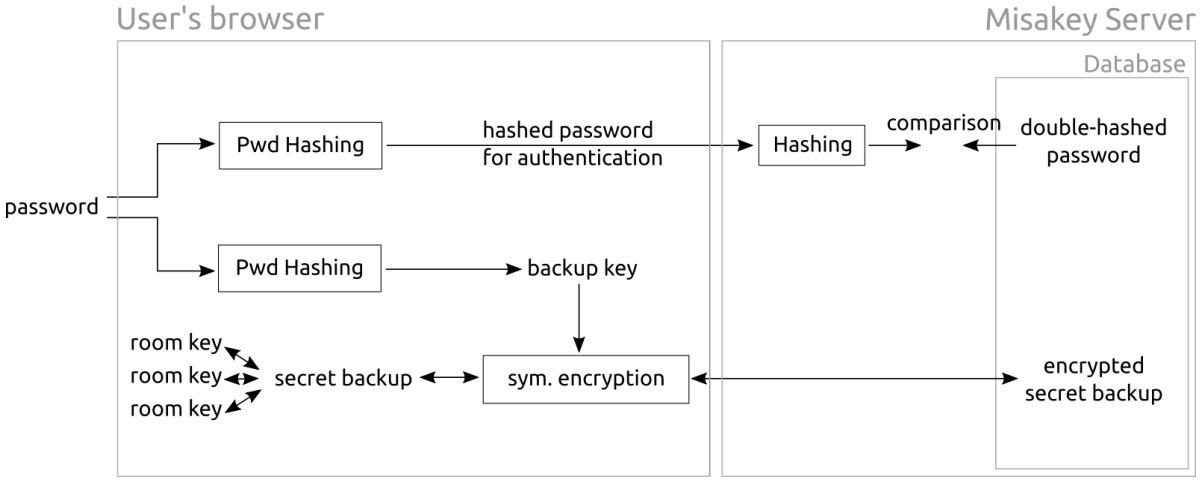
In Client-side password hashing, it is the Web application that performs the password hashing instead of the server. This means the server never sees the password, it only receives a hash of it.

As a result, with client-side password hashing we are able to use the user's password to derive the user's backup key. This key derivation operation is done with Argon2 again, only with different parameters.

Client-side hashing has another benefit being that it relieves the server from the resource-intensive task of hashing passwords. This is why

client-side password hashing is also known under the name server relief (see a description of it in the libsodium documentation).

Note that, even with client-side password hashing, the server should still hash what it receives from the client before storing it on disk, just to make sure that an attacker stealing the database cannot use it to log in as another user. Only this time the hashing algorithm does not have to be resource-intensive since the value that must be hashed is not a password but a (Argon2-)hashed password, and brute-force or dictionary attacks won't work against it.



The Headache of Users Losing Their Password

We have already seen a few examples of how end-to-end encryption makes everything more complicated. Well, you ain't seen nothing yet! Because we still haven't talked about users forgetting their password.

With most applications, forgetting one's password isn't really a problem : you click “I forgot my password”, enter your email address, and you instantly receive an email containing a link. You click on the link, you are asked to choose a new password, and you are done. This is so easy that I saw people preferring doing this rather than trying to remember their password or going to the next room to get the notebook where their passwords are.

Of course, Misakey has a “I forgot my password” button as well. And when you click on it, you receive an email to validate your identity, and

then you are able to set a new password and log in with it. The difference is that, because of end-to-end encryption, you just lost access to all of your data. Indeed without your password, your browser cannot decrypt your secret backup, and so it does not have access to your decryption keys, and you won't be able to read the content of any of your chat rooms, or any file that you saved in Misakey. So setting a new password makes you able to log in again, but you will be unable to access end-to-end encrypted data that you received before the password change.

Note that this isn't really different if we have a backup key that is not derived from the user's password but from a different “passphrase” as in Element : in this case, forgetting one's password isn't a big deal, but forgetting one's passphrase is catastrophic as well. And users are much more likely to forget their “passphrase” just because it's a weird concept they are not used to, and they were told to use something different than their password.

So what do we do? First, we must warn users about the consequences of using the “I forgot my password” button, so that they won't use it as a practical way to log in. Second, in every part of our application we must not assume that we have the decryption keys we need, and we must offer the best possible user experience when we don't have them. And third, we must implement mechanisms to help the user recover access to data that was lost due to using “I forgot my password”.

There are two such “recovery mechanisms” we are going to release soon in Misakey. One consists in letting the user try to remember past passwords she had forgotten: her application derives a backup key from the password and attempts to decrypt the old secret backup with this backup key. The other mechanism consists in letting the user export her backup key at any time: she can store it somewhere and use it later instead of her password to decrypt past secret backups. Note that, when the user exports her backup key, she is responsible for its security.

The astute reader will remark that, if the backup key is derived from the user's password, it changes every time the user changes her password. As a result, if the user changed her password between the last time she exported her backup key and the time she uses “I forgot my password”, the exported backup key will not decrypt the archived secret backup. The solution is to have two keys: one key that encrypts the secret backup, that does not change and is the one the user can export, and another key that is derived from the user password. The latter is used to encrypt and decrypt the former. The details of this mechanism will be

detailed in a further version of this text.

Why We Don't Store Keys on the Device's Hard Disk

Readers may be surprised that the situation is so desperate when a user uses the “I forgot my password” button. Isn't it possible that the decryption keys are still present on one of the user's devices?

First of all, even if we find ways to recover the user's decryption keys, we still have to consider the worst case when designing the application, so it does not change our efforts to make the application usable even when some decryption keys are missing.

Anyway, the answer is: no, the keys are most likely not present on any of the user's devices. The decryption keys are stored in a part of the device's memory that disappears as soon as the browser tab is closed (the “JavaScript Execution Context”, to be precise). Actually it is even worse than that: the keys will also disappear if the user refreshes the page, and they may disappear on a phone if the user switches from the browser to another application and doesn't go back to the Misakey browser tab before some time.

Now, modern browsers would let us store data in a more persistent memory. But we don't want to do it, for security reasons. Web browsers store the data of Web applications as files on the device's disk. On a computer, this means that any other program can access these files. It's a bit better on a modern smartphone, but still, we would not feel comfortable storing decryption keys there. If Misakey was a native application we could potentially require a “secure storage space” from the operating system, but we are not a native application.

As a result, every time you open Misakey, your browser has to download and decrypt your secret backup from the Misakey server. Only then it is able to read the messages and files in your chat rooms.

But remember, to decrypt the secret backup, your browser needs the backup key. We don't want to store the backup key on the device's disk, for the same reason that we don't want to store the decryption keys of your chat rooms. As a result, for quite some time the Misakey application would re-ask your password every time you would re-open the app, even if you were using it just seconds ago. Closing the tab, refreshing it ... all of these events would cause a new password prompt. Even worse: we use the OpenID Connect protocol during login, and this

protocol requires several redirections of the browser, which also flushes the secret backup out of the memory. This means that when you logged in, you had to type your password twice : a first time during login, and a second time after you are logged in, when you try to open a first chat room. This is not a smooth user experience at all, so solving this problem was a top priority for us.

For some time I was seriously contemplating storing decryption keys on the device. But there was another cryptographic trick that could save the day: cryptographic secret sharing.

Making Secrets Less Exposed with Secret Sharing

Secret Sharing is a cryptographic technique which turns a secret into several secret shares. By combining the shares together you can recover the original secret, but each share taken individually is completely useless. The most intuitive use case for secret sharing is if you are afraid of losing your password, so you would like to write it down somewhere, or to give it to a friend. With secret sharing you can do this in a more secure way: each friend and each place where you wrote something down only gets a share or your password. If an attacker steals one of the shares (or one of your friends turns against you) your password is still safe: the only way for an attacker to get your password is to steal all the shares.

First, a warning about how not to implement secret sharing. You may think that you can actually take parts of the secret. Like, if your password is p4ssw0rd123 (that's a terrible password, by the way), you would send p4ssw0 to one friend and rd123 to the other. Don't do this: it is not secure at all. It does not satisfy the property we expect from secret sharing that “having a share does not help you to recover the secret”.

If you think that with half of the secret, an attacker has half the amount of work required to find the entire secret, here is some mathematics for you: say a cryptographic key has 128 bits. Each bit can be 0 or 1, so there are 2^{128} different possibilities the attacker must try if she does not

known anything about the key[6]. This amount is so high[7] that it is considered infeasible to try them all. But if you already know half of the bits, now there are only 64 bits left to try, so 2^{64} possibilities. This is still quite a lot (18 billion billions), but that's definitely feasible with a modern computer. Remember that 128 bits, on the other hands, was completely impossible to crack. I like to use the following proverb for this phenomenon: half the key is not half the security. To be precise, with a 128 bits keys, half the key is (2^{128} / 2^{64}) = 2^{64} =18 billion billions times less secure.

So how do we do secret sharing securely, then? Many people will tell you about Shamir Secret Sharing[8] (SSS): don't listen to them. SSS is very secure, that's not the problem, but it's also quite complex. You shouldn't implement it yourself, so you will have to find a cryptography library that provides it.

There is a much simpler way to do secret sharing: XOR-based secret sharing (or just "XOR secret sharing"). It is trivial to implement and it suits the large majority of use cases. SSS is only better than XOR secret sharing when you want to turn your secret in a large amount N of shares and you want to use a threshold, that is a number T lesser than N such that any combination of T shares will suffice to recover the secret (useful if some shares are lost, or some of your friends are unable or unwilling to send you the share you gave them). For the large majority of use cases, the number of shares N will not be big enough to justify the use of SSS, and people should just use XOR secret sharing.

For a description of how XOR secret sharing works, see the Wikipedia article on Secret Sharing. Now let's see how we have been using secret sharing at Misakey.

Using Secret Sharing on Chat Room Keys

The first application of secret sharing at Misakey was not on the secret backup mechanism; instead it was applied to the mechanism by which users invite other people to chat rooms. The default way to invite

---

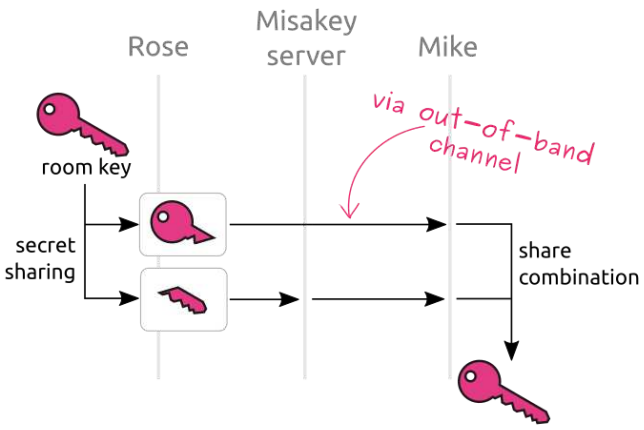
[6] there are two possible values for a single bit (0 or 1), 2^2 (that is, 2×2 = 4) values for a two-bits value (00, 01, 10 and 11), 2^3 (that is, 2×2×2 = 8) values for a three-bits value, and so on...

[7] that is 340,282,366,920,938,463,463,374,607,431,768,211,456 , to be exact

[8] see Section 2 of this paper for a description of SSS; the original paper is unfortunately not accessible for free

someone to a chat room in Misakey is to generate an invitation link for the chat room, which contains the address of the chat room in Misakey as well as the secret key necessary to decrypt its content. This is the “secret that is sent through an out-of-band channel” that we talked about earlier.

Hopefully the user will be careful about security when sending this invitation link, but accidents happen. The main risk factor is time: maybe nothing bad will happen in the few months after the link was sent, but maybe one of the two persons will have its mailbox or chat account hacked several years after, and the link is probably still present in it. In security, it is a good thing to consider that you always end up having some keys being leaked given a long enough amount of time.



To mitigate the consequences of an attacker stealing an invitation link, the user's browser does not put the decryption key of the chat room in the invitation link. Instead, the browser creates two shares out of the key, puts one share in the invitation link, and sends the other one to Misakey. This is compatible with end-to-end encryption because what Misakey received is not the decryption key, it is just a share of it, and Misakey's server will never see the other share.

Now if an attacker gets to steal the invitation link, she does not have the decryption key yet: she only has a share of it, and she has to get the second one from Misakey's server to get the chat room decryption key. This gives us an opportunity to block bad people from getting keys by applying various security checks before we accept to send our share of the key. In the near future users will be able to set access policies to

chat rooms, and these policies will be applied by the server before deciding whether to send the key share of the room or not.

We also have the opportunity to destroy our own key share at the request of the user, making all previous invitation links for this chat room completely useless. If we destroy our share of the chat room key, even an attacker stealing both the invitation link and all of Misakey's database will be unable to decrypt the content of the chat room.

Using Secret Sharing on the Backup Key

We said that we did not want to store the backup key in the browser's persistent storage because we were afraid that it would get stolen one day. Again, the risk that the backup key gets stolen right after we store it is pretty low. The risk is rather about the key staying for a long time on disk, and one day something happens and the key gets stolen.

Again, we solve this problem with secret sharing: the browser creates two shares for the backup key, one is stored locally, and the other one is sent to Misakey's server. Now when the user's cryptographic secrets are flushed out of memory, the browser can get the key share from the persistent storage, the second share from Misakey, and by combining the two it can recover the backup key and get access to all the encrypted data.

This means that an attacker getting access to the local storage of a user's browser still has some work to do before it can have the user's backup key. Also, Misakey's server will destroy the share it has when the device who uploaded the share has not been active for some time. This way, no matter how long the share of the backup key is kept in the device's storage, it is our server that controls for how long this share is of any use.

As a result, the user does not have to type her password too often, while we avoid storing decryption keys on devices' hard disk.

Wrapping Up

Having all their data “in the cloud” is very practical for users since they can access it from any device, but more and more people are worried about cloud service providers using their data without their consent, or these providers being victim of an attack resulting in users' data being leaked publicly.

End-to-end encryption is a solution to this problem that is applicable

when the cloud mostly acts as an intermediary passing data between various users but not having to process the data itself. With end-to-end encryption, the server, or any attacker stealing the content of the database, does not have the key to decrypt the users' data. This greatly mitigates the consequences of a security breach, and it also makes the servers a less attractive target for attackers.

However, end-to-end encryption is quite challenging to implement and to deploy. There are four main reasons for this:

- users are not as good as our servers at remembering secrets, meaning that they forget their password
- the server cannot help with many things, including recovering access to the user's data when they forgot their password
- users use several devices that must be kept in sync, and with end-to-end encryption this means keeping keys in sync as well
- storing cryptographic secrets on the device would be very practical but is not always safe

One day, there will probably be tools that let developers build end-to-end encrypted applications without having to design the end-to-end encryption with all the corner cases introduced by it (maybe this will be Misakey in a few years?), but we are not there yet, so we have to build end-to-end encryption ourselves, even if we can borrow ideas from the other existing encryption protocols out there.

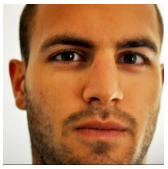
We have touched upon a few technical challenges that we encountered while designing and implementing end-to-end encryption at Misakey, as well as how we solved them with a smart use of pretty standard cryptographic techniques (regular encryption, secret sharing, and using several layers of keys to make key management simpler).

Of course we are not done improving Misakey, and a number of new features, as well as security and performance improvements are already planned :

- recovering one's password by using secret sharing and giving shares to other Misakey users
- transparently changing the keys of chat rooms on a regular basis (a.k.a. “key rotation”)
- searching in all chat rooms without having to download and

decrypt all of their content

About the Author and Misakey



Cédric Van Rompay is a software developer at Misakey, specialized in cryptography and security. He holds a degree in computer science from Télécom Paris and EURECOM and a PhD in cryptography from the same institutions.

cedricvanrompay.fr

Misakey: we believe in challenging the privacy statut quo, we believe in doing differently. The way we challenge the privacy statut quo is by making products that are easy to use, highly secure and privacy friendly. We made an app to protect the billions of documents shared everyday by email.

www.misakey.com

Visit www.misakey.com

Discuss with the author