

Dashlane Security White Paper

October 2018

Contents

1.	General Security Principles.....	2
a.	Protection of User Data in Dashlane.....	2
b.	Local Access to User Data	2
c.	Local Data Usage After Deciphering.....	3
d.	Use of 2FA Applications to Increase User Data Safety.....	3
e.	Authentication.....	3
f.	Communication.....	4
g.	Details on Authentication Flow.....	6
h.	Keeping the User Experience Simple	8
i.	Use of 2FA Application to Secure the Connection to a New Device.....	8
j.	2-Factor Authentication.....	8
k.	Sharing Data Between Users	9
l.	Using Password Changer to Further Increase User Security	11
m.	Account Recovery	12
2.	Impact on Potential Attack Scenarios.....	13
a.	Minimal Security Architecture	13
b.	Most Common Security Architecture	14
c.	Dashlane Security Architecture	15
d.	Anti-Clickjacking Provisions	18
e.	Same-origin Policy.....	18
f.	Memory Protection.....	18
g.	Intel SGX	19

1. General Security Principles

a. Protection of User Data in Dashlane

Protection of user data in Dashlane relies on 4 separate secrets:

- The **User Master Password**
 - It is never stored on Dashlane servers, nor are any of its derivatives (including hashes)
 - By default, it is not stored locally on disk on any of the user's devices; we simply use it to (de)cipher the local files containing the user data
 - It is stored locally upon user request when enabling the feature "Remember my Master Password"
 - In addition, the user's Master Password never transmits over the internet, nor do any of its derivatives (including hashes)
- In some cases (local storage), we use an **Intermediate Key** (random 32-byte) encrypted with the derived Master Password.
- A unique **User Device Key** for each device enabled by a user
 - Auto generated for each device
 - Used for authentication
- A **Dashlane Secret Key**
 - Used to secure communication between the Dashlane application and the browser plugins.

b. Local Access to User Data

Access to the user's data requires using the **User Master Password** which is only known by the user. It is used to generate the symmetric AES 256-bit key for ciphering and deciphering the user's personal data on the user's device.

The user's data ciphering and deciphering is performed using *OpenSSL*:

- A 32-byte salt is generated using the *OpenSSL* `RAND_bytes` function (ciphering) or reading it from the AES file (deciphering).
- The User Master Password is used, with the salt, to generate the AES 256-bit key that will be used for (de)ciphering. We use Argon2d, by default, with the following parameters: iterations = 3, memory = 32Mo, parallelization = 2. We also support PBKDF2-SHA2 with 200,000 iterations.
- The 16-byte initialization vector is chosen randomly.
- Then, the data is (de)ciphered using CBC-HMAC mode.
- When ciphering, the salt is written in the AES file.

c. Local Data Usage After Deciphering

Once the user has input their Master Password locally in Dashlane and their user data has been deciphered, data is loaded in memory.

The Dashlane client operates within significant constraints to use deciphered user data effectively and securely:

- Dashlane processes decipher and access individual passwords to autofill them on websites or to save credentials without having to ask the user for the Master Password each time.
- Users require that these actions are performed quickly.
- The passwords are sent from different processes through named pipes or web sockets from core to plugins (but are encrypted using AES first).
- The Argon2d (or PBKDF2) derivation used to compute the AES keys adds significant latency (on purpose to protect from brute force attacks).

See in §2.f for more on memory management.

d. Use of 2FA Applications to Increase User Data Safety

At any time, a user can link their account to a 2FA application on their mobile device (an example among others is Google Authenticator). All of their data, both the data stored locally and the data sent to Dashlane servers for synchronization purposes are then ciphered with a new key, which is generated by a combination of the **User Master Password** and a randomly generated key called the **User Secondary Key** stored on Dashlane server, as described in the following steps:

- The user links their Dashlane account with their 2FA application.
- Dashlane servers generate and store a **User Secondary Key**, which is sent to the user's client application.
- All personal data are ciphered with a new symmetric AES 256 bits generated client side with both the **User Master Password** and the **User Secondary Key**.
- The **User Secondary Key** is never stored locally.
- The next time the user tries to log into Dashlane, they will be asked by Dashlane servers to provide a One-Time Password generated by the 2FA application. Upon receiving and verifying this One-Time Password, Dashlane servers will send the **User Secondary Key** to the client application, allowing the user to decipher their data.

Doing so, user data can be deciphered only by having both the **User Master Password**, and the **2FA** application linked to the user's account.

e. Authentication

As some of Dashlane's services are cloud-based (data synchronization between multiple devices for instance) there is a need to authenticate the user on Dashlane servers.

Authentication of the user on Dashlane servers is based on the **User Device Key** and has **no relationship with the User Master Password**.

When a user creates an account or adds a new device to synchronize their data, a new User Device Key is generated. The User Device Key is composed of two parts:

- First, a predictable part based on some hardware and software characteristics of the user's device.
- Second, 38 characters (lower letters, capital letters, and numbers) generated using the *OpenSSL* `RAND_byte` function.

This User Device Key is then stored locally in the user data, ciphered as all other user data as explained earlier, and sent to our servers. When a user has gained access to their data using their Master Password, Dashlane is able to access their User Device Key to authenticate them on our servers without any user interaction.

As a result, Dashlane does not have to store the user Master Password to perform authentication.

f. Communication

All communications between the Dashlane application and the Dashlane servers are secured with HTTPS. HTTPS connections on the client side are performed using *OpenSSL*. On the server side, we use a *DigiCert* High Assurance CA-3 certificate¹.

The HTTPS communications between Dashlane application and the Dashlane servers are using SSL/TLS connections.

Main TLS protocol steps are as follows:

- The client and the server negotiate to choose the best cipher and hash algorithm available on both side.
- The server sends a digital certificate.
- The client verifies the certificate by contacting a Certificate Authority.
- The client encrypts a random number with the server's public key and sends it to the server.
- The server decrypts this number, and both sides use this number to generate a symmetric key, used to encrypt and decrypt data.

Finally, communication between the Dashlane browser plugin and the Dashlane application is secured using with AES 256 with the *OpenSSL* library:

- A 32-byte salt is generated using the *OpenSSL* `RAND_bytes` function (ciphering) or reading it from the inter-process message (deciphering).
- The **Dashlane Private Key** is used, with the salt, to generate the AES 256-bit key that will be used for (de)ciphering.

¹ Key Length: 2048-bit, Signature algorithm = SHA2 + RSA

- The IV is set to a random 16-byte value.
- Then, the data is (de)ciphered using CBC-HMAC mode.
- When ciphering, the salt is written on inter-process message

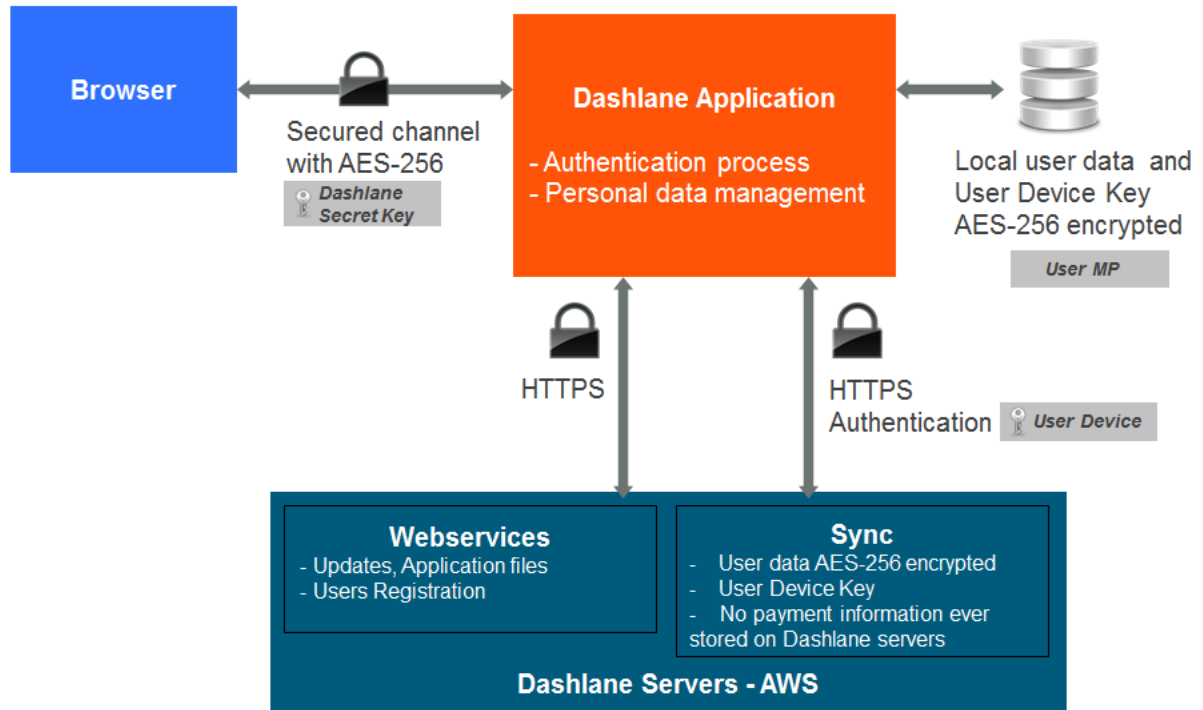


Figure 1: Use of Authentication Mechanisms in Dashlane

g. Details on Authentication Flow

The initial registration for a user follows the flow described in Figure 2: *Authentication Flow During Registration*.

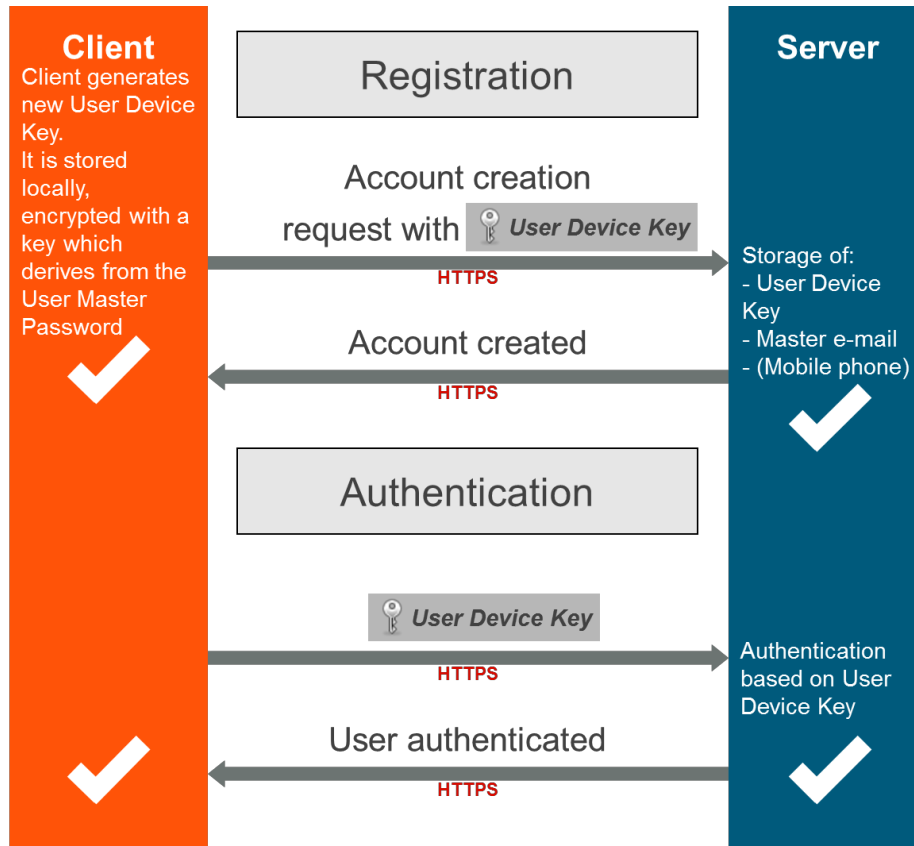


Figure 2: Authentication Flow During Registration

As seen in Figure 2, the User Master Password is never used to perform server authentication, and the only keys stored on our servers are the User Device Keys.

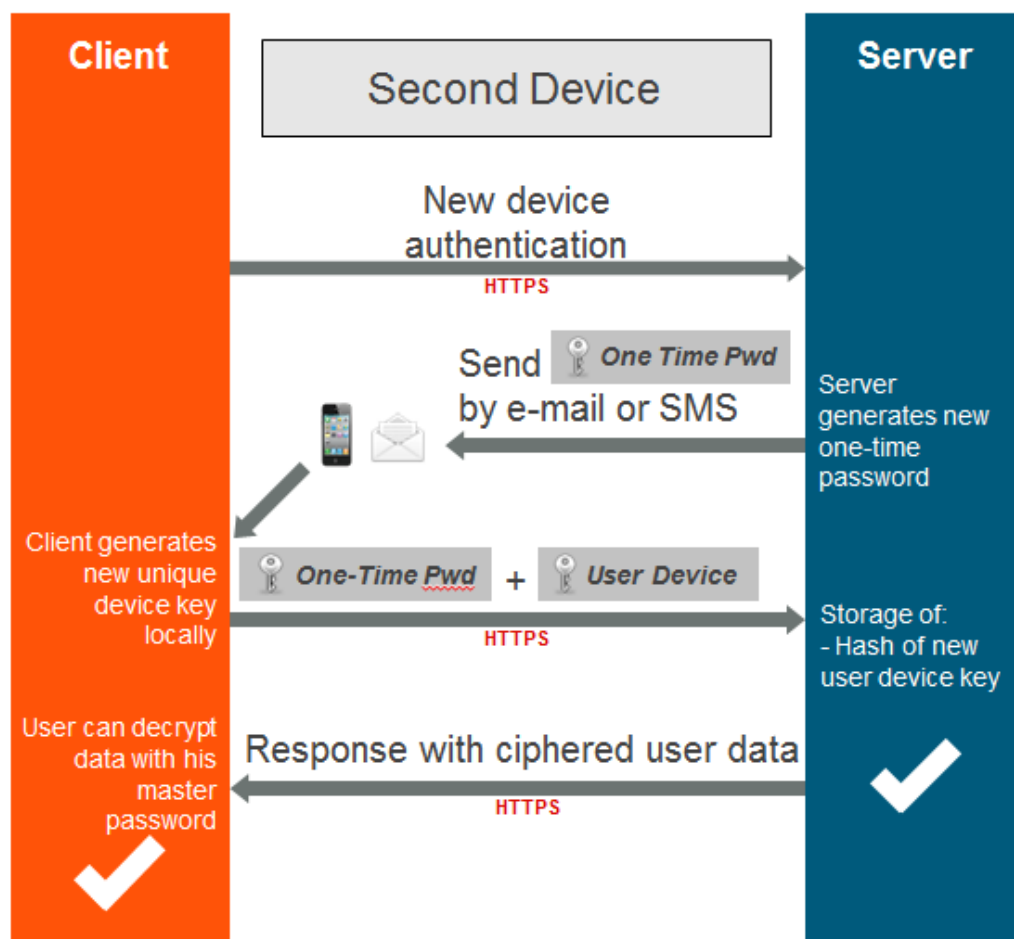


Figure 3: Authentication When Adding a New Device

When adding a second device, the important point is that Dashlane needs to make sure the user adding the additional device is indeed the legitimate owner of the account. This is to gain additional protection in the event the User Master Password has been compromised and an attacker who does not have access to their already-enabled device is trying to access the account from another device.

As shown on Figure 3, when a user is attempting to connect to a Dashlane account on a device that has not yet been authorized for this account, Dashlane generates a One-Time Password (a Token) that is sent to the user either to the email address used to create the Dashlane account initially or by text message to the user's mobile phone if the user has chosen to provide their mobile phone number.

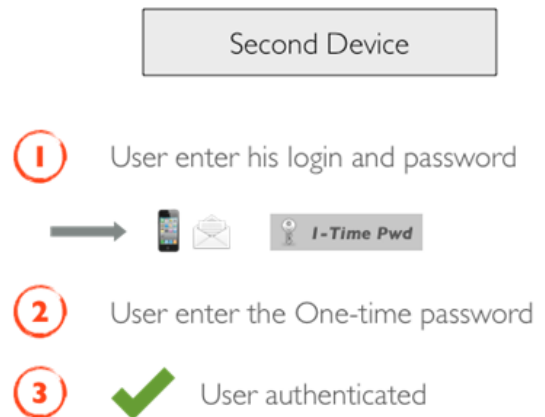
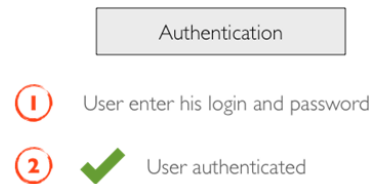
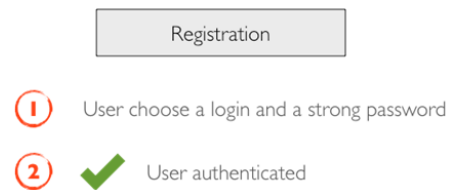
In order to enable the new device, the user has to enter both their Master Password and the Token. Only once this two-factor authentication has been performed will Dashlane servers start synchronizing the user data on the new device. All communication is handled with HTTPS and the user data only travels in AES-256 encrypted form. Please note again that the User Master Password never transmits over the internet.

h. Keeping the User Experience Simple

All along, our goal is to keep the user experience simple and to hide all the complexity from the user. Security is growing more and more important for users of cloud services but they are not necessarily ready to sacrifice convenience for more security.

Even though what goes on in the background during the initial registration steps is complex (see Figure 2), the user experience is very simple. All they have to do is to pick a (strong) Master Password, and all the other keys are generated by the application without user intervention.

When adding an additional device, the process is equally simple while remaining highly secure through the use of two-factor authentication described in Figure 3.



i. Use of 2FA Application to Secure the Connection to a New Device

At any time, a user can link their Dashlane account to a 2FA application on their mobile device. When they attempt to connect into a new device, instead of sending them a One-Time Password by email, Dashlane asks the user to provide a One-Time password generated by the 2FA application.

After receiving and verifying the One-Time Password provided by the user, Dashlane servers will store the **User Device Key** generated by the client application, as described in Figure 3.

j. 2-Factor Authentication

Dashlane offers 2-Factor Authentication that can be activated from the security settings in the desktop application to force the usage of a second factor each time the user logs into Dashlane.

Supported two-factor methods include 2FA applications such as Google Authenticator or U2F compatible devices such as Yubikeys. U2F is an open protocol from the FIDO Alliance (<https://fidoalliance.org>). Dashlane is a member of the FIDO Alliance.

k. Sharing Data Between Users

Dashlane allows users to share credentials and secure notes with other users, or with groups of users, in such a way that Dashlane never directly accesses a user's data at any point. In fact, Dashlane's servers never have access to the content of shared data.

Dashlane's sharing relies on asymmetric encryption; upon account creation, a unique pair of public and private RSA keys are created by the Dashlane application for each user. The private key is stored in the user's personal data, and the public key is sent to Dashlane's servers. RSA public and private keys are generated using the OpenSSL function `RSA_generate_key_ex`, using a key length of 2048 bits, with 3 as a public exponent.

Here is the process for a user, Alice, to share a credential with another user, Bob:

- Alice asks Dashlane's servers for Bob's public key.
- Alice generates a 256-bit AES key, using a cryptographically secure random function on each platform. This key is unique for each shared item and is called an `ObjectKey`.
- Alice encrypts the `ObjectKey` using Bob's public key, creating a `BobEncryptedObjectKey`.
- Alice sends the `BobEncryptedObjectKey` to Dashlane's servers.
- Alice encrypts her credential with the `ObjectKey`, using AES-CBC and HMAC-SHA2 creating an `EncryptedCredential`.
- Alice sends the `EncryptedCredential` to Dashlane's servers.
- When Bob logs in, Dashlane's servers inform him that Alice wants to share a credential with him. Bob must manually accept the item in his Dashlane application and sign his acceptance using his private key.
- Upon acceptance, Dashlane's servers send Bob the `BobEncryptedObjectKey`, and the `EncryptedCredential`.
- Bob decrypts the `BobEncryptedObjectKey` with his private key and gets the `ObjectKey`.
- Bob decrypts the `EncryptedCredential` with the `ObjectKey` and adds Alice's plain text credential to his own personal data.

Sharing data with a group of users follows the same security principle: Use a user's RSA public and private keys to send protected AES keys, sign a user's action, and use intermediary AES keys to exchange data.

To summarize:

- Each user has a pair of public and private RSA 2048-bit keys:
 - o Public keys are used to encrypt information only a specific user can decrypt.
 - o Private keys are used to sign actions users are performing.
- For each credential or secure note shared, an intermediary AES 256-bit key is created and used to perform data encryption and decryption.

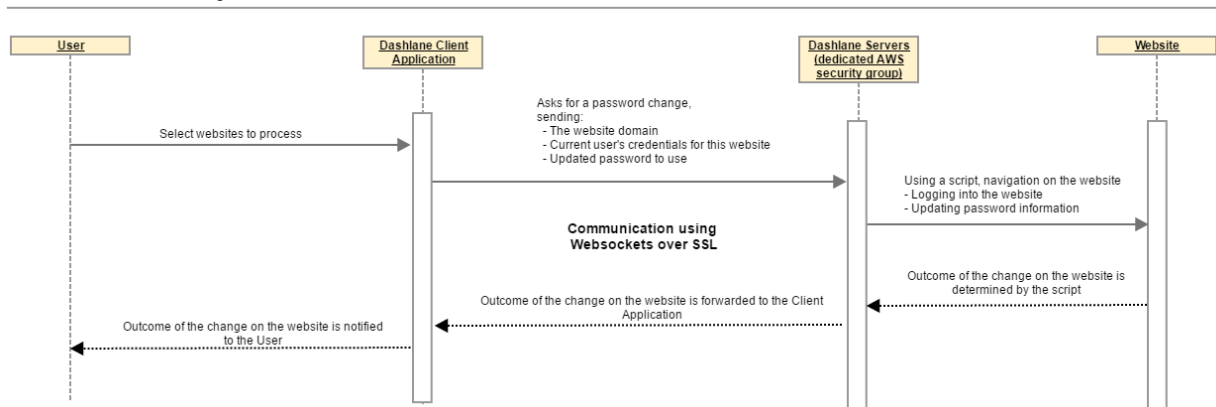
I. Using Password Changer to Further Increase User Security

The Password Changer feature of Dashlane offers a 1-click experience to change a password for a particular website. This makes changing passwords for compromised websites easier. Furthermore, it provides users a convenient way to regularly update their passwords without going through the hassle of manually updating passwords for accounts they have. Password Changer makes a very important, rarely followed security practice a lot easier.

To change a password for a particular website, a Dashlane's client sends the current saved password to Dashlane's servers along with a new strong password generated on the client. This communication is done using secure WebSockets (WebSockets over SSL/TLS – the SSL termination is done using AWS Elastic Load Balancers as for any other Dashlane webservices) to prevent Man-in-the-Middle attacks. The servers try to log in to the targeted website and change the user's password using either a browser navigation or a call to an API, depending on the website. Dashlane prompts the user for additional information if needed (e.g. security question) using the same secure WebSocket connection. At the end of the operation, it notifies the user with the result. In case of success, the client updates the password locally.

The servers (AWS EC2 instances) that are used to supply Password Changer are separated from the rest of the Dashlane's server infrastructure (dedicated instances and distinct AWS security groups). Additionally, on the server side, sensitive information (e.g. logins and passwords) is stored in RAM only. It's removed from RAM right after the result is sent back to the client (the password change takes 45 seconds in average), or after five minutes in case of a client disconnection.

Dashlane Password Changer



m. Account Recovery

Account Recovery allows Dashlane Business users to regain access to Dashlane by resetting their Master Password. Our patent pending process preserves a zero-knowledge architecture. No one is able to access the user's data or recover the lost Master Password. Master Passwords are never stored on any servers nor transmitted in any form.

Our solution allows users to both reset their Master Password and recover the data stored on an authorized device. Account Recovery is an optional feature admins can activate for their Dashlane Business account in the Admin Console.

To enable recovery, the user's local key—itsself encrypted with the user's Master Password—is also ciphered using a unique user recovery key, which is generated and used for all of the user's devices when they opt into Account Recovery. This user recovery key is then ciphered using a unique server-side recovery key, which is only known to Dashlane and the user's client devices. When an admin enables Account Recovery, their public key is used to cipher the server-side recovery key which, as aforementioned, was already used to encrypt the user's recovery key. An admin can then, via their private key, later access the user's recovery key protected by the server-side recovery key.

When a user requests Account Recovery, they are asked to verify their account and create a new Master Password. A critical step of the recovery process is the verification of the identity of the user. It is up to the admin, acting as a trusted third-party, to ensure the user requesting recovery is indeed the owner of the account. If an admin approves the request, the server-side recovery key, which protects the user's recovery key, is securely exchanged from the admin to the user through a public/private key system. On the user's device, the user's recovery key is then decrypted using the server-side recovery key, provided by Dashlane after the user's identity and request have been validated. The user's recovery key is then used to decipher the user's local key, which in turn is used to decipher the user's data. The recovered data is then re-encrypted with the new Master Password and re-synced to the Dashlane servers.

As this process involves a Master Password change, all of the user's devices have to be re-registered to Dashlane for the user to access their newly encrypted data.

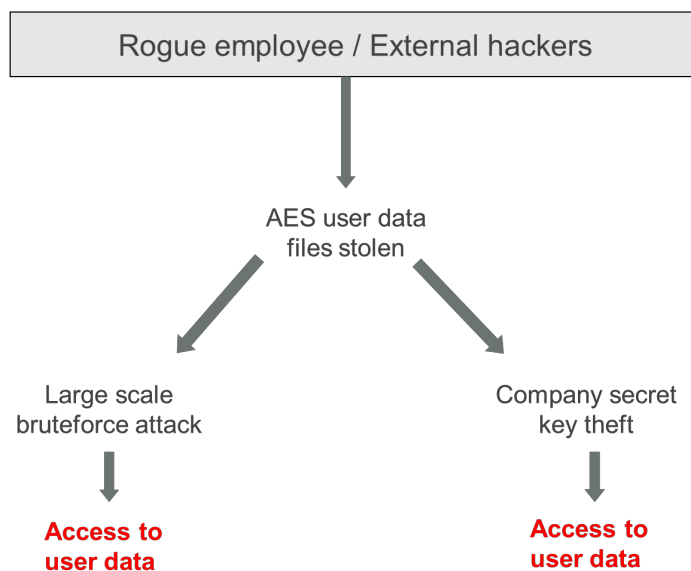
2. Impact on Potential Attack Scenarios

Today, cloud-based services make various choices to encrypt their user data. These choices have important consequences in terms of security.

a. Minimal Security Architecture

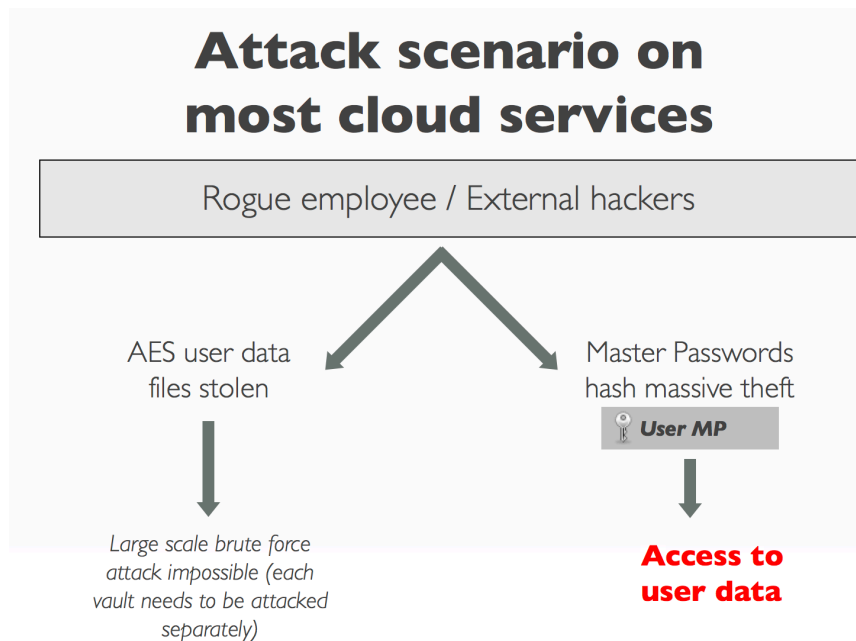
Cloud services can use a **single private secret**, usually under their control, **to encrypt all user data**. This is obviously a simpler choice from an implementation standpoint, plus it offers the advantage of facilitating **deduplication** of data which can provide important economic benefits when the user data volume is important. Obviously, this is not an optimal scenario from a security standpoint since if the key is compromised (hacker attack or rogue employee), all user data is exposed.

Attack scenario



b. Most Common Security Architecture

A better alternative is to use a different key for each user. The most common practice is to ask the user to provide a (strong) Master Password and to derive the encryption key for each user from their Master Password. However, to keep things simple for the user, many services or applications tend to also use the user's Master Password as an authentication key for the connection to their services. This implies that an attacker could access a user's vault by just knowing the master password. It could also easily lead to implementation errors (missing salt/rainbow tables attacks, wrong/weak hashing, etc.).



c. Dashlane Security Architecture

In order to make this attack scenario impossible, we have made the decision to separate the key used for user data encryption and the key used for server based authentication (See Figure 4: *Limits on Attack Potential With Dashlane's Security Architecture*). The user data is encrypted with a key which is a derivative of the User Master Password. A separate User Device Key (unique to each device-user couple) is used to perform authentication on Dashlane Servers. This User Device Key is automatically generated by Dashlane. As a result:

- Encryption keys for user data are not stored anywhere
- No Dashlane employee can ever access user data
- User data is protected by the Master Password even if Dashlane's servers are compromised

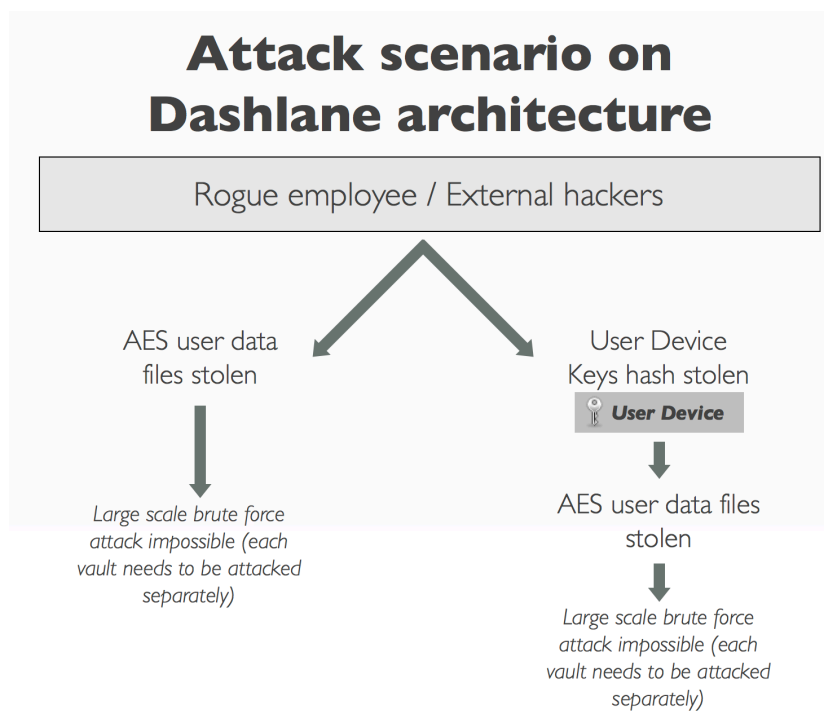


Figure 4: Limits on Attack Potential With Dashlane's Security Architecture

Even if this scenario happens, a rogue employee or an external hacker would have a very hard time executing a brute force or dictionary attack on the AES user data files, as we use the Argon2d (or PBKDF2-SHA2) algorithm. As the user data are encrypted using a salted key which is a derivative of their User Master Password, no rainbow attacks are possible.

As an example, this is a benchmark of attempts to decipher AES files using a Xeon 1.87GHz (4 cores):

	Time to get the password on a Xeon 1.87 GHz (4 cores)		
Type of brute force attack	AES 256	Argon2 (3 iterations, 32 Mb)	AES 256 with PBKDF2-SHA2 with 200,000 iterations
4 million-term dictionary	2.8 seconds	46 days	~450 hours
Alphanumerical (small caps + digits) password of 7 characters	15.7 hours	111,669 years	~1,000 years
Alphanumerical (small caps + digits) password of 8 characters	23.6 days	6,923, 519 years	~45,000 years

This table represents the time it would take on a Xeon 1.87 GHz (4 cores) to break a password used to protect Dashlane user data. Without using Argon2 or PBKDF2, those numbers show that even with a strong password, an attacker would be able to crack the user password in less than a month.

Using Argon2 or PBKDF2, and given that Dashlane enforces reasonably strong password requirements² (and so User Master Passwords are not contained in a dictionary), an attack would be impractical.

Obviously, there is a limit to any security architecture. If the user’s computer is physically compromised and an attacker is able to install a keylogger allowing them to capture all keystrokes, then no password-based security system will prevent data theft or piracy. This is why the end user still remains responsible for physically protecting their computer from non-authorized access and for making sure they are not installing potentially infected software. Our point is that in any event, a Dashlane user is significantly more secure than if they store sensitive

² At least one upper case, one lower case, 1 digit, and at least 8 characters

personal data in Word or Excel documents, uses their memory or a system that necessitates weak password patterns, or stores passwords in their browser caches.

d. Anti-Clickjacking Provisions

In order to protect Dashlane users from rogue websites that would attempt to use clickjacking tactics or other JavaScript based attacks to extract data from the Dashlane application, we have made sure none of the webpage-based interactions involving user data unrelated to this website use JavaScript.

Instead, all the interactions³ involving user data have been written in C++, and this compiled code has seen the use of various packing and protection methods to further complicate reverse engineering attempts and make clickjacking and other JavaScript attacks extremely difficult to perform. This of course won't be relevant if the user's computer has been compromised by a rogue program.

For example, the popups used to trigger form-filling on a webpage are C++ popups and are external from the JavaScript. As a result, a rogue website cannot trigger a click that would cause Dashlane to believe that the user has actually clicked, and therefore, cannot extract information unless the user explicitly clicks in the field.

e. Same-origin Policy

Dashlane automatically logs users into websites. In order to avoid providing user's information to rogue websites, the same-origin policy is always respected.

First, a credential saved by Dashlane when it has been used on a website with the URL of mysubdomain.mydomain.com will not be automatically filled on another website with the URL of myothersubdomain.mydomain.com. This prevents the credential of a specific website from being provided to another website which would share the same top-level domain name.

Also, a credential saved by Dashlane when it has been used on a website with a URL beginning with https will not be automatically filled on another website with a URL beginning with http.

f. Memory Protection

A problem can arise if an attacker takes control of the user's client device. In that scenario, the attacker could retrieve the deciphered user data from the memory.

This is an extreme scenario as, in that case, the attacker can take control of many parts, including adding a keylogger to capture the Master Password.

- Mobile operating systems (Android, IOS) ensure that no process can ever access the memory of another process and *are not directly affected*.
- Sandboxed process: Windows store or Mac app store apps *can't access other process memory, either*.

³ The only exception being interactions where data specific to the website is provided like the automatic login where we do not create any additional risk by using JavaScript

- Non-Sandboxed desktop apps are an issue. They can access memory with classic system API (CreateRemoteThread, ReadProcessMemory, WriteProcessMemory on Windows) or classic debugging tools.
- On Windows, Dashlane binaries are compiled with ASLR enabled.

Dashlane is working on adding additional memory protections (add intermediate cipher keys, wipe chunks of memory before releasing, etc.), but we also need to take into account that if the attacker has control over the process memory, they can already cause a lot of harm and can bypass such countermeasures:

- Hook process decryption functions
- Add a keylogger
- Hook SSL http functions and retrieve passwords
- Tamper certificates authorities
- Debug, trace, add watches, and bypass added security

Finally, we believe the system integrity and security between processes is a system function and Dashlane cannot (and should not) reinvent the wheel and add useless complexity that could lead to other vulnerabilities and have negative side-effects on antivirus.

g. Intel SGX

Dashlane Windows application supports the Intel SGX protocol, which adds another layer of security for Dashlane customers using Intel CPU compatible with SGX instructions.

When SGX-compatible hardware is detected by Dashlane, it enables the use of the secure enclave to further protect the encryption key. When this feature is enabled, the key is derived two times:

- One time using the classic Dashlane derivation function (today Argon2d or PBKDF2)
- A second time using the SGX enclave to do another derivation. The salt of the derivation is confined in the enclave and can't be read by anyone. The encryption is also done in the enclave, so an attacker [can never access this key](#).