ExpressVPN Keys Security White Paper

Updated October 31, 2023

Table of Contents

1. Introduction

In 2022, 86% of all attacks against web applications started with stolen credentials, mainly weak or reused passwords[1]. It's an enormous problem… with a simple but criminally underutilized solution.

A **password manager** not only makes it easy to replace weak credentials with strong ones, it also greatly reduces damage caused by any one credential's exposure. To encourage more widespread adoption of this solution, we at ExpressVPN decided it was time to draw on our over 10 years of security expertise to develop our own password manager: **ExpressVPN Keys**.

ExpressVPN Keys (or, simply, "Keys") is not a standalone app; it lives in our flagship ExpressVPN apps for iOS and Android, and in a browser extension for Chrome (and other Chromium-based browsers) for use on Mac, Windows, and Linux computers. Everything stored in Keys is securely synced across all platforms for easy access, and Keys users can store and view as many items (logins, secure notes, credit/debit cards) across as many devices as they like.

Keys has many security features, including **biometric unlock**, **two-factor authentication**, **breach alerts**, and a security "scoring" system for each login, which we call **Password Health**. In the interest of transparency and greater security awareness, we created this white paper to outline our implementation of these and other features. We also take the opportunity to discuss the design, philosophy, and cryptography principles on which Keys is built, maintained, and improved.

We welcome any questions or comments you may have about Keys or this white paper, which you can send to us by contacting ExpressVPN Support. If you find a bug or potential vulnerability, please report it immediately through our bug bounty program so our engineers can validate and triage the issue.

Now, let's talk about Keys!

[1] 2023 Data Breach Investigations Report, Verizon:
https://www.verizon.com/business/resources/reports/dbir/2023/summary-of-findings/

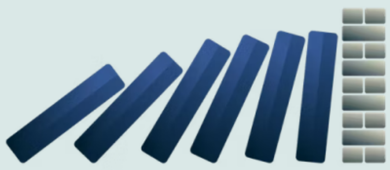
2. Strategies and principles

The following strategies have always guided how we protect VPN customer data, and they apply just as readily to how we protect the items in a Keys user's encrypted **vault**:

ExpressVPN's four key strategies

1. MAKE SYSTEMS DIFFICULT TO COMPROMISE

The front line in our defenses is making our systems **secure**. We employ many different techniques—from cutting-edge encryption to hardware security devices—to ensure that it's difficult to break into any of them.



2. MINIMIZE POTENTIAL DAMAGE

Despite our efforts, it is still possible that a motivated attacker may break through our defenses. We address this risk by applying **guardrails** to minimize the damage a potential attacker can do from their initial foothold.

3. MINIMIZE THE TIME OF COMPROMISE

In addition to minimizing the severity of potential damage, our processes also help to limit the **duration** of compromise and the amount of time that attackers can stay lurking.

4. VALIDATE OUR SECURITY CONTROLS

All of our software and services are rigorously **tested** to ensure they work as intended and meet the high standards of privacy and security that we promise to our customers.

For examples of how these principles are implemented across all ExpressVPN products, visit the ExpressVPN Trust Center.

In designing a new password manager from the ground up, however, we also employed a few additional principles:

Kerckhoffs's principle

The renowned mathematician Claude Shannon once said, "One ought to design systems under the assumption that the enemy will immediately gain full familiarity with them" (Shannon, 1949).

In cryptography, this is known as **Kerckhoffs's principle**: the idea that a system should be secure even if adversaries know everything about the system—as long as they don't know the secret key (Kerckhoffs, 1883). In the context of ExpressVPN Keys, this means your stored items are secure even if a persistent adversary has gained complete knowledge of our cryptosystem—as long as that adversary doesn't have your primary password.

In other words, we don't rely on "security through obscurity." We want our cryptosystem to be fully transparent so that it can be studied, tested, and improved; in fact, that is why we are writing this white paper!

Your secret stays with you

When you first set up ExpressVPN Keys, the app guides you to set a strong **primary password**. Because the security of all your stored items hinges on the secrecy of that primary password, we don't want anyone but you to have it, not even us.

To minimize risk, we designed a client-server protocol that ensures that all computations requiring your primary password are done in your local system and *never leave your device*. That's because sending your primary password in any form (even encrypted) increases the risk of compromise. We want your primary password to remain secure even in the unlikely event that network attackers are harvesting your traffic with the goal of decrypting it later. Therefore we will never require you to send your primary password in any form.

See our section on zero-knowledge encryption for more detail.

Good software design

In designing Keys, we made several decisions at the very start that allowed developers to focus on security without adding unnecessary complexity or reinventing the wheel:

- First, we wrote a self-contained **core library** in Rust as the base component, which allows for a well-defined interface with compatibility across many platforms in a type-safe and memory-safe way.

- Next, we used the Concise Binary Object Representation (CBOR) for **serialization** and **deserialization** of data in our communications, reducing risk of dangerous deserialization attacks that are common in other frameworks.

- Finally, we made a conscious choice to pick standard, well-tested cryptographic **primitives**. In particular, we use the class of ciphers that allow for authenticated encryption with associated data: AEAD (Rogaway, 2002), which simultaneously ensures confidentiality, integrity, and authenticity.

Trust but verify

Even if we designed the perfect system, we would still need to verify its implementation. After all, bugs can potentially creep in if developers are not careful, edge cases may crop up without proper testing, and we must always be ready for issues in the software supply chain. To ensure ExpressVPN Keys lives up to the security guarantees described in its design:

- Our codebases go through a battery of internal **penetration testing**.

- We regularly commission independent, third-party **audits** of our apps.

- We remain open to security research by the public, and anyone can report security issues through our **bug bounty program**.

Now, let's dive into how Keys actually works.

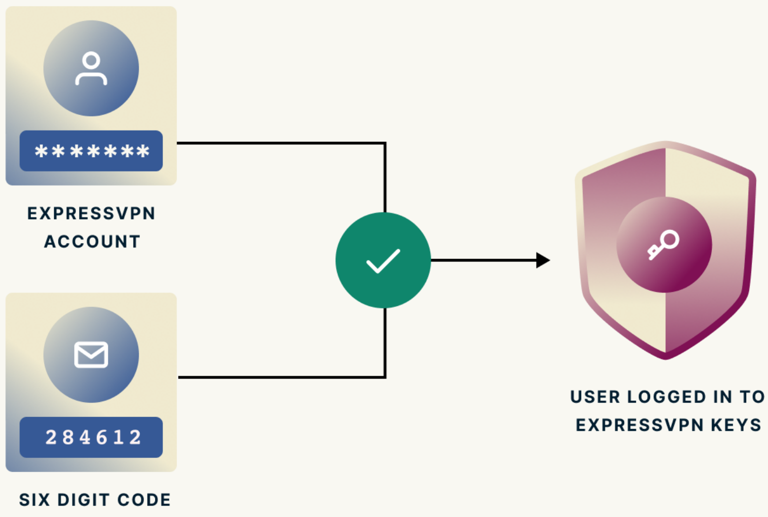
3. Authentication

When you sign in to ExpressVPN Keys, various cryptographic and network operations occur in the background. Here we'll walk you through how your communications to the server are protected by multiple authentication mechanisms, both through standard cryptographic protocols and through additional protections implemented at the application layer. We'll illustrate how these mechanisms protect your traffic and your account from network attackers who might be passively listening or attempting to modify traffic to our servers.

Step-up authentication

Keys accounts are tied to ExpressVPN accounts, so before you can create or access your Keys account, we need to confirm that you are the owner of the ExpressVPN account that is signing up for Keys.

To perform this authentication, the first step is to validate your ExpressVPN account credentials. Then we perform another round of authentication using a second factor to ensure that the owner of the account is the same person requesting access to Keys[2]. This **step-up authentication** takes the form of a six-digit code sent to the primary email address associated with the ExpressVPN account. This code must be entered into the app, after which the user is authenticated to ExpressVPN Keys, and an authentication token (in the form of a JSON Web Token (JWT) signed with RS256[3]) is returned. This token contains an additional scope allowing it access to ExpressVPN Keys, but it will still *not* be able to access the passwords in the vault, since the user has not provided the secret primary password (more on this in the next section).



[2] Note that the above process does not apply to iOS users who have already completed Apple's 2FA process.
[3] This is the recommended "`alg`" setting for JSON Web Tokens according to RFC 7518. We follow current best practices in authenticating tokens as per RFC 8725.

Network security

As you use the password manager (including the authentication described above), layers of authentication and authorization protect all API requests made by ExpressVPN Keys. These layers are intended to make it nearly impossible for an attacker to gain access to, download, and/or tamper with your stored items.

First, all network communication to ExpressVPN API servers are protected through TLS[4], and HTTPS is used in all communications. No HTTP endpoint—which would otherwise be in plaintext—is exposed. A security policy is configured and enforced at the gateway for the negotiation of appropriate cipher suites. Additionally, TLSv1.3 is supported, so cipher suites providing PFS (perfect forward secrecy) are supported and used if the client supports it. This means traffic sent to our server is much harder to compromise, since unique keys are generated, used, and rotated throughout. On top of this, TLS also provides the standard security guarantees for your network traffic with us:

a. **encrypting** your traffic so network attackers cannot access the plaintext

b. **calculating** a message authentication code to prevent modification to your communications with us, and

c. **verifying** that you are indeed talking to the real ExpressVPN API servers

Backing the secure communications is a public certificate tying the ExpressVPN API domain name to the server. This public certificate contains a 2048-bit RSA public key[5] and is signed by a trusted CA (Certificate Authority), with the certificates themselves rotated automatically every 3 months (an example of minimizing the time of compromise).

Next, we implement another layer of authentication at the application layer, built around a 4096-bit RSA key pair known as the user's *identity* that is generated when the account is created. The RSA key pair consists of two parts:

1. The **private key** (also called the **Private Identity Key**). This is used to produce a cryptographic signature to prove to the server that the user is the one making the request. More precisely, this key is used to sign the payload/contents of an API request. The private key is kept secret through the secret primary password provided by the user and this part of the keypair never leaves the device in plaintext.

2. The **public key** (also called the **Public Identity Key**). This is used to verify the contents of the API request and establish that the user making the API request has access to the Private Key. As suggested by the name, this part of the key pair is meant to be public and can be used by anyone to validate a signature produced by the private key. The

[4] We follow the guidelines presented in SP 800-52 Rev. 2, Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations
[5] Barker, E., & Dang, Q. (2015, January). NIST SP 800-57 Part 3 Rev. 1, Recommendation for Key Management, recommends 2048-bit RSA keys to be used for Public Key Infrastructure (PKI).

ExpressVPN Keys server stores the public key and uses it to validate the signature on any API request.

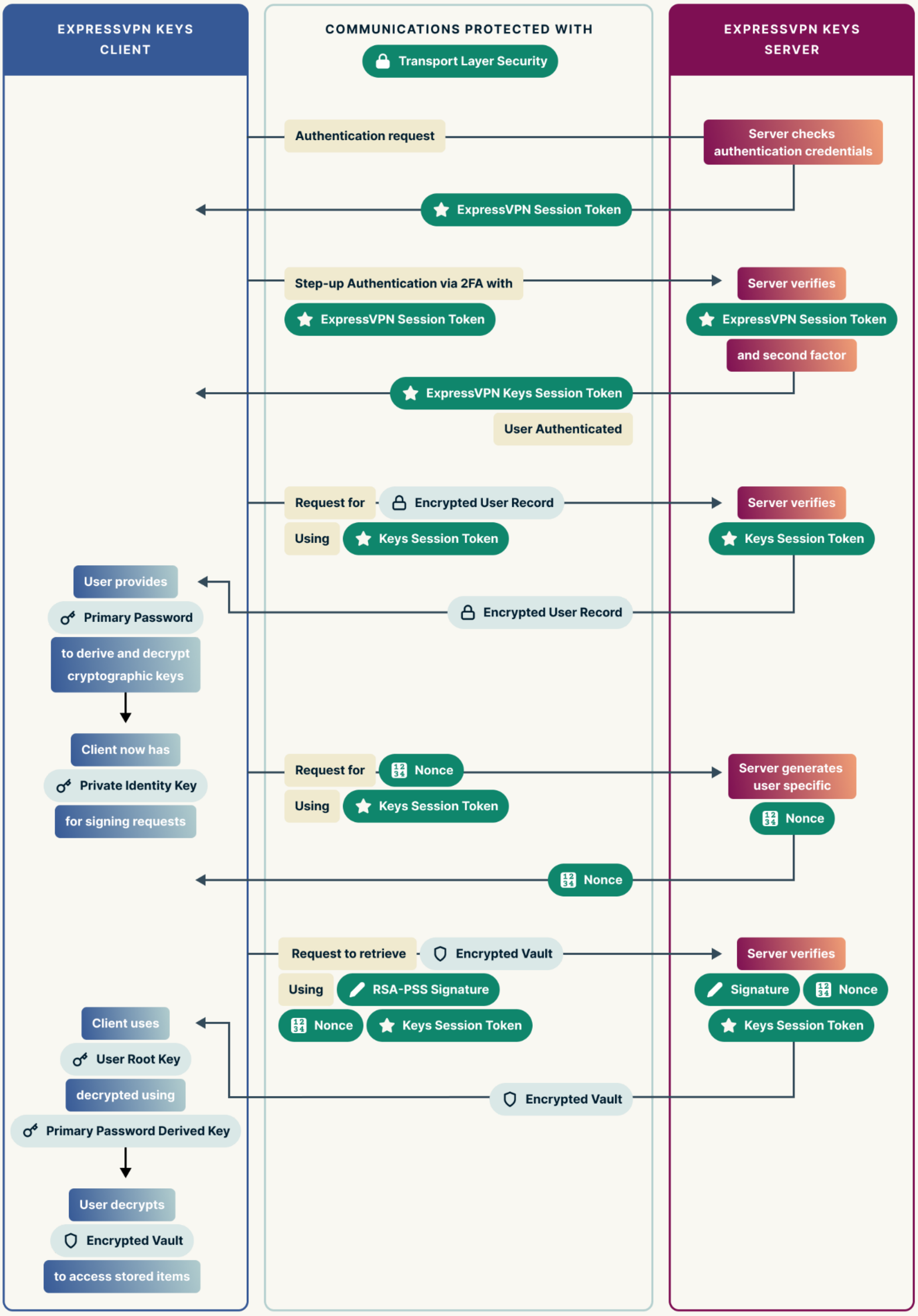
A word about “keys” vs. “Keys”

For the purposes of this paper, references to cryptographic keys will be uncapitalized except when named in a specific, singular key such as the **Private Identity Key**. Wherever “Keys” is capitalized and plural, you can safely assume we’re talking about the password manager itself.

Once the client establishes communication with the ExpressVPN Keys server, the first and only API endpoint used by the client that does *not* require a payload signed with the user identity is the API endpoint used to fetch the encrypted user record[6]. After the user decrypts the encrypted user record using the secret primary password, all further communications to the API server (including those that fetch the user’s stored items, and those which modify them) are protected by this additional layer of authentication. Specifically, ExpressVPN Keys uses RSA-PSS (Bellare & Rogaway, 1998) with the SHA512 hash algorithm and a 512-bit salt to sign each request with the private key. An attacker without the private key is unable to sign requests on behalf of the user and would therefore have those forged requests rejected by the ExpressVPN Keys API server.

As an extra layer of protection, we include an additional single-use signed cryptographic **nonce** within the payloads sent to the server. This nonce is a randomly[7] generated 64-bit unsigned integer[8] tied to a specific user and is generated server-side when requested by a user. This nonce is only valid for a single use or for 5 minutes, whichever comes first. The single-use property prevents **replay attacks**, where in the very unlikely event that a passive adversary manages to break TLS to observe, collect, and resend your traffic to our servers, then our systems know to reject the replayed request because the nonce has already been used. The latter property ensures that the request is *timely*—if a user requested a token from the server but did not use it within 5 minutes, an attacker will not be able to abuse the fact that the token was stored unused for some period of time.

[6] This avoids a “chicken and egg” situation where we need the private key to sign the request, but we don’t yet have it since it’s tied to an encrypted user record stored on the server.
[7] We use a cryptographically secure random number generator currently based on the ChaCha12 block cipher (see StdRng for more information).
[8] A 64-bit integer allows for approximately 18 quintillion possible unique numbers, which in more countable terms is about 18,000,000,000,000 million. The likelihood that there is a collision with existing nonces from the same user, assuming the user has 10 devices, can be approximated with the equation *n*2/2*m* (where *n* is the number of existing per-user nonce, and *m* is the range for a nonce) or more concretely 102 / (2×264) which is 0.000…00027 (with 18 zeros), which means it is *effectively impossible* for an attacker to replay a request. See the Birthday Problem for a theoretical explanation.


EXPRESSVPN KEYS CLIENT
COMMUNICATIONS PROTECTED WITH
Transport Layer Security
EXPRESSVPN KEYS SERVER
Authentication request
Server checks authentication credentials
ExpressVPN Session Token
Step-up Authentication via 2FA with
ExpressVPN Session Token
Server verifies
ExpressVPN Session Token
and second factor
ExpressVPN Keys Session Token
User Authenticated
Request for
Encrypted User Record
Using
Keys Session Token
Server verifies
Keys Session Token
User provides
Primary Password
to derive and decrypt cryptographic keys
Encrypted User Record
Client now has
Private Identity Key
for signing requests
Request for
Nonce
Using
Keys Session Token
Server generates user specific
Nonce
Nonce
Request to retrieve
Encrypted Vault
Using
RSA-PSS Signature
Nonce
Keys Session Token
Server verifies
Signature
Nonce
Keys Session Token
Client uses
User Root Key
decrypted using
Primary Password Derived Key
Encrypted Vault
User decrypts
Encrypted Vault
to access stored items

Even after authentication, however, the user's vault is *not yet accessible until after a primary password is provided*. In the next section, we will elaborate on how the vault works. Specifically, we will detail the operations that ExpressVPN Keys performs locally on your device using the secret primary password.

4. Zero-knowledge encryption

Zero-knowledge encryption allows you to store items in Keys in a way that only you—the person with the secret primary password—can access. When you store items in your vault, your primary password and any other plaintext information never leaves your device. Therefore, ExpressVPN Keys and its servers have *no way* of accessing this information in plaintext. Even if a sophisticated attacker were to somehow get access to our systems, all information is encrypted—no plaintext information corresponding to your vault is ever stored in the ExpressVPN Keys database.

To understand how this works, let's take a look at what happens when you first create your account with your primary password and how we derive the necessary cryptographic ingredients from that primary password.

Account creation

In ExpressVPN Keys, the onboarding process is relatively simple. After step-up authentication, you simply have to provide your preferred primary password, and all cryptographic keys will be generated locally on your device in the background.

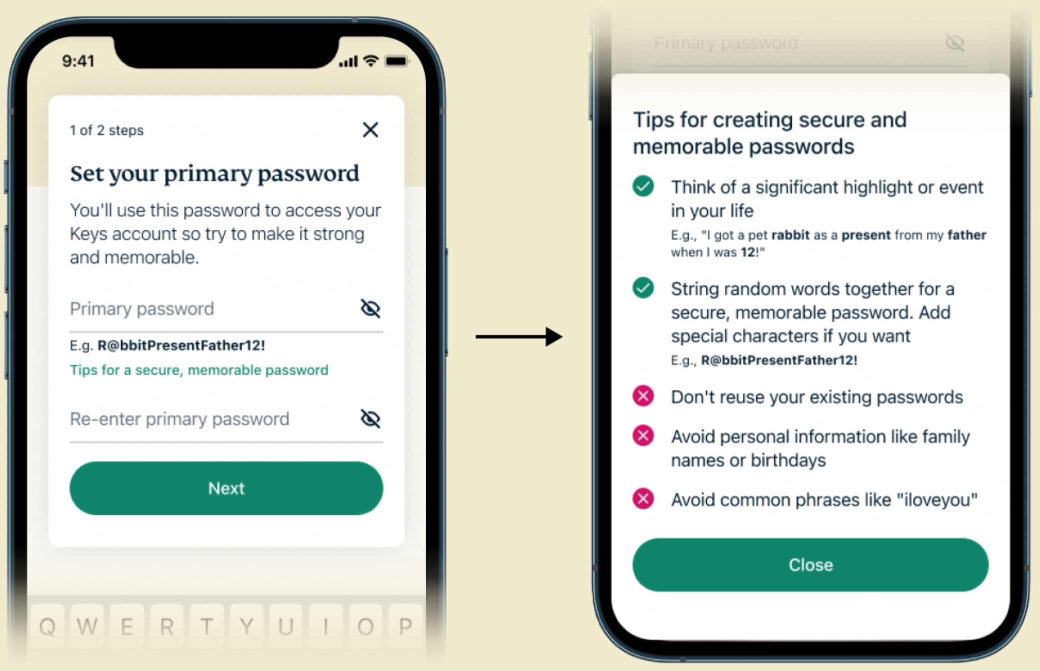
A word about your primary password

This password is the key to your entire vault, therefore it is *critically important* that it is neither leaked nor guessable by an attacker.

To help you choose a strong, **unguessable** password, ExpressVPN Keys evaluates your proposed primary password using the industry standard zxcvbn password strength estimator (Wheeler, 2016). zxcvbn measures the guessability of your password without relying on outdated password rules around the number of capital letters and special symbols[9]. If your password is considered weak then you will be warned, and if it is too weak then you will not be able to proceed.

[9] Length and complexity requirements are counterproductive as users tend to work around them in predictable ways, and should no longer imposed as requirements beyond the guidance in NIST Special Publication 800-63B, Digital Identity Guidelines: Authentication and Lifecycle Management

Your primary password should also be **memorable** enough that you won't be tempted to copy it somewhere else in plaintext (or in plain sight, such as by sticking a note on your computer). That's why we also suggest using "random" words with personal meaning (but not guessable information like family names or birthdays).


9:41

1 of 2 steps

Set your primary password

You'll use this password to access your Keys account so try to make it strong and memorable.

Primary password

E.g. **R@bbitPresentFather12!**

Tips for a secure, memorable password

Re-enter primary password

Next

Q W E R T Y U I O P

Primary password

Tips for creating secure and memorable passwords

- ✓ Think of a significant highlight or event in your life
  E.g., "I got a pet **rabbit** as a **present** from my **father** when I was **12**!"
- ✓ String random words together for a secure, memorable password. Add special characters if you want
  E.g., **R@bbitPresentFather12!**
- ✗ Don't reuse your existing passwords
- ✗ Avoid personal information like family names or birthdays
- ✗ Avoid common phrases like "iloveyou"

Close

Primary Password Derived Key (ppdk)

Using a key derivation function called PBKDF2, we transform your primary password into a cryptographic key called the **Primary Password Derived Key** (*ppdk*):

$$ppdk := PBKDF2_{SHA512}(primary\ password,\ salt,\ 210000,\ 32)$$

The PBKDF2 function uses a SHA512 HMAC as the pseudorandom function, your primary password, a 32-byte salt[10], and runs the key derivation function for 210,000 iterations[11] before returning a 32-byte key: the *ppdk*.

[10] Note that the salt is uniquely generated for every user and uses OpenSSL's RAND_bytes function or the Web Crypto API's getRandomValues function, depending on the platform. This unique salt prevents the use of rainbow tables to precompute the output from a hash function, significantly increasing the computation required for an attacker to recover the original primary password.
[11] This is based on recommendations from OWASP.

The *ppdk* never leaves your device; in fact it is never stored anywhere and exists only briefly as part of the vault unlock process. We use the *ppdk* to encrypt downstream cryptographic key material and, since only you know the primary password from which it is generated, attackers without knowledge of your primary password won't be able to decrypt it and access your stored items.

Recovery Code Derived Key (rcdk)

We also generate a **recovery code** locally on your device when you first sign up for Keys. The recovery code (which we'll call *k* below) is a 15-byte string (giving 120-bits of security[12]) generated from a cryptographically secure pseudorandom generator implemented in OpenSSL or the Web Crypto API, depending on the platform.

We encode the 15-byte string into printable characters (alphanumerics) via Base32 encoding (e.g., *AAAB-TP5X-3F7A-XTUA-4DA4-YA4Q*), so users will be able write down the characters of the recovery key and store it in a secure place. The pseudocode for the recovery code generation algorithm is illustrated below:

Unset
```
function generate_recovery_code():
    r := openssl_RAND_bytes(15)
    k := BASE32_NOPAD.encode(r)
    return format("{}-{}-{}-{}-{}-{}", key[0:4], key[4:8], key[8:12],
key[12:16], key[16:20], key[20:24])
```

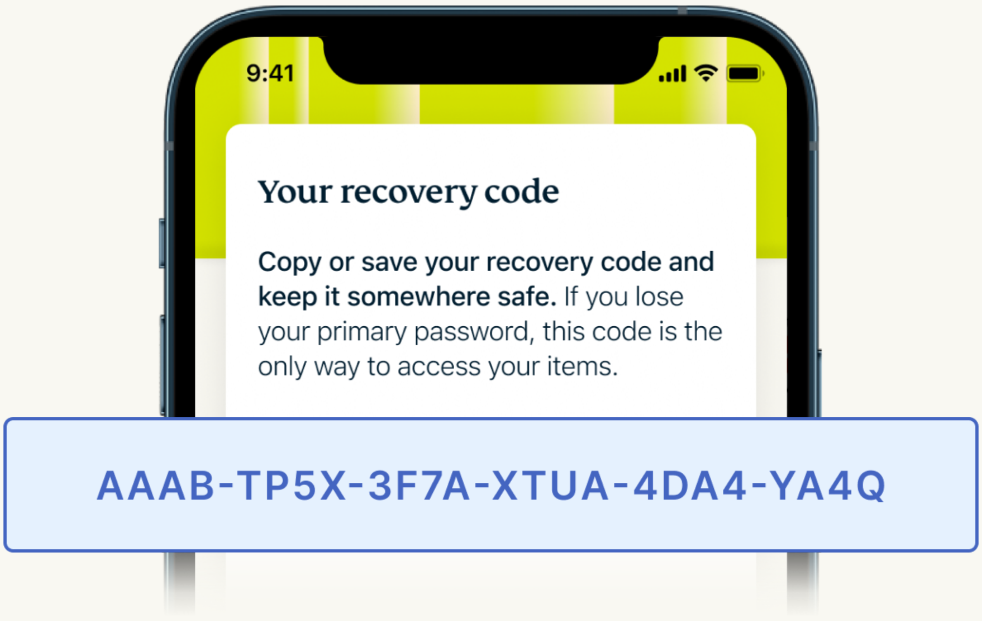
In precisely the same way as described above for your primary password, your recovery code is used to generate a cryptographic key: the **Recovery Code Derived Key** (*rcdk*) using the same key derivation function and parameters. Your *rcdk* can only be generated by you and never leaves the device, so attackers without knowledge of your recovery code won't be able to decrypt downstream cryptographic key material and access your stored items.

[12] SP 800-131A Rev. 2 - Transitioning the Use of Cryptographic Algorithms and Key Lengths recommends a key size of at least 112 bits of security.

A word about your recovery code

It is very important to keep this recovery code in a safe location. If you ever forget your primary password, this recovery code will be the only way to recover access to the vault and change your primary password.

ExpressVPN Keys has no knowledge of your primary password or your recovery code, so if you lose both, your vault will be permanently encrypted and ***all your data will be effectively lost***.



Your recovery code

Copy or save your recovery code and keep it somewhere safe. If you lose your primary password, this code is the only way to access your items.

AAAB-TP5X-3F7A-XTUA-4DA4-YA4Q

The *ppdk* and *rcdk* both have the ability to unlock your vault. In order to allow two different keys to have access to the same information, we use something called a *User Root Key*.

User Root Key, User Document Key, and Encrypted User Record

The *User Root Key* is a 256-bit, randomly generated AES key used to protect the cryptographic key that actually encrypts and decrypts your stored items. That key, known as the *User Document Key*, is also a 256-bit, randomly generated AES key. This two-tier approach enables users to change their primary password without ever having to decrypt the entire vault and re-encrypt using a new *User Document Key*. What happens in the background is that the *User Root Key* is simply re-encrypted with the new password.

This also enables changing the recovery code if needed: the app can present the user with a new recovery code and re-encrypt the *User Root Key* to be re-encrypted using the newly derived key.

For an additional layer of security, if the user changes both the primary password and the recovery code at the same time, then we also opportunistically rotate the *User Root Key* and encrypt the new one with both keys.

The *Encrypted User Record* is first generated locally using the following information:

```
Unset

EncryptedUserRecord {
    primary_password_key: EncryptedUserRootKey,
    recovery_code: EncryptedUserRootKey,
    identity: PublicKey
    body: EncryptedUserRecordBody
}
```

The *identity* above is the same *identity* used for authentication in the previous chapter. The *primary_password_key* is an encrypted vector that decrypts to the *User Root Key*.

Suppose the following function prototype for AES-256-GCM[13]:

```
AES-256-GCM_encrypt(data_to_encrypt, key, IV)
```

The *primary_password_key* and *recovery_code* are defined as follows, using a random 12 byte initialization vector (IV)[14].

[13] The Advanced Encryption Standard (AES) is a symmetric key block cipher that splits input data into 128-bit padded blocks and encrypts them with various keys derived from the user provided key. We use 256-bit randomly generated keys, which gives the maximum possible strength, and we apply the Galois/Counter Mode (GCM) allowing for authenticated encryption with associated data. This means that an authentication tag is generated, allowing for detection if the ciphertext is ever modified. We use OpenSSL's implementation of AES-GCM in ExpressVPN Keys.
[14] NIST SP 800-38D - Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC allows the use of an RBG-based construction of IVs as long the randomness is sufficient and the number of encryption invocations do not exceed 2^{32} invocations.

```
user_root_key[15] := rand_bytes[16](32)
primary_password_key := AES-256-GCM_encrypt(user_root_key, ppdk, RAND_bytes(12))
recovery_code := AES-256-GCM_encrypt(user_root_key, rcdk, rand_bytes(12))
```

We then use the user_root_key and encrypt the *UserRecordBody* using the below operation:

```
body := AES-256-GCM_encrypt(UserRecordBody, user_root_key, rand_bytes(12))
```

The body stored in the `EncryptedUserRecord` is the encrypted version of the UserRecordBody data structure below:

```
Unset
UserRecordBody {
    document_key: UserDocumentKey
    user_private_key: PrivateKey
}
```

where `document_key` is a *User Document Key* that is generated **locally** as below:

```
document_key[17] := rand_bytes(32)
```

This *User Document Key* is a 256-bit AES key that is used to encrypt items (i.e., your passwords, credit/debit cards, secure notes, associated metadata, or any other information that may be stored in your vault). This key is *extremely important*—any access to this key enables access to your passwords and other data in the vault. As noted above, the *User Document Key* is secured and encrypted with the *User Root Key*, which in turn is further secured by a key derived from your primary password (*ppdk*) or from your recovery code (*rcdk*).

Lastly, in the *User Record*, the `user_private_key` (or the *Private Identity Key*) is the RSA private key associated with the identity seen in the `EncryptedUserRecord`, and will be used to sign messages to the API servers (see previous chapter), and is secured in a similar manner. In the end, when your ExpressVPN Keys account is created, only the *Encrypted User Record* leaves your device, where it is transmitted and stored on the ExpressVPN Keys servers.

[15] The *User Root Key* is never stored on the filesystem in plaintext and is decrypted as needed to access the *User Document Key* to read the contents of your vault.
[16] This is Web Crypto API's getRandomValues function on our browser extension, and OpenSSL's RAND_bytes on other platforms.
[17] The *User Document Key* is never stored on the filesystem in plaintext and is decrypted as needed to access the contents of your vault.

The above process may seem complex, but it is the foundation for the security of ExpressVPN Keys. Because the *User Document Key* and the *User Root Key* never leave your device unencrypted, no one at ExpressVPN, nor any attacker who may compromise any infrastructure at ExpressVPN, has access to the keys needed to decrypt your stored items or the *Encrypted User Record*. This is the core benefit of zero-knowledge encryption: *only you*, with your primary password or recovery code, have the power to decrypt your stored items.

A day in the life of a password

To explore how this all works in the context of your passwords, let's look at the life of a typical item you might store in Keys: a single login (username and password).

Storage

This is what happens when you **store** a login in Keys:

1. You sign in to ExpressVPN Keys with step-up authentication.

2. The *Encrypted User Record* is retrieved from the ExpressVPN Keys server.

3. You provide the *primary password*[18] so the *ppdk* can be generated locally.

4. The *ppdk* is used to decrypt the encrypted *User Root Key* in the *Encrypted User Record.*

5. The *User Root Key* is used to decrypt the encrypted *User Document Key* and *Private Identity Key* in the *Encrypted User Record Body*.

6. The metadata of the login to be stored (e.g., the username and the domain where the login should be filled) is first encrypted[19] using AES-256-GCM and the *User Document Key* and stored in the *metadata* field of the document record *d*.

7. The actual *password* (or the body of the document) is then encrypted[20] using AES-256-GCM and the *User Document Key* and stored in the body field of the same document record *d*.

---

[18] The recovery code would also allow the user to retrieve the *User Document Key* and the primary password is used for the sake of illustration.
[19] While the original construction using AES-GCM is already IND-CPA secure (ciphertexts are sufficiently randomized given the same input and therefore IND-CDBA secure—meaning an read-only adversary cannot glean any meaningful information from the vault), the metadata is further padded by up to 128 bytes to prevent a length side channel from leaking the actual length of the metadata (or specifically, the domain), therefore it is nearly impossible to glean the actual domain from encrypted ciphertext of the metadata. We note that AES-GCM is an AEAD cipher and is resistant against a read-write adversary as per the MAL-CDBA game. See On the Security of Password Manager Database Formats (Gasti and Rasmussen, 2012) for more details.
[20] Similarly, AES-GCM with up to 128 bytes of additional padding makes it resistant to both read and read-write adversaries. The padding is especially important here since short passwords (up to a 16-byte resolution) may be targeted by adversaries.

8. An API call to /sync is made, where new encrypted document record(s) *d* are signed with the *Private Identity Key* using RSA-PSS and sent upstream to the server.

9. The server verifies the signed message using the *Public Identity Key* and stores the encrypted *document record* in the database.

Autofill

And this is what happens when, after you visit a website or app whose login you've stored in Keys, you want to **autofill** that login's username and password:

1. You sign in to ExpressVPN Keys via step-up authentication.

2. The *Encrypted User Record* is retrieved from the ExpressVPN Keys server.

3. You provide the *primary password* to the ExpressVPN Keys application so the *ppdk* can be generated locally.

4. The *ppdk* is used to decrypt the encrypted *User Root Key* in the *Encrypted User Record*.

5. The *User Root Key* is used to decrypt the encrypted *User Document Key* and *Private Identity Key* in the *Encrypted User Record Body*.

6. An API call to `/documents` is made, signed with the *Private Identity Key* using RSA-PSS, and the encrypted documents are retrieved from the server.

7. The server verifies the signed message using the *Public Identity Key* and returns the list of documents stored for the particular user.

8. The *metadata* of the documents are decrypted using the *User Document Key*, allowing Keys to match against domains with existing password entries.

9. Only when requested[21], the *body* of the document ***d*** is decrypted with the *User Document Key*, revealing the password stored for the domain.

10. ExpressVPN Keys autofills the password on the domain, as instructed.

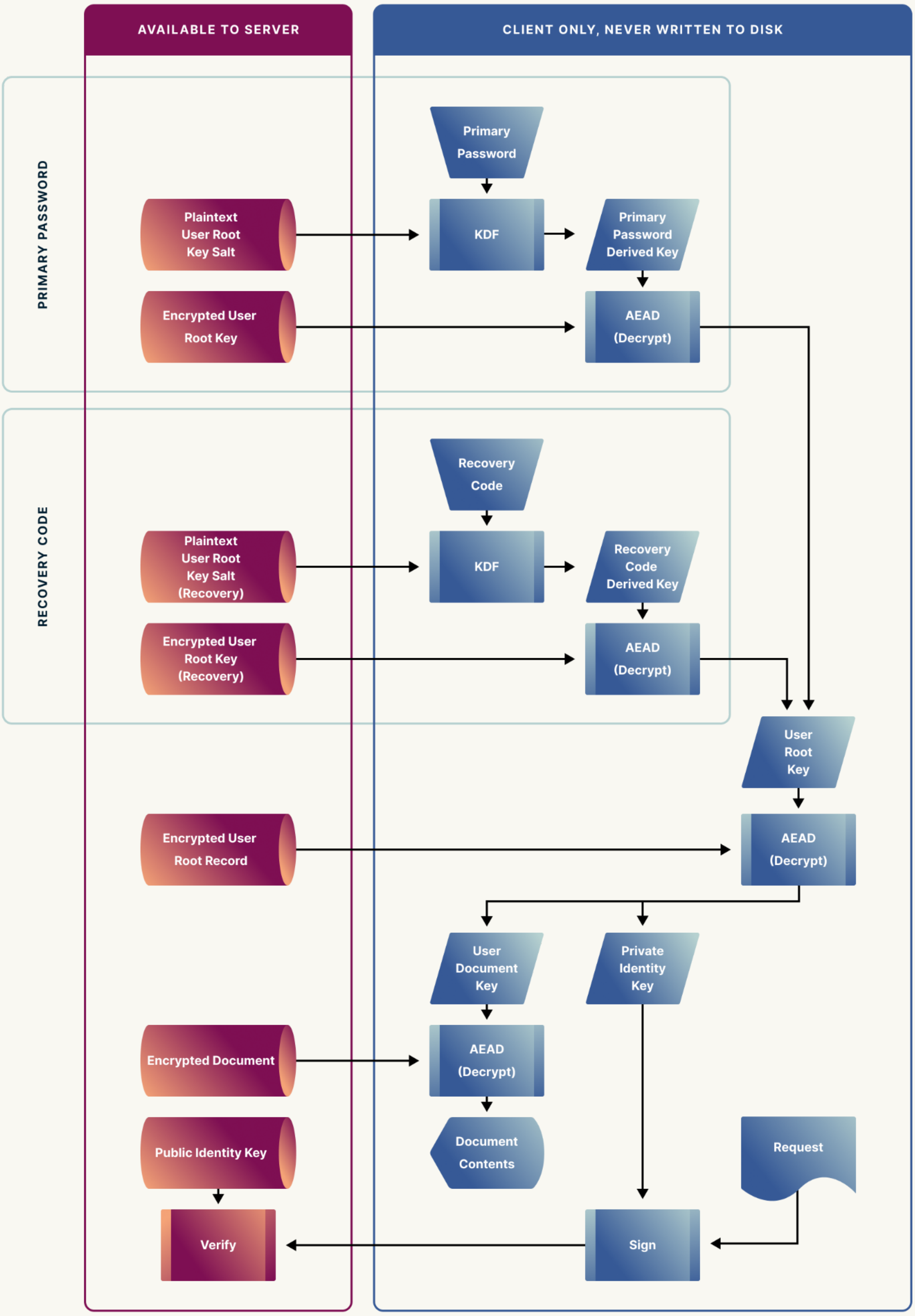
Most of this happens in the background, of course. As the user, you experience this process in a matter of moments as you sign in to Keys, visit a site or app, and witness your stored username and password conveniently autofilled into the login screen.

[21] See the section on Autofill security for more on when we allow this.

Summary of cryptographic keys

To help you keep track of all the “keys” used in Keys, here’s a summary of cryptographic keys in table and diagram forms:

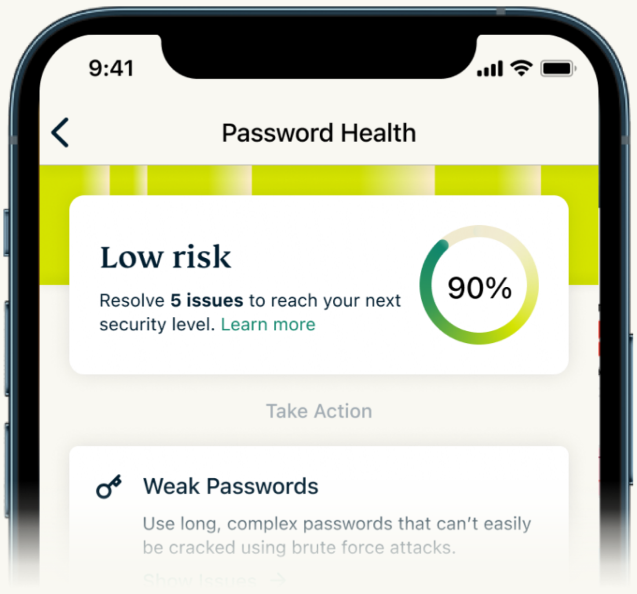
Name	Algorithm	Purpose	Stored Server-side?	Encrypted With?
Primary Password Derived Key (ppdk)	Generated via PBKDF2 with SHA512 HMAC, 32 byte salt, 200000 iterations Used for AES-256-GCM Key size: 256-bits	AES key derived using the user provided ***primary password***, used to encrypt/decrypt the *User Root Key*	NO	N/A
Recovery Code Derived Key (rcdk)	Generated via PBKDF2 with SHA512 HMAC, 32 byte salt, 200000 iterations Used for AES-256-GCM Key size: 256-bits	AES key derived using the randomly generated ***recovery code***, used to encrypt/decrypt the *User Root Key*	NO	N/A
User Root Key	Generated ***randomly*** using a CSPRNG Used for AES-256-GCM Key size: 256-bits	Used to encrypt/decrypt the Encrypted User Record containing the User Document Key and Private Identity Key	YES, encrypted	Encrypted using the *ppdk* and the *rcdk*
User Document Key	Generated ***randomly*** using a CSPRNG Used for AES-256-GCM Key size: 256-bits	Used to encrypt/decrypt *documen*t metadata and body	YES, encrypted	Encrypted using the *User Root Key*
Public Identity Key	Generated ***randomly*** using a CSPRNG Used for RSA-PSS verification Key size: 4096-bits	Used by the server to verify API requests to the Password Manager is from a legitimate user with access to the vault	YES	N/A
Private Identity Key	Generated ***randomly*** using a CSPRNG Used for RSA-PSS signing Key size: 4096-bits	Used by the client to sign API requests to the Password Manager server to authenticate as a user with access to the vault	YES, encrypted	Encrypted using the *User Root Key*


AVAILABLE TO SERVER
CLIENT ONLY, NEVER WRITTEN TO DISK
PRIMARY PASSWORD
RECOVERY CODE
Primary Password
Plaintext User Root Key Salt
KDF
Primary Password Derived Key
Encrypted User Root Key
AEAD (Decrypt)
Recovery Code
Plaintext User Root Key Salt (Recovery)
KDF
Recovery Code Derived Key
Encrypted User Root Key (Recovery)
AEAD (Decrypt)
User Root Key
Encrypted User Root Record
AEAD (Decrypt)
User Document Key
Private Identity Key
Encrypted Document
AEAD (Decrypt)
Document Contents
Public Identity Key
Request
Verify
Sign

5. Password Health

Beyond storing and retrieving your logins, ExpressVPN Keys also helps you monitor and improve their security with a feature we call **Password Health**.

Password Health gives a percentage score for the "health" of your vault. This score takes into consideration several factors for each login stored in your vault. The score is calculated locally on your device without sending any passwords (or any other element of your stored login) to any server.



9:41
Password Health
Low risk
Resolve 5 issues to reach your next security level. Learn more
90%
Take Action
Weak Passwords
Use long, complex passwords that can't easily be cracked using brute force attacks.

In order of decreasing importance, the factors for each login included in the health score are:

1. *Has the password been **exposed** in a known data breach?*[22]
   ExpressVPN Keys lets you know whether your passwords have been exposed in data breaches aggregated by HaveIBeenPwned. Your passwords are never shared with ExpressVPN, HaveIBeenPwned, or any external parties during this process, as we'll explain in Section 5.1. If you do not wish to check for exposed passwords, you can always disable this feature in the app settings.

2. *Is the password used for **more than one** login in your vault?*
   Reusing passwords puts you at greater risk from data breaches. If any one of the sites are breached then the attacker potentially gains access to all of the sites where the same password was reused. ExpressVPN Keys checks whether your passwords are used by more than one login in a privacy-preserving way, ensuring your passwords are

[22] This feature is currently available on iOS and Android, and coming soon to the browser extension.

not decrypted and loaded into memory unless absolutely necessary (i.e., if we find that two hash prefixes match and must briefly decrypt to check for an exact match).

3. *How **strong** is the password?*
   ExpressVPN Keys uses the industry-standard zxcvbn to provide a measure of a password's resistance to guessing or brute-force attacks. A password is considered weak if it could be easily crackable in a matter of hours or days.

4. *Does the login contain a **2FA** code (if the domain is known to support it)?*
   ExpressVPN Keys uses data provided by the 2FA directory to determine if the domain of the login supports 2FA via TOTP, a time-based one-time password algorithm. The check is performed locally using a built-in database, so no request is needed to any server. 2FA improves your security by providing each account with a constantly changing one-time code. If a code is leaked then, unlike a password, it will become useless shortly afterwards. Therefore, for sites that support TOTPs, ExpressVPN Keys will encourage you to configure it. For sites not covered by the 2FA directory, you can still configure a TOTP.

5. *Does the login contain a domain that uses **HTTP**?*
   Connections made using HTTP (as opposed to HTTPS) are not encrypted, meaning your carefully generated secure password would be sent to the site in plaintext, vulnerable to interception or observation by an attacker.

If a login falls short on any of the above factors, Keys will guide you through updating the relevant factor to improve the security of that login, and the overall Password Health of your vault.

Data breach detection

ExpressVPN Keys uses HaveIBeenPwned (HIBP) to provide monitoring and alerting of whether your passwords have ever been compromised by a data breach.

The HIBP APIs are carefully designed using k-anonymity to ensure that your data is never shared directly with the provider of the service and that even an attacker observing the traffic to those services does not gain useful insights into the data.

This is achieved by dividing the results into buckets of at least *k* entries. This ensures that the response associated with your password or email is indistinguishable from at least *k*-1 other entries. The buckets have been defined by HIBP such that *k* is always at least several hundred responses.

HIBP achieves this by taking an SHA1 hash of the relevant data (either a password or an email address) and requesting the buckets of results corresponding to the first *N* hexadecimal characters of that hash. For passwords, *N* is 5 (meaning there are 1,048,576 buckets) and for emails, *N* is 6 (corresponding to 16,777,216 buckets). Both values of *N* were chosen to ensure that each bucket contains hundreds of results, typically on the order of 800 results. The results then include all the known hash suffixes (the remaining 40-*N* hexadecimal characters) and the corresponding leak or breach information.

Let’s say one of your stored passwords is “Passw0rd”. The SHA1 of the string “Passw0rd” is

EBFC7910077770C8340F63CD2DCA2AC1F120444F

Taking the first five characters, we get the range EBFC7. We request that range from HIBP and, at time of writing[23], we received a response with 823 results[24]. Within those results we search for the hash and find a single result:

910077770C8340F63CD2DCA2AC1F120444F:**93254**

Indicating that the password “Passw0rd” has been seen in data breaches 93,254 times. Note that any attacker who might observe the request for range EBFC7 would not be able to distinguish this from the 822 other results in that bucket.

If instead we use a randomly generated password like “!zH%g7\"B/0DLhcl5”, which hashes to:

0C15D86EBD64CCD8A73AAD731B575C69F942943A

Then we look up the range 0C15D and we do not find the suffix 86EBD...2943A in the 849 results, indicating that “!zH%g7\"B/0DLhcl5” has never been seen in a data breach.

To allow ExpressVPN Keys to make these checks without requiring that the actual password is decrypted unless absolutely necessary, the 5-character prefix is stored in document metadata so as to be available when the vault is unlocked (unlike the password itself, which remains encrypted until specifically required).

In addition to passwords, Keys can also help you check whether your email address has been found in a data breach through HIBP[25]. The process is the same as above except HIBP uses the first 6 characters as the range. For example, the email address “expressvpnkeys-user@example.com” hashes to:

[23] HIBP frequently ingests new data breaches from a variety of sources, so this number will grow.
[24] For the purposes of this example, we have not requested padding from the HIBP API. ExpressVPN Keys always requests padding, which helps obscure the request even from attackers who can observe the TLS encrypted request by preventing them from making estimations based on the sizes of responses.
[25] Note that breach checks using an email address are performed only when requested by the user, while breach checks for stored items in Keys are performed continuously as part of Password Health.

B103C9FF83BBF97F00BF5ED14AF44813E60AAAD9

Then we look up the range B103C9 and check for the suffix FF83B...AAAD9.

The k-anonymity design of the HIBP APIs mean that an attacker does not learn anything about the data we are requesting. This same property extends to the operators of the HIBP service themselves: we don't have to trust them at all even though they can see all the requests being made.

ExpressVPN Keys takes a further step and provides additional privacy guarantees to prevent any association of the request to your account or IP address. Instead of making requests directly to the HIBP servers, ExpressVPN Keys instead makes requests through a proxy that we run for this purpose. HIBP never sees the real IP address of an ExpressVPN Keys user; instead they see an IP address associated with ExpressVPN Keys infrastructure. The result is that HIBP is not aware that specifically *you* were interested in ranges EBFC7 and 0C15D (each containing 800+ results) but only that *some* ExpressVPN Keys users were interested in those ranges. This adds an additional layer of anonymity to the requests made by you and other ExpressVPN Keys users.

ExpressVPN Keys takes one final step to preserve the anonymity of requests for commonly used passwords. For example, the password "password" is found in bucket 5BAA6[26] 9,659,365 times. The other 814 results in that bucket have only been seen 16,160 times between them. This means that an attacker who observes a request for bucket 5BAA6 can guess, with a reasonable certainty of success, that the password behind the request is "password"[27].

To guard against this, ExpressVPN Keys embeds a corpus of the top 25,000 most reused passwords from the HIBP database locally, so that it can warn users that the passwords have been leaked *without* having to make any external API requests.

[26] "password"'s SHA-1 is 5BAA61E4C9B93F3F0682250B6CF8331B7EE68FD8
[27] This is why ExpressVPN Keys encourages the use of strong, randomly generated passwords. Even if they happen to hash to bucket 5BAA6, the attacker's guess would be wrong.

6. Supply chain security

Internally developed code

All components of the ExpressVPN application, including ExpressVPN Keys, are rigorously reviewed by both human developers and automated systems before being accepted and merged. These systems are used to enforce security properties all the way from the code that ships the application to that code included in the shipped ExpressVPN application. These systems are designed from the ground up to provide defense-in-depth, making it difficult for adversaries to ship malicious code and preventing the compromise of any single machine or developer from affecting the ExpressVPN Keys codebase or wider ExpressVPN codebase.

The first step in this process is **human review**. No code can be accepted into a release branch of ExpressVPN Keys unless it has been approved by a developer *other than the author*. When a pull request (PR) is marked as approved, a continuous integration (CI) bot then checks that the approver is not the author of any commit in that PR, and if a reviewer is indeed conflicted (i.e., authored some of the code) then the PR is not approved for merge and a second, non-conflicted reviewer must approve.

The next step is **automated checks**. All commits must be signed by a hardware security device (YubiKey) that is tied to a developer at ExpressVPN. Every PR and every commit is checked by a CI bot to ensure that this is the case. The presence of an unsigned commit, or one that is not signed by an approved YubiKey, will block the PR from merging. The YubiKeys used for this require a physical touch operation *and* a passphrase before they will sign anything. This means without possession of the physical key itself, even if a developer's machine were compromised, the private keys that are embedded into the YubiKey are not at risk of being stolen.

Finally, even after a code change makes it onto a release branch, the signatures created by the YubiKeys are still checked. Only CI builds performed on branches that consist *entirely* of suitably signed code produce any artifacts, therefore any shippable artifacts are very unlikely to be tainted with unsigned, malicious code.

Third-party code

As part of ExpressVPN Keys CI/CD systems, we also perform various checks to ensure the source code that we consume from third parties remains secure and up to date.

First, we regularly run tooling (`cargo deny`) against the ExpressVPN Keys codebase, which checks for new RustSec advisories issues against our third-party dependencies. This means that if a new issue is discovered, we can immediately start work on an update that corrects the vulnerability.

Second, we commit to our version control a manifest (`Cargo.lock`) that enumerates all of our third-party dependencies. This manifest includes a SHA256 checksum of the dependency, meaning that an attacker cannot substitute a compromised version of a dependency without getting a change committed directly to our version control system, which as we have shown above is a monumental task for an adversary. This means we always know the version of the code we are shipping and that it is not changing unexpectedly. To ensure we do not miss improvements and updates to third-party code, we also run tooling that lets us know about new versions of dependencies that are not tied to a RustSec advisory.

The above checks apply not just to the code that makes it into the ExpressVPN app but also to the service side infrastructure, ensuring that no malicious code can be pushed from a single point of compromise and making sure that dependencies on our servers are up to date.

Finally, permission to deploy to the production infrastructure requires approval from a specific subset of developers, thus limiting the risk of insider threat through the principle of least privilege.

ExpressVPN's build verification process has been independently audited by experts at PwC Switzerland.[28] For more on third-party assessments, see the section on external security audits.

[28] Results of this and other audits are available to ExpressVPN customers on our Privacy and Security Audits page.

7. Hardening our apps

Thus far we've talked about how we secure communications between the server and the ExpressVPN Keys application, as well as the cryptography that secures your vault. But what happens after the vault is unlocked is equally important.

In this section, we explore the security measures in our mobile apps and browser extension that further minimize the risk of your stored items being exposed.

Stored items remain encrypted until used

Unlocking your vault decrypts the metadata about the vault and each document within it, but it does not decrypt the body of any document. This means that login passwords, TOTPs, the bodies of your secure notes, and your credit/debit card numbers and their security codes remain encrypted until exactly when they are used, i.e.:

- when you choose to reveal them in the UI, or
- copy them to your clipboard, or
- use them to autofill a login field

Autofill security

When you visit a website with your vault unlocked, ExpressVPN Keys will prompt you to automatically fill in (**autofill**) your login credentials for that website, if they are in your vault.

While the act of populating usernames and passwords into a webpage might seem simple, there are a lot of complexities around how to do that securely, especially given the possibility that the website itself could be hostile. The website could be lying about its identity (e.g., pretending to be *gmail.com*), and with some malicious scripting could attempt to harvest all credentials by forcing the password manager's browser extension to autofill them into a field that the attacker controls.[29] Similar attacks can be employed on mobile devices as well.

We employ two mechanisms to prevent credential harvesting threats:

1. We strictly require **user interaction** before credentials are populated, so the user is always aware of what credential they are filling into the form. Anything that seems out of the ordinary will immediately sound alarm bells.

2. We also require strict checks on the **domain** where the credentials are populated. In particular, we will only fill in the credential if the domain matches[30] the one for the credential in ExpressVPN Keys. If the subdomains are different, a warning is displayed

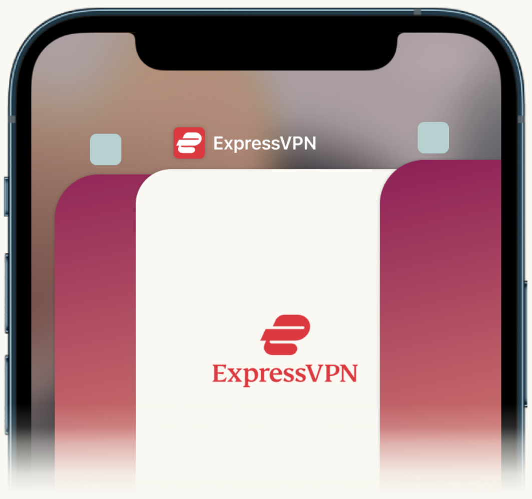
[29] This was studied extensively in Password Managers: Attacks and Defenses (Silver et al., 2014).
[30] This is a match on the domain name (and all subdomains) and each port is treated as a unique match.

and the user has to approve the request before the credentials are autofilled. We determine the correct domain via trusted API calls[31] on the browser extension, Android, and iOS, rather than anything mutable from an attacker's perspective.

The two mechanisms described significantly raise the difficulty level of attempting an attack on Keys' autofill mechanism. We further augment these defenses with regular internal and external security assessments to ensure that our implementation of the autofill is hardened to various attack vectors, even potentially novel ones.

Security screen

When multitasking on mobile applications, you could potentially make a mistake and take a screenshot of your password manager screen when it is in the background, potentially with the password exposed. With **security screen**[32] enabled, your screenshots will not capture anything while ExpressVPN Keys is in the background.



Biometric unlock

Enabling biometric unlock in Keys on mobile lets you access your stored items using your fingerprint or face *instead* of your primary password. This feature is protected by each platform's native authentication security:

On iOS, we allow authentication by Face ID and Touch ID on all devices running iOS 15 and above.

On Android, we allow biometric unlock *only* on devices with the most secure class of biometric sensor (Class 3).

[31] On Android, this performs package verification as per AutofillService documentation. On Chrome, we use the properties of the Tab object, accessed from the background script. On iOS, we use ASCredentialServiceIdentifier.
[32] Note that this is a feature on mobile versions of ExpressVPN Keys and is not available on the browser due to limitations on the operating system and browser extension interface.

8. Server and cloud infrastructure

Since unencrypted user data never leaves the user's device, ExpressVPN Keys' cloud server infrastructure is never called upon to process or store it. Still, our infrastructure has been architected to follow strict security best practices.

First, Keys infrastructure is completely separate from any other ExpressVPN infrastructure. This includes accounts, subscription services, and all VPN servers around the world. This separation of concerns means that ExpressVPN Keys' infrastructure is never exposed to any personally identifiable information (PII), nor affected by any of your VPN traffic.

Like all ExpressVPN infrastructure, Keys infrastructure has been carefully designed around the principle of least privilege, encryption at rest, as well as proactive monitoring and alerting to detect any issues.

By default, Keys infrastructure contains no permanent administrative services. All administrator access is performed using bastion hosts that are created on demand and destroyed when not in use. Access to create these bastion hosts is granted by a specific administrative role that has minimal privileges, and access is gated by 2FA backed by hardware devices (YubiKeys) and a passphrase known only to the owner of the hardware security device. Once they are created, the bastion hosts are only accessible from a specific list of trusted source IPs and again with a YubiKey.

Direct administrative access is rarely, if ever, required in Keys infrastructure—everything possible has been automated. Therefore, any creation of identity and access management (IAM) roles or cloud credentials that are able to create a bastion host—even the bastion host creation event itself—will generate a security alert that must be acknowledged or else it is escalated to a dedicated security operations center (SOC).

Different administrative duties are each given a distinct role (even though they may be performed by the same person), and each role is granted only the minimum privileges to perform that specific task. Access to any role requires authentication using a YubiKey device. All access credentials are disabled when not in use, and this is monitored automatically such that if a set of credentials are left enabled, this will be flagged for action by the SOC.

Where automation requires infrastructure access, we likewise ensure that dedicated roles with the minimal required privileges are used and that every environment has separate users and separate credentials so that environments cannot impact one another. This includes CI/CD jobs where the final deployment step requires authorization by an administrator before the job is granted access to the required credentials.

Other properties that are automatically monitored include the creation of new accounts and the creation of new security credentials. All events are also fed into a security information and event management (SIEM) system, which monitors and alerts on any anomalous activity.

9. Security audits

Internal

As we developed ExpressVPN Keys, our internal security team conducted an extensive review of the threat landscape and vulnerabilities that can affect password managers, resulting in the following battery of test cases and scenarios we use when performing our own security assessments:

GENERAL

- Ensure unauthorized modifications to the documents in the password vault can be detected
- Ensure third-party dependencies contain no known exploitable vulnerabilities
- Ensure the cryptographic libraries and primitives used have no known exploitable weaknesses
- Ensure the keys are appropriately sized according to industry standards and recommendations
- Ensure that the encrypted vault cannot be decrypted other than through the primary password or recovery code

EXPRESSVPN KEYS SERVER

- Ensure logs do not store sensitive information
- Ensure the database does not contain any secrets other than the encrypted vault
- Ensure the encrypted vault entries do not leak information about the length or contents of the data it is protecting
- Ensure proper access control (e.g. no other user can access a different user's password vault without proper authorization)
- Ensure tampered requests are rejected
- Ensure replayed requests are rejected
- Ensure there is no way of performing a race condition to corrupt data when multiple devices are syncing at the same time

CHROME EXTENSION

- Ensure communications through *postMessage* are properly checked for the appropriate origins (e.g. from a malicious page to the password manager extension, or from a malicious extension to the password manager extension)
- Ensure that credentials do not fill into untrusted portions in the Document Object Model (DOM), particularly attacker-controlled iframes
- Ensure clickjacking of the password manager extension is prevented
- Ensure user consent is given before autofilling to the form fields
- Ensure the extension is not susceptible to cross-site scripting (XSS) or CSS injection attacks from any user-controllable value
- Ensure no prototype pollution vulnerability exists in the codebase from any user-controllable value

MOBILE APPLICATIONS

- Ensure the domain/package being autofilled is checked appropriately before autofilling
- Ensure users are aware of the risks if they choose to use the password manager on a rooted device
- Ensure that if biometric authentication and decryption is enabled, that the authentication and decryption mechanism cannot be bypassed, even if the device is lost and rooted afterwards
- Ensure any IPC mechanism exposure does not cause leakage of any sensitive information, even the encrypted vault itself
- Ensure that all data, even encrypted, is removed after logging out

External

To help us continually improve Keys, we also commission security audits with reputable external vendors. In a series of third-party audits, we requested Cure53, an independent security vendor, to evaluate the security posture of ExpressVPN Keys on iOS, Android, and the Chrome extension. Specifically, Cure53 focused on whether a threat actor could perform the following exploits on the ExpressVPN Keys application:

- Gain ability to access and steal the vault via any remote means, including inter-process communication (IPC) mechanisms
- Extract or modify secrets stored in the password manager vault

In all three assessments, the security stance of ExpressVPN Keys was perceived positively by the assessors. We have reproduced excerpts from the reports here, with the full report linked below each:

"Generally speaking, the overall yield of findings is relatively moderate in comparison with similarly scoped audits, which would typically ***represent a positive indication of the inscope items' perceived security strength."***

Cure53 Pentest-Report ExpressVPN Keys Browser Extension 09.-10.2022

"The ExpressVPN Keys password manager was also deemed resilient against Unicode related origin confusion attacks... Given the importance of login credentials to the overall framework, the security of the data at rest was rigorously assessed. In this regard, the cryptographic functions utilized by the password manager to store the credentials ***garnered a solid impression on the whole."***

Cure53 Pentest-Report ExpressVPN Android Client App & Integrations 08.2022

"Additionally, the password manager integration and associated features were examined in-depth by Cure53. Positively, ***no client-side issues were detected*** *in the UI components. The matching of domains, subdomains, and the integration of the Autofill feature within the iOS ecosystem were also* ***soundly handled in general****."*

Cure53 Pentest-Report ExpressVPN iOS App 08.-09.2022

Further, ExpressVPN Keys is continuously tested by security researchers from around the world. All components of ExpressVPN Keys are in scope as part of our bug bounty program. If you find any issues that violate the security guarantees laid out in this document, please report them to us through this program at your earliest convenience.

Bibliography

Barker, E. (2009, September). *Recommendation for Digital Signature Timeliness*. NIST SP 800-102, Recommendation for Digital Signature Timeliness. https://doi.org/10.6028/NIST.SP.800-102

Barker, E., & Dang, Q. (2015, January). *Recommendation for Key Management, Part 3: Application-Specific Key Management Guidance*. NIST SP 800-57 Part 3 Rev. 1, Recommendation for Key Management, Part 3: Application-Specific Key Management Guidance. https://doi.org/10.6028/NIST.SP.800-57pt3r1

Barker, E., & Roginsky, A. (2019, March). *Transitioning the Use of Cryptographic Algorithms and Key Lengths*. NIST SP 800-131A Rev. 2, Transitioning the Use of Cryptographic Algorithms and Key Lengths. https://doi.org/10.6028/NIST.SP.800-131Ar2

Bellare, M., & Rogaway, P. (1998, August). PSS: Provably Secure Encoding Method for Digital Signatures. *IEEE P1363*.

Bormann, C., & Hoffman, P. E. (2020, December). *Concise Binary Object Representation (CBOR)*. RFC 8949. https://www.rfc-editor.org/info/rfc8949

Dworkin, M. (2007, November). *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST SP 800-38D, Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. https://doi.org/10.6028/NIST.SP.800-38D

Gasti, P., & Rasmussen, K. B. (2012). On the Security of Password Manager Database Formats. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings* (pp. 770-787). Springer. https://doi.org/10.1007/978-3-642-33167-1_44

Grassi, P., Newton, E., Fenton, J., Perlner, R., Regenscheid, A., Burr, W., Richer, J., Lefkovitz, N., Danker, J., Choong, Y.-Y., Greene, K., & Theofanos, M. (2017, June). *Digital Identity Guidelines: Authentication and Lifecycle Management*. NIST SP 800-63B - Digital

Identity Guidelines: Authentication and Lifecycle Management. https://doi.org/10.6028/NIST.SP.800-63b

Jones, M. B. (2015, May). *JSON Web Algorithms (JWA)*. RFC 7518. https://www.rfc-editor.org/info/rfc7518

Jones, M. B., Bradley, J., & Sakimura, N. (2015, May). *JSON Web Token (JWT)*. RFC 7519. https://www.rfc-editor.org/info/rfc7519

Josefsson, S. (2006, October). *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. https://www.rfc-editor.org/info/rfc4648

Kaliski, B. (2000, September). *PKCS #5: Password-Based Cryptography Specification Version 2.0*. RFC 2898. https://www.rfc-editor.org/info/rfc2898

Kerckhoffs, A. (1883, January). La cryptographie militaire. *Journal des sciences militaires*, *IX*, 5-38.

Li, Z., He, W., Akhawe, D., & Song, D. (2014). The Emperor's New Password Manager: Security Analysis of Web-based Password Managers. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014* (pp. 465-479). USENIX Association.

Mckay, K., & Cooper, D. (2019, August). *Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations*. NIST SP 800-52 Rev. 2, Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations. https://doi.org/10.6028/NIST.SP.800-52r2

Oesch, S., & Ruoti, S. (2020). That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (pp. 2165-2182). USENIX Association.

Rescorla, E. (2018, August). *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. https://www.rfc-editor.org/info/rfc8446

Rescorla, E., & Dierks, T. (2008, August). *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. https://www.rfc-editor.org/info/rfc5246

Rogaway, P. (2002, November). Authenticated-encryption with associated-data. *Proceedings of the 9th ACM Conference on Computer and Security, CCS 2002, Washington, DC, USA*, 98-107. https://doi.org/10.1145/586110.586125

Rogaway, P. (2011, February 10). Evaluation of Some Blockcipher Modes of Operation. https://web.cs.ucdavis.edu/~rogaway/papers/modes.pdf

Samarati, P., & Sweeney, L. (1998). Protecting Privacy when Disclosing Information: k-Anonymity and Its Enforcement through Generalization and Suppression. *Technical Report SRI-CSL-98-04*. https://www.csl.sri.com/papers/sritr-98-04/

Schaad, J. (2005, June). *Use of the RSASSA-PSS Signature Algorithm in Cryptographic Message Syntax (CMS)*. RFC 4056. https://www.rfc-editor.org/info/rfc4056

Shannon, C. (1949, October 4). Communication Theory of Secrecy Systems. *Bell System Technical Journal*, *28*(4), 662. https://doi.org/10.1002/j.1538-7305.1949.tb00928.x

Sheffer, Y., Hardt, D., & Jones, M. B. (2020, February). *JSON Web Token Best Current Practices*. RFC 8725. https://www.rfc-editor.org/info/rfc8725

Silver, D., Jana, S., Boneh, D., Chen, E., & Jackson, C. (2014, August). Password Managers: Attacks and Defenses. *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, 449-464.

Wheeler, D. L. (2016). zxcvbn: Low-Budget Password Strength Estimation. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016* (pp. 157-173). USENIX Association.

Glossary

- **2FA:** two-factor authentication, a security mechanism that requests the user to provide two separate, distinct forms of identification before privilege is granted

- **AEAD:** see **authenticated encryption with associated data**
- **AES:** Advanced Encryption Standard, an industry specification for encrypting data with symmetric keys
- **AES-256-GCM:** a mode for AES that uses 256-bit keys and the Galois/Counter Mode that ensures both confidentiality and integrity of the data
- **API:** Application Programming Interface, a set of definitions and contracts that enable different systems to communicate and interact with each other
- **authenticated encryption with associated data:** an encryption scheme that ensures confidentiality of the data and authenticity, potentially with additional information that has its integrity protected
- **authentication:** the process of validating and verifying the identity of a user before granting access privileges to a system
- **authenticity:** the property of being unforgeable, meaning that there exists cryptographic evidence that the request was made by the sender
- **authorization:** the process of giving access privileges to a system
- **autofill:** an input method that takes data from a data source and automatically fills the data on a web page or application, based on the context of the fields to be filled in
- **Base32:** an encoding of byte data to a restricted set of 32 writable symbols as defined in RFC4648
- **bastion host:** a hardened server used to manage access to an internal/private network from an external network to minimize the attack surface exposed onto the external network
- **bit:** binary digit consisting of either 0 or 1
- **brute-force attacks:** the process of enumerating all possibilities in order to gain access to a system
- **bug bounty program:** a program that allows researchers to report security issues and receive recognition and compensation for valid issues
- **byte:** 8 bits, typically the smallest addressable unit of memory on computers, which encodes a single character
- **CA:** see **certificate authority**

- **CBOR**: see **Concise Binary Object Representation**
- **CD:** see **Continuous Delivery**
- **certificate authority:** an entity that stores, signs, and issues digital certificates
- **certificate:** also known as a public key certificate, this is an electronic identifier for a given entity with a digital signature issued by a certificate authority
- **Chromium:** the open-source web browser on which many other web browsers are based
- **CI:** see **Continuous Integration**
- **CI/CD:** Continuous Integration/Continuous Delivery
- **cipher suites:** a set of cryptographic algorithms used in TLS to establish secure communications, with different cipher suites offering different levels of security
- **cloud credentials**: an electronic proof of identity used to authenticate to cloud resources
- **commit** a snapshot of the current state of the codebase in Git
- **Concise Binary Object Representation:** a binary data serialization format as defined in RFC 8949
- **confidentiality:** the property of secrecy or privacy of data, usually achieved with encryption
- **Continuous Delivery:** a software development practice where code changes are automatically prepared for release or production
- **Continuous Integration:** a software development practice where code changes are automatically tested when pushed upstream
- **defense-in-depth:** the practice of leveraging multiple layers of security measures to achieve a security goal
- **deserialization:** a process by which a byte stream representing an object is re-structured back into a fully functional object at runtime
- **document:** a generic storage for a record in the vault, consisting of encrypted metadata and the encrypted contents (see **stored item**)

- **Document Object Model:** the data representation of the objects that comprise the structure and content of a document on the web
- **DOM:** see **Document Object Model**
- **Encrypted User Record:** a data record containing the *ppdk*-encrypted *User Root Key*, the *rcdk*-encrypted *User Root Key*, the *Public Identity Key*, and *Encrypted UserRecordBody*
- **encryption:** the process of protecting data by scrambling it with cryptographic functions such that only an actor with the key can access the unscrambled data
- **encryption at rest:** the property that any data written to storage is always encrypted (i.e., plaintext is never stored).
- **HaveIBeenPwned:** a website that allows users to check whether their personal data has been compromised in data breaches
- **HIBP:** see **HaveIBeenPwned**
- **HMAC:** a hash-based message authentication code that uses a cryptographic hash function together with a cryptographic key
- **HTTP:** Hypertext Transfer Protocol, for transmitting hypermedia documents like webpages in a web browser
- **HTTPS:** Hypertext Transfer Protocol Secure, used to transmit hypermedia documents securely by leveraging **TLS**
- **IAM role:** Identity and Access Management role, represents an identity that has specific permissions to carry out the subset of operations that it is authorized to perform
- **iframe:** an HTML element on the DOM that loads another document within the current page
- **IND-CDBA:** indistinguishability of databases, a property of a password manager database that implies a read-only adversary cannot infer information on the records stored in the password manager database
- **IND-CPA:** indistinguishability under chosen-plaintext attack, a security property that implies an adversary that has access to an encryption oracle can infer useful information for future encryptions based on the ciphertexts of those encryptions
- **integrity:** a security property that implies an adversary is not able to tamper with a piece of information without being detected

- **inter-process communication:** a mechanism that allows different applications to communicate and interact with each other
- **IPC:** see **inter-process communication**
- **JWT:** JSON Web Token, a standard for credentials as defined in RFC 7519
- **k-anonymity:** a property of a dataset that measures the degree of anonymity provided by the dataset and reduces the risk of being uniquely re-associated with a particular record
- **KDF:** see **Key Derivation Function**
- **Kerckhoffs's principle:** the principle that a cryptosystem should be secure even if everything about the system, except the key, is public knowledge
- **Key Derivation Function:** a function used to derive secret keying material from a secret and other information
- **MAC:** see **message authentication code**
- **MAL-CDBA:** malleability of chosen database game, a security property that implies a password manager database can detect that it is being tampered with by a read-write adversary that has access to an encryption oracle
- **message authentication code:** a digital tag used to authenticate a message and its integrity
- **nonce:** a number that is only used once, employed in conjunction with a request to prevent replay attacks
- **OpenSSL:** an open source software library for the TLS protocol
- **Password Health:** a score in ExpressVPN Keys representing the security of the logins in your vault, taking into account previous exposure, re-use, strength, presence of 2FA, and protocol used
- **PBKDF2:** Password-Based Key Derivation Function 2, a cryptographic function used to generate cryptographic keys from passwords, as defined in RFC 2898
- **perfect forward secrecy:** a security feature of changing cryptographic keys so that sessions are not compromised even in the event that the long term secret is compromised
- **personally identifiable information:** data that can be used to identify a person, including but not limited to names, addresses, phone numbers, and email addresses

- **PFS:** see **perfect forward secrecy**
- **PII:** see **Personal Identifiable Information**
- **plaintext:** any data that is not encrypted, sometimes called cleartext
- **ppdk:** see **Primary Password Derived Key**
- **PR:** see **Pull Request**
- **primary password:** the password used to unlock a Keys user's vault, provided when the user first signs in to ExpressVPN Keys
- **Primary Password Derived Key:** a cryptographic key derived from the primary password; used to decrypt the encrypted *User Root Key* in the *Encrypted User Record*
- **primitives:** well-established, low-level algorithms (e.g., hash functions, key exchanges) used as building blocks in a larger cryptographic system
- **principle of least privilege**: the principle that every module of a computer system should only have access to the functions necessary for its legitimate use, and no more
- **Private Identity Key:** an RSA private key unique to each user, used to sign requests made to the ExpressVPN Keys API server
- **prototype pollution:** a JavaScript vulnerability that enables an adversary to add arbitrary properties to global object prototypes which may be inherited by other objects
- **Public Identity Key:** an RSA public key unique to each user, used to verify requests made to the ExpressVPN Keys API server
- **Pull Request:** a mechanism in software development where a contributor signals to others the intention to merge code changes from one source to another source
- **rcdk:** see **Recovery Code Derived Key**
- **recovery code:** a Base32 encoded, randomly generated string that can be used to recover access to the ExpressVPN Keys vault if the primary password is forgotten or lost
- **Recovery Code Derived Key,** a cryptographic key derived from the recovery code; used to decrypt the encrypted *User Root Key* in the *Encrypted User Record*
- **replay attack:** an attack in which an adversary collects legitimate traffic then retransmits it in an attempt to gain illegitimate access

- **RS256**: RSA+SHA256 as used in the JWT specifications to sign the JWT, defined in RFC 7518

- **RSA:** a public key cryptosystem used for secure data transmission, where a key pair (public, private) is generated for use in cryptographic operations such as encryption or signing

- **RSA-PSS:** also known as RSASSA-PSS, a probabilistic signature scheme that replaces RSA PKCS#1 v1.5, defined in RFC 4056

- **salt:** random data passed as an additional input to a one-way function to increase resilience to brute-force attacks

- **Security Information and Event Management system:** a security solution that provides real-time analysis of logs and data from applications and hardware to produce security alerts

- **security operations center:** a function within an organization that continuously monitors for threats, detects adversarial activity, and investigates and responds to cyberattacks

- **security screen:** a privacy feature in ExpressVPN Keys on mobile platforms that prevents information on the application screen from being captured in a screenshot while the application is running in the background

- **SHA1:** Secure Hash Algorithm 1, a hash function takes an input and outputs a 160-bit message digest, defined in RFC 3174

- **SHA512:** a hash function takes an input and outputs a 512-bit message digest, defined in RFC 6234

- **SIEM:** see **Security Information and Event Management system**

- **Signature:** a mathematical scheme for verifying the authenticity of messages

- **SOC:** see **Security Operations Center**

- **step-up authentication:** a mechanism that requests an additional factor for authentication when a user of ExpressVPN wishes to access ExpressVPN Keys

- **stored item**: a login (username/password), credit/debit card details, or secure note stored in a user's vault (see **document**)

- **TLS:** Transport Layer Security, a cryptographic protocol that provides end-to-end security for data sent between parties over the internet

- **TOTP:** time-based one-time password, a form of 2FA that uses the current time as a source of uniqueness, defined in RFC 6238
- **User Document Key:** the cryptographic key used to encrypt and decrypt documents' metadata and documents containing username and password information
- **User Root Key:** the cryptographic key used to decrypt the encrypted *UserRecordBody*
- **UserRecordBody:** a data structure, encrypted by default, that stores the *User Document Key* and *Private Identity Key*
- **vault:** the collection of all documents and their metadata, uniquely tied to the user
- **Web Crypto API:** a W3C standard that defines an interface to perform cryptographic operations in web applications
- **YubiKey:** a hardware authentication device to protect access to systems, typically used as a second factor
- **zero-knowledge encryption:** the process of encrypting user data with key material that cannot be inferred or derived by the service provider
- **zxcvbn:** an open source password strength estimator, see https://github.com/dropbox/zxcvbn