ISEC- 601

Teaching Aide
Password-Based Key Derivation Function 2

PBKDF2 Building blocks & How it’s Used

History

HMAC

How it works (Key Derivation, Salting, Iterations)

Development / how pbkdf2 progressed- Cooper

Security Assessment

Main Application

Future / New Standard / Alternatives

Conclusion

History

- Most users select password in low randomness (password, 123456)
- This make the number of attempt an attacker need to try fairly low
- The higher entropy password are more difficult to brute force

Password Length	All Characters	Only Lowercase
3 characters	0.86 seconds	0.02 seconds
4 characters	1.36 minutes	.046 seconds
5 characters	2.15 hours	11.9 seconds
6 characters	8.51 days	5.15 minutes
7 characters	2.21 years	2.23 hours
8 characters	2.10 centuries	2.42 days
9 characters	20 millennia	2.07 months
10 characters	1,899 millennia	4.48 years
11 characters	180,365 millennia	1.16 centuries
12 characters	17,184,705 millennia	3.03 millennia
13 characters	1,627,797,068 millennia	78.7 millennia
14 characters	154,640,721,434 millennia	2,046 millennia

History

Rank	Password	Frequency
1	123456	753,305
2	linkedin	172,523
3	password	144,458
4	123456789	94,314
5	12345678	63,769
6	111111	57,210
7	1234567	49,652
8	sunshine	39,118
9	qwerty	37,538
10	654321	33,854
11	000000	32,490
12	password1	30,981
13	abc123	30,398
14	charlie	28,049
15	linked	25,334
16	maggie	23,892
17	michael	23,075
18	666666	22,888
19	princess	22,122
20	123123	21,826

- Top 20 passwords of LinkedIn users in 2012
- Standard hashing ineffective algorithms such as MD5 os SHA-1 are fast but low entropy

History

Imagine you need to transmit data over insecure channel. What would you do?
Encrypt it using a password, right?
Most people would do that.
Most people would also choose a short, easy-to-remember password.
This makes it for a huge vulnerability to brute-force attacks

Is there anything that would help us mitigate that?

History

One way to do this is to encrypt data with a *Key* that was derived from a *Password*. Here is where Key Derivation Functions (KDF) come in handy.

The first standardized KDF was Password-Based KDF 2 (PBKDF2). Introduced in 2000 by RSA Laboratories, in RFC: 2898

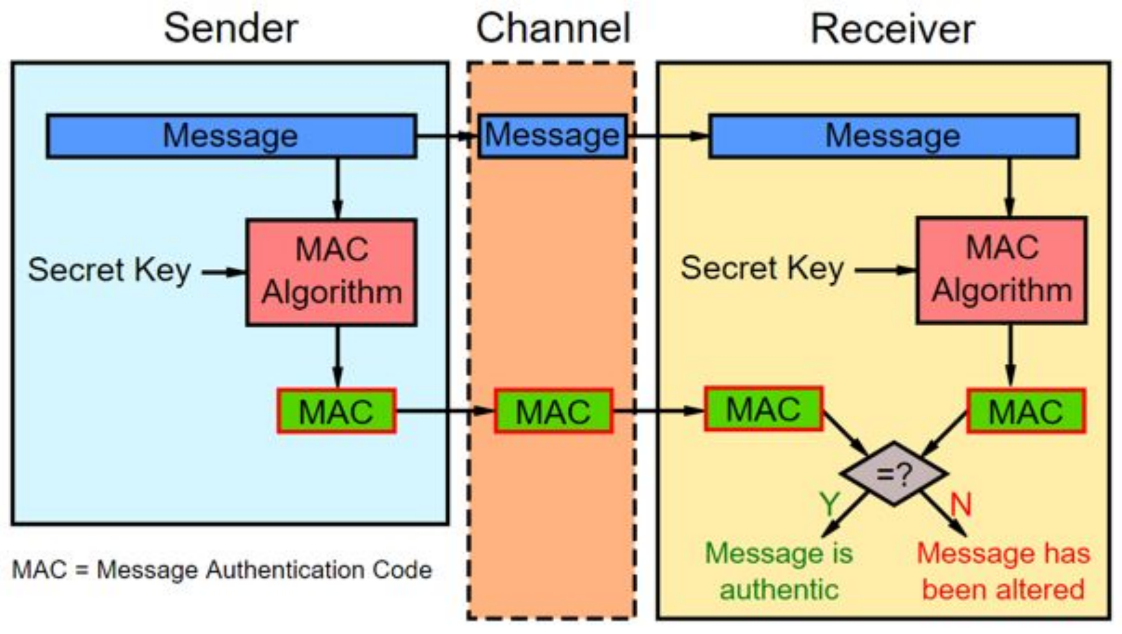
Before it, there was a PBKDF1, which didn't produce large enough keys, and thus, was vulnerable.

Authenticity and Integrity

- HMAC is a type of MAC
- Using the Password as HMAC key establish Authenticity
- HMAC ensures Integrity

- How can we identify who is behind a device ? (Authenticity)

- How can we ensure that the message was not altered during transmission? (Integrity)



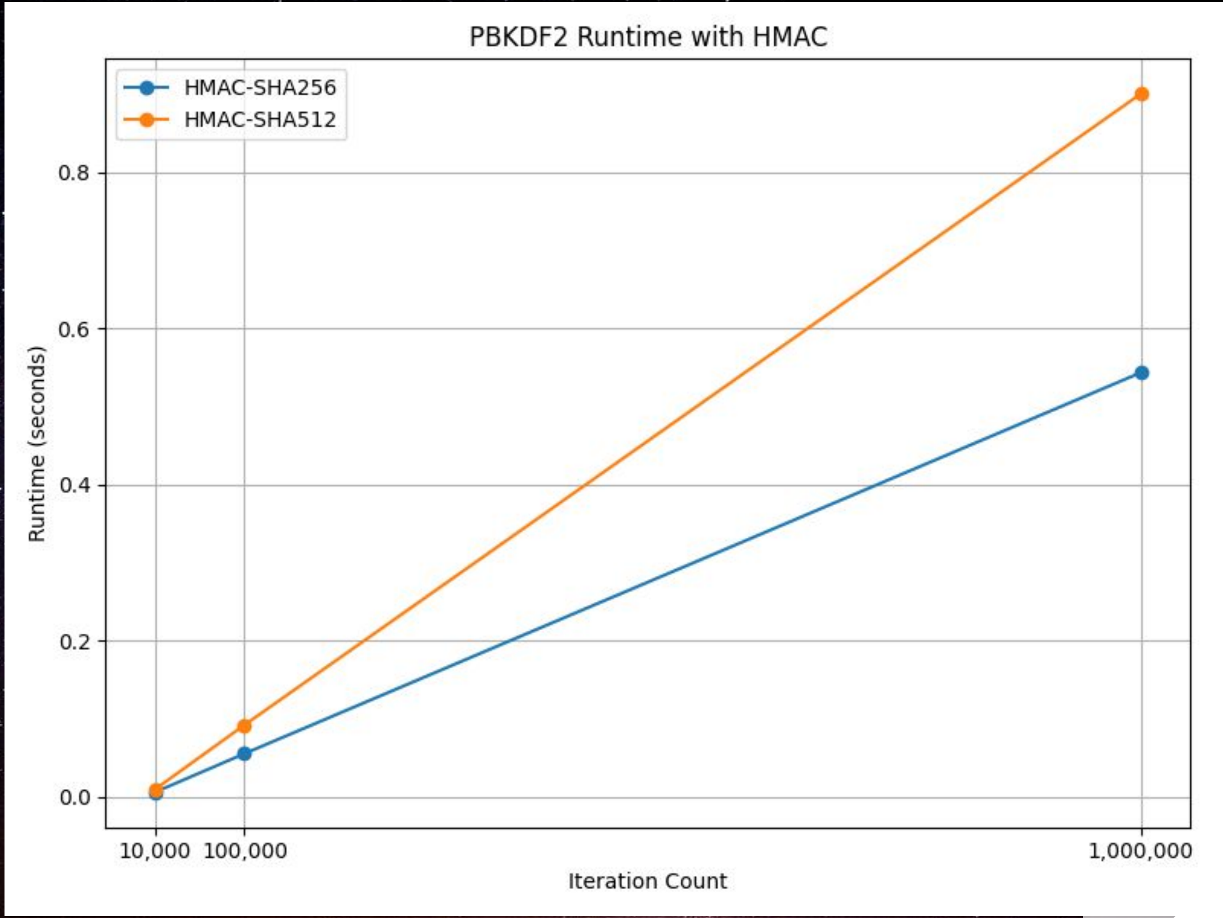
HMAC

- MACs are called “cryptographic checksums” (Integrity)
- [m || h(m)] -> hashing the message alone is insecure
- If only the sender and receiver know the key we can assume authenticity
- Tag = [M || h(k || m)] k is shared by the sender and receiver
- MAC is still vulnerable to extension attacks
- HMAC(K, M) = H((K' ⊕ opad) // H((K' ⊕ ipad) // M))
- Ipad 0x36 constant and opad 0x5c with long hamming distance
- only someone with the secret key can generate or verify the HMAC.
-

HMAC as the PRF

- HMAC combines a secret key with a cryptographic hash function (SHA-512)
- HMAC is often used as a Pseudo-Random Function
- It is collision-resistant
- It produces outputs of predictable fixed size

Short keys are more vulnerable to brute-force attack.



How it works

As stated by IETF: “PBKDF2 applies a pseudorandom function to derive keys”

PBKDF2 allows various pseudorandom functions to be included (PRF)
The length of an output of such a function is denoted as hLen (measured in bytes). It is also a length of each individual block that will make up the final key.

PBKDF2 function takes in four variables:

- P - Password from which the key is derived (string)
- S - salt (string)
- c - iteration count (integer)
- dkLen - intended length of the derived key (integer, bytes)

And outputs DK (derived key)

Let’s take a closer look!

How it works

1) First, PBKDF2 takes hLen*(2^32-1) and compares it to dkLen.
   If DkLen is bigger, PBKDF2 will return an error

This means that 2^32 is the maximum number of blocks PBKDF2 can produce to make up the key.

How it works

2) Next, we get 2 variables - l and r

l is the number of blocks with length hLen within the derived key (rounded up)
l = CEIL(dkLen/hLen), where CEIL is a ceiling function

and r is is the length of the last block in bytes
r = dkLen - (l-1)*hLen

These two help with determining how many blocks are needed to reach the key length (l) and for computing the necessary padding size for the last block (if its length is not a multiple of hLen

How it works

3) Next, PBKDF2 applies a function F to compute each hLen-block T of the final key.

The function F takes in the password P, salt S, iteration count c, and block index i

Meaning, block T_3 will be computed with F(P,S,c,3), and block T_l with F(P,S,c,l)

How it works

The function F itself is executed for each block c-number of times, each time producing U_x, which goes from U_1 to U_c.

On the first iteration (when computing U_1), the previously determined PRF takes in P, S, and the index of corresponding block i (taking only the first four most significant bytes of it).

The password P is concatenated with salt S and counter i before being fed into the PRF.

How it works

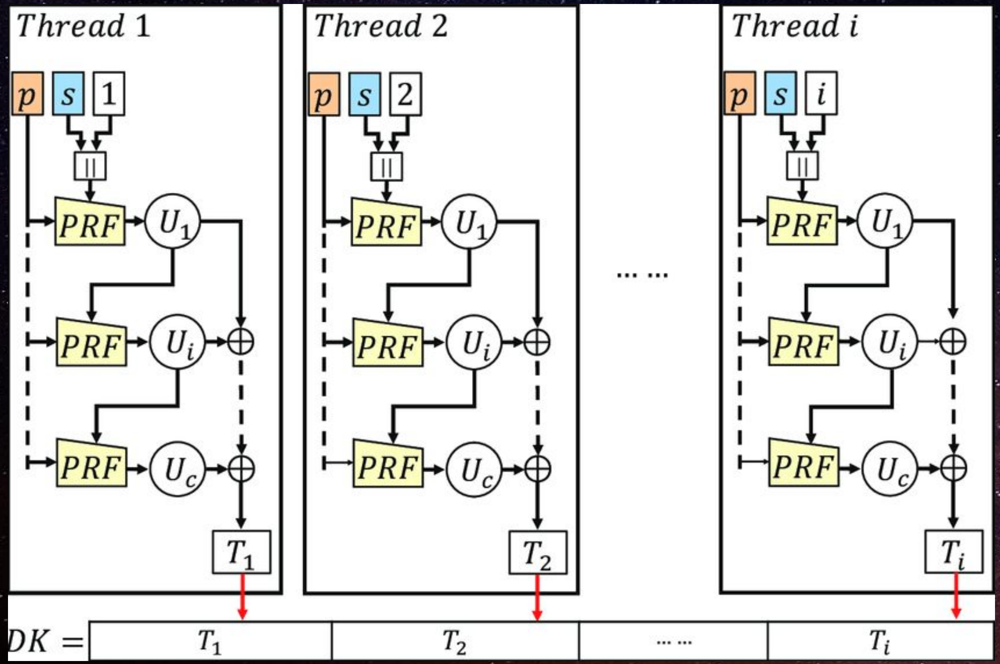
On each concurrent iteration password P is concatenated with the output of the previous round of PRF (i.e. to compute U_2, PRF will use P and U_1).

This is repeated until we compute U_c, after which, each of computed U_x is XOR'ed with other U_x within the same block.
In other words, $T_2 = F(P,S,c,2)=U_1 \text{ XOR } U_2 \text{ XOR } \dots \text{ XOR } U_C$

How it works

4) Finally, after computing all of T_x, they are concatenated.

To produce the derived key (DK), all we need is to take the first dkLen bytes from the result of the concatenation.



EVOLUTION OF THE PBKDF2

1970s-1990s

Pre-KDF & PBKDF1

2000s

The Birth of PBKDF2

2010s

Widespread Adoption & Scaling Iterations

2015s-Now

Modern Context & Alternatives

Pre-KDF & PBKDF1 (1970s–1990s)

In the early era, passwords were often stored using direct hashes (MD5, SHA-1) with little or no salt, leaving systems vulnerable to dictionary and rainbow table attacks. PKCS #5 v1.5 (1993) introduced **PBKDF1**, the first formal password-based derivation scheme, which applied a hash repeatedly to password and salt. However, PBKDF1 was limited to short key lengths and relied on weak iteration counts. This stage was about recognizing the need for systematic password hardening, but the tools were still primitive by today's standards.

PBKDF1 (1970s–1990s)

PBKDF1 (P, S, c, dkLen)

$T_1 = \text{Hash}(P \| S)$,
$T_2 = \text{Hash}(T_1)$,
...
$T_c = \text{Hash}(T_{c-1})$,

$DK = Tc<0..dkLen-1>$

The Birth of PBKDF2 (2000)

The release of **PKCS #5 v2.0 (RFC 2898)** in 2000 marked the formal introduction of **PBKDF2**. It improved on PBKDF1 by allowing arbitrary key lengths and replacing simple hashing with **HMAC** as the pseudorandom function. Importantly, PBKDF2 established the three pillars of modern key derivation: unique salts, high iteration counts, and flexible key sizes. This became the baseline recommendation for password-based encryption, influencing standards like PKCS #12 and NIST guidelines.

Widespread Adoption & Scaling Iterations (2010s)

During the 2010s, PBKDF2 became a global standard for password hashing and key derivation, powering **WPA2 Wi-Fi, BitLocker, VeraCrypt, and password managers**. Security guidance, such as **NIST SP 800-132 (2010)**, formally endorsed PBKDF2, recommending ≥128-bit salts and iteration counts tailored to hardware capabilities. As hardware improved, recommended iterations rose from thousands to hundreds of thousands. However, researchers highlighted a weakness: PBKDF2 is **CPU-bound**, making it easier for attackers with GPUs/ASICs to parallelize brute-force attempts.

Modern Context & Alternatives (2015–Present)

The **Password Hashing Competition (2015)** accelerated interest in more robust, **memory-hard** KDFs like **scrypt and Argon2**, which resist parallel cracking far better. While Argon2id has since been standardized (RFC 9106), PBKDF2 remains widely deployed due to its simplicity, library support, and FIPS compliance. Modern advice (e.g., OWASP 2023) is to have iteration counts set from **hundreds of thousands to millions** and migrate where possible to Argon2id for new designs. Today, PBKDF2 represents the “compatibility anchor”: old but reliable, still suitable when carefully tuned, yet increasingly supplemented by stronger next-generation algorithms.

PBKDF2 vs Rainbow Table

Since rainbow tables contain lists of precompiled hash values for the passwords, deriving a key and using a key instead a password itself would require a hacker to first compute the derived value, which, while not really mitigating the thread, is supposed to slow the brute-force attacks down by a certain factor (depending on the number of iterations of PBKDF2 and whether GPU cluster is used (will be discussed later on))

PBKDF2 Advantages

It is deterministic and resistant to preimage and collision attacks

It is highly customizable depending on the use case and amount of security needed, as the number of iterations can be easily adjusted to increase encryption and decryption time (increases security, but wait times for end user are longer) or decrease them (decreases security, but make it more usable for end user)

Additionally, PBKDF2 is scalable as it allows the use of different PRF’s, meaning that, if the old PRF becomes vulnerable, a new one can put in its place without changing the rest of the encryption-decryption system

Security Considerations

PBKDF2 is a cryptographic key derivation function, which is based on iteratively deriving Hash-based Message Authentication Code (HMAC)

- It is easy to implement PBKDF2 into many systems due to its simplicity
- It reduces the password requirements for general users since their passwords are converted to a fixed-size key, regardless of the complexity of the original password
- PBKDF2 is very scalable due to its ability to have variable number of iterations (with slower derivation leading to higher resistance, but also higher login time, and vice versa)

Security Considerations

- On the flipside, it does not mitigate mentioned attacks, instead, it is meant to slow them down. In any of the above attacks, an attacker will require more time to check against each individual password

- While PBKDF2 is CPU-intensive, is not very resistant to GPU attacks, where an attacker can compute against multiple passwords simultaneously, diminishing benefits of using PBKDF2

Security Considerations

As NIST suggested in their NIST SP 800-63-3, “the iteration count SHOULD be as large as verification server performance will allow, typically at least 10,000 iterations”. This was proposed in June 2017.

In July 2025 NIST released NIST SP 800-63-4, where PBKDF2 is no longer even mentioned as recommended function to use.

Application

Industry Applications of PBKDF2 Password Hashing:

- Microsoft Windows Data Protection API (DPAPI)
- Keeper (for password hashing)
- LastPass (for password hashing)
- 1Password (for password hashing)
- Enpass (for password hashing)
- Dashlane (for password hashing)
- Bitwarden (for password hashing)
- Standard Notes (for password hashing)
- Mac OS X Mountain Lion (for user passwords)
- Apple's iOS mobile operating system (for protecting user passcodes and passwords)
- WinZip (AES Encryption Scheme)
- Django (web framework, as of release 1.4)

WPA2-PSK

- Often PBKDF2 is used to create an encryption key from a defined password
- PBKDF2 uses HMAC-SHA1 to derive a Pairwise Master Key(PMK)
- A PSK(Pre-shared Key) is a shared secret between two parties

- IEEE 802.11i standard defines WPA in Wifi pre-shared key as:
  - PSK = PBKDF2(PassPhrase, ssid, ssidLength, 4096, 256)
- It uses AES CCMP
- WPA2 relies entirely on PBKDF2
- It is vulnerable to offline attacks and lacks forward secrecy

WPA2 vs WPA3

- WPA3 introduces SAE (Simultaneous Authentication of Equals)
- WPA3 requires interaction with the access point for each authentication attempt
- It supports forward secrecy
- It uses GCMP-256

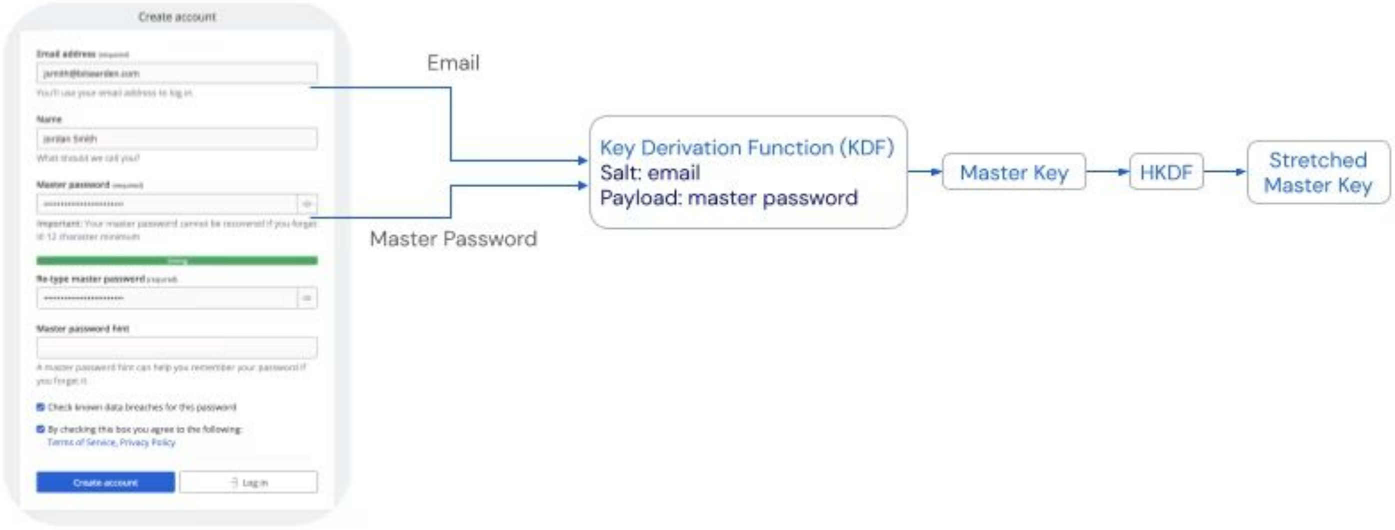
Wi-Fi security protocol	Key management approach	Encryption size	Protocols used
WEP	Static keys	64-, or 128-bit	RC4 (Rivest Cipher 4)
WPA	Dynamic keys	128-bit	RC4 (Rivest Cipher 4)
WPA2	Dynamic keys	128-bit or 265-bit	AES (Advanced Encryption Standard) using CCMP (Counter Mode with Cipher Block Chaining Message Authentication Code Protocol)
WPA3	Dynamic keys (unique keys, individualized data encryption)	192- and 256-bit	GCM (Galois-Counter Mode) using SAE (Simultaneous Authentication of Equals)

Bitwarden Schemes

Password storage:

- Bitwarden is an open source password management service
- When a user registers, it uses a key derivation function (PBKDF2) with 700,000 iteration rounds to stretch the master password with user's email address as a salt, using HMAC-SHA256 as its PRF
- The resulting salted value is 256-bit Master key that is stretched again with HKDF
- HMAC-based extract and expend stretched the key to 512 bits
- Bitwarden employees cannot see user's password

Bitwarden Client



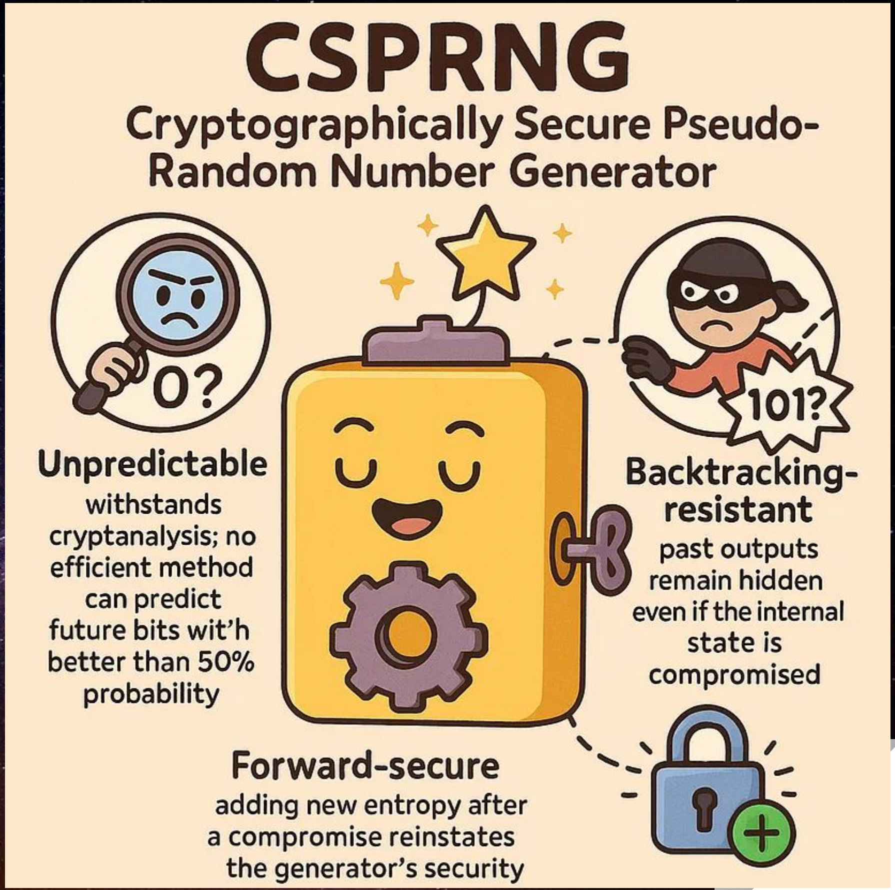
CSPRNG

A CSPRNG is a deterministic algorithm

1. Unpredictable (weaker than indistinguishability)
2. Backtracking-resistant
3. Forward-Secure

Bitwarden uses a CSPRNG (Cryptographically Secure Pseudorandom Number Generator)

Bitwarden generates 512-bit Symmetric Key (MAC: 256-bits, Encryption Key: 256-bits) and IV-128 bits



Bitwarden Schemes

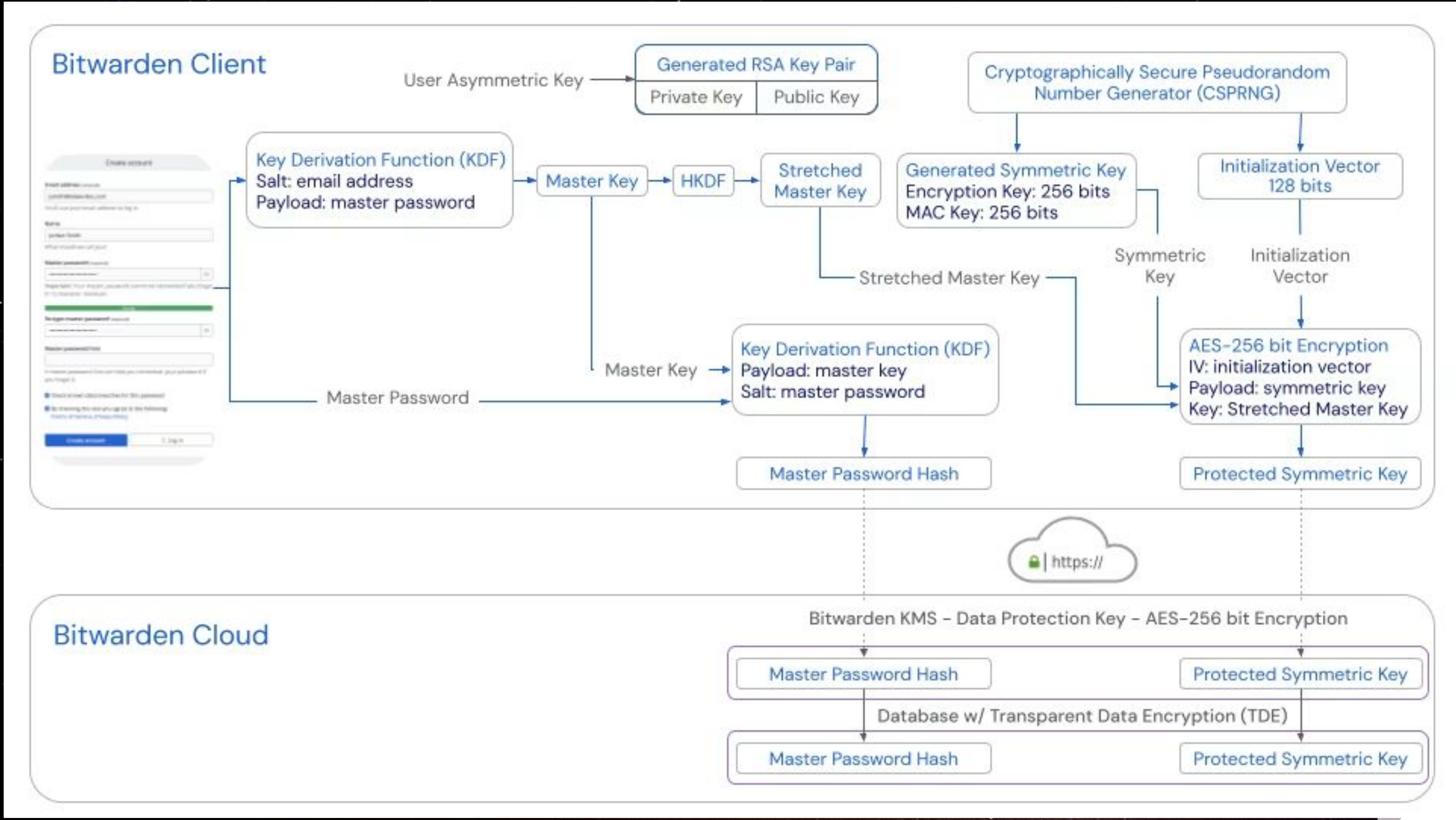
- The **Symmetric Key** as a payload is encrypted with **AES-256** bit encryption using the **Stretched Master Key** and **Initialization Vector** producing the protected Symmetric Key

- A **Master Password Hash** is generated using PBKDF-SHA256 with a payload of the Master Key and with a salt of the master password

- The Protected Symmetric key and Password Hash are sent to the Bitwarden Server via generated asymmetric encryption RSA Key pair

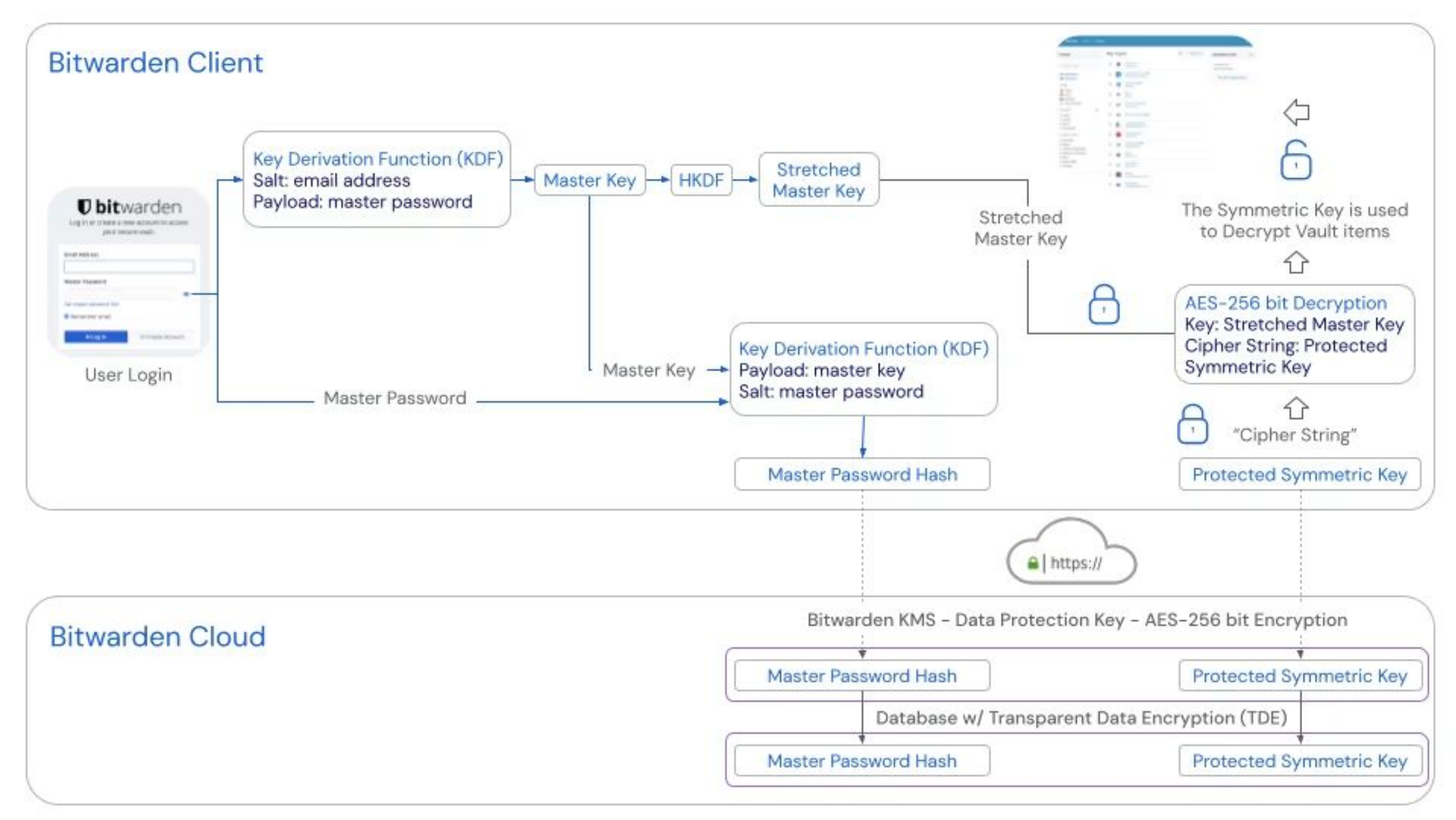
- The Master Password Hash is used for authentication

- Bitwarden does not keep the master password itself stored locally or in-memory on the Bitwarden client



Bitwarden Client
User Asymmetric Key
Generated RSA Key Pair
Private Key
Public Key
Cryptographically Secure Pseudorandom Number Generator (CSPRNG)
Key Derivation Function (KDF)
Salt: email address
Payload: master password
Master Key
HKDF
Stretched Master Key
Generated Symmetric Key
Encryption Key: 256 bits
MAC Key: 256 bits
Initialization Vector 128 bits
Stretched Master Key
Symmetric Key
Initialization Vector
Master Key
Key Derivation Function (KDF)
Payload: master key
Salt: master password
Master Password
AES-256 bit Encryption
IV: initialization vector
Payload: symmetric key
Key: Stretched Master Key
Master Password Hash
Protected Symmetric Key
https://
Bitwarden Cloud
Bitwarden KMS - Data Protection Key - AES-256 bit Encryption
Master Password Hash
Protected Symmetric Key
Database w/ Transparent Data Encryption (TDE)
Master Password Hash
Protected Symmetric Key

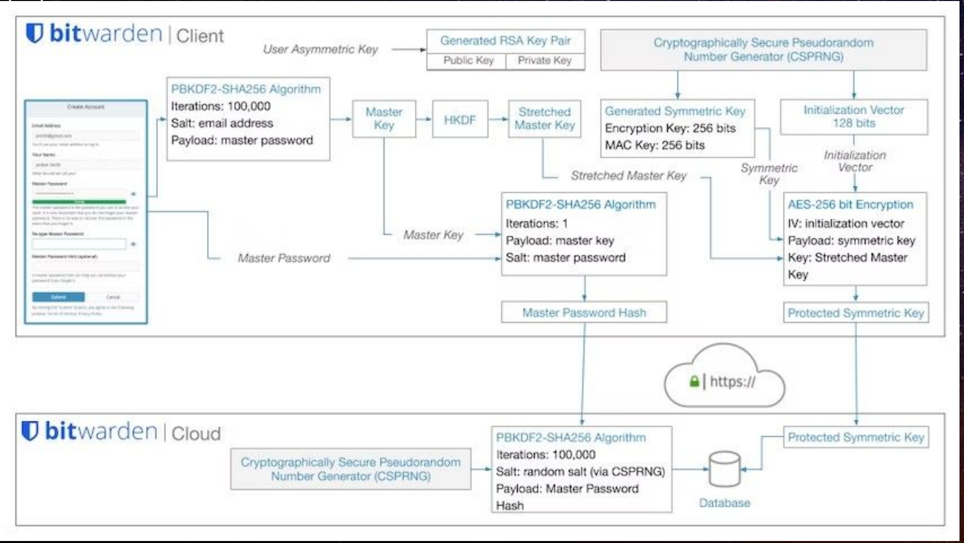
Finally, the Stretch Master key and Protected Symmetric key (AES-256) are used to decrypt vault items (usernames and passwords)



(1) A critical vulnerability lead to a lot of users losing their credentials (LastPass data breach). In the past, it used 100,100 iterations, well below OWASP recommendation of 310,000. Worst of all, some accounts only had 5,000 iterations. (Before)

(2) This shows the maximum time to crack PBKDF2 with SHA-256 using 12 GPUs (RTX 4090)

(1)



(2)

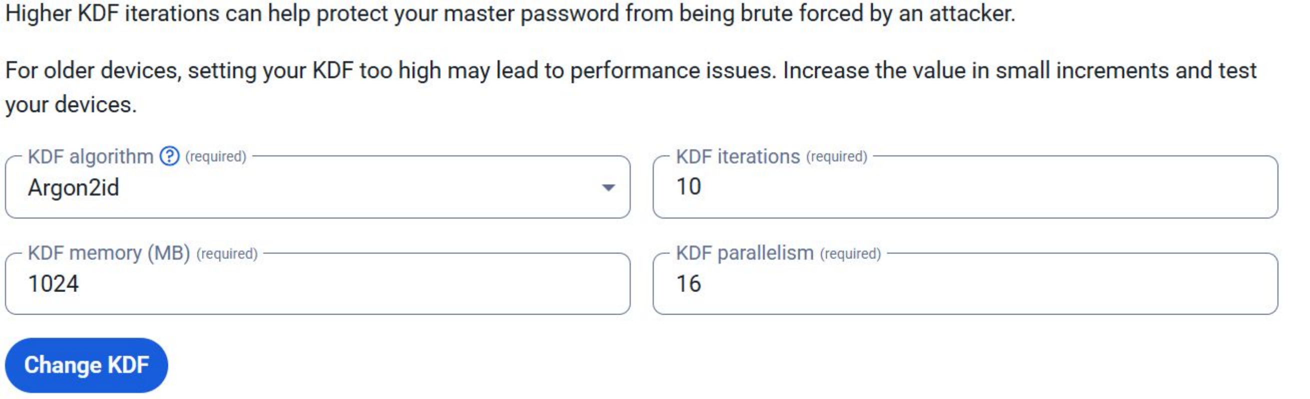
TIME IS TAKES A HACKER TO BRUTE FORCE YOUR LastPass•••| PASSWORD

*Following the 2022 data breach

Hardware: 12 x RTX 4090 | Password hash: PBKDF2-SHA256 * 600,000

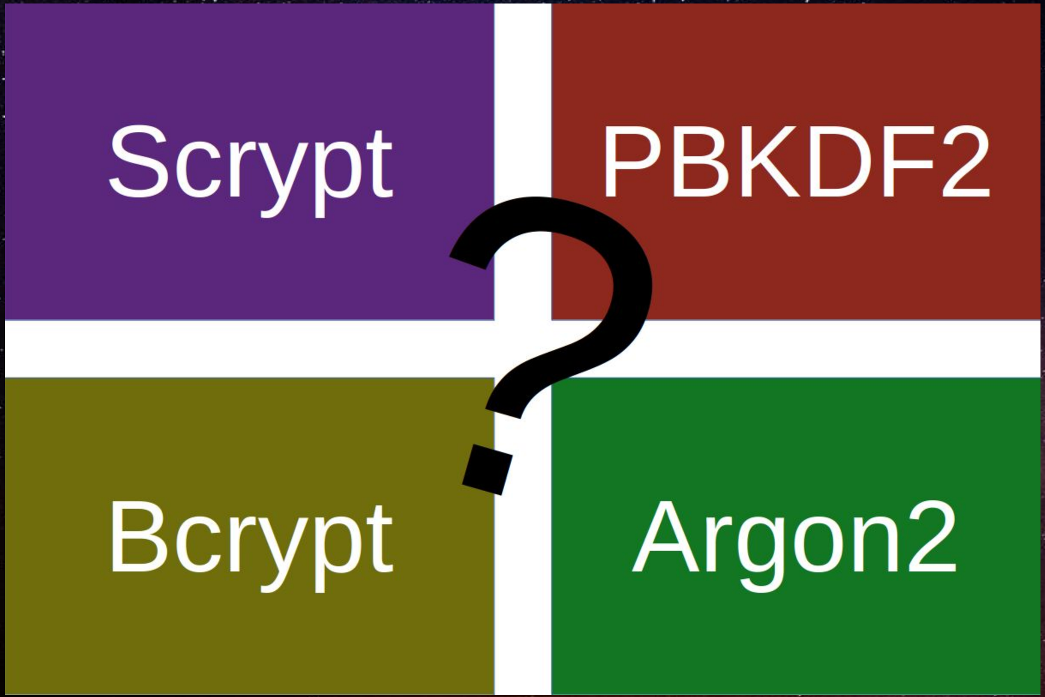
Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters, Symbols
4	Instantly	3 secs	44 secs	1 min	2 mins
5	1 sec	1 min	38 mins	2 hours	3 hours
6	6 secs	31 mins	1 day	4 days	1 weeks
7	1 min	13 hours	2 months	8 months	2 years
8	10 mins	2 weeks	10 years	42 years	110 years
9	2 hours	1 year	529 years	2k years	7k years
10	17 hours	27 years	27k years	159k years	537k years
11	7 days	699 years	1m years	9m years	37m years
12	2 months	18k years	74m years	614m years	2bn years
13	2 years	472k years	3bn years	38bn years	184bn years
14	19 years	12m years	201bn years	2tn years	12tn years
15	190 years	319m years	10tn years	146tn years	904tn years
16	1k years	8bn years	544tn years	9qd years	63qd years
17	19k years	215bn years	28qd years	562qd years	4qn years
18	190k years	5tn years	1qn years	34qn years	310qn years

- Bitwarden latest options uses Argon2id
- Use a Password Strength testing tool (https://bitwarden.com/password-strength/)
- Use MFA or better passkey (biometric or FIDO2)
- Replace PBKDF2 with Argon2id
- The downside of increasing the iterations for PBKDF2 or KDF iterations for Argon2id is a slower decryption



Alternatives to PBKF2

Considering PBKDF2's weakness to GPU-based attacks, the likely question that comes to mind is whether there are any better alternatives?



Alternatives to PBKF2

The first alternative called Bcrypt was introduced back in 1999 used in PHP and OpenBSD.

Uses some protection against GPU attacks, but has fixed memory usage and is not safe in long term

Alternatives to PBKF2

Another alternative is called Scrypt. It was introduced in 2009 and standardized in 2016 as RFC 7914. It allows for adjustment of both, time- and memory-based parameters.

However, it has an issue called time-memory tradeoff, due to which, when Scrypt performs more computations (e.g. more time), it uses less memory. With right parameters, it is possible to achieve constant memory usage, making it vulnerable to the same issue that Bcrypt and PBKDF2 have.

Alternatives to PBKF2

With these concerns in mind, there was an open competition in 2013 called "Password Hashing Competition". As a result, in 2015, the new hashing function called Argon2 was created.

It allows to separately set up time, memory costs, as well as parallelism degree, making it secure against GPU-based attacks.

Argon2 was standardized in RFC 9106 in 2021.

Conclusion

Despite the availability of more secure and modern alternatives, such as Argon2, PBKDF2 is still widely used in many critical systems, including WPA/WPA2 encryption, GRUB2, Winrar, Linux Unified Key Setup, VeraCrypt, etc.

It will take time until PBKDF2 will become obsolete, and until then it is important to understand its weaknesses and advantages to identify whether its usage is acceptable or will it compromise the system.

References

[1] B. Kaliski, A. Rusch, and K. Moriarty, PKCS #5: Password-based Cryptography Specification version 2.1, Jan. 2017. doi:10.17487/rfc8018

[2] P. A. Grassi et al., Digital Identity Guidelines: Authentication and lifecycle management, Jun. 2017. doi:10.6028/nist.sp.800-63b

[3] D. Temoshok et al., Digital Identity Guidelines:Authentication and Authenticator Management, Jul. 2025. doi:10.6028/nist.sp.800-63b-4

[4] A. Visconti, O. Mosnáček, M. Brož, and V. Matyáš, "Examining PBKDF2 security margin—case study of Luks," Journal of Information Security and Applications, vol. 46, pp. 296–306, Jun. 2019. doi:10.1016/j.jisa.2019.03.016

[5] "Password hashing competition," Password Hashing Competition, https://www.password-hashing.net/ (accessed Sep. 28, 2025).

References (contd.)

[6] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "RFC 9106," RFC 9106: Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications, https://www.rfc-editor.org/rfc/rfc9106 (accessed Sep. 28, 2025).

[7] H. Choi and S. C. Seo, "Optimization of PBKDF2 using HMAC-sha2 and HMAC-LSH families in CPU environment," IEEE Access, vol. 9, pp. 40165–40177, 2021. doi:10.1109/access.2021.3065082

[8] B. Kaliski, "PKCS #5: Password-based cryptography specification version 2.0," RFC Editor, https://www.rfc-editor.org/rfc/rfc2898.html (accessed Sep. 28, 2025).

[9] N. Vettivel, "Securing passwords using hashing," Medium, https://nishothan-17.medium.com/securing-passwords-using-hashing-8ce558e14b6d (accessed Sep. 28, 2025).

[10] S. Nakov, "Mac and key derivation," Practical Cryptography for Developers, https://cryptobook.nakov.com/mac-and-key-derivation (accessed Sep. 28, 2025).

References (contd.)

[11] D. Chatterjee, “Cryptographically secure pseudo-random number: Introduction,” Medium, https://medium.com/@cozy03/cryptographically-secure-pseudo-random-number-introduction-5b6f19d20ae7 (accessed Sep. 28, 2025).

[12] Bitwarden, “Bitwarden Security whitepaper,” Bitwarden, https://bitwarden.com/help/bitwarden-security-white-paper/#tab-onboarding-2VYGcnxLwmYH4J977Xd38H (accessed Sep. 28, 2025).

[13] Arjen et al., “Bitwarden design flaw: Server side iterations,” Almost Secure, https://palant.info/2023/01/23/bitwarden-design-flaw-server-side-iterations/ (accessed Sep. 28, 2025).

[14] C. Neskey, “Examining the LastPass breach through our password table,” Hive Systems, https://www.hivesystems.com/blog/examining-the-lastpass-breach-through-our-password-table (accessed Sep. 28, 2025).