WIKIPEDIA
The Free Encyclopedia

Salt (cryptography)

In cryptography, a **salt** is random data that is used as an additional input to a one-way function that hashes data, a password or passphrase.[1] Salts are used to safeguard passwords in storage. Historically, only the output from an invocation of a cryptographic hash function on the password was stored on a system, but, over time, additional safeguards were developed to protect against duplicate or common passwords being identifiable (as their hashes are identical).[2] Salting is one such protection.

A new salt is randomly generated for each password. Typically, the salt and the password (or its version after key stretching) are concatenated and fed to a cryptographic hash function, and the output hash value (but not the original password) is stored with the salt in a database. Hashing allows later authentication without keeping and therefore risking exposure of the plaintext password if the authentication data store is compromised. Salts don't need to be encrypted or stored separately from the hashed password itself, because even if an attacker has access to the database with the hash values and the salts, the correct use of said salts will hinder common attacks.[3]

Salts defend against attacks that use precomputed tables (e.g. rainbow tables),[4] as they can make the size of table needed for a successful attack prohibitively large without burdening users.[5] Since salts differ from one another, they also protect weak (e.g. commonly used, re-used) passwords, as different salted hashes are created for different instances of the same password.[3]

Cryptographic salts are broadly used in many modern computer systems, from Unix system credentials to Internet security.

Salts are closely related to the concept of a cryptographic nonce.

Example usage

Here is an incomplete example of a salt value for storing passwords. This first table has two username and password combinations. The password is not stored.

Username	Password
user1	password123
user2	password123

The salt value is generated at random and can be any length; in this case the salt value is 16 bytes long. The salt value is appended to the plaintext password and then the result is hashed, which is referred to as the hashed value. Both the salt value and hashed value are stored.

Username	Salt value	String to be hashed	Hashed value = SHA256 (Password + Salt value)
user1	D;%yL9TS:5PalS/d	**password123**D;%yL9TS:5PalS/d	9C9B913EB1B6254F4737CE947EFD16F16E916F9D6EE5C1102A2002E48D4C88BD
user2	)<,-<U(jLezy4j>*	**password123**)<,-<U(jLezy4j>*	6058B4EB46BD6487298B59440EC8E70EAE482239FF2B4E7CA69950DFBD5532F2

As the table above illustrates, different salt values will create completely different hashed values, even when the plaintext passwords are exactly the same. Additionally, dictionary attacks are mitigated to a degree as an attacker cannot practically precompute the hashes. However, a salt cannot protect common or easily guessed passwords.

Without a salt, the hashed value is the same for all users that have a given password, making it easier for hackers to guess the password from the hashed value:

Username	String to be hashed	Hashed value = SHA256
user1	**password123**	57DB1253B68B6802B59A969F750FA32B60CB5CC8A3CB19B87DAC28F541DC4E2A
user2	**password123**	57DB1253B68B6802B59A969F750FA32B60CB5CC8A3CB19B87DAC28F541DC4E2A

Common mistakes

Salt re-use

Using the same salt for all passwords is dangerous because a precomputed table which simply accounts for the salt will render the salt useless.

Generation of precomputed tables for databases with unique salts for every password is not viable because of the computational cost of doing so. But, if a common salt is used for all the entries, creating such a table (that accounts for the salt) then becomes a viable and possibly successful attack.[6]

Because salt re-use can cause users with the same password to have the same hash, cracking a single hash can result in other passwords being compromised too.

Salt length

If a salt is too short, an attacker may precompute a table of every possible salt appended to every likely password. Using a long salt ensures such a table would be prohibitively large.[7][8]

Benefits

To understand the difference between cracking a single password and a set of them, consider a file with users and their hashed passwords. Say the file is unsalted. Then an attacker could pick a string, call it `attempt[0]`, and then compute `hash(attempt[0])`. A user whose hash stored in the file is `hash(attempt[0])` may or may not have password attempt[0]. However, even if `attempt[0]` is *not* the user's actual password, it will be accepted as if it were, because the system can only check passwords by computing the hash of the password entered and comparing it to the hash stored in the file. Thus, each match cracks a user password, and the chance of a match rises with the number of passwords in the file. In contrast, if salts are used, the attacker would have to compute `hash(attempt[0] || salt[a])`, compare against entry A, then `hash(attempt[0] || salt[b])`, compare against entry B, and so on. This prevents any one attempt from cracking multiple passwords, given that salt re-use is avoided.[9]

Salts also combat the use of precomputed tables for cracking passwords.[10] Such a table might simply map common passwords to their hashes, or it might do something more complex, like store the start and end points of a set of precomputed hash chains. In either case, salting can defend against the use of precomputed tables by lengthening hashes and having them draw from larger character sets, making it less likely that the table covers the resulting hashes. In particular, a precomputed table would need to cover the string `[salt + hash]` rather than simply `[hash]`.

The modern shadow password system, in which password hashes and other security data are stored in a non-public file, somewhat mitigates these concerns. However, they remain relevant in multi-server installations which use centralized password management systems to push passwords or password hashes to multiple systems. In such installations, the root account on each individual system may be treated as less trusted than the administrators of the centralized password system, so it remains worthwhile to ensure that the security of the password hashing algorithm, including the generation of unique salt values, is adequate.

Another (lesser) benefit of a salt is as follows: two users might choose the same string as their password. Without a salt, this password would be stored as the same hash string in the password file. This would disclose the fact that the two accounts have the same password, allowing anyone who knows one of the account's passwords to access the other account. By salting the passwords with two random characters, even if two accounts use the same password, no one can discover this just by reading hashes. Salting also makes it extremely difficult to determine if a person has used the same password for multiple systems.[11]

Unix implementations

1970s–1980s

Earlier versions of Unix used a password file `/etc/passwd` to store the hashes of salted passwords (passwords prefixed with two-character random salts). In these older versions of Unix, the salt was also stored in the passwd file (as cleartext) together with the hash of the salted password. The password file was publicly readable for all users of the system. This was necessary so that user-privileged software tools could find user names and other information. The security of passwords is therefore protected only by the one-way functions (enciphering or hashing) used for the purpose. Early Unix implementations limited passwords to eight characters and used a 12-bit salt, which allowed for 4,096 possible salt values.[12] This was an appropriate balance for 1970s computational and storage costs.[13]

1980s–

The shadow password system is used to limit access to hashes and salt. The salt is eight characters, the hash is 86 characters, and the password length is effectively unlimited, barring stack overflow errors.

Web-application implementations

It is common for a web application to store in a database the hash value of a user's password. Without a salt, a successful SQL injection attack may yield easily crackable passwords. Because many users re-use passwords for multiple sites, the use of a salt is an important component of overall web application security.[14] Some additional references for using a salt to secure password hashes in specific languages or libraries (PHP, the .NET libraries, etc.) can be found in the external links section below.

See also

- Password cracking
- Cryptographic nonce
- Initialization vector
- Padding
- "Spice" in the Hasty Pudding cipher
- Rainbow tables
- Pepper (cryptography)

References

1. Fenton, James L.; Grassi, Paul A.; Garcia, Michael E. (June 2017). "NIST Special Publication 800-63-3" (https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-3.pdf) (PDF). *NIST Technical Series Publications*.
2. Anderson, Ross (2020). *Security engineering: a guide to building dependable distributed systems* (Third ed.). Indianapolis, Indiana. ISBN 978-1-119-64281-7. OCLC 1224516855 (https://www.worldcat.org/oclc/1224516855).
3. Rosulek, Mike (January 3, 2021). "Chapter 11: Hash Functions" (https://joyofcryptography.com/pdf/chap11.pdf) (PDF). *The Joy of Cryptography*. pp. 204–205.
4. Godwin, Anthony (10 September 2021). "Passwords Matter" (http://bugcharmer.blogspot.com/2012/06/passwords-matter.html). *The Bug Charmer* (Blog). Retrieved 2016-12-09.
5. Boneh, Dan; Shoup, Victor (January 4, 2020). *A Graduate Course in Applied Cryptography* (https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_5.pdf) (PDF). pp. 693–695.
6. "Secure Salted Password Hashing - How to do it Properly" (https://crackstation.net/hashing-security.htm). *crackstation.net*. Retrieved 2021-03-19.
7. Menezes, Alfred J.; Oorschot, Paul C. van; Vanstone, Scott A. (1997). *Handbook of Applied Cryptography*. CRC Press. p. 288. ISBN 0-8493-8523-7.
8. "Secure Salted Password Hashing - How to do it Properly" (https://crackstation.net/hashing-security.htm#salt).
9. "Password Storage - OWASP Cheat Sheet Series" (https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html). *cheatsheetseries.owasp.org*. Retrieved 2021-03-19.
10. "How Rainbow Tables work" (http://kestas.kuliukas.com/RainbowTables/). *kestas.kuliukas.com*.
11. Stallings, William; Lawrie Brown (2015). *Computer security: principles and practice* (Third ed.). Boston. ISBN 978-0-13-377392-7. OCLC 874734678 (https://www.worldcat.org/oclc/874734678).
12. Morris, Robert; Thompson, Ken (1978-04-03). "Password Security: A Case History" (https://web.archive.org/web/20130821093338/http://cm.bell-labs.com/cm/cs/who/dmr/passwd.ps). *Bell Laboratories*. Archived from the original (http://cm.bell-labs.com/cm/cs/who/dmr/passwd.ps) on 2013-08-21.
13. Simson Garfinkel; Gene Spafford; Alan Schwartz (2003). "How Unix Implements Passwords" (https://www.safaribooksonline.com/library/view/practical-unix-and/0596003234/ch04s03.html). *Practical UNIX and Internet Security* (3rd ed.). O'Reilly Media. ISBN 9780596003234.
14. "ISC Diary – Hashing Passwords" (https://www.dshield.org/diary.html?storyid=11110). Dshield.org. Retrieved 2011-10-15.

External links

- Wille, Christoph (2004-01-05). "Storing Passwords - done right!" (http://www.aspheute.com/english/20040105.asp).
- OWASP Cryptographic Cheat Sheet (https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cryptographic_Storage_Cheat_Sheet.md)
- how to encrypt user passwords (http://www.jasypt.org/howtoencryptuserpasswords.html)