Passwords

Cryptology (1)

Martin Stanek

2025

KI FMFI UK Bratislava

Content

- passwords for authentification
- storing passwords, time-memory trade-off
- key derivation functions
- HOTP and TOPT

Introduction

- the most frequent authentication method
  - alone or combined with other methods (something you know/have/are)
- constructions for confidentiality / integrity, such as
  - protocols for authentication and key agreement using shared secret
  - protection of private keys stored in files
- some problems with passwords:
  - default passwords (can be found on the Internet)
    - a global problem (DDoS attacks, IoT, Mirai etc.)
  - low entropy of a password, easy to guess
  - strong passwords hard to remember
  - passwords stored insecurely, e.g. in cleartext
  - passwords sent via insecure channel, e.g. telnet
  - shared among systems (it worsens the impact of a successful attack)

Attacks and policies

- attacks:
  - brute-force search
  - dictionary attacks
  - precomputation (for example rainbow tables)
- password policy, for example:
  - password length, “diversity” of characters (groups) used in a password
  - max./min. password age
  - checking password history, login name or other public account data
  - block account after x unsuccessful login attempts
  - delays after unsuccessful login attempts, etc.
- choosing a password
  - randomly generated (hard to remember … password managers (?))
  - user chosen (predictability, similarity with other passwords etc.)
  - phrase-derived passwords, …

The most common passwords

- How not to choose a password
- source: NordPass, Top 200 Most Common Passwords, 2024 Edition
  - personal and corporate lists

1. 123456
2. 123456789
3. 12345678
4. password
5. qwerty123
6. qwerty1
7. 111111
8. 12345
9. secret
10. 123123
11. 1234567890
12. 1234567
13. 000000
14. qwerty
15. abc123
16. password1
17. iloveyou
18. 11111111
19. dragon
20. monkey

Entropy of passwords chosen by users

- estimates of entropy according to NIST SP 800-63-2 (94 character alphabet):

length	no checks (bits)	rules * (bits)
6	14	23
8	18	30
10	21	32
20	36	42
40	56	62

(*) dictionary tests and composition rule (character groups)

- expect worse reality; NIST overestimated the security of passwords

Entropy of passwords – other estimates

- estimating entropy is not easy
  - substring from a dictionary, number sequences, personal information, …
- various methods implemented
  - providing feedback on password strength to users
  - web sites, password managers, specialized applications, …
- comparison of KeePassXC, zxcvbn library (bit security):

password	KeePassXC	zxcvbn
`12345678`	1.58	2.00
`dragon1`	6.32	9.54
`u39;2$mMiaEJ2`	77.90	43.19
`shipfumeshead`	32.54	32.13

NIST SP 800-63-3 Digital Identity Guidelines

- NIST SP 800-63-2 is superseded by the SP 800-63 suite
- NIST SP 800-63B-4: Digital Identity Guidelines, Authentication and Authenticator Management (2025)
- requirements for passwords, few examples:
  - 1FA $\Rightarrow$ length ≥ 15; MFA $\Rightarrow$ length ≥ 8
  - Verifiers SHALL NOT impose other composition rules (e.g., requiring mixtures of different character types).
  - Verifiers SHALL NOT require subscribers to change passwords periodically. However, verifiers SHALL force a change if there is evidence that the authenticator has been compromised.
  - Verifiers SHALL NOT permit the subscriber to store a hint that is accessible to an unauthenticated claimant.
- Appendix A with a discussion on the strength of passwords

Storing passwords

cleartext
- database/file leak ⇒ all passwords compromised
- passwords readable by admin, from backups etc.

password hash: $H(p)$
- equal passwords ⇒ equal hashes
- precomputed hashes applicable for various systems

hash of the password and a *salt*: $H(p \parallel s)$
- salt – random, not necessary secret
- fast H ⇒ fast brute-force

"slow" hashing of password and salt: $H^c(p \parallel s)$
- iteration count c – to slow down the computation c-times, e.g., $c = 1\,000$
- verification: 2 ms ↦ 2 seconds (acceptable?)
- attack: 10 days ↦ 27.4 years (sufficient?)

Time-Memory trade-off

Time and memory for password searching

- assumption: $h = H(p)$, attacker knows h
- N – size of the password space
- brute-force: time $T \approx N$, memory $M \approx 1$, no precomputation needed
  - example: SHA-1, random alphanumeric (62 characters) password of length 10
    Apple M3 Pro GPU $\approx$ 3 000 MH/s $\Rightarrow$ 8.87 years
    Nvidia RTX 5090 $\approx$ 70 000 MH/s $\Rightarrow$ 138.8 days
- precomputation of all hashes (only once, time $\approx N$), subsequent search in the table: $T \approx 1, M \approx N$
  - example: SHA-1, alphanumeric passwords of length 10 … 33 572 PB (passwords and hashes, i.e. $62^{10} \cdot (10 + 20)$ bytes)

Time and memory for password searching (2)

- time-memory trade-off (TMTO)
  - applicable for inverting any function
  - computing a preimage of a hash function
  - finding a key
    - in a block cipher: $f(x) = E_x(m)$ (K/CPA)
    - for MAC: $f(x) = \mathrm{Mac}_x(m)$, etc.

Hellman's TMTO for passwords

$$
\begin{array}{ccccccccccc}
p_{1,1} & \xrightarrow{H} & h_{1,1} & \xrightarrow{g} & p_{1,2} & \cdots & \xrightarrow{H} & h_{1,t-1} & \xrightarrow{g} & p_{1,t} & \xrightarrow{H} & h_{1,t} \\
p_{2,1} & \xrightarrow{H} & h_{2,1} & \xrightarrow{g} & p_{2,2} & \cdots & \xrightarrow{H} & h_{2,t-1} & \xrightarrow{g} & p_{2,t} & \xrightarrow{H} & h_{2,t} \\
\vdots & & & & & \vdots & & & & & & \vdots \\
p_{m,1} & \xrightarrow{H} & h_{m,1} & \xrightarrow{g} & p_{m,2} & \cdots & \xrightarrow{H} & h_{m,t-1} & \xrightarrow{g} & p_{m,t} & \xrightarrow{H} & h_{m,t}
\end{array}
$$

Hellman's TMTO for passwords

$$
\begin{array}{ccccccccccc}
p_{1,1} & \xrightarrow{H} & h_{1,1} & \xrightarrow{g} & p_{1,2} & \cdots & \xrightarrow{H} & h_{1,t-1} & \xrightarrow{g} & p_{1,t} & \xrightarrow{H} & h_{1,t} \\
p_{2,1} & \xrightarrow{H} & h_{2,1} & \xrightarrow{g} & p_{2,2} & \cdots & \xrightarrow{H} & h_{2,t-1} & \xrightarrow{g} & p_{2,t} & \xrightarrow{H} & h_{2,t} \\
\vdots & & & & \vdots & & & \vdots & & & & \vdots \\
p_{m,1} & \xrightarrow{H} & h_{m,1} & \xrightarrow{g} & p_{m,2} & \cdots & \xrightarrow{H} & h_{m,t-1} & \xrightarrow{g} & p_{m,t} & \xrightarrow{H} & h_{m,t}
\end{array}
$$

- store $\langle p_{i,1}, h_{i,t} \rangle_{i=1}^{m}$ sorted/indexed by the second coordinate
- inverting H:
  1. for $i = 0, 1, \ldots, t-1$: test for $(H \circ g)^i(h)$ in the last column
  2. after a match, say $(H \circ g)^i(h) = h_{r,t}$, we compute $p = (g \circ H)^{t-1-i}(p_{r,1})$ ("false alarms" possible)
- memory $M \approx m$; time $T \approx t$ (on-line), precomputation $\approx mt$

Hellman's TMTO – covering the password space (1)

- the attack can find only those passwords that are in some chain
- if $g \circ H$ is a single-cycle permutation on password space, then we have the TMTO with $TM = N$ (unrealistic)
- usually the mapping behaves like a random mapping
  - collisions: prob. increases for increasing number of elements in the table
  - chains can cycle or merge

Hellman's TMTO – covering the password space (2)

- problem: it is hard to cover more than N/t elements in a single table
  - let's assume a covering of $mt \geq N/t$ elements and we add another chain
  - probability that no collision with already covered elements occurs:

$$\Pr < \left(\frac{N - N/t}{N}\right)^t = (1 - 1/t)^t \approx \frac{1}{e}$$

  … and the probability lowers further with increasing covering

- single table can be used for approx. N/t elements

Hellman's TMTO – covering the password space (3)

- solution: use t independent tables (for distinct choices of g)
- experimental results: if $mt^2 \approx N$, then each table covers approx. $0.8 \cdot mt$ elements
- probability of success for t tables is approx.

$$1 - (1 - 0.8 \cdot mt/N)^t \approx 1 - \left(1 - \frac{0.8 \cdot mt^2/N}{t}\right)^t$$
$$\approx 1 - e^{-0.8mt^2/N} \approx 1 - e^{-0.8} \approx 0.55$$

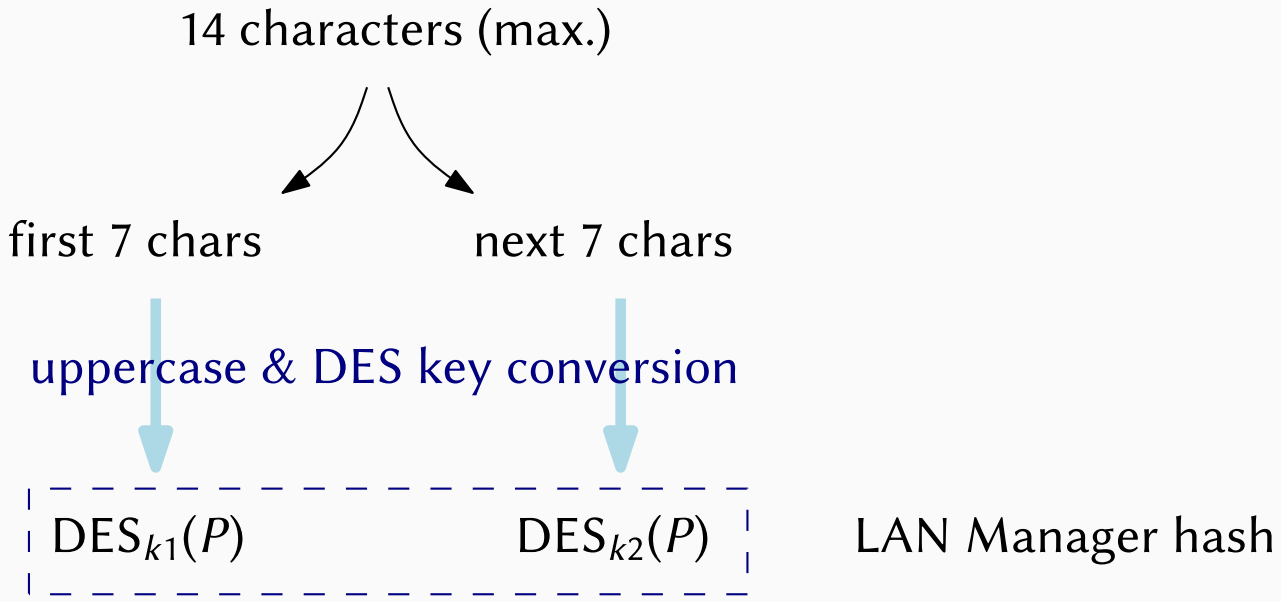
- time (on-line) $T \approx t^2$, memory $M \approx mt$

Hellman's TMTO – covering the password space (4)

- we want to cover N elements, i.e., $mt \cdot t \approx N$;
- TMTO curve: $TM^2 \approx t^4 m^2 \approx N^2$
  - interesting point on the curve: $T = M = N^{2/3}$ ($t \approx m \approx N^{1/3}$)
  - example: SHA-1, alphanumeric password of length 10 – approx. 4.9 minutes (Apple M3 Pro GPU, no time for lookups counted); 546 GB (a pair counted as 20+10 bytes)
- improvements:
  - distinguished points – fixed part of values in the last column, e.g., first d bits are zero, thus reducing table lookups (disk operations)
  - rainbow tables – distinct g_i for each column: reduction of collision probability, more costly search, single table; overall constant-time speedup
    example: ophcrack – cracking Windows LAN Manager passwords

How not to store passwords (1)

- LAN Manager hash (Windows, P is a fixed plaintext)



- each half can be attacked independently

How not to store passwords (2)

- Adobe hack (2013)
  - encrypted passwords, 3DES in ECB mode (single key)
- Ashley Madison (2015)
  - bcrypt, some passwords with additional MD5 hash
- plaintext passwords (2019)
  - Google, Facebook, Twitter, etc.

Key derivation functions

Passwords used for cryptographic constructions

- PKCS #5 v 2.1 (RFC 8018) Password-Based Cryptography Specification
  - derivation of symmetric keys from passwords (encryption, MAC)
  - password checking (non-standardized, just a note in RFC)
  - PBKDF2 (Password based key derivation function)
    - input: password P, salt S, iteration count c, output length d (in bytes)
- salt
  - random bit string of sufficient length (for example 128), secrecy not required
  - potentially many different keys for a single password
  - makes precomputation of keys for dictionary passwords useless ⇒ the attacker must wait for the salt value
  - deterministic alternative for random generation of the salt: KDF(P, M), where M is the message to be processed (not secure if message space is small)
- iteration count (makes the brute-force attack harder)
  - increase the work factor for computation, min. 1000 recommended

PBKDF2

- output: $T_1 \parallel T_2 \parallel \ldots$ (as needed)
- max. output length $\left(2^{32} - 1\right) \cdot H_l$, where H_l is the length of underlying h.f.'s output
  - 80 GB for SHA-1
- computation: $T_i = F(P, S, c, i)$, where

$$F(P, S, c, i) = U_1 \oplus U_2 \oplus \ldots \oplus U_c$$

$U_1 = \mathrm{PRF}(P, S \parallel \mathrm{INT}(i))$ INT returns 4-byte value

$U_2 = \mathrm{PRF}(P, U_1)$

...

$U_c = \mathrm{PRF}(P, U_{c-1})$

- standard PRF is HMAC-SHA-1: $\mathrm{PRF}(a, b) = \mathrm{HMAC}_a(b)$
  - HMAC-SHA-256 and similar constructions are commonly used as well

- C. Persival (2009)
- idea: make the brute-force even harder
  - password cracking easy to parallelize
  - GPU, custom ASIC (Application Specific Integrated Circuit)
  - PBKDF2 – small memory
  - large memory requirements increase the circuit area (and its price)
- the attacker can choose:
  - moderate time and (relatively) large memory requirements
  - small memory and large time requirements
- another memory-hard alternative: Argon2
  - Biryukov, Dinu, Khovratovich (2015)

scrypt – theory: ROMix (1)

- ROMix(B, N) (sequential memory-hard function)

  parameters: hash function H with k bit output
  Integerify function, bijection $\{0,1\}^k \to \{0, \ldots, 2^k - 1\}$

  input: B – bit string of k bits
  N – work factor ($N < 2^{k/8}$)

  computation: $V_i = H^i(B)$ for $0 \leq i < N$
  $X = H^N(B)$
  for $i = 0, \ldots, N - 1$:
  $j = \text{Integerify}(X)$
  $X = H(X \oplus V_j)$
  return X

- pseudorandom order of accessing V_j values

scrypt – theory: ROMix (2)

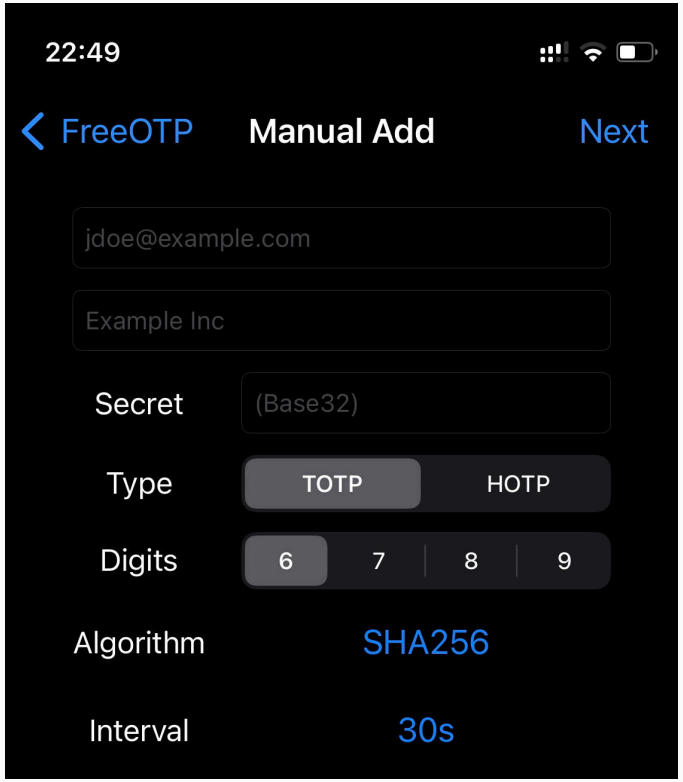
- real scrypt specification – RFC 7914
  - scryptROMix and scryptBlockMix functions, scryptROMix is a variation of ROMix, parameters (work factor N, block size, parallelism)
- instantiation: PBKDF2-HMAC-SHA256, Salsa20/8 core

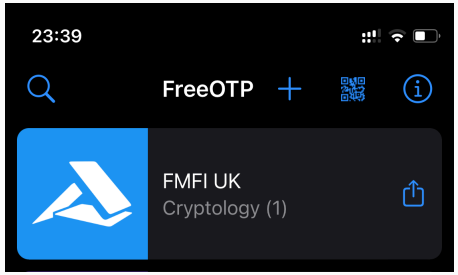
HOTP and TOTP

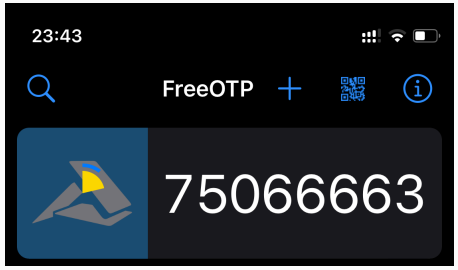
One-time passwords

- multifactor authentication, 2-step verification, …
  - something you know/have/are
  - often mobile phone: SMS, push notifications, authenticator app
- one-time passwords
- HOTP and TOTP
  - HOTP: HMAC-Based One-Time Password Algorithm (RFC 4226)
  - TOTP: Time-Based One-Time Password Algorithm (RFC 6238)

FreeOTP example








```
otpauth://totp/FMFI%20UK:Cryptology%20(1)?secret=ONUG65LMM
RRGKYTFOR2GK4TUNBQW45DINFZTCMRT&algorithm=SHA256&digits=8
&period=30&lock=false
```

HOTP

- actors: HOTP generator (client), HOTP validator (server)
- $\mathrm{HMAC}_K(\cdot)$, default based on SHA-1
- parameters:
  - K – shared secret (static symmetric key, ≥ 128 bits)
  - C – counter value (8B, synchronized, starts with 0)
  - Digits – output length (≥ 6)

$$\mathrm{HOTP}(K, C) = \mathrm{Truncate}(\mathrm{HMAC}_K(C))$$

- Truncate – transform HMAC output to HOTP value
  - focus on uniformity and implementation clarity
- client increments C, and then calculates the next HOTP value
- server recalculates and compares received HOTP value
  - server increments C after a successful authentication

HOTP – remarks

- authentication protocol over a secure channel, such as TLS, IPsec
- security of shared secret is important (obviously)
- validation failure (HOTP values do not match)
  - resynch protocol (look-ahead window)
  - look-ahead parameter s – server validates against s consecutive values
  - if unsuccessful → failed attempt
- brute-force attack prevention
  - brute-force attack is, in theory, the best attack possible
  - throttling parameter – the maximum number of failed attempts
- in some scenarios, server can request multiple HOTP values
- bidirectional authentication possible

TOTP

- extension of HOTP: counter value C replaced by time
  - short-lived OTP values (instead of valid until next successful authentication“)
- HMAC based on SHA-1 (default), SHA-256, SHA-512
- parameters:
  - X – time step in seconds (usually $X = 30$ seconds)
  - time – current Unix time (seconds since 1.1.1970)
  - $T = \lfloor \text{time}/X \rfloor$ – number of time steps

$$\text{TOTP}(K, T) = \text{HOTP}(K, T)$$

TOTP – remarks

- time step size: security vs. usability
- “one-time only” requirement: the server must not accept the second attempt after the successful validation
- delay window – accept TOTP value from the previous time step
  - time when the value was entered vs. time when it is validated
  - recommended 1 time step
- resynchronization
  - clock drift
  - server can set limits on forward and backward time drifts
  - remember the drift and adjust for next validation

Exercises

1. *Find a method for storing passwords used by your favorite Linux distribution.*
2. *What is the maximum time step the NIST SP 800-63B-4 requires for TOTP?*
3. *Find the least secure and the most secure alphanumeric passwords of length 2 according to the `zxcvbn` library.*