Passwords

Martin Stanek

Department of Computer Science
Comenius University
stanek@dcs.fmph.uniba.sk

Cryptology 1 (2021/22)

Content

Passwords – introduction

- the most frequent authentication method
  - alone or combined with other methods
    (something you know/have/are)
- constructions for confidentiality / integrity, e.g.
  - protocols for authentication and key agreement using shared secret
  - protection of private keys stored in files
- some problems with passwords:
  - default passwords (e.g. can be found on the Internet)
    a global problem (DDoS attacks 2016, 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)

Passwords – introduction (2)

- attacks:
  - brute-force search
  - dictionary attacks
  - precomputation (e.g. 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, . . .

Worst passwords (1)

- October 2013; Adobe (data of 38 million active users)
- overall more than 130 million accounts/passwords (incl. inactive)
- top 20 passwords (educated guess, helpful “hints”, ECB mode):

1.	123456	(≈ 1.9 million)	11.	1234567890	
2.	123456789	(≈ 446 thousand)	12.	000000	
3.	password	(≈ 345 thousand)	13.	abc123	
4.	adobe123	(≈ 211 thousand)	14.	1234	
5.	12345678		15.	adobe1	
6.	qwerty		16.	macromedia	
7.	1234567		17.	azerty	
8.	111111		18.	iloveyou	
9.	photoshop		19.	aaaaaa	
10.	123123		20.	654321	

The most frequent passwords

How not to choose passwords
source: NordPass, based on breached passwords in 2020 (more than 275 million passwords)

1. 123456
2. 123456789
3. picture1
4. password
5. 12345678
6. 111111
7. 123123
8. 12345
9. 1234567890
10. senha
11. 1234567
12. qwerty
13. abc123
14. Million2
15. 000000
16. 1234
17. iloveyou
18. aaron431
19. password1
20. qqww1122
21. 123
22. omgpop
23. 123321
24. 654321
25. qwertyuiop

Entropy of passwords?

- estimates of entropy for user chosen password 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)

- one can expect worse situation when humans select passwords (NIST overestimated the security of passwords)
- entropy is not the best approach to measure password strength

  Weir et al. (2010): Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords

Entropy of passwords – other estimates

- estimation of 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 KeePass a zxcvbn library (bit security):

password	KeePass	zxcvbn
`qwerty`	12	2.32
`password1`	8	7.57
`JE38bslk@psl`	67	39.86
`spidersarecoolandfun`	72	50.66

NIST SP 800-63-3 Digital Identity Guidelines

- NIST SP 800-63-2 is superseded by the SP 800-63 suite (2017)
  - SP 800-63-3 Digital Identity Guidelines
  - SP 800-63A Enrollment and Identity Proofing
  - SP 800-63B Authentication and Lifecycle Management
  - SP 800-63C Federation and Assertions
- no entropy estimates, only guidelines given (length preferred over complexity rules)
- for example, see Appendix A in NIST SP 800-63B:
  Users should be encouraged to make their passwords as lengthy as they want, within reason. . . .

  Length and complexity requirements beyond those recommended here significantly increase the difficulty of memorized secrets and increase user frustration. As a result, users often work around these restrictions in a way that is counterproductive. Furthermore, other mitigations such as blacklists, secure hashed storage, and rate limiting are more effective at preventing modern brute-force attacks. Therefore, no additional complexity requirements are imposed.

How strong are real passwords?

- 2012 LinkedIn
  - password hashes leaked – approx. 6,5 million users
  - SHA-1, no salt used
  - approx. 60% passwords broken
  - experiment (F. Pesce):
    - dictionary attack, no GPU or specialized HW used, no rainbow table
    - 4 hours – recovery of approx. 900 thousand users passwords
    - continuing the attack . . . approx. 2 million passwords compromised
  - May 2016 – story continues
    - 167 million accounts, 62 million unique hashes
    - KoreLogic:

      2 hours . . . 65% hashes cracked
      1 day . . . 78% hashes cracked (86% accounts)
      2 days . . . 80% hashes cracked

Storing passwords (informal discussion)

- cleartext
  - database/file leak $\Rightarrow$ all passwords compromised
  - passwords readable by admin, from backups etc.
- password hash: $H(p)$
  - equal passwords $\Rightarrow$ equal hashes
  - precomputed hashes "applicable" for various systems/installations
- hash of the password and a "salt" $H(p \,||\, s)$
  - salt – random string (for each password), not necessary secret
  - hash function speed $\Rightarrow$ fast brute-force (many passwords can be tested in short time)
- "slow" hashing of password and salt $H^c(p \,||\, s)$
  - iteration count c – to slow down the computation of the hash c-times, e.g. $c = 1\,000$
  - password verification: for example 2 ms vs. 2 seconds (acceptable?)
  - attack: for example 10 days vs. more than 27 years (sufficient?)

Time and memory for password searching

- assumption: $h = H(p)$, attacker knows h
- N – size of the password space
- trivial attacks:
  - brute-force: time $T \approx N$, memory $M \approx 1$, no precomputation needed

    example: SHA-1, random alphanumeric (62 characters) password of length 8, 14.5 million hashes/s (i7-2600 @ 3.40 GHz) ≈ 174 days
    using GPU is much better, e.g. single Nvidia GTX 1080 runs about 8500 million SHA-1 hashes/s ≈ 7 hours

  - precomputation of all hashes (only once, time ≈ N), subsequent search in the table: $T \approx 1$, $M \approx N$
    example: SHA-1, alphanumeric passwords of length 8 . . . 6114 TB (passwords and hashes, i.e. $62^8 \cdot (8 + 20)$ 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 $f(x) = \mathrm{Mac}_x(m)$, for stream cipher f maps the key and IV into a running key

Hellman's TMTO for passwords – idea

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

- 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 for passwords – idea

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

- 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 for passwords – covering the 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
- 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:

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

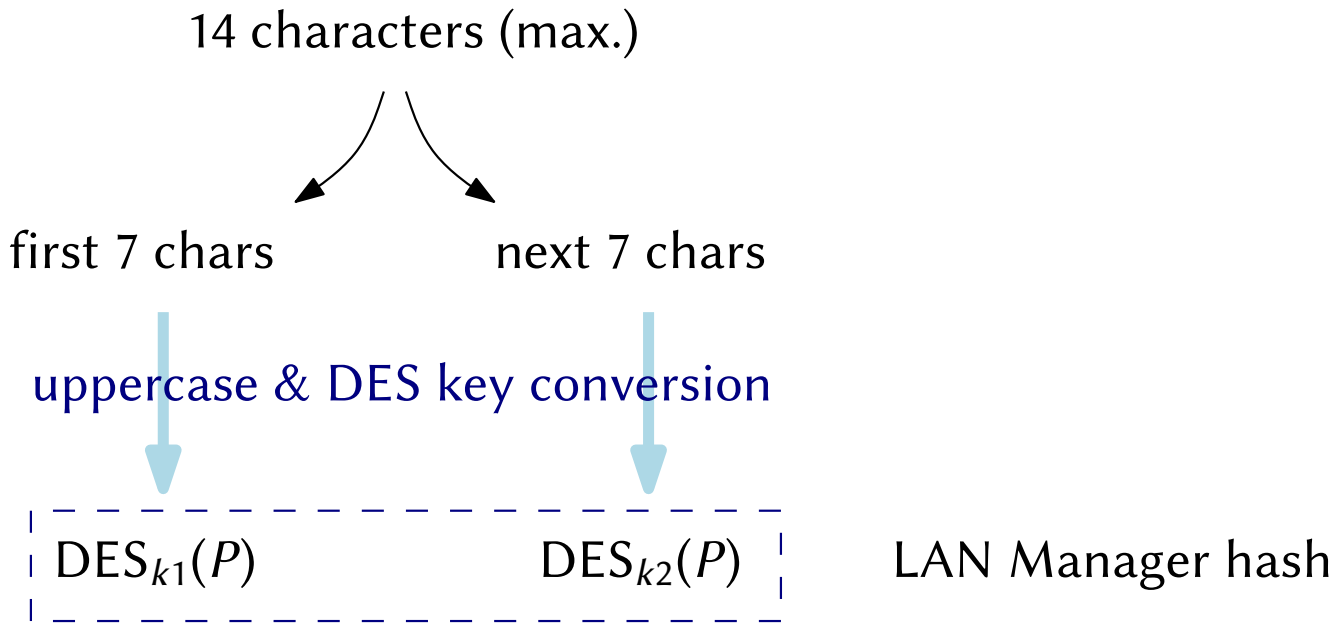
  …and the probability lowers further with increasing covering
- single table can be used for approx. N/t elements

Hellman's TMTO for passwords – covering the space (2)

- 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, and the prob. of success for t tables is approx. $1-(1-0.8mt/N)^t \approx 1-(1-(0.8mt^2/N)/t)^t \approx 1-e^{-0.8mt^2/N} \approx 1-e^{-0.8} \approx 0.55$
- time (on-line) $T \approx t^2$, memory $M \approx mt$
- 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 8 – approx. 4.2 minutes (no time for lookups counted); 101.5 GB (a pair counted as 20+8 bytes)
- improvements:
  - distinguished points – fixed part of values in the last column, e.g. first d bits are zero, thus reducing table lookups (i.e. 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) $\Rightarrow$ equal passwords map into equal ciphertexts
- block length 8 bytes (passwords divided into blocks), i.e. equal blocks mapped into equal ciphertext blocks
- password hints leaked as well – easy to guess some passwords

How not to store passwords (3)

- 000Webhost (2015)
  - webhosting service, 13 million plaintext passwords
- Ashley Madison (2015)
  - data leak (10 GB compressed)
  - 36 million accounts
  - bcrypt used to store passwords (iteration count 2^{12}, salt)
  - 15.26 million accounts additional plain MD5 hash stored $\Rightarrow$ more than 11 million passwords cracked in 10 days

How not to store passwords (4)

random examples of storing plaintext passwords (2019)

- Facebook (03/2019)
  - Facebook Lite (primary), Facebook, Instagram
  - “hundreds of millions” of user passwords (since 2012)
  - searchable by employees
- Google (05/2019)
  - “some portion” of G Suite users (since 2005)
- Twitter (05/2019)
  - more than 330 million users (entire user base)
  - bug (found internally) - storing plaintext passwords in an internal log
- Robinhood (07/2019)
  - commission-free stock trading platform
  - “some users” affected

Password used for cryptographic constructions

- PKCS #5 v 2.1 (RFC 8018) Password-Based Cryptography Specification
- various use of passwords:
  - 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 (e.g. 64), secrecy not required
  - potentially many different keys for a single password
  - makes precomputation of keys for dictionary passwords useless $\Rightarrow$ 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 if message space is small)
- iteration count (makes the brute-force attack harder)
  - increase the work factor for function computation, min. 1000 recommended in the RFC 8018 (based on NIST SP 800-132)

PBKDF2

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

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

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

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

$$\ldots$$

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

- 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
- alternatives to PBKDF2: bcrypt (based on Blowfish block cipher, frequently used), scrypt (increasing memory requirements)

scrypt

- 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

scrypt – theory: ROMix

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

  parameters: h.f. 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 $0 \le i < N$
  $X = H^N(B)$
  iterate for $i = 0,\ldots,N-1$:
  j = Integerify(X)
  $X = H(X \oplus V_j)$
  return X

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