Hesla

Problémy:
- jednoduchá hesla lze snadno uhodnout (brute-force, slovníky...)
- složitá hesla se těžko pamatují
- uživatelé jsou lidé ⇒ papírky s hesly, tedy heslo na více místech
- jak nastavit politiku hesel?

A co když ze serveru unikne DB uživatelů s hesly? ~ problém trvalý tomuto

- hesla hešujeme ... ale útočník stále může provést brute-force útok offline
- předpočítané tabulky hešů, 1. pokus:
  - heslo → hash → jiné heslo
    - f funkce vzorkující prostor možných hesel
    - případ fce f
  - řetězky vzniklé iterováním f: start → → → → konec
    - N hodnot
    - prostor hesel pokryjí řetězky
    - z každého si uložím začátek a konec
    - z neznámého hše zkusím N kroků, než se trefím do konce řetězku; pak najdu začátek a od něj ... předchůdce hše.
    - v ideálním případě zmenším paměťové nároky tabulky N-krát za cenu N-krát pomalejšího hledání
  - problém: řetězky srůstají ⇒ k 1 konci může patřit víc začátků, R řetězků pokryje $\ll N \cdot R$ hesel.
- duhové tabulky (Rainbow tables)
  - v každém kroku řetězku použiji jinou vzorkovací f-ci – "barvy"
  - pokud se 2 řetězky potkají, brzy se zase rozejdou ⇒ téměř disj.
  - při hledání potřebuji zkoušet všechny možné pozice v řetězku ⇒ hledání zpomalují N^2-krát, paměť redukují cca N-krát.

Příklad: project-rainbowcrack.com
pro SHA-1, ASCII, 1-8 znaková hesla: 460 GB tabulka

Obrana proti brute-force útokům:
- "solení" hesel: hesla hešují s noncí, tu uložím spolu s hešem
  ⇒ z hešů nepoznám, že 2 hesla jsou stejná
  ⇒ duhové tabulky nepomohou

- iterování hešů: nám 1000× zpomalení ověřování hesla neradí útočníkovi zato zdarma :(
  - ↳ tzv. key stretching – iterováním hešů také můžeme vyrobit PRNG seedovaný heslem a získat tak z hesla klíč vhodné délky
- jiný přístup: neukládáme část nonce, takže musíme zkoušet :(
  - ↳ tzv. key strengthening

→ Příklad: PBKDF2 (Password-Based Key Derivation Function)

- odvození z libovolné PRF (pseudonáhodná fce s klíčem, typicky HMAC)
- výstup: $B_1 B_2 B_3 \dots$ (podle požadované délky výstupu)

přičemž: $B_i = U_1^i \oplus U_2^i \oplus \dots \oplus U_c^i$ ← # iterací

$U_{j1}^i = PRF(password, salt \| i)$

$U_{j,n}^i = PRF(password, U_{j,n-1})$

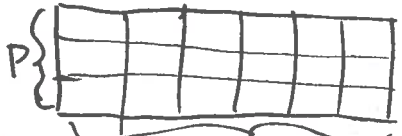
↑ pokud je moc dlouhé, tak jeho heš (tím vznikne z ekviv. hesla)

Další vývoj: snažíme se zkomplikovat paralelizaci na GPU/FPGA
⇒ typicky zvýšením požadavků na paměť
(to vše jen kvůli slabým heslům, silná nepotřebují ani iterovat)

↓

Argon2 (náčrtek)

Parametry: M = množství paměti
P = stupeň paralelismu
T = # iterací



1KB bloky tak, aby jich bylo celkem M paměti

Vstupy: heslo
sůl
(taj. klíč, asociovaná data)

ne moc blízkým, aby fungovala parelelizace

- na začátku vyplníme první 2 sloupce heši vstupů a parametrů
- ∀ iterace postupně vyplní všechny sloupce, výsl. přixoruje k pův. obsahu
- používá kompresní fci (1KB, 1KB) → 1KB odvozenou z Blake2 (to je heš odvozený z ChaCha20 ze soutěže SHA-3)

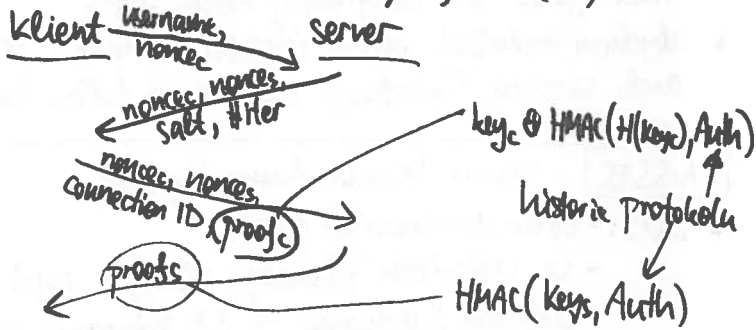
↑ nepravinné

komprimuje blok doleva od akt. (cyklicky) s vybraným předch. blokem.

Varianty:
- deterministicky (podle PRNG)
- v závislosti na datech z levého bloku

Interaktivní autentikace heslem (4A)

- **challenge-response:** server pošle nonci a sůl, klient hash(heslo || sůl || nonce)
  - ... riziko: server si musí pamatovat plain-textová hesla
    - ... nebo jejich hashe, ale ty pak musí poznat i klient ⇒ nepanuje
- **protokol SCRAM** (Salted Challenge-Response Authentication Mechanism)
  - určím sůl a #iterací
  - z hesla odvodím K_c = HMAC(heslo, "Client key") a K_s = HMAC(heslo, "Server key") — PBKDF2(heslo, sůl, #iter.)
  - server si pamatuje: username, sůl, #iterací, K_s, hash(K_c)
  - průběh:



  - pokud znám H(key$_c$), mohu rekonstruovat key$_c$ (pak ho chci rychle zahodit...)

Kerberos – distribuovaná správa klíčů pomocí sym. kryptografie [MIT 1988]

- protokolu se účastní "principálové" – klienti a servery
  - Ticket Granting Service – má s každým principálem společný taj. klíč
  - když A chce mluvit s B, pořídí si tiket T_{AB}:
    - A pošle TGS: $\{B, time\}_{K_{A,TGS}}$
    - TGS vyrobí session key K_{AB} a pošle A: $\{K_{AB}\}_{K_{A,TGS}}$, $\{T_{AB}\}_{K_{B,TGS}}$
      kde tiket $T_{AB} = (B, \{A, \text{adresa } A, \text{time range}, K_{AB}\}_{K_B})$
    - A přepošle T_{AB} ⟷ B
    - B případně pošle $\{time\}_{K_{AB}}$, aby se také autentikoval
    - dále už šifrujeme pomocí K_{AB} zprávy mezi A, B.

! potřebujeme synchronizované hodiny aspoň ± pár minut
↳ jak to udělat bezpečně? — po které si pamatujeme pakety

Ve skutečnosti: (Kerberos v5)

- TGS je služba jako každá jiná ⇒ klient pro ni má také tiket (TGT)
- místo klíčů z tiketů (které mohou mít dlouhou životnost) používáme autentikátory pro specifické sessions:

$A_{A,B} = \{$ID A, timestamp, session key$\}_{K_{AB}}$ ← z tiketu

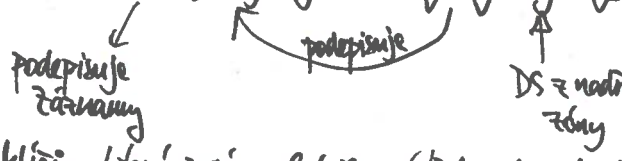
↳ na 1 použití – během replay okna si pamatuji všechny, co jsem viděl

- klient se může prvotně autentikovat heslem: ~~autorizační server~~ autentikační umí vydat TGT zašifrovaný hešem hesla
- abychom zabránili offline útokům na hesla: preautentikace – pošlu auth. serveru timestamp zašifrovaný hešem hesla

---

DNSSEC – Secure Domain Name System

- DNS: - strom doménových jmen
  - ve vrcholech záznamy různých typů (A, AAAA, MX, ...)
  - delegace subdomén → NS záznamy, zóny vs. domény
- každá zóna obsahuje klíč (záznam DNSKEY)
- klíčem podepisujeme záznamy (pro jméno + typ → záznam RRSIG)
- nadřazená zóna podepíše klíč podřízené (záznam DS, kde je NS)
- můžeme používat více klíčů: - rotace klíčů
  - zone-signing key vs. key-signing key



- "root of trust" – množina klíčů, které známe lokálně (třeba od root zóny)
- zónu stačí podepsat offline a pak jen servírovat hotové RRSIGy
- Jak se podepisuje neexistence záznamu? Podepisujeme díry!

X.Y.Z NSEC X'.Y.Z ⟨typy záznamů pro X.Y.Z⟩
↳ nejbližší další jméno v kanonickém pořadí