Password storing functions bcrypt and scrypt

How do they work?

Table of content

Table of content

Saving passwords

- **User has to be authenticated, how?**
- **Store something (only) he knows**
  - Plain text. Bad, everyone knows it immediately, easily transferable to other sites
  - Hashed secret. Better, but still easily attackable with rainbow tables
  - (repeated) hashed secret with random data. Pretty good, rainbow table attacks are very hard
- **But we can make it even harder**
- **We can make it deliberately slow/memory consuming**

Key derivation functions[1]

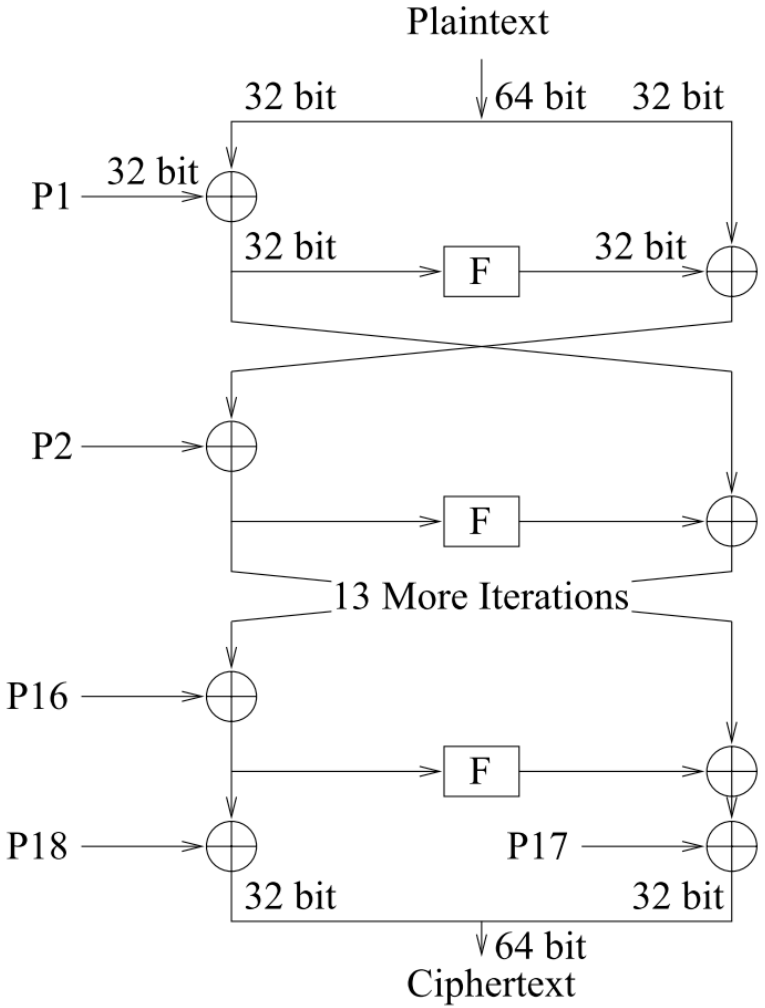
- KDF(password, salt)
- Salt prevents attacks with pre-computed password tables (rainbow-tables)
- Attacks still possible with salt and result
- Good KDF should have adaptable runtime
- It should make extracting information as hard as guessing

Blowfish[2] [3]

- Encryption algorithm from Bruce Schneier
- Uses 18 32-bit subkeys P_i and 4 arrays with 256 32-bit subkeys S_i derived from encryption key
- Encrypts data in 64-bit blocks over 16 rounds
- P and S arrays are called state

Blowfish

- $F(a,b,c,d)=((S_1[a] + S_2[b]) \text{ XOR } S_3[c]) + S_4[d] \bmod 2^{32}$
- $R_i = L_{i-1} \text{ XOR } P_i$
- $L_i = R_{i-1} \text{ XOR } F(R_i)$



Eksblowfish[2]

- Blowfish with additionally cost and salt
- Key-setup phase similar to Blowfish key-setup
- Result depends on password and salt
- Hard to precompute
- cost = number of rounds
- salt = 128 random bits
- password = up to 56 bytes

Eksblowfish

EksBlowfishSetup $(cost, salt, key)$
  $state \leftarrow$ InitState ()
  $state \leftarrow$ ExpandKey $(state, salt, key)$
  repeat (2^{cost})
    $state \leftarrow$ ExpandKey $(state, 0, salt)$
    $state \leftarrow$ ExpandKey $(state, 0, key)$
  return *state*

- **InitState copies digits of π into P then S**

Eksblowfish

- ExpandKey XORs P with encryption key
- Encrypts first 64 bit of salt and replaces P_1 and P_2
- XOR with second 64 bit of salt
- Second half of salt encrypted with new P values
- Replaces P_3 and P_4 and XORed with first 64 bit of salt
- ...
- The same with arrays S

Bcrypt[2]

- Uses password and salt to encrypt magic value
- Uses eksblowfish to setup state
- Attackable with parallel computing, FPGAs, ASICs
- Calculation needs 4KB of RAM

Bcrypt

bcrypt (*cost*, *salt*, *pwd*)
  state ← EksBlowfishSetup (*cost*, *salt*, *key*)
  ctext ← “OrpheanBeholderScryDoubt”
  repeat (64)
    ctext ← EncryptECB (*state*, *ctext*)
  return Concatenate (*cost*, *salt*, *ctext*)

- **ECB is electronic codebook**
- **Every block is encrypted only with the same key**

PBKDF2[4]

- PBKDF2(p, s, c, dkLen)
- $F = U_1$ XOR U_2 XOR ... XOR U_c
- $U_1 = H(p, s \,||\, int(a))$
- $U_i = H(p, U_{i-1})$
- $T_i = F(p, s, c, i)$
- $dk = T_1 \,||\, T_2 \,||\, T_l$<0..r-1>

- p = Password
- s = Salt
- c = Number of rounds
- r = Number of bytes in last block
- dklen = Output length in bytes

HMAC[5]

- Used to authenticate message (integrity and authenticity)
- Uses same key for signing and verifying, unlike signature
- K' derived from K by expanding it with 0x00 or hashing
- HMAC(K', M)=H((K' XOR 0x5c...) || H((K' XOR 0x36...) || M))
- HMAC_SHA256 is HMAC with SHA256 as hash

ROMix[1]

- ROMix(B, N)
- B = Input
- N = Work metric
- X = B
- N times: $V_i = H(V_{i-1})$
- N times: $X = H(X \text{ XOR } V_{\text{int(X) mod N}})$
- output = X

SMix[1]

- $SMix_r(B, N)$
- B = Input
- r = Block size parameter
- $X = B_{2r-2}$
- 2r times: $X = H(X \text{ XOR } B_i)$
- And $Y_i = X$
- output = $Y_0, Y_2, \ldots, Y_{r-2}, Y_1, Y_3, \ldots, Y_{2r-1}$

Scrypt[1]

- Scalable computation time and memory consumption
- scrypt(P, S, N, r, p, dkLen)
- P = Password
- S = Salt
- N = Number of iterations in ROMix
- r = Block size parameter (memory consumption)
- p = Parallelization parameter (computation cost)
- dkLen = output length

Scrypt

- **MFLen = length of SMix block in bytes**
- **B_0, ..., B_{p-1} = $PBKDF2_{HMAC\_SHA256}$(P, S, 1, p * MFLen)**
- **p-1 times: B_i = $SMix_r$(B_i, N)**
- **dk = $PBKDF2_{HMACH\_SHA256}$(P, B_0 || ... || B_{p-1}, 1, dkLen)**

Scrypt

- Scrypt has better scaling properties
- It seems bcrypt still more implementations
- Even after 17 years bcrypt has no real weaknesses
- Still recommended to use scrypt for new projects

Examples

https://github.com/Tarsnap/scrypt

uint8_t buffer[64];

crypto_scrypt(“pleaseletmein”, 13, “SodiumChloride”, 14, 1048576, 8, 1, buffer, 64)

buffer =

21 01 cb 9b 6a 51 1a ae ad db be 09 cf 70 f8 81

ec 56 8d 57 4a 2f fd 4d ab e5 ee 98 20 ad aa 47

8e 56 fd 8f 4b a5 d0 9f fa 1c 6d 92 7c 40 f4 c3

37 30 40 49 e8 a9 52 fb cb f4 5c 6f a7 7a 41 a4

Examples

- **https://www.npmjs.com/package/scrypt**

  var res = scrypt.hashSync("pleaseletmein", {"N":1048576, "r":8, "p":1}, 64, "SodiumChloride");

  console.log("Result: "+res.toString("hex"));

  21 01 cb 9b …

Further reading

- **Scrypt successor? Argon2[6] won the Password Hashing Competition 2015[7]**
- **Secure Remote Password Protocol[8]**

Thank you!

- **Any questions?**

daniel@bilanovic.de

PGP: A969 EC2D 2714 9F0B CF64 4906 DFEB AAEC 1E68 90F9

Sources

- [1] https://www.tarsnap.com/scrypt/scrypt.pdf
- [2] https://www.openbsd.org/papers/bcrypt-paper.pdf
- [3] https://www.schneier.com/academic/blowfish/
- [4] https://tools.ietf.org/html/rfc2898#section-5.2
- [5] https://tools.ietf.org/html/rfc2104
- [6] https://password-hashing.net/argon2-specs.pdf
- [7] https://password-hashing.net/
- [8] http://srp.stanford.edu/ndss.html

Additional links

- **How Facebook stores passwords: http://video.adm.ntnu.no/pres/54b660049af94**
- **How Dropbox stores passwords: https://blogs.dropbox.com/tech/2016/09/how-dropbox-securely-stores-your-passwords/**