Password KDF and Disk encryption

Alessandro Barenghi

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB)
Politecnico di Milano

alessandro -dot- barenghi - at - polimi -dot- it

April 20, 2017

Password Storage

How to store a password

- Scenario: common password authentication scheme: the server needs to verify the password provided by the client through knowing the correct one
- Currently, one of the most widespread authentication scenarios
- Straightforward way: The server stores the cleartext password of the client and matches it against the one provided
- In case of a breakthrough all the credentials of the users are disclosed and perfect impersonation is possible!
- Cleartext password storage should never be used, in favour of more proper storage methods

Password Storage

Hashes and Bruteforcing

- The most straightforward solution is to store the value of a strong hash of the actual user password on the server
- Upon authentication, the user sends the password, which gets hashed by the server and checked against the stored hash: if they match the user is granted access
- An attacker with access to the stored hashes must find a valid password which hashes the same as one of the stored hashes to gain access to the system (first preimage attack)
- This is reasonably computationally demanding, and can be made worse by iterating a large number of times the hash function: instead of storing $H(\texttt{pwd})$, store $H^n(\texttt{pwd})$ (typically $n \approx 1000$)

Password Storage

Time-To-Memory Trade-off

- Storing $H^n(\texttt{pwd})$ raises the computational cost to bruteforce passwords over a large keyspace
- It becomes more appealing to employ a computation-storage tradeoff (i.e. store some precomputed values) to enhance the password breaking speed
- Straightforward approach: store a large number of precomputed $H^n(\cdot)$ hashes together with their input in a search-efficient structure (f.i. Red-Black/B+/B* Tree) and lookup the hash value
- Provided the structure stores hashes for the whole input space, the success probability is 1, i.e. the technique is deterministic
- Main issue: high disk space requirements: such a structure for 7-letter alphanumeric passwords and $H^n(\cdot) = \mathrm{SHA\text{-}1}^n(\cdot)$ is $\approx 57.6\,TB$!

Password Storage

Time-To-Memory Trade-off: Rainbow Tables

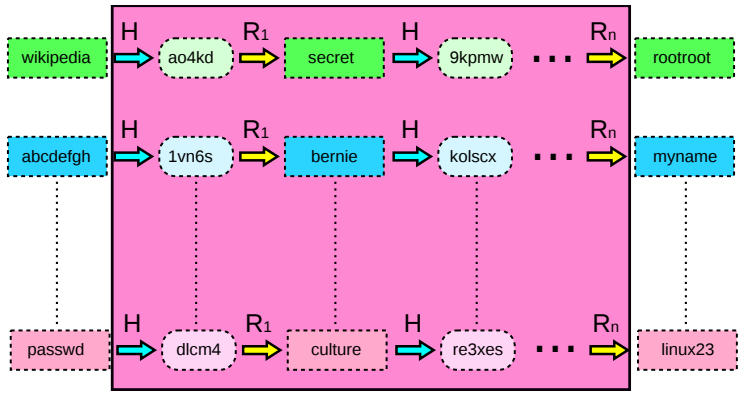
- The best known time to memory tradeoff technique for password bruteforcing are the so-called Rainbow Tables
- The key idea is somehow close to the one of the BSGS algorithm (and thus similar speedups can be expected)
- **Key Idea**: The attacker picks at random a password and iteratively computes its hash. Since the output of a hash is (usually) not a valid password, one or more so-called *reduction* function R_i are employed to map an hash output into a valid password to be re-hashed
- In this way, the attacker obtains a long chain looking like

$$\texttt{pwd}_1 \rightarrow H(\texttt{pwd}_1) \rightarrow R_1(H(\texttt{pwd}_1)) = \texttt{pwd}_2 \rightarrow H(\texttt{pwd}_2) \ldots \rightarrow \texttt{pwd}_n$$

- The attacker stores $\texttt{pwd}_1$ and $\texttt{pwd}_n$ in a lookup table, known as Rainbow table (as the different chains are usually marked with different colors)

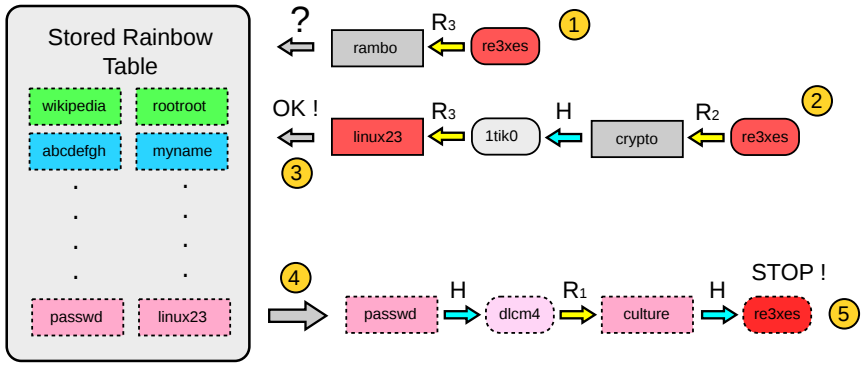
Password Storage

RT Structure



Password Storage

RT Lookup



Password Storage

Rainbow Tables: assumptions

- The reduction function(s) are usually simple truncations of the hash output, followed by a trivial mapping to printable ascii characters (f.i. dropping the 8th bit, or simply employing a map on the desired character set for the passwords)
- The key assumption is that iterating the hash-reduce procedure will perform a random walk over all the possible values for the password, thus yielding a complete table
- As this is not exactly true, the Rainbow Tables are **not guaranteed to succeed** in breaking passwords; however, given a sufficiently large table (i.e. one where (number of chains $\times$ chain length)$\approx$ keyspace), the probability comes close enough to 1 (i.e. $> 99.9\%$) to be of practical use.
- The length of the chains is picked depending on the available storage: longer chains yield smaller tables with a longer lookup computation

Password Storage

Salting

- How to avoid time-to-memory tradeoff attacks? Randomize the result of the hash employing a salt!
- A *salted hash* is the result of storing $H(\texttt{pwd}||\texttt{salt})$, where `salt` is a small random value
- As the `salt` is hashed together with the password, the same password will never hash to the same value, thus effectively hindering the construction of lookup/rainbow tables for hashed values, as an attacker would need to tabulate the whole keyspace$\times\lceil log_2(\texttt{salt})\rceil$
- The `salt` is usually stored together with the salted hash, as its role is only to hinder the precomputation
- Salting does not strengthen in any way the password choice: choosing "password" as password is still silly!

Password Storage

Salting Case Study - Unix Crypt

- POSIX `crypt()` function is one of the most diffused salted hash primitives, and is widely used to store password in a variety of Unices
- The first edition employed a modified Enigma simulator as hashing function and was **not** salted (strongly deprecated)
- The second edition replaced it with 35 runs of the DES algorithm, employing the salt as the key and a combination of the password bytes as the plaintext (deprecated)
- The third instance employed 1000 runs of MD5 and a salt mixing scheme which involved alternating salt and password bytes as the input of the MD5 primitive, without reinitializing the hash state (deprecated as bruteforce speeds reach 600kKeys/s on modern GPUs)
- A modern revision employs 3000 runs of SHA-256/512

Password Storage

A more methodic approach

- The current *standardized* state of the art for key derivation is the Password Based Key Derivation Function 2 (PBKDF2) [**?**]
- The PBKDF2 derives a key k, of length l_k starting from an arbitrary length password `pwd` and a salt s, with a given work factor c
- The core of the PBKDF2 is a 2-parameters pseudorandom function H (e.g. HMAC(text,key)) with output length l_h
- The derived key is built as $i = \lceil \frac{l_k}{l_h} \rceil$ tiles t_i obtained as

$$s_0 = s||i, \quad \forall j, 0 < j < (c-1),\ s_j = H(s_{j-1}, \texttt{pwd}) \qquad t_i = \bigoplus_{j=0}^{c} s_j$$

- For instance, WPA-2-PSK uses PBKDF2 with H =HMAC-SHA1, `pwd` =passphrase, s =ssid, $c = 4096$ and $l_k = 256$

Password Storage

Is this enough?

- The security margin of previous hashes is in terms of mandating a given amount of **computation** per password
- While it is a good stopgap measure, GPU/ASIC password breakers:
  - Exploit pipelining to extract parallelism from KDF computations
  - Exploit technology scaling to fit more pipelines per area (=cost)
- Raising the work factor of the previous KDFs does not hinder these speedups
- Can we devise a KDF explicitly tackling this?

Preventing efficient password breaking

Memory hard algorithms

- In an ideal KDF the computation work factor should be linked to the area implementing it
- The best approach to do so is to bind the **computational** complexity of the KDF to the **spatial** complexity
  - Building memory in dedicated password breakers increases area linearly

Definition

Memory Hard algorithm (RAM computation model): A *memory hard* algorithm $\mathcal{A}$ on a RAM, with $T_{\mathcal{A}}(n)$ time and $S_{\mathcal{A}}(n)$ space requirements, is characterized by $S_{\mathcal{A}}(n) \in \Omega(T_{\mathcal{A}}(n)^{1-\varepsilon})$

Preventing efficient password breaking

Sequential memory hard algorithms

- A memory hard function is perfect to drive raise the area requirements for a single KDF bruteforcing unit
- However, it does not prevent the exploitation of the internal parallelism to build pipelined units

Definition

A *Sequential memory hard function* is an algorithm $\mathcal{A}$, computable in $T_{\mathcal{A}}^{\text{RAM}}(n)$ on a RAM, which can be computed on a PRAM in $T_{\mathcal{A}}^{\text{PRAM}}(n)$ time and $S_{\mathcal{A}}^{\text{PRAM}}(n)$ space no faster than

$$O(T_{\mathcal{A}}^{\text{RAM}}(n)^2) = S_{\mathcal{A}}^{\text{PRAM}}(n) \cdot T_{\mathcal{A}}^{\text{PRAM}}(n)$$

Preventing efficient password breaking

ROMix An example of sequential memory hard function

Given a hash function $H(\cdot)$ with output length l_h, a l_h-bit input string p, an integer work factor $N \leq 2^{l_h/8}$, $\text{ROMix}(p, N)$ is a memory hard function:

- $\texttt{tmp} \leftarrow p$
- for $i = 0$ to $N - 1$ do
  - $v[i] \leftarrow \texttt{tmp}$
  - $\texttt{tmp} \leftarrow H(\texttt{tmp})$
- for $i = 0$ to $N - 1$ do
  - $j \leftarrow \texttt{tmp} \bmod N$
  - $\texttt{tmp} \leftarrow H(\texttt{tmp} \oplus v[j])$
- return tmp

Preventing efficient password breaking

Putting it all together

- Employing ROMix together with a way to derive its l_h-bit input p from a password and a salt, we obtain an algorithm for sequential memory hard KDF:
  1. $p \leftarrow \text{PBKDF2}(\texttt{password}, \texttt{salt}, 1, l_h)$
  2. $o \leftarrow \text{ROMix}(p, N)$
  3. return $\text{PBKDF2}(\texttt{pwd}, o, 1, l_h)$
- The work factor N of this construction tunes both the computation and the memory required
- The original proposal, scrypt [**?**], allowed also to run multiple instances of ROMix in parallel, changing the output lenght of the PBKDF2 in (1)

Preventing efficient password breaking

Effectiveness figures for 130nm ASIC - $ to break a password in a year

KDF	8 ASCII char	40-char passphrase
DES Crypt	< 1\$	< 1\$
MD5 Crypt	130\$	1.4k\$
PBKDF2-SHA256 (0.1s)	18k\$	200k\$
PBKDF2-SHA256 (5.0s)	4.8M\$	52M\$
scrypt-SHA256 (0.06s)	920k\$	10M\$
scrypt-SHA256 (3.8s)	19B\$	210B\$

- scrypt is significantly more effective in preventing efficient bruteforcing in both interactive login ($< 0.1s$ work factor) and KDF ($\approx 5s$) settings

Encrypting data at rest

Problem statement

- Encryption of data in mass memory (a.k.a. data at rest) may take place at:
  - **Single File**: a single file is encrypted with a proper symmetric cipher
    - `gpg` has a symmetric file encryption mode with a password-derived key
  - **Filesystem overlay**: contents of all the files of a FS are encrypted, no metadata encryption
    - `ecryptfs` provides a FUSE-based FS encryption overlay (home encryption in Ubuntu)
  - **Block level**: the entire disk, or logical volume is encrypted transparently w.r.t. the FS
    - Bitlocker (Windows), `dm-crypt` (linux), FileVault (MacOS), and Ciphershed (a Truecrypt fork, cross-platform)
- In this lesson we will tackle disk (or volume/partition/slice) encryption

Encrypting data at rest

Problem statement

- An encryption scheme for data at rest (i.e. in mass memory) requires:
  - Confidentiality: encrypted data should not be distinguishable from random data, even if vast amounts of ciphertext (2^{60}B+) are available
  - Integrity: the encryption scheme should provide a way to spot alterations in the stored data
- Symmetric encryption employing a password derived key is the way data-at-rest encryption is handled
- There is a need to devise a proper mode of operation to fit our desiderata

Encrypting data at rest

IEEE P1619

- The state of the art standardized data-at-rest encryption scheme is IEEE P1619, which uses the XTS mode of operation
- The XTS mode of operation is the result of the combination of two strategies:
  - Xor-Encrypt-Xor (XEX) mode of operation [**?**] devised by Phil Rogaway to have an efficient encryption/decryption scheme
  - CipherText Stealing (CTS) is employed to avoid padding at the end of the plaintext, as it may not be feasible (end-of-disk)

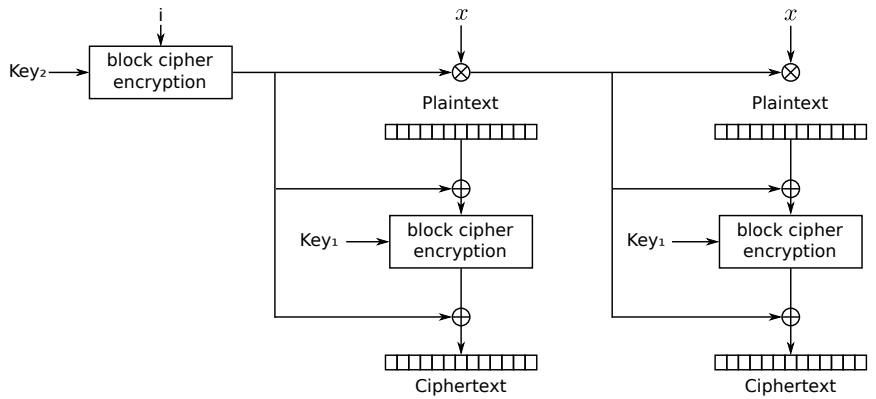
Encrypting data at rest

Xor-Encrypt-Xor (XEX) Mode of operation - 1

- The Xor-Encrypt-Xor mode of operation evolves the ECB performing plaintext and ciphertext whitening, and employs two symmetric keys
- To generate the pads for each cipher block (16B for AES) in a physical disk block (usually 512-8192 B) XEX does the following:
  1. Encrypt under k_2 the physical disk block index i and use the ciphertext pad_1 as a whitening pad for the first cipher block
  2. Derive the next pad considering the bits of pad_1 as coefficients of a polynomial over $\mathbb{Z}_{2^{128}}$ (modulo $x^{128} + x^7 + x^2 + x + 1$), and multiply them by x (i.e. left shift pad_1 by 1 bit and reduce as needed)
- Typically the key for a XEX block mode is split into two halves, one to ECB-encrypt the data and the other to generate the pads
  - e.g., to encrypt and generate pads with AES-128 you need a 256b key

Encrypting data at rest

Xor-Encrypt-Xor (XEX) Mode of operation - 2

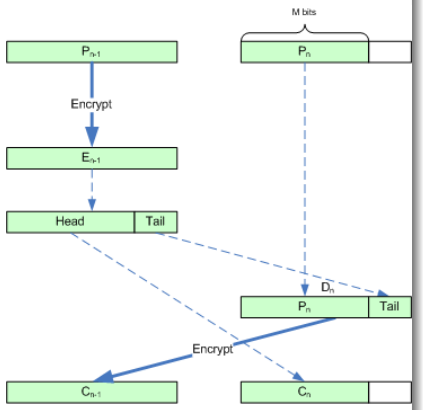


Encrypting data at rest

Ciphertext Stealing

- To solve the issue of the possible need for padding, i.e. `ptx_len` mod `block_len` $= m \neq 0$, ciphertext stealing goes as follows.

1. Encrypt the penultimate plaintext block, obtaining c_{n-1}
2. Pad the final plaintext block with `block_len`-m bytes of c_{n-1}
3. Encrypt the padded final block, and store it as the penultimate **ciphertext** block
4. Store the first m bytes of c_{n-1} as the final, reduced size ciphertext block
5. To decrypt, start decrypting the penultimate block and roll back



Bibliography I