MTAT.07.003 Cryptology II
Spring 2012 / Exercise session ?? / Example Solution

Exercise (Security of standard password hashing). *Salted hashing is common mechanism for storing passwords. Each user is first granted an identifier* id*. Every time the password is hashed, the identifier* id *is appended to it and then a system-wide hash function is used to compute the digest. The authentication is successful if the digest coincides with the digest stored in* `passwd` *file.*

1. *Formalise the system by using abstract primitives for appropriate actions. Describe the functional requirements that are needed for seamless authentication. Compare the formalisation with the lottery system described in the first exercise.*
2. *Define an attack scenario where the attacker tries to reverse engineer passwords from salted hashes. Formalise the corresponding security condition. Can this security condition be met in practice?*
3. *Extend the attack description to the setting where the identifier* id *depends on the identity of a user. What does such a design choice give to the attacker? Modify the corresponding attack scenario and derive the corresponding security requirement.*

Solution. Formally, the password authentication consists of two main functions $\mathsf{SumbitHash}$ and $\mathsf{VerifyPasswd}$ where $\mathsf{SubmitHash} : \mathcal{X} \times \mathcal{I} \to \mathcal{C}$ and $\mathsf{VerifyPasswd} : \mathcal{X} \times \mathcal{I} \times \mathcal{C} \to \{0, 1\}$ such that

$$\forall x \in \mathcal{X}\ \forall \mathsf{id} \in \mathcal{I} : \quad \mathsf{VerifyPasswd}(x, \mathsf{id}, \mathsf{SubmitHash}(x, \mathsf{id})) \equiv 1\ .$$

In practice, both functions are implemented by using a single hash function of type $h : \mathcal{X} \times \mathcal{I} \to \mathcal{D}$ where $\mathcal{D}$ is the digest space, $\mathcal{X}$ is password space and $\mathcal{I}$ is the set of all possible identities. The hash function with the complex domain $\mathcal{X} \times \mathcal{I}$ is usually constructed from a (moderately slow) general purpose hash function of type $f : \{0,1\}^* \to \{0,1\}^{256}$ by using a special envelope for representing elements of $\mathcal{X} \times \mathcal{I}$ as bit strings. The most simplest way is of course concatenation of x and id denoted by $x\|\mathsf{id}$. However, there are many better alternatives, as well. Hence, we abstract away exact details how a general purpose hash function is converted to a hash function with dedicated domain and state all notions in terms of dedicated hash functions.

The function $\mathsf{SumbitHash}$ is commonly implemented as $h(x, \mathsf{id})$ and thus we must evaluate a predicate $h(x, \mathsf{id}) = y$ in order to test whether x is the correct password for the identity id and the digest y. In that sense, password based authentication is completely equivalent to the lottery system described above.

The first attack scenario, where an attacker tries to reverse passwords from q hashes, can be modelled as the following game

```
$\mathcal{G}_1^{\mathcal{A}}$
    $x_1, \ldots, x_q \xleftarrow{u} \mathcal{M}$
    $\mathsf{id}_1 \ldots, \mathsf{id}_q \xleftarrow{u} \mathcal{I}$
    for $i \in \{1, \ldots, q\}$ do $y_i \leftarrow h(x_i, \mathsf{id}_i)$
    $(x_*, k) \leftarrow \mathcal{A}(h(y_1, \ldots, y_q)$
    return $[h(x_*, \mathsf{id}_k) = y_k]$
```

provided that all users generate passwords randomly and identities are also randomly assigned. The game models an attack without specific target identity given access to q password hashes $y_1, \ldots, y_q$. One can make the game more realistic by specifying a distribution of passwords—a huge dictionary of words together with their occurrence probabilities. Of course, the complete specification of dictionary distribution is usually out of reach and thus one often only postulates certain aspects of the distribution, such as the total amount of passwords and maximal occurrence probability (min-entropy).

In many practical systems, the password length is limited to eight alphanumeric symbols. As a result, the total size of the password space is $(2 \cdot 26 + 10)^8 \lesssim 2.2 \cdot 10^{14}$. This space can be completely traversed in 692 computer years assuming that we can compute $10,000$ evaluations of hash function in a second on an ordinary computer. If an adversary uses a moderate botnet of size $10,000$ then the whole space can be looked through in 25 days. Consequently, hashes in password files provide only a moderate level security.

More importantly, it is almost impossible to increase security margin in large-scale systems. First of all, average users are not motivated or cannot memorise reliably passwords with more characters. Second, we cannot make the hash function much slower, as servers must handle thousand login attempts in a second.

Hash salting only assures that one has to relaunch the attack when he or she needs to compromise different identity. If the salt id depends on the user name then one might be able to compute single password-hash table once and use it to attack many independent systems. Under our assumptions the full size of such dictionary is around 6400 Terabytes, which is a lot but tractable for large governmental agencies. Hence, it is not a good idea if a user `root` has the same salt in many systems.

More formally, if the salt depends on the identity, then a corrupted system administrator might allure users to take specific user names. In the most severe case, the adversary can directly specify salts for all users, which leads to the following security game:

```
$\mathcal{G}_2^{\mathcal{A}}$
    $\mathsf{id}_1 \ldots, \mathsf{id}_q \leftarrow \mathcal{A}$
    $x_1, \ldots, x_q \xleftarrow{u} \mathcal{M}$
    for $i \in \{1, \ldots, q\}$ do $y_i \leftarrow h(x_i, \mathsf{id}_i)$
    $(x_*, k) \leftarrow \mathcal{A}(h(y_1, \ldots, y_q)$
    return $[h(x_*, \mathsf{id}_k) = y_k]$ .
```

Note that it does not make sense to model the adversaries influence on passwords. First of all, if the adversary can influence x_k then we can assume that it controls the identity id_k anyway. Secondly, the description of the hash function h is public and thus corruption of specific users does not give adversary new power for guessing the passwords of uncorrupted users.