AN RSA DEVICE FOR FIGHTING PASSWORD-FILE THEFT

P. FORTUNY AYUSO

Abstract. We propose a new authentication scheme which stores passwords in a safe way using RSA and greatly diminishes the risk associated with (encrypted) password-file theft.

1. The Threat

Lately, the use of rainbow tables together with large collections of dedicated GPUs has called to attention the weakness inherent in storing passwords using standard hash functions. As a matter of fact, users being humans, dictionary attacks against this kind of databases seem to us to be deemed to be succesful, at least in a relevant number of cases.

A fast way to overcome this is using a scheme like RSA's PBKDF2 (RFC 2898). However, this is just a restatement of a hashing algorithm (with more iterations etc. but relying on the security of a specific hashing protocol and on the slowness of the process). We present a different approach based —as RSA is— on the complexity of the factorization and the discrete logarithm problems.

The drawback is that our algorithm requires —like so many other authentication mechamisms— a hardware component different from the one the password file is stored on. But we consider the benefits outweigh the costs, especially because set up is easy, fast and cheap.

2. The RSA way

RSA encryption *with padding* provides a way to store a message without revealing any information on it, and secure against brute-force attacks like using dictionaries or rainbow-tables, as the cracker would need not only to guess the password but also the (random) padding in order to verify that the encrypted data match certain plaintext.

Using a hash function prior to RSA encryption would provide another layer of security. We follow this approach in the rest of the paper.

Fix a cryptographic hash function H and an RSA padding scheme for some RSA key pair $(private, public)$. For example, EME-OAEP, as specified in PKCS#1 v2.0 (the padding denoted by OpenSSL as RSA_PKCS1_OAEP_PADDING). Call $\mathrm{E}_{\mathrm{RSA}}$ the encryption function

Date: July 6, 2011.

(using the *public* key) and $\mathrm{D}_{\mathrm{RSA}}$ the decryption function (using the *private* key).

Our proposal is to store the following: given a `username` and a password *pass*, store, together with `username`, the following datum:

$$\mathrm{E}_{\mathrm{RSA}}(\mathrm{H}(pass)). \tag{1}$$

Our scheme requires a `server` to verify the authenticity of the authentication token presented by the user (the password). This `server` holds two private keys: one for decryption (corresponding to the public one used to encrypt *pass*) and one for signing. The respective public keys are known to the `client` (which is, in this set up, the computer the user is trying to log in to).

3. The `AUTHENTICATE` Protocol

It is obvious that logging into a system using our proposed password database scheme requires much more than the usual login protocols. However, it is clearly much simpler than the full X509 standard (as used, for example, on `https` connections) and provides the extra layer of security which ensures that, for example, an SQL injection providing a password token (as stored in the password database) will be worth nothing.

The `AUTHENTICATE` protocol we present provides, as it should, a way to ensure non-replaying of security tokens.

We propose the following `AUTHENTICATE` protocol. Assume the pair (*private*, *public*) is the RSA encryption key pair. As above, we assume a cryptographically secure hash function H has been fixed.

In all this protocol, a string like $a : b$ means "the concatenation of a and b as text strings, inserting a ':' in between".

i) The `client` starts the protocol by connecting to the `server`.
ii) The `server` provides the client with a nonce n (short enough, say 32 bits long).
iii) The `client`, after fetching the user's authentication credentials (say pwd), gets the stored value $q_1 = E_{RSA}(H(pass))$ as in (1), computes $q_2 = E_{RSA}(n : H(pwd))$, generates another nonce m and sends the string $m; q_1; q_2$ to the `server`.
iv) The `server` computes $p_1 = \mathrm{D}_{\mathrm{RSA}}(q_1)$, $p_2' = \mathrm{D}_{\mathrm{RSA}}(q_2)$, verifies that p_2' is of the form $n : p_2$. If this is not so, authentication is denied (there has been a communication error). If it is so, then
    (a) If $p_1 = p_2$ then set $t = 1$.
    (b) If $p_1 \neq p_2$ then set $t = 0$.

    Let k be the message $m : t$ (m is the nonce sent by the `client` in step iii). Let s be the signature $\mathrm{S}_{\mathrm{RSA}}(k)$ (using the signing private key, which —as we said above— is different from the encrypting one). Send $k; s$ to the `client`.

v) The client verifies that s is the signature of k using the signing public key. If it is not so, then autentication is denied. If it is, then check that k is of the form $m : a$, where m is the nonce sent by the client. If it is not so, authentication is denied (communication error). If it is so, then authentication is granted if $a = 1$ and denied in any other case.

It is clear that the only way to get authenticated is to provide the `client` with some token *pwd* such that $\mathrm{D}_{\mathrm{RSA}}(\mathrm{E}_{\mathrm{RSA}}(n : H(pwd)) = n : H(pass)$, where *pass* is the "correct" authentication token. This requires knowledge of n (which is unique for each session) and the ability to compute $H(pass)$. This latter can only be done if (knowing $\mathrm{E}_{\mathrm{RSA}}(H(pass))$, that is, in possession of the password file) one is able to decrypt an RSA-encrypted plaintext, given the public key only.

4. Safety against rainbow tables

If the password database is stolen, the attacker has the information provided $\mathrm{E}_{\mathrm{RSA}}(H(pass))$, where *pass* is the true password chosen by the appropriate user.

If the attacker has access to the password database, he needs to decypher $\mathrm{E}_{\mathrm{RSA}}(H(pass))$ knowing just the public key and, even if he is able to do this, he still needs to recover *pass* from $H(pass)$ in order to be able to authenticate.

5. Further comments

Our scheme is certainly slower than any other implementations we are aware of. However, we deem the security enhancment worth the cost.

There remains the issue of securing the authentication `server`. In order to do this, we propose a dedicated computer —which we call a `Sibyl`— which, after configuration, only performs the `AUTHENTICATION` protocol on a network connection (and behaves normally on a terminal, for example).

In any case, our scenario (which is fairly standard) is a service whose password database is at risk but which is able to store the private key *safely*.

Dpto. de Matemáticas, Universidad de Oviedo
E-mail address: `info@pfortuny.net`