Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: June 10, 2014

R. Kisteleki
RIPE NCC

December 7, 2013

Password Storage Using Public Key Encryption
draft-kistel-encrypted-password-storage-00

Abstract

Current password storage methods predominantly use cryptographic hash functions in order to avoid storing users' passwords in clear text. Unfortunately, recent advancements in hardware design (notably GPUs) allow an attacker to try millions or even billions of password guesses per second which makes "decryption" of simple passwords feasible in short amounts of time.

This document describes a password storage scheme that incorporates public key encryption in order to slow down password verification. Since public key algorithms are several orders of magnitude slower than hash functions, the result makes it much harder for an attacker to discover users' passwords from the stored, encrypted format.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on June 10, 2014.

Copyright Notice

Table of Contents

1. Introduction

1.1. Problem Description

The vast majority of information services use usernames and passwords in order to authenticate users of the service. Instead of storing these passwords in clear text form, the best current practice involves adding some entropy ("salt") to the password, cryptographically hashing the result, and storing the resulting value, as well as the input salt, in the password database.

If an attacker gets hold of this database (via breaking into a system and copying the password database, or using an application bug to reveal it in some other way), they can apply massive amounts of offline CPU/GPU power, use rainbow tables, etc. to find out the original passwords. Modern hardware can be used to apply brute force and execute staggering amounts of password tries in short amounts of time. One can also use precomputed values (rainbow tables) to speed up the process even further.

One of the reasons for why this can be successful is that the hashing algorithm can be implemented in hardware -- one can do millions-billions of password tries per second on a current GPU. Current

practices (for example PKBDF2 [PKBDF2]) try to address this by applying multiple rounds of hashing in order to slow down this mechanism. But in practice the number of rounds is mostly set to a default of 100 or 1000 or such, so precomputing tables is still feasible.

One solution to this problem is to incorporate a "known slow", one way algorithm into the mix, thereby making it more difficult for an attacker to do large amount of tries too quickly. Preferably the algorithm should have no generally and cheaply available hardware implementation. Also, it should be a generally known and widely implemented algorithm. For example, RSA public key encryption could be used.

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Algorithm Description

The proposed method stores user passwords as follows:

1. Pick a suitable hash function (e.g. SHA-256) and public key size (e.g. 2048 bits).

2. Generate a public-private key pair, but keep only the public key part and destroy the private key immediately.

3. In order to store a password, create the hash over the concatenation of the salt and the password, then encrypt it with the public key generated above. The result is hashed again, which results in a limited size output. The pseudocode for the storage algorithm is therefore:

   output = hash(rsa_encrypt(hash(salt+password), public_key))

4. When verifying the password, the same algorithm is applied to the input; then the result is compared with the stored value as usual.

During the encryption step, OAEP or PKCS1 v1.5 padding cannot be used because they are not deterministic in terms of output, which means comparison of stored vs. recomputed would be impossible. Therefore the RSA encryption should be applied without using a padding scheme. The salted hash given as the input to the RSA encryption provides sufficient randomness for this particular purpose.

3. Algorithm Properties

It is reasonable to assume that if an attacker gets hold of the password file, they will also obtain a copy of the corresponding public key. In this case, every password guess attempt still requires an RSA encryption operation, which makes it considerably slower to compute passwords using a brute force approach.

It is believed to be computationally infeasible to reveal passwords in case of an attacker getting hold of the password file but not the public key.

The key space provided by asymmetric algorithm used makes it infeasible to maintain and use rainbow tables for the decryption of the passwords (the same password and salt results in a different encoded form because the use of different public keys).

Use of this method also slows down the password verification for the regular login use case; the size of the asymmetric key used affects the performance of both the benevolent and rogue use cases. It is therefore RECOMMENDED for the operator to choose the key size based on the expected and peak password verification (login) rate. Even small key sizes can introduce significant complexity for an attacker while not affecting the regular password verification times too much.

The operator MAY choose to use multiple public keys at the same time. For example, the operator can choose to use a new key of the same -- or even different -- size from a certain point in time for storage of newly created passwords, while older passwords can still be verified

using the previous key material. As long as all the used public keys used are accessible to the operator, this makes is possible to migrate passwords to be encrypted by the new key over time.

In addition to the algorithm description and salt used, each stored encrypted password SHOULD be accompanied by a reference to the public key used during the encryption process. For example, using the "$" character as the delimiter the format can be:

<algorithm-id>$<pub-key-id>$<salt>$<encrypted-password>

4. Security Considerations

If the public key used to encrypt the passwords is no longer available, then no passwords can be verified any more. Therefore the operator MUST ensure that the public key used in this method is available at all times.

The private part of the used RSA key SHOULD be destroyed immediately after generation.

5. Acknowledgements

The author would like to thank Richard Barnes and Stephen Kent for their feedback during the preparation of this draft.

6. Normative References

[PKBDF2] Wikipedia, "PBKDF2", 2013, <http://en.wikipedia.org/wiki/PBKDF2>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

Author's Address

Robert Kisteleki
RIPE NCC
Amsterdam
NL

Email: robert@ripe.net