Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 7, 2012

S. Bellovin
Columbia University

January 4, 2012

Hashed Password Exchange
draft-bellovin-hpw-00.txt

Abstract

Many systems (e.g., cryptographic protocols relying on symmetric cryptography) require that plaintext passwords be stored. Given how often people reuse passwords on different systems, this poses a very serious risk if a single machine is compromised. We propose a scheme to derive passwords limited to a single machine from a typed password, and explain how a protocol definition can specify this scheme.

1. Introduction

Today, despite the lessons of more than 30 years [[cite Morris and Thomson]], many systems store plaintext passwords. This is often done for good reasons, such as authenticating cryptographic exchanges or as a convenience to users with many passwords; see, for example, the password store in many browsers or the Keychain in MacOS. That said, this practice does pose a security risk to users, since their passwords are in danger if the system is compromised.

The big problem is not compromise of the actual password used on that system; while regrettable, it is inherent in the service definition. Rather, the problem is that users tend to reuse passwords on different systems. If a password is compromised on one machine, the user is at risk on many different systems. Accordingly, we describe a scheme for storing a single-site-only password, derived from the user's typed password; a compromise of a service thus affects just that service.

To accomplish this, we specify a "Hashed Password Exchange" standard, or rather, a metastandard. Rather than specifying a precise way to store and use hashed passwords, we give rules for specifying hashed passwords for use in a given protocol or application. We take advantage of the fact that unlike 1979, when users used very dumb terminals to transmit passwords directly to the receiving applications, most passwords these days are entered into user-controlled software; these programs in turn transmit the passwords to the verifying applications. There is thus intelligence on the user's side; we will use this to irreversibly transform the entered password into some other string. By the same token, the receiving system must apply the same transform to the authenticator supplied at user enrollment time or password change time. Because two independent pieces of software must apply the same transformation, the algorithm must be precisely specified in standards documents.

1.1. Requirements Notation

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 [RFC2119].

2. Definitions and Goals

We use the following definitions:

Username An arbitrary string, the syntax of which is application-dependent, employed by both the user and the verifying system to uniquely identify a given user.

Entered Password The authenticator typed by the user to his or her own software. The usual quality rules (length, special characters, etc.) can be applied; that is out of the scope of this standard.

Effective Password The actual, over-the-wire, string transmitted by the user's software.

Service A particular application on a particular machine or cluster of machines appearing as a single machine

Service URI A URI [RFC3986] for which this effective password should be valid. Only the scheme name, userinfo, and host name portions are discussed here; use of path information is protocol-dependent. In the userinfo field, only the username is used. An example is given below.

Our scheme has the following goals:

1. No two users of a given service should have the same effecive password, even if the entered passwords are the same.

2. No two effective passwords for the same user should be the same for different services, even if the entered passwords are the same.

3. It should be infeasible to invert the hashing function to retrieve the entered password from an effective password and service URI.

4. It should be computationally expensive to mount dictionary attacks on compromised effective passwords.

3. The Hashed Password Scheme

Fundamentally, we calculate the effective password by iterating HMAC [RFC2104], using the entered password as the key and the service URI as the data. This meets all four of our goals:

1. Since the username is part of the service URI, different users will have different URIs, and hence different effective passwords.

2. Since the hostname is part of the URI, different services for any given user will have different URIs, and hence different effective passwords.

3. For any reasonable underlying hash function, it is believed to be infeasible to invert HMAC; see [RFC2104] for details.

4. By iterating a sufficient number of times, dictionary attacks can be made arbitrarily expensive.

We do not use a salt in this scheme. The primary purposes of a salt are to achieve our first and second goals, which we achieve in other ways. A salt also protects against precomputation of possible passwords of known users in anticipation of a later password file compromise; however, since the salt must be used in calculating the effective password, it would have to be known to the user as well as the server, and users typically have multiple devices on which they enter passwords. Using a salt would require that users know it and reenter it, which we regard as infeasible and of limited benefit.

Usernames and the hostname portions of service URIs must be canonicalized before applying HMAC. Legal characters in a username are upper and lower case US-ASCII letters, period, hyphen, underscore, and digits. All other characters MUST be percent-encoded, per section 2.1 of [RFC3986]. Hostnames MUST be canonicalized per [RFC5890][RFC5891] and converted to lower case. How usernames and hostnames are entered is application- and implementation-dependent, and not part of this specification. The hostname used is either the string users type or unambiguously derivable from it per specified rules.

The URI scheme name is given by the protocol specification and MUST NOT be entered directly by the user.

The iteration count is protocol- and use-dependent, and given in the protocol specification.

The effective password, then, is calculated by iterating HMAC some

number of times over the message

    scheme://username@hostname

with the entered password as the key.

3.1. Examples

```
ipsec://someuser@gw.example.net
imap://someuser@mail.example.com
submission://someuser@mail.example.com
```

Note that although someuser can specify the same entered password for both 'imap' and 'submission' on mail.example.com, the effective passwords will be different.

4. Specifying Hashed Password Exchange

The following elements must be in any protocol specification that uses Hashed Password Exchange.

- The scheme name MUST be specified. Generally, this will be taken from the IANA name assigned to the port, but this is not required. Thus, a mail submission URI (TCP port 587) might use the scheme name "submission".

- The rules for deriving the hostname from what users enter MUST be specified. They may be as simple as "use the name the user specifies, e.g., imap.example.com", or they may account for common alternatives: "If the specified host name does not begin with 'www.', prepend it; thus, both 'example.com' and 'www.example.com' would use the hostname 'www.example.com' in forming the URI.

- The iteration count MUST be specified. The value -- typically in the hundreds of thousands with today's technology -- SHOULD be different for different services, and MAY be adjusted based on the platforms on which the calculations are typically done. Note that the iteration is done at password change time rather than run-time, so expense is not a major concern. (Just how long the iterations should take will depend on the protocol designers' understanding of likely platforms and usage patterns. Something that will be run exclusively on fast devices and with stored hashed passwords should use a higher count; something where run-time user password entry on a slow device is considered likely should use a lower count.)

- The rules, if any, for canonicalizing the entered password MUST be specified. It is perfectly acceptable to accept an arbitrary octet string here, but that is not required.

- The hash function to be used with HMAC MUST be specified. MD5 [RFC1321] is more than sufficient; however, the tradeoff is likely to be between what code is likely to be available in implenetations versus the iteration count. SHA-512 [RFC6234] is much slower than MD5, but since the goal is constant time, this matters very little; thus, MD5 would have a higher iteration count than SHA-512 would for the same protocol.

- The encoding rules for sending the effective password over the wire. The output of HMAC is an arbitrary byte string. Given the length of typical HMAC output and the infrequency with which they are sent, transmission efficiency is not a major concern, so a simple hexadecimal encoding is fine. Implementations MAY specify truncation; however, they SHOULD NOT use effective passwords

shorter than 16 octets before encoding.

- o If the protocol permits negotiation of authentication methods, a separate code point MUST be assigned to this scheme.

How passwords are changed -- that is, how new effective passwords are supplied to the verifying machine -- is beyond the scope of this specification. If the entered password is sent directly at password change time, quality checks can be enforced; however, this exposes entered passwords to attacks who have compromised the verifying machine. This is not a major risk, since the rate of password change is low. Conversely, client-side code (e.g., Javascript) can make advisory recommendations on password strength; while the server cannot enforce this, since it will see only effective passwords, very few users will have the will and the skill to override this.

If effective passwords are used only for the usual password verification and not for cryptographic purposes, they should be treated with the care used for ordinary password, i.e., salted, hashed, etc. There is little need for extra iterations, though, since the iteration used in calculating them already provides strong protection against dictionary attacks, and it is unlikely that the extra server-side iterations will be significantly larger than the iterations already performed to comply with this specification.

5. Security Considerations

To be written.

6. Normative References

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

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

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, August 2010.

[RFC5891] Klensin, J., "Internationalized Domain Names in Applications (IDNA): Protocol", RFC 5891, August 2010.

[RFC6234] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011.

Author's Address

S.M. Bellovin
Columbia University
1214 Amsterdam Avenue
MC 0401
New York, NY 10027
US

Phone: +1 212 939 7149
Email: bellovin@acm.org