Introduction

Media covers the theft of large collections of passwords on an almost daily basis. Media coverage of password theft discloses the password storage scheme, the weakness of that scheme, and often discloses a large population of compromised credentials that can affect multiple web sites or other applications. This article provides guidance on properly storing passwords, secret question responses, and similar credential information. Proper storage helps prevent theft, compromise, and malicious use of credentials.

Information systems store passwords and other credentials in a variety of protected forms. Common vulnerabilities allow the theft of protected passwords through attack vectors such as SQL Injection. Protected passwords can also be stolen from artifacts such as logs, dumps, and backups.

Specific guidance herein protects against stored credential theft but the bulk of guidance aims to prevent credential compromise. That is, this guidance helps designs resist revealing users' credentials or allowing system access in the event threats steal protected credential information. For more information and a thorough treatment of this topic, refer to the Secure Password Storage Threat Model here http://goo.gl/Spvzs. The link to this document is http://goo.gl/2loKu.

Guidance

1. Do not limit the character set or length of credentials

Some organizations restrict the 1) types of special characters and 2) length of credentials accepted by systems because of their inability to prevent SQL Injection, Cross-site scripting, command-injection and other forms of injection attacks. These restrictions, while well-intentioned, facilitate certain simple attacks such as brute force.

Do ***not*** apply length, character set, or encoding restrictions on the entry or storage of credentials. Continue applying encoding, escaping, masking, outright omission, and other best practices to eliminate injection risks.

2. Use a cryptographically strong credential-specific salt

A salt is fixed-length cryptographically-strong random value. Append credential data to the salt and use this as input to a protective function. Store the protected form appended to the salt as follows:

```
[protected form] = [salt] + protect([protection func], [salt] + [credential]);
```

Follow these practices to properly implement credential-specific salts:

- Generate a unique salt upon *creation of each* stored credential (not just per user or system wide);
- Use *cryptographically-strong random* [*3] data;
- As storage permits, use a 32bit or 64b salt (actual size dependent on protection function);
- Scheme security does not depend on hiding, splitting, or otherwise obscuring the salt.

Salts serve two purposes: 1) prevent the protected form from revealing two identical credentials and 2) augment entropy fed to protecting function without relying on credential complexity. The second aims to make pre-computed lookup attacks [*2] on an individual credential and time-based attacks on a population

intractable.

3. Impose infeasible verification on attacker

The function used to protect stored credentials should balance attacker and defender verification. The defender needs an acceptable response time for verification of users' credentials during peak use. However, the time required to map `<credential> → <protected form>` must remain beyond threats' hardware (GPU, FPGA) and technique (dictionary-based, brute force, etc) capabilities.
Two approaches facilitate this, each imperfectly.

3a) Leverage an adaptive one-way function - Adaptive one-way functions compute a one-way (irreversible) transform. Each function allows configuration of 'work factor'. Underlying mechanisms used to achieve irreversibility and govern work factors (such as time, space, and parallelism) vary between functions and remain unimportant to this discussion. Select:

- PBKDF2 [*4] when FIPS certification or enterprise support on many platforms is required;
- Scrypt [*5] where resisting any/all hardware accelerated attacks is necessary but support isn't.

Example `protect()` pseudo-code follows:

```
return [salt] + pbkdf2([salt], [credential], c=10000);
```

Designers select one-way adaptive functions to implement `protect()` because these functions can be configured to cost (linearly or exponentially) more than a hash function to execute. Defenders adjust work factor to keep pace with threats' increasing hardware capabilities. Those implementing adaptive one-way functions must tune work factors so as to impede attackers while providing acceptable user experience and scale. Additionally, adaptive one-way functions do not effectively prevent reversal of common dictionary-based credentials (users with password 'password') regardless of user population size or salt usage.
Work Factor - Regardless of the chosen adaptive one-way function, developers must choose a work factor. No one single work factor fits all design situations. Experiment with and configure work factors relative to total function execution time, rather than to a particular chosen value (*WF).

3b) Leverage Keyed functions - Keyed functions, such as HMACs, compute a one-way (irreversible) transform using a private key and given input. For example, HMACs inherit properties of hash functions including their speed, allowing for near instant verification. Key size imposes infeasible size- and/or space-requirements on compromise--even for common credentials (aka password = 'password').

Designers protecting stored credentials with keyed functions:

- Use a single "site-wide" key;
- Protect this key as any private key using best practices;
- Store the key outside the credential store (aka: not in the database);
- Generate the key using cryptographically-strong pseudo-random data;
- Do not worry about output block size (i.e. SHA-256 vs. SHA-512).

Example `protect()` pseudo-code follows:

```
return [salt] + HMAC-SHA-256([key], [salt] + [credential]);
```

Upholding security improvement over (solely) salted schemes relies on proper key management.

4. Design protection/verification for compromise

The frequency and ease with which threats steal protected credentials demands “design for failure”. having detected theft, a credential storage scheme must support continued operation by *marking* credential data compromised and engaging alternative credential validation workflows as follows:

1. Protect the user’s account
    a. Invalidate authN ‘shortcuts’ disallowing login without 2nd factors or secret questions
    b. Disallow changes to account (secret questions, out of band exchange channel setup/selection, etc.)
2. Load & use new protection scheme
    a. Load a new (stronger) protect(credential) function
    b. Include version information stored with form
    c. Set ‘tainted’/‘compromised’ bit until user resets credentials
    d. Rotate any keys and/or adjust protection function parameters (iter count)
    e. Increment scheme version number
3. When user logs in:
    a. Validate credentials based on stored version (old or new); if old demand 2nd factor or secret answers
    b. Prompt user for credential change, apologize, & conduct OOB confirmation
    c. Convert stored credentials to new scheme as user successfully log in

Supporting workflow outlined above requires tight integration with Authentication frameworks and workflows.

References

* [1] - Morris, R. Thompson, K., Password Security: A Case History, 04/03/1978, p4:
http://cm.bell-labs.com/cm/cs/who/dmr/passwd.ps
* [2] - Space-based (Lookup) attacks:
* Space-time Tradeoff: Hellman, M., Crypanalytic Time-Memory Trade-Off, Transactions of Information Theory, Vol. IT-26, No. 4, July, 1980
http://www-ee.stanford.edu/~hellman/publications/36.pdf
* Rainbow Tables -
http://ophcrack.sourceforge.net/tables.php
* [3] - For example: http://docs.oracle.com/javase/6/docs/api/java/security/SecureRandom.html
* [4] - Kalski, B., PKCS #5: Password-Based Cryptography Specification Version 2.0, IETF RFC 2898, September, 2000, p9
http://www.ietf.org/rfc/rfc2898.txt
* [5] Percival, C., Stronger Key Derivation Via Sequential Memory-Hard Functions, BSDCan ‘09, May, 2009
http://www.tarsnap.com/scrypt/scrypt.pdf
* [WF] For instance, one might set work factors targeting the following run times:

- Password-generated session key - fraction of a second;
- User credential - ~0.5 seconds;
- Password-generated site (or otherlong-lived) key - potentially a second or more.
-