SitePassword Design Decisions

This document explains a number of design decisions for SitePassword using a format I learned when working with Intel. I'm leaving out the "Stakeholders" sections because I'm the only one.

Summary of decisions

Offline guessing mitigation: Create a lot of collisions by making super password less guessable than site password

Hash function: PBKDF2

Slow hash function parameters: 1 iteration, 64 Mb key, SHA256

Conversion to characters: u mod |A|, where u is 8 bits from the hash and |A| is the alphabet size

Finding an acceptable password: Iterate PBKDF2, 1 iteration, 8*L bit key, SHA-256

Offline guessing mitigation

An adversary who learns one site password can attempt to guess the user's super password.

Decision

Create a lot of collisions (guesses that produce the correct password for one site but not others) forcing the adversary to try them online.

Objection

1. Users might not choose super password that are less guessable that site passwords.
2. It's not clear how other design decisions affect its security.

Alternatives

1. Multi-factor for super password
   1.1. It's not clear how to integrate multi-factor into the different implementations for the extension and web page.
   1.2. It's not clear how to handle different device types.
   1.3. Many people don't like multi-factor

2. Computing for a long time and caching the result
   2.1. Computing for a long time starts with a secret, typically the super password. The problem is when users have more than one.

Reasons for Decision

1. The security analysis is straightforward.
2. The protection is strong.

Hash function

Like most password calculators, SitePassword uses a hash function to compute the site password.

Decision

PBKDF

Objections

1. It isn't slow enough for any compute time that meets the latency requirements.
2. It isn't memory- or GPU-hard.
3. It isn't resistant to custom hardware.

Alternatives

1. scypt is memory-hard
   1.1. Not as mature
   1.2. Not built into the browser
2. Argon2 comes in memory- and GPU-hard versions
   2.1. Not as mature
   2.2. Not built into the browser

Reasons for Decision

1. PBKDF is widely used, and therefore well studied, for storing the hash of passwords.
2. A version, PBKDF2, is builtin to the browsers, so the computation time is closer to what an adversary would get when mounting a guessing attack against a super password.
3. Settings described below increase its memory hardness.

Hash parameters

PBKDF2 allows the independent choice of key size and number of iterations. The goal is to make the hash as slow as possible while still meeting the latency requirement. PBKDF2 also allows the choice of internal hash function.

Decision: PBKDF2 Parameters

Compute 1 iteration to produce an 64 Mb key with SHA-256.

Objections

1. That's a lot more bits than can ever be converted to characters and still meet the latency requirement.
2. Other hashes are supported.

Alternatives

1. 50,000 iterations to produce a 2 Kb key in the same time.
   1.1. Less memory intense.
2. SHA-512
   2.1. Not as commonly used.
3. SHA-1
   3.1. SHA-1 is deprecated for reasons not relevant here, but using it doesn't look good.

Reasons for Decision

Computing a 64 Mb key is more memory intensive, reducing the number of instances of the guessing program the adversary can run on a single machine.

Converting hash bits to characters

PBKDF2 produces a key made up of bits, but passwords are made up of characters. Some form of conversion is needed.

Decision

Create an array of characters accepted by the site, replicating the special characters to give them approximately the same probability of being selected as the integers. Use a uint8 modulo the alphabet length as an index into the character array.

Objections

1. That waste bits. Only 6 bits for an array 64 or shorter, 7 bits for longer alphabets, are needed.
2. Modulo arithmetic is slow.

Alternatives

1. Avoid modulo arithmetic by replicating the characters until the array has 256 elements.
   1.1. Some characters will appear more times than others giving an adversary trying to guess a site password a clue, which can be mitigated by making the site password one character longer.
   1.2. Using modulo arithmetic only takes ~2X as long. (Measured on a laptop.)
2. Use Base64 encoding.
   2.1. Base64 may not produce special characters from the required set.
   2.2. There's no need for an interoperable conversion.
3. Use the minimum number of bits for the index.
   3.1. The code is more complicated.

Reasons for Decision

Simple code wins out over more complete use of the bits in the hash.

Finding an acceptable site password

Many sites have restrictions on what passwords they accept, such as the number of characters in the password as well as the number of upper/lower case letters, digits, and special characters. Some even require special characters from a specified list.

Decision

If the result of 1 iteration to produce a 64 Mb key doesn't produce a valid password, iterate PBKDF2 with 1 iteration to produce a 8*L bit key, where L is the number of characters in the site password. The input to the first iteration is the rejected password instead of the full 64 Mb hash converted to characters.

Objections

1. The standard way of iterating is to use the whole hash, not the password generated from it, so the security might be weaker.
2. Doesn't always find an acceptable password in the allowed time.

Alternatives

1. Use a sliding window over the characters produced by the hash.
   1.1. Doesn't always find an acceptable password.
   1.2. There is no proof that the adversary gains no advantage.
2. Rotate the alphabet on each try.
   2.1. Doesn't always find an acceptable password.
   2.2. There's no proof that the adversary gains no advantage from this algorithm.
3. Use all the bits of the previous hash converted to characters as input to the first hash iteration.
   3.1. Converting 64 Mb to characters takes ~1 second, which is way too long. Making that key shorter would reduce this time at the cost of a smaller memory footprint.
   3.2. Further iterations do use the entire hash as input to the next iteration.
4. Use one byte of the hash to choose a character and another byte of the hash to pick an effectively random location in the site password for that character. Try another byte if that location is occupied. Continue until all characters have been assigned or all bytes of the hash have been used.
   4.1. Always finds a valid password if the rules are consistent.
   4.2. Uses many more bytes of the hash, which gives the adversary greater ability to filter collisions.
   4.3. Users might think a password with all special characters is more secure.

Reasons for Decision

The randomness of the output of a hash is well understood to give the adversary no advantage. Using less than the full hash as input to the first iteration is no worse security than using the super password for the initial hash.