SitePassword Design Decisions

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

Offline attack mitigation

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

Decision

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

Objection

1. That's never been tried before.
2. It's not clear how other design decisions affect its security.

Alternatives

1. Strong super passwords
    1. User's don't alway choose strong super passwords.

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

3. Computing for a long time and caching the result
    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 cryptography is straightforward.
2. Converting bits of the hash to characters doesn't bias the character selection.

Hash function

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

Decision

PBKDF2

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. scyipt is memory-hard
    1. Not as mature
    2. Not built into the browser
2. Argon2 comes in memory- and GPU-hard versions
    1. Not as mature
    2. Not built into the browser

Reasons for Decision

1. PBKDF2 is widely used, and therefore well studied, for storing the hash of passwords.
2. It's 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. Other protections described later provide a different form of protection than memory- and GPU 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 to make the adversary's job harder while still meeting the latency requirement. It also allows the choice of internal hash function.

Decision: PBKDF2 Parameters

Compute 1 iteration to produce a 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.
2. SHA-512
    1. SHA-512 is not as commonly used.
3. SHA-1.
    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.

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 16 Kb key.

Objections

1. A password that needs the iterations will have fewer pre-images in order to finish fast enough to try many times if needed.
2. The version of PBKDF2 built in to the browser is async. Async in general and async in loops is a common source of bugs.

Alternatives

1. Use a sliding window over an array of characters produced by the hash.
    1. Each step of the sliding window algorithm produces what is effectively a random set of characters. Since all the attacker sees in the acceptable site password, the fact that computed from the hash come from overlapping loses no security. There is no proof of this statement

2. Rotate the character array on each try.
    1. Rotating the array of characters on each try copies the array. That's a performance hit, albeit a negligible one.
    2. There's no proof that the adversary gains no advantage from this algorithm.

Reasons for Decision

The randomness of the output of a hash is well understood to give the adversary no advantage. A 16 Kb key produces enough pre-images even if the adversary filters out those that look too complex for a person to remember.

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 the same probability of being used as the integers. Use a byte as an index modulo the length of the character array to index into the array.

Objections

1. That waste bits. Only 6 bits for an array 64 or shorter, to 7 bits for longer arrays are needed.

Alternatives

1. Use Base64 encoding.
    1. Base64 may not produce special characters from the required set.
    2. There's no need for an interoperable conversion.
2. Use the minimum number of bits for the index.
    1. The number of pre-images will be larger for a given key size, but it's already large enough.
    2. The code is more complicated.
3. Replicating the special characters simplifies the the attacker's job.
    1. We already assume the attacker knows the password rules, including the allowed set of special characters. Even if there is only one that gets repeated 10 times, all the attacker knows is the probability that the special character appears more than once in the site password.

Reasons for Decision

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