Parallel

Steve Thomas (Steve at tobtu dot com)

Specification

Symbols/functions

```
**                      Exponential
%                       Modulus
||                      Concatenate two strings
^                       Xor two strings
hash(string)            Calculates a hash of a string
HASH_LENGTH             Length of hash output in bytes
truncate(string, N)     Truncates a string to N bytes
zeros(N)                Creates an N byte string of zeros
WRITE_BIG_ENDIAN_64(N)  Converts an integer into a string
length(string)          Return the byte length of a string
```

Password hash

Inputs:

```
string  password
string  salt
integer t_cost
integer outlen
```

Algorithm:

```
t_cost_1        = t_cost % 2 ** 16
t_cost_2        = floor(t_cost / 2 ** 16)
t_cost_upgrade  = floor(2 ** floor((t_cost_2 - 1) / 2) * (3 - (t_cost_2 % 2)))
                = 1, 2, 3, 4, 6, 8, 12, ... (Note floor(-1 / 2) = -1)
t_cost_parallel = 1920 * floor(2 ** floor((t_cost_1 - 1) / 2) * (3 -
(t_cost_1 % 2)))
                = 1920 * {1, 2, 3, 4, 6, 8, 12, ...} (Note floor(-1 / 2) = -1)

if outlen > HASH_LENGTH
      return error

key = hash(hash(salt) || password)

// Work
for i = 0 to t_cost_upgrade - 1
      // Init work
      work = zeros(HASH_LENGTH)

      for j = 0 to t_cost_parallel - 1
            work = work ^ hash(WRITE_BIG_ENDIAN_64(j) || key)

      // Finish
      key = hash(hash(work || key))
      key = truncate(key, outlen) || zeros(HASH_LENGTH - outlen)

return truncate(key, outlen)
```

KDF

Inputs:

```
string  password
string  salt
integer t_cost
integer outlen
```

Algorithm:

```
t_cost_parallel = 1920 * floor(2 ** floor((t_cost - 1) / 2) * (3 - (t_cost %
2)))
                = 1920 * {1, 2, 3, 4, 6, 8, 12, ...} (Note floor(-1 / 2) = -1)


key = hash(hash(salt) || password)

// Init work
work = zeros(HASH_LENGTH)

// Work
for i = 0 to t_cost_parallel - 1
      work = work ^ hash(WRITE_BIG_ENDIAN_64(i) || key)

// Finish
i = 0
while length(out) < outlen
      out = out || hash(READ_BIG_ENDIAN_64(i) || work || password)
      i = i + 1
return truncate(out, outlen)
```

Statement

There are no deliberately hidden weaknesses (backdoor, etc.).

Initial Security Analysis

This is best for low memory applications or when FPGAs or GPUs are present. It's very simple and is as resistant to collisions as the underling hash function.

The "outlen" for the password hash should be no less than 16 bytes and doesn't need to be more than 32 bytes. A 16 byte minimum is to prevent simple collisions because of the birthday paradox. The suggested max of 32 bytes is to prevent wasted space storing a large hash. Since there is virtually no chance of a collision at that size. The rest assumes you are between those values. The KDF has no such suggested limits.

I'm a fan of a three step password KDF:

1. Transform password and salt into a fixed size value "key"
2. Generate a "proof of work" value dependent on the key
3. Extract the output key from the proof of work value and the password

Step one makes sure the second step's runtime doesn't depend on the password's length. Step three makes sure any entropy lost in step two is reclaimed. We can claim that if proof of work value can be guess in at least the amount of work needed to generate the proof of work value then the output key is at least as hard as the minimum of the cost of guessing the key and the cost to guess the password * cost of to generate proof of work value.

For step one I picked "hash(hash(salt) || password)" to avoid simple collisions given full control of salt and password. For HMAC(key:password, message:salt) you can find a collision by having a password be larger than the block size. I wanted to avoid endian issues with hash(saltSize || salt || passwordSize || password). Although if you have a 255 byte max length for salt then this would work hash(saltSizeUint8 || salt || password). Anyways there really isn't anything wrong with these collisions, but since they are easy to avoid why not.

For step two I originally chose "hash(i || j || key)" because it is a good balance between optimized defender, unoptimized defender, and attacker. Optimized defender will precalculate the first round or two with "i" and unoptimized defender won't. The attacker will also precalculate the first round or two with "i" and might choose to precalculate "j" too, but the attacker probably won't because of the extra complexity and memory usage for only a round or two. Big endian was chosen so an optimized defender using a hashing algorithm that uses 32 bit words (e.g. SHA1 or SHA256) can precalculate three rounds since the high 32 bits of "j" change infrequently or don't at all.

I removed "i" because it wasn't needed. It was originally there to prevent hash collisions between keys of different upgrade rounds, but the attacker would never keep track of these to exploit this. Also the probability of a collision is so low that it shouldn't ever happen under normal circumstances. Now that it's gone the differences between optimized defender, unoptimized defender, and attacker are smaller.

Efficiency Analysis

The attacker-defender ratio is near 1. Any advancements in cracking are advancements for the defender. If ASICs come out that can crack this hash can be used by the defender.

Intellectual Property Statement

The scheme is and will remain available worldwide on a royalty free basis, and I am unaware of any patent or patent application that covers the use or implementation of the submitted algorithm.

Tweaks

The only real change was the removal of "WRITE_BIG_ENDIAN_64(i)" from the password hash. The KDF is unchanged. This was done to lower the difference between unoptimized defender and attacker. As an added bonus the password hash and KDF are more similar.

I originally meant for this to be used with any hashing algorithm. I was using SHA512 as a place holder. In the original paper when I mentioned the hash function, instead of saying SHA512, I said "the underling hash function".

Fixes (not tweaks but errors in the documentation):
t_cost_parallel was suppose to be multiplied by 1920 but not done in the pseudocode.
Renamed t_cost_sequential to t_cost_upgrade do to confusion of it's purpose.
Formatting: extra spaces and mismatched parentheses.