battcrypt (Blowfish All The Things)

Steven Thomas (steve at tobtu dot com)

Specification

Symbols/functions

```
**                      Exponential
%                       Modulus
||                      Concatenate two strings
^                       Xor two strings
SHA512                  Calculates a SHA512 of a string
blowfish_init_448_bit_key  Initializes blowfish with a 448 bit key
blowfish_encrypt_cbc    This is a running CBC (ie the last block of the
                          previous call is the IV for the next call)
truncate(string, N)     Truncates a string to N bytes
zeros(N)                Creates an N byte string of zeros
```

Password hash

Inputs:

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

Algorithm:

```
t_cost_1       = t_cost % 2 ** 16
t_cost_2       = floor(t_cost / 2 ** 16)
t_cost_main    = 2 ** floor(t_cost_1 / 2) * ((t_cost_1 % 2) + 2)
               = 2, 3, 4, 6, 8, 12, ...
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)
mem_size       = 4 * 2 ** m_cost

key = SHA512(SHA512(salt) || password))
for u = 0 to t_cost_upgrade - 1
       // Initialize blowfish
       blowfish_init_448_bit_key(truncate(key, 56), IV:zeros(8))

       // Initialize data
       data = SHA512(BIG_ENDIAN_64( 0) || key) ||
              SHA512(BIG_ENDIAN_64( 1) || key) ||
              ...
              SHA512(BIG_ENDIAN_64(31) || key)

       // Initialize mem
       for i = 0 to mem_size - 1
              data = blowfish_encrypt_cbc(data)
              mem[i] = data
       data = blowfish_encrypt_cbc(data)

       // Work
       for i = 0 to t_cost_main - 1
              for j = 0 to mem_size - 1
                     r = last64Bits_bigEndian(data) & (mem_size - 1)
                     mem[j] = blowfish_encrypt_cbc(data ^ mem[j] ^ mem[r])
                     data = data ^ mem[j]

       // Finish
       key = SHA512(SHA512(data || key))
       key = truncate(key, outlen) || zeros(64 - outlen)

return truncate(key, outlen)
```

Key derivation function

Inputs:

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

Algorithm:

```
t_cost_main = 2 ** floor(t_cost / 2) * ((t_cost % 2) + 2)
            = 2, 3, 4, 6, 8, 12, ...
mem_size    = 4 * 2 ** m_cost

key = SHA512(SHA512(salt) || password))

// Initialize blowfish
blowfish_init_448_bit_key(truncate(key, 56), IV:zeros(8))

// Initialize data
data = SHA512(BIG_ENDIAN_64( 0) || key) ||
       SHA512(BIG_ENDIAN_64( 1) || key) ||
       ...
       SHA512(BIG_ENDIAN_64(31) || key)

// Initialize mem
for i = 0 to mem_size - 1
       data = blowfish_encrypt_cbc(data)
       mem[i] = data
data = blowfish_encrypt_cbc(data)

// Work
for i = 0 to t_cost_main - 1
       for j = 0 to mem_size - 1
              r = last64Bits_bigEndian(data) & (mem_size - 1)
              mem[j] = blowfish_encrypt_cbc(data ^ mem[j] ^ mem[r])
              data = data ^ mem[j]

// Finish
work = SHA512(data || key)
i = 0
while length(out) < outlen
       out = out || SHA512(READ_BIG_ENDIAN_64(i) || work || password)
       i = i + 1
return truncate(out, outlen)
```

Difference (KDF vs password hash):

The KDF is the same as the password hash except there is no upgrade loop and the finish step.

Statement

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

Initial Security Analysis

The goal of this algorithm was to take Blowfish and make it more memory-hard. Blowfish was picked because it is slow on GPUs and well studied. My starting point was bcrypt and after modifying how Blowfish works and several different configurations of key expansion, encrypting, and multiple Blowfish cores. I realized two things if I change how Blowfish works there is no guarantee of security and the S-boxes must stay in cache otherwise it is too slow. I re-realized Blowfish's key expansion is just CBC encrypting zeros and encrypted blocks overwrite the P array and S-boxes once they are encrypted. The choice for CBC encrypting happened when I thought to search for if the necessary functions exist in PHP (mcrypt_module_open, mcrypt_generic_init, mcrypt_generic, and mcrypt_generic_deinit). I originally chose 4KiB block size because that's the size of Blowfish's S-boxes but have since dropped it to 2 KiB. This is something that could be tweaked in a later version, but this should be constant once finalized. I now realize I basically made a simplified scrypt with Blowfish. So some of the security analysis done on scrypt can be applied to this.

t_cost is split into two time costs and adjusted: t_cost_upgrade and t_cost_main. t_cost_upgrade is for upgrading a password hash when at rest. This only increases the amount of work and does not increase the amount of memory. This was a design choice since I'm guessing that authentication/web servers will want to use a small amount of memory and doubling it each time hashes are upgraded might cause problems.

I went through a few formulas each one has issues with some r values (E() is Blowfish encrypt):

1. data ^= mem[j] = E(data ^ mem[r]) r = j-1
2. mem[j] ^= data = E(data ^ mem[r]) r = j-1
3. mem[j] = E(mem[j] ^ mem[r]) r = j
4. mem[j] ^= data = E(data ^ mem[j] ^ mem[r]) r = j and r = j-1
5. data ^= mem[j] = E(data ^ mem[j] ^ mem[r]) r = j and r = j-1

#1 is fast and when r = j-1 it only cancels mem[r] during encryption.
#2 has a problem where data is canceled when r = j-1.
#3 is slow in PHP also this doesn't carry over data until mem[r] hits a modified block and mem[j] ^ mem[r] can cancel each other.
I like #4 and #5 because both data and mem[j] change depending on data, mem[j], and mem[r], but this is slower and with #4 you need to avoid r = j and r = j-1.

In all but #1 and #5 you must avoid the problem areas of r being j, j-1, or both. This is easy to do if you add one (#1-#3) or two (#4,#5) new blocks to mem. Then if r is greater than or equal to j-1 (#1,#2,#4,#5) or j (#3) then adding one (#1-#3) or two (#4,#5) to r. Since #1 doesn't have major problems when r is j-1 we can simply use it as is. I was debating between #1 and #5 but ultimately when with #5 since it uses mem[j] instead of overwriting it.

t_cost_1 might be secure at 0 and should be at 1. And by secure I mean the attacker won't want to use less memory. This represents hitting twice as many blocks of memory than there are. You are expected to hit over 86.5% of the memory blocks. Note that scrypt is expected to hit over 63.2% of the memory blocks since it only does one pass.

t_cost_1 (t_cost_main)	Miss rate m_cost = 0	Miss rate m_cost = 1	Miss rate m_cost = 2	Miss rate m_cost = 3	Miss rate m_cost = ∞
0 (2)	1 in 9.989	1 in 8.470	1 in 7.887	1 in 7.629	1 in 7.389
1 (3)	1 in 31.57	1 in 24.65	1 in 22.15	1 in 21.07	1 in 20.09
2 (4)	1 in 99.77	1 in 71.74	1 in 62.21	1 in 58.20	1 in 54.60
4 (8)	1 in 9'955	1 in 5'146	1 in 3'870	1 in 3'387	1 in 2'981
6 (16)	1 in 99'100'000	1 in 26'480'000	1 in 14'970'000	1 in 11'470'000	1 in 8'886'000

* Miss 1 in (1 / (1 - 1 / (4 * 2 ** "m_cost")) ^ ("t_cost_main" * (4 * 2 ** "m_cost")))
** Limit m_cost => ∞ is miss 1 in (e ** t_cost_main)

Even if Blowfish is found to be broken it is very unlikely that this break will cause battcrypt to have problems. Since everything is hashed before and after with the initial key. SHA512 would need to be very broken for there to be a problem with this. The easiest place to cause a collision is with the upgrade loop and key truncation, but if the key is truncated to a sufficiently large amount (128 bits) accidental collisions will not be an issue. The best spot for malicious collisions is in SHA512(SHA512(salt) || password) but that will only happen if SHA512 collisions are possible.

Blowfish is a 64 bit block cipher and when encrypting with CBC one should not encrypt more than I believe 2 ** 32 blocks or 32 GiB. If this is found to be a problem re-keying every X mem blocks would fix it, but I don't think this is a problem since we're talking about running this for a few minutes.

Efficiency Analysis

When comparing battcrypt to bcrypt it was a happy coincidence that you use the settings t_cost = 1 and m_cost = bcrypt_cost - 2. battcrypt has the speed advantage of doing 0.752% to 1.726% less Blowfish blocks than bcrypt with costs 5 to 15 respectively. The formulas for the number of Blowfish blocks for cost settings can be found in the benchmark function in battcrypt.php.

battcrypt implemented in PHP takes about twice as long as the equivalent amount of work done with a compiled bcrypt implementation. This is very efficient since most other algorithms would be much slower in PHP than their compiled counterparts. This is because blowfish_init_448_bit_key and blowfish_encrypt_cbc can both be done with built-in functions in PHP 4.0.2 and later with mcrypt_module_open, mcrypt_generic_init, mcrypt_generic, and mcrypt_generic_deinit.

I didn't have time to write a password cracker for GPUs but I would expect battcrypt to be slower than bcrypt with equivalent settings. bcrypt is faster on CPUs than GPUs so I would expect the same to be true for battcrypt even at very low memory settings.

On modern CPUs, interleaving two or three Blowfish calculations approximately doubles the speed. When all cores/threads are used each thread drops to around 1.75 times faster. So for 8 threads on FX-8120 that's 14x faster than the defender. I was debating whether to add parallelism and opted for not because it would make this more complicated. Also on an authentication/web server it probably would not make sense to have your password hashing algorithm spawn a bunch of short lived threads. Yes if I added a fixed amount of parallelism, say 2x, then interleaving them would decrease the attacker-defender ratio, but at a cost of complexity.

There is an option for server offload if the salt and settings are public. Basically the user calculates everything but instead of the last "SHA512(SHA512(data || key))" they do "X = SHA512(data || key)" and send that to the server. All the server has to do is check if " truncate(SHA512(X), outlen)" is what is stored in the database. Using the username as the "salt" is one way although this isn't a proper salt as salt is defined as cryptographic random data. This would be pepper.

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.