Tortuga

Password hashing based on the Turtle algorithm

Teath Sch <teathsch@jailcity.com>

Introduction

Password hashing algorithms are a staple in any cryptographic toolbox, yet they remain underutilized and misused. They provide two main functions: key stretching and key derivation. Common cryptographic hash functions are often used instead of dedicated password hashing algorithms. This approach is often hazardous because ad hoc constructions of password hashing algorithms aren't misuse resistant. In fact, they are notoriously prone to catastrophic failure. The Tortuga algorithm works well for both key stretching and key derivation, however, the basic concept is untested in practice and hasn't been subjected to any review.

The new algorithm is a keyed sponge function with a recursive Feistel network as the permutation. Specifically, the permutation used is the Turtle algorithm originally designed by Matt Blaze [1].

The basic idea is to create a block cipher on the fly which requires an enormous key. This achieves memory-hardness. The importance of memory hardness in password hashing applications was thoroughly explored by Colin Percival, the inventor of the scrypt algorithm [2]. In a Feistel block cipher construction, the parts of the key that have already been used can be discarded, however this doesn't provide an opportunity for a non-trivial memory optimization because the intermediate ciphertexts must be saved. If N is the block size in bits of a cipher, then a Feistel network always requires at least N bits of memory at any point in its execution. An attacker could discard portions of the ciphertext only to recompute them later, but the loss in speed will be such that it can only be regained by parallelizing the attack in a way which will require at least as much circuitry (give or take a small constant depending on the platform) as was gained by discarding ciphertexts.

Description

The key and block sizes of the permutation are variable. The key size is the smallest power of four greater than the size of the password in bytes. The block size is the square root of the key size. This relationship is inherent in the 4-RFN Turtle algorithm. The w parameter (word size) to the Turtle algorithm is fixed to be one byte. The permutation parameter to the Turtle algorithm is:

permutation(n, word, key) = (word + key[n]) mod 256

Note: this is not the main permutation of our sponge construction. The Turtle algorithm uses this small sub-permutation recursively to generate the necessary block cipher on the fly.

The rate of the sponge function is set to block_size / 4. The capacity is set to rate * 3;

```
rate     = block_size / 4
capacity = rate * 3
bitrate  = rate + capacity
```

The algorithm happens five steps.

1. Permutation Key generation
2. Password hardening
3. Absorbing
4. Iteration
5. Squeezing

Auxillary Function: encode_uint()

The following C code demonstrates the encode_uint() function

```
void encode_uint(uint8_t * res, uint32_t x) {

   res[0] =  x        & 0xFF;
   res[1] = (x >>  8) & 0xFF;
   res[2] = (x >> 16) & 0xFF;
   res[3] = (x >> 24) & 0xFF;
}
```

Step One: Permutation Key generation

The permutation key is derived from the salt. It is hardened against length extension attacks by initializing key buffer with the salt size.

Pseudocode:

```
function genkey(key, keylen, salt) {

    for (i = 0, j = 0; i < keylen; i += 4, ++j) {

        key[i:i+4]^=turtle(encode_uint(i), key[i:i+4]);
    }

    if (length(salt) > 16) {

        return genkey(key, keylen, salt[16:end])
    }

    return key;
}

key = genkey(encode_uint(length(salt)), keylen, salt);
```

Step Two: Password hardening

To protect against length extension attacks, the length of the password is encrypted using an instantiation of the Turtle cipher and initializing the sponge state buffer with the output.

```
sponge_state = turtle[p](length(password), key)
```

This is equivalent to prepending the zero-padded length of the password to itself.

Step Three: Absorbing

The password is absorbed into the sponge state in the standard way. The password is zero padded to a multiple of ***rate***. ***rate*** sized chunks of the password are xor'd into the sponge state and the main keyed permutation is called each time.

Step Four: Iteration

This is where the key stretching happens.

min_iterations = (t_cost * 16 / key_bytes + 1) * 2

The main permutation is called on the sponge state min_iterations times.

Repeat min_iterations times:

sponge_state = turtle[p](sponge_state, key)

Note that min_iterations was adjusted to reflect the size of the permutation so that the overall running time is mostly unrelated to the m_cost parameter. This works quite well in practice. For example, changing the **t_cost** parameter from 5000 to 10000 will roughly double the running time, but changing the **m_cost** parameter from 50 to 5000 will have little effect on the running time.

Step Five: Squeezing

The final hash value is extracted in the standard way for sponge constructions. ***rate*** bits are repeatedly copied from the sponge state interleaved with calls to the main permutation, This is repeated until ***outlen*** bytes have been extracted.

Statement on Hidden Weaknesses

There are no deliberately hidden weaknesses and/or backdoors in the Tortuga algorithm. There are no deliberately hidden weaknesses and/or backdoors in the Tortuga reference implementation.

Use Cases

Tortuga is appropriate for key stretching applications. It is also good for key derivation because the output length is variable.

Security Analysis

Cryptanalysis of the recursive Feistel structure for the Turtle cipher is known to be in NP [1]. The actual security of an instantiation depends on the sub-permutation and its relationship with the mixing operation of the Feistel structure. The sub-permutation uses addition while the mixing operation using bitwise addition. This alternation between operations of different algebraic orders is known to provide confusion and diffusion with increasing numbers of rounds [3].

The capacity of the sponge function is related to the m_cost parameter. The “resistance level” for a hermetic sponge is the capacity divided by a safety margin [4].

Efficiency Analysis

Because Tortuga has such a simple description, various optimizations are possible. No attempts at platform dependent optimizations have been carried out yet, but it should relatively easily to translate Tortuga into “close-to-metal” implementations.

Formal operation counts/lower bounds on complexity have not been computed yet, but the algorithm should be fairly easily to analyze in this regard.

Reference Code

Complete reference code in C is available. At the time of writing, no Web site exists for Tortuga. To get a copy of the current code, send an email to teathsch@jailcity.com.

Intellectual Property Statement

The Tortuga algorithm and reference implementation are and will remain available worldwide on a royalty free basis. The authors/designers are unaware of any patent or patent application that covers the use or implementation of the Tortuga algorithm.

References

[1] Matt Blaze. Efficient Symmetric-Key Ciphers Based on an NP Complete Subproblem. http://www.crypto.com/papers/turtle.pdf

[2] Colin Percival. Stronger Key Derivation Via Sequential Memory-Hard Functions. https://www.tarsnap.com/scrypt/scrypt.pdf

[3] David J. Wheeler and Roger M. Needham. Tea, a Tiny Encryption Algorithm. http://www.cix.co.uk/~klockstone/tea.pdf

[4] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge functions. CANS 2001, Sanya, December 10. Page 25.

Test Vectors

```
t_cost: 987
m_cost: 610
pass:   1
salt:   Satoshi
Result: F940DE8BF66DE96394076005B67A030BEFFF3D1854B8C8F50F66FA3EB4DA912E

t_cost: 3
m_cost: 34
pass:   987654321
salt:   abcde
Result: CB678F5DDD11B296EC08B191B901BBA9474615C30250AB984691518A14724921

t_cost: 89
m_cost: 2584
pass:   987654321
salt:   Satoshi
Result: 3D7F810372058BB6BC792763B211B3EE822F12DB97C61E7A3FAF12221316AE09

t_cost: 1
```

m_cost: 8

pass: qwerty

salt: Satoshi

Result: 2B65E24D7CB72554B18AE38D3AC5FC7F0D7CB9CA6B45C2CDFCB78534B10A232D

t_cost: 0

m_cost: 6

pass: 12

salt: abcde

Result: EFF7015868935391EE167FDF290070A31B49D6DE8FE7A168485343F1BE363FAF

t_cost: 6

m_cost: 3

pass: 1f9fndsa

salt: abcde

Result: BD9364302755B73AC693A5735CF82F2DF722FE4B0DA3C44057A5A73AD6637523

t_cost: 3

m_cost: 6765

pass: changeme

salt: 123456

Result: D4C01AEAA133662E27BFD8DB3D2219A8DCA6A3C7669750141D48711FE9AF796D

t_cost: 2

m_cost: 3

pass: ufne7hkq

salt: Satoshi

Result: 44D9CE45732A7D487D17FC592E3DE3C2BD48555714495EA563BA6D58DD478C49

t_cost: 55

m_cost: 1

pass: 12

salt: abcde

Result: 3FAF09B07003AB29669EEFD7E19828533371EE567FFF494030A33BE9161E4F07

```
t_cost: 2
m_cost: 377
pass:   1
salt:   abcde
Result: 1F5F32BAECA6FEE6EE9C946CE7228C60CB9255006D5643837D8BCB2F62F731B4

t_cost: 8
m_cost: 610
pass:   12
salt:   Satoshi
Result: 9B5F30E6C29D13220A459C126ADB090D9D09EEEDFF82534D6C6B70991886C5F6

t_cost: 0
m_cost: 3
pass:   changeme
salt:   123456
Result: 44EA42F9D3A8CE2E8381DCDA82F56FE846C6CF9D54DA5251BB78AEEE0B99AC5A

t_cost: 2
m_cost: 3
pass:   ufne7hkq
salt:   123456
Result: 14FEEF36C33C9E4D7A2D54168B468794BEE9B22124BE37167B6C5EA59A1564F6

t_cost: 28657
m_cost: 2
pass:   12
salt:   Satoshi
Result: C037938B7EF2A753338C807FC34B46829FB3A364E0F713DBCE72A793C35C80DF

t_cost: 377
m_cost: 3
pass:   ufne7hkq
salt:   123456
Result: 24F603B68FD46EF1F279D43E1F96F3FC9E7D1ADDD4F63BA697547E3912D184BE
```

```
t_cost: 6
m_cost: 0
pass:   changeme
salt:   123456
Result: 82F56FE846C6CF9D54DA5251BB78AEEE0B99AC5A52EDF7C8A6D65755242AC269

t_cost: 1597
m_cost: 3
pass:   ufne7hkq
salt:
Result: E4590E9D47FE5DECDDE324D176B51FCE0564F58B04292E8D27FE8DECED83E4C1

t_cost: 1597
m_cost: 3
pass:   1f9fndsa
salt:   Satoshi
Result: 547C21C9231E36253513CC3C81294B26C6F5A54BA42CF1E9736E463515037CCC

t_cost: 89
m_cost: 4181
pass:   qwerty
salt:
Result: 083D09478394FB47E8F217073B9D1BAFB135629614EEAB5723D5489BE2E4523A

t_cost: 610
m_cost: 987
pass:   12
salt:   abcde
Result: A4669637CEFE66B2A78AAAF9EE5924925C9DDC893842E539775B9262E3FFC128

t_cost: 144
m_cost: 0
pass:   987654321
salt:
```

```
Result: DEAA6F1C928060279E84CECA3FAC2210C077EED49EAA2F5CD240E067DEC48ECA

t_cost: 2584
m_cost: 2584
pass:   987654321
salt:   123456
Result: ACA55A573C53BBFD0E3F3903300B1F611638B64E21E844D3C234134DBDD4F3DF

t_cost: 10946
m_cost: 4
pass:   1f9fndsa
salt:
Result: F59B085CCDD9137E42714D2B68DC85A1437E12D9C5CB28DCED89031EC2117D7B

t_cost: 4
m_cost: 4181
pass:   password
salt:   abcde
Result: 2989783E6FA637184AF92A5EBD24E98A970042E01697B4366033CE64AF2F20E4

t_cost: 377
m_cost: 987
pass:   changeme
salt:   123456
Result: 414C209902A547109E01EB168B621A4846199A503279BE3FC80286CF786586A6

t_cost: 89
m_cost: 8
pass:   qwerty
salt:   Satoshi
Result: 0D7CB9CA6B45C2CDFCB78534B10A232D1A85BCFF2D5C39CAEB6562CDBCF7A5D4

t_cost: 4
m_cost: 6
pass:
```

```
salt:
Result: 19E0B9E582190E2D31B4B99851B582D96675D1E4596089F52219CE5D411479B8

t_cost: 17711
m_cost: 6765
pass:   1
salt:   Satoshi
Result: E34FB3FBEBD4CC8BA4AA5626472694D9192A6D0A49F6C769538F3FA4CC365048

t_cost: 10946
m_cost: 233
pass:   ufne7hkq
salt:
Result: 40112A95649A597497381D65A631512F3C29A4DED02B47D8A4A7A8725FA5F4E0

t_cost: 28657
m_cost: 89
pass:   changeme
salt:   123456
Result: 8FE02FC5CFF179C800B6E2DEAB8EEA4216F9744066D6E558AB6848F6DF672A48

t_cost: 610
m_cost: 987
pass:
salt:   123456
Result: F65E49BB424000232967FEEF7BEA23007DAC20AF714642181D412C8433799260

t_cost: 3
m_cost: 2
pass:   987654321
salt:   Satoshi
Result: 12F47E529F7CDE3CD44FA294164A7F4C6E243C8FD2A47E425FBCAEBC64CF8284
```