BSB Utilities Inc.

Whitenoise Laboratories inc.

Software Specifications

For

Tinnitus Utilizing Whitenoise Substitution Stream Cipher

Written by
Stephen Boren email: sboren@bbr.ca phone:
Andre Brisson email: brisson@lightspeed.ca

Table of Contents

1 ALGORITHM DESCRIPTION

Whitenoise Substitution Stream Cipher is a multi-key-Super key hierarchical cryptographic process. This cryptographic system utilizes a method of encryption that can reasonably be described conceptually as an algorithmic representation of a multi-dimensional encrypting cipher matrix. The Whitenoise algorithm takes several sub keys and then creates a very long non-repeating key stream.

The resulting key stream is then used to randomize the plaintext, this randomized plaintext is then put through a substitution cipher.

The Whitenoise component. The mechanism for computing the "SuperKey" from the "sub keys" works as follows. Let $S_j^{\{i\}}$ denote the *j*-th byte of the *i*-th "sub key". Let $\ell^{\{i\}}$ denote the length of the *i*-th "sub key." For example, we might have $\ell^{\{1\}}$ = 13, $\ell^{\{2\}}$ = 17, and so on. Create from the "sub key" *i* the unending sequence of bytes

$$S_0^{\{i\}}, S_1^{\{i\}}, S_2^{\{i\}}, \cdots, S_{(\ell^{\{i\}}-1)}^{\{i\}}, S_0^{\{i\}}, S_1^{\{i\}} \cdots$$

Let $S_j^{\{i\}}$ denote the *j*-th byte of the above sequence, if *j* is any number 0 to ∞; or, in other words, we implicitly reduce the subscript of $S_j^{\{i\}}$ modulo $\ell^{\{i\}}$. Then, the *j*-th byte of the "SuperKey," call it Z_j, is defined by

$$z_j = s_j^{\{1\}} \oplus s_j^{\{2\}} \oplus \cdots \oplus s_j^{\{n\}}$$

Here, "$\oplus$" denotes the XOR operation. In other words, to be more explicit,

$$z_j = s_{j \bmod \ell^{\{1\}}}^{\{1\}} \oplus s_{j \bmod \ell^{\{2\}}}^{\{2\}} \oplus \cdots \oplus s_{j \bmod \ell^{\{n\}}}^{\{n\}}$$

Where j mod $\ell^{\{i\}}$ returns an integer in the range 0,1,2,..., $(\ell^{\{i\}} - 1)$

The "SuperKey" has a j value that ranges from 0 to $((\ell^{\{1\}} \times \ell^{\{2\}} \times \ell^{\{3\}} \cdots \ell^{\{n\}}) - 1)$.

Let $P_0,P_1,P_2,P_3,..$ be the bytes of the plaintext, and $C_0,C_1,\ldots$ the bytes of the ciphertext, in order. Also, $z_0,z_1,\ldots$ denotes the bytes of the "SuperKey" (computed as already described in Section 1). We define the ciphertext by $C_i := P_i$ xor $S[z_i]$.
The ciphertext C is formed by concatenating the bytes $C_0,C_1,..$,and then C is returned as the result of the encryption process.

Decryption works in reverse in the obvious manner.

2 Algorithm Implementation

The interface to Whitenoise Substitution is as follows. The input to the stream cipher is two seed values, used to create the key. In this implementation this is done during fob creation and never done by the user.

ENCRYPTION INTERFACE:
  INPUTS:
    a key K = created by (seed1, seed2)
    a big number counter T(set by key creation and updated as it is used.
    a L-bit plaintext P, for some L
  OUTPUT:
    a L-bit ciphertext C, which is the encryption of P

DECRYPTION INTERFACE:
  INPUTS:
    a key K = created by (seed1, seed2)
    a big number counter T(same as above)
    a L-bit ciphertext C, for some L
  OUTPUT:
    a L-bit plaintext P, which is the decryption of C

The counter T is set during key creation for the first message, then incremented by the size of the plaintext for the next message, and so on, for the third message, …etc.

In this implementation, the user is the only one using the encrypt/decrypt, so no keys need to be transmitted.
We can assume that the counter T is known to any eavesdropper/thief who can acquire all ciphertexts. The attacker cannot exert any influence over T; T will always simply count from starting point on up. The counter value T might not appear in the actual programming API for any real implementation of Whitenoise Substitution-- for instance, the Whitenoise Substitution implementation might instead be stateful and implicitly maintain T as part of its private state -- but the effect is the same, and this way we can specify Whitenoise Substitution as though it were a stateless, deterministic mathematical function.

The key K is pair of values, termed seed1 and seed2. seed1 is a N-digit(our big number library uses char digits to do limitless math, it is only necessary to preset the offsets) value chosen secretly, uniformly at random, independently of everything else, and never disclosed. seed2 is a 32-bit value chosen secretly, uniformly at random, independently of everything else, and never disclosed. The value N is a security parameter, and will typically be in the range of 500-700 digits or 1600-2400 bits. Any choice in this range should lead to acceptable security. The value N may be safely made public, without endangering the security of the system, and it might be the same for all users (for instance, it might be hard-coded in the software).
All security resides in the secrecy & unpredictability of the key K.

Whitenoise Substitution encryption is decomposed into two components:
(1) key setup, and (2) output generation. The interface to key setup is as follows:

KEY SETUP INTERFACE:

```
INPUTS:
  a key K = (seed1, seed2)
OUTPUTS:
  an integer n, in the range 10,11,..,30
  a list (l^1,l^2,..,l^n) of n lengths
  a list (s^1,s^2,..,s^n) of n sub keys
  a big number counter T
  an array S[256] of bytes
```

The integer n is the number of sub keys. Each length l^i is a prime number in the range 2,3,5,..,1021 (there are 172 different primes in this range), and l^i represents the length in bytes of the i^{th} sub key. Each sub key s^i is l^i bytes long. S[] is the substitution; it is a permutation on bytes. We will sometimes write s^i_j to denote the j^{th} byte of s^i, for j in the range 0,1,..,l^i-1. All the outputs of the key setup phase must be kept secret, and they may be different for each message enciphered.

The interface to output generation is as follows:

OUTPUT GENERATION INTERFACE:

```
INPUTS:
  an integer n, in the range 10,11,..,30
  a list (l^1,l^2,..,l^n) of n lengths
  a list (s^1,s^2,..,s^n) of n sub keys
  a big number counter T
  an array S[256] of bytes
  a L-bit plaintext P, for some L
OUTPUT:
  a L-bit ciphertext C, which is the encryption of P
```

Whitenoise Substitution is obtained by plugging the implementation of the key setup phase into the implementation of the output generation phase. To fully specify Whitenoise Substitution, it suffices to separately specify each of these two phases.

KEY SETUP ALGORITHM:

1. Treat seed1 as the decimal representation of an integer in the range of 500-700 digits.
2. Let X := seed1
3. Let Y := $\sqrt{X}$ is the irrational number generated by square rooting X
4. Let Z_1, Z_2, Z_3, Z_4, .. be the digits after the decimal point in the decimal representation of Y. Each Z_i is in the range 0,..,9.
5. Call srand(seed2). // only the first time
6. Call rand() to get the irrational starting point, start.
7. Let start := rand() mod 100. start is in the range 0,1,..,99.
8. Throw away Z_1 and Z_2 all the way to Z_{start}.

9. Let tmp := $10*Z_{(start+1)} + Z_{(start+2)}$. Throw away those used values.
10. Let n := 50 + (tmp mod 50). n is in the range 50,51,..,99.
11. For i := 1,2,..,n, do:
12. Let j = 3*(i-1)
13. Let tmp be the next byte from the Z stream.
14. Let tmp := $100*Z_{j+1} + 10*Z_{j+2} + Z_{j+3}$
15. Let t := 172 - (tmp mod 172). t is in the range 1,2,..,172.
16. Let u be the t^{th} prime among the sequence 2,3,5,..,1021.
17. If u is equal to any of l^1, l^2, .., $l^{\{i-1\}}$, set t to (t+1)mod 172 goto 16
18. Set l^i = u.
19. Next I : goto 11 until all subkey sizes are set.
20. For i := 1,2,..,n, do:
21. For j := 0,1,2,..,l^i, do:
22. Let k := 4*j
23. Let tmp be the next byte from the Z stream.
24. Let tmp := $(1000*Z_k + 100*Z_{k+1} + 10*Z_{k+2} + Z_{k+3})$ mod 256
25. Let s_j^i := tmp
26. Next j : Next subkey byte
27. Next I : Next subkey
28. For i := 0,1,2,..,255, do:
29. Let j := 4*i
30. Let tmp := $(1000*Z_j + 100*Z_{j+1} + 10*Z_{j+2} + Z_{j+3})$ mod 256
31. If tmp is equal to any of S[0], S[1], .., S[i-1], set to (tmp+1)mod 256 goto 31
32. Set S[i] := tmp.
33. Next i
34. Let offset := $Z_iZ_{i+1}\ldots Z_{i+9}$
35. Return n, (l^1,l^2,..,l^n), (s^1,s^2,..,s^n), S[256] and offset.
36. Save in keyfile and add seed1 and start value to DB
37. Increment seed1 and goto 2 //repeat until enough keys are created

It is possible to use srand() and rand() as they are only used to create the keys at the manufacturing stage and saving the resulting keys on the FOB. In a distributed system seed2 would just be used directly as only 1 key is made from the two seeds.

3 Key Creation Rules

Tinnitus is a personal security system where by using a key FOB which you plug into your computer and contains your key. You can encrypt all the files that you wish to secure. Therefore, there is no key distribution system required. It is not for use in any kind of broadcast situation. Without your FOB you cannot access the encrypted files.

Each FOB contains their own unique key(business version will contain two such keys for use with master key for limited communication abilities). How those keys are created is in the following definitions.

First the system must be seeded with two random seed values. The first seed value will be in the range of 500 to 700 decimal digits. The second seed value is a 32 bit value simply used to seed the rand function. Now to create the keys the square root of the first value is used to create a pseudo random sequence to be used to create the key. The second seed value is used to initialize using srand function then rand is then used to set the starting offset(0 to 100). The next key is created using seed1 + 1 and the next rand call, and this is continued…(of course each seed1 is tested to make sure it is not a perfect square).

For Tinnitus, the maximum subkey size is 1021 bytes. This means there are 172 different subkey lengths that follow the rule that subkey length must be prime. The first number to be calculated is how many subkeys to be used. This is calculated by taking the first two digits generated by our random stream. This is MODed by 50 and add 50 to give the key structure of between 50 and 99 subkeys. Then using the following system repeatedly to generate each subkeys length. A byte is generated then is MODed by 172 then you take 172 – this value to choose the prime value. If this value is already used it takes the next available larger value. Then the subkeys are filled in turn byte by byte. The substitution cipher is then created by grabbing a byte in sequence to fill it out. Again if there is a repetition, it simply takes the next available byte. Once the key is created it is saved into a file using the file format defined in the next section and the next key is started until the entire sequence of keys requested have been created.

For Tinnitus, there is no key distribution system. This is because the key is only being used for personal security on the users' computer. Therefore the key is created during the manufacturing process of each FOB and is unique for each device. For retrieval purposes the seed values used to create each key will be stored for authenticated retrieval in the event of a lost FOB. Each key is then stored on each FOB.

As each file is encrypted a new file named {OLDFILENAME}.{OFFSET}.wn. The OLDFILENAME includes the extension to allow for easy decryption and maintaining the same file format for functionality. As each file is encrypted, it is immediately decrypted and compared to the original and then both the test copy and the original file are deleted using a clean sweep deletion process(entire file rewritten as 0's and then 1's and then deleted).

4 KEY FILE FORMAT

The Tinnitus key file format is defined as below.

```
typedef struct wnkeyfiletype {
    char       fileid[2]; // must be WN to identify file format
    long       version;  // file type version number to allow changes
    BIGNUMBER  offset; // the offset is a large number stored as a
                       // string of decimal digits delineated by “”’s
    long       numsk; // the number of subkeys
    long       sklen[numsk]; // the individual subkey lengths
    char       sk1[sklen[1]]; // the 1st subkey
    char       sk1[sklen[2]]; // the 2nd subkey
    char       sk1[sklen[3]]; // the 3rd subkey
       …       …       …       …
    char       sknumsk[sklen[numsk]]; // the numskth subkey
    char       substit[256]; // the substitution cipher key
} WNKEYFILE;
```

The Tinnitus Key file format is fairly standard. The only different value is the offset that is stored in a string of decimal digits that are delineated by “’s. An example of this would be “987654321”, this allows for values ranging up to 1000 digits long which may be necessary for Tinnitus to function. This prevents the reuse of sections of the key stream if the offset is implemented properly.