Published : IEEE International Conference on Computer Science and Software Engineering, 2008

A Multi Way Tree for Token Based Authentication

Vipin M, Sarad AV, Sankar K
AU-KBC Research Centre
Anna University
Chennai, India
{vipintm, avsarad, sankar}@au-bkc.org

Abstract—**Tokens are popular in two factor authentication, where the first authentication credential is a fixed secret key s_1 associated with the user and the second authentication credential is a dynamic secret key s_2 generated by the token assigned to the user. Popular hardware tokens use a synchronized clock T along with a static secret key encoded into the token to generate a one time secret s_2. The contribution of the paper is to eliminate this clock synchronization with the authentication server S and to detect unauthorized login attempts when an intruder is in possession of s_1 or s_2 but not both, with a high degree of certainty.**

***Keywords-** Access control, two factor authentication, multi way tree.*

I. INTRODUCTION

Tokens are used to generate one time keys in two factor authentication schemes. In a two factor authentication scheme, the first authentication credential is a fixed secret key s_1 associated with the user and the second authentication credential is a one time [1] secret key s_2 generated by the token assigned to the user. The server S authenticates s_1 followed by s_2. If the first level of authentication fails, the second level is not carried out. The motive of a two factor authentication is to thwart a malicious attacker from gaining access to a secure service, when s_1 is compromised either due to attacks such as a virus, trojan, keylogger, TEMPEST[2] equipment or social engineering tricks such as tricking a user to reveal the secret key s_1. S/KEY^{TM} [3-4] is a popularly used one time password system based on Lamport's algorithm [5]. This approach suffers from the disadvantage that the number of times the one way hash function needs to be iterated is equal to the maximum number of secure logins required by the user. The other alternative is to store the hash table but is again storage intensive. It is of advantage to reduce the time and space complexity for one time secret generation and one such efficient scheme is RSA SecurID$^{©}$ [6-7]. It uses a token to generate a one time 6-8 digit secret every minute, based on the clock and a secret key encoded into the token [8]. The RSA tokens have been upgraded to use an AES based hash due to weaknesses in the alleged SecurID hash function [9]. Any skew that may occur in the clock needs to be corrected and requires synchronization. Further, the time has to be carefully mixed with the key dependent hash function in order to minimize information leakage at the hashed output. The time varying key however has the advantage that the secret s_2 cannot be used as a fixed point for an attacker, effectively preventing an attacker from guessing s_2 with a high degree of success. This assumes significance since one time secrets are short (6-8 digit word) taking into account ease of use which is essential from the user perspective.

It is possible to retain the above useful properties without clock synchronization and can be done in a time and storage efficient manner. Such an efficient scheme involving a Linear Feedback Shift Register (LFSR), a cryptographically secure one way hash function and a multi way tree is proposed. Section II. briefly describes LFSR's, a cryptographically secure one way hash function and a multi way tree. Section III describes the protocol used for one time secret generation and the authentication step. Section IV discusses the security considerations for the scheme proposed and section V concludes the main results of the paper.

II. MATHEMATICAL BACKGROUND

This section discusses the primitives used in the authentication protocol namely maximum period linear feedback shift registers, a cryptographically secure one way hash function and a multi way tree. LFSR's are chosen for its operational speed and simplicity. One way hash functions are used for secure authentication.

A. LFSR and One Way Hash Functions

LFSR's in GF(2) has the advantage that it is extremely fast in hardware and software and hence used to generate pseudo random bits. Let $y_i \in \{0,1\}$, $1 \le i \le l$ be the taps of the LFSR and $w=\{w_1,w_2,\ldots,w_l\}$ be the initial loading of the LFSR, $w_i \in \{0,1\}$. Define $Z=\{0,1\}^l$. The feedback function $f: Z \rightarrow Z$ is

$$f(w_1,w_2,\ldots,w_l) = (\sum_{k=1}^{l} y_k . w_k \ , w_1,w_2,\ldots,w_{l-1})$$

When the LFSR is clocked, the contents of the tapped register cells are added modulo 2 and fed back. The bit in the right most register cell is the LFSR output. For the output of the LFSR to be of maximum period 2^n-1, the polynomial of degree n corresponding to the taps should be primitive modulo 2. An extensive table of primitive polynomials modulo 2 is available in [10].

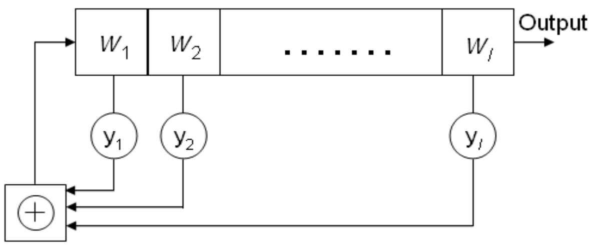

Figure 1. Fibonacci configured LFSR

Choose a primitive polynomial with degree n=160. The initial loading of the LFSR is a static secret key encoded into the token. A cryptographically secure one way hash function H [11] must satisfy a number of properties including its ability to hash an arbitrary message M to fixed length message digest. It should have a good avalanche effect; where every bit flip in the input to the hash function should result in nearly half the number of bits of its output to change. A robust cryptographically secure one way hash function such as RIPEMD-160 [12-13] may be used; where 160 bits is the length of the message digest generated.

B. *Multi Way Tree*

A complete multi way tree *MTree* structurally similar to a B-tree [14] is used in the authentication scheme. Define the order of the *MTree* to be the number of elements present in each node of the tree.

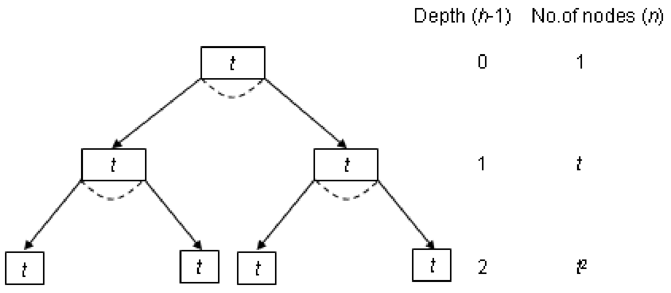

Figure 2. Multi way tree *MTree*

Let the order of the *MTree* be set to an arbitrary positive integer t. i.e. each node has t entries as illustrated in Figure 2. The branching factor of the *MTree* is defined as the maximum number of branches originating from any given node and is also set to t. For a *MTree* of height h, total number of nodes

$$n = 1 + t + t^2 + \ldots + t^h = (t^{h+1} - 1) / (t-1) \qquad (1)$$

Total number of elements in *MTree is*

$$= [(t^{h+1} - 1)/(t-1)].t \qquad (2)$$

It is also trivial to see that the number of leaf elements for an *MTree* with height h is $t^{h-1}.t$

$$= t^h \qquad (3)$$

III. TWO FACTOR AUTHENTICATION PROTOCOL

The user U supplies two secret credentials s_1 and s_2 to a trusted remote server S, whose task is to authenticate U. The first secret s_1 is static and the second secret s_2 is dynamic. The secret s_1 is a password or a passphrase known only to user U and server S. The one time secret s_2 is constructed on demand at both (user and server) sides through a common shared secret *seed*. The static secret key encoded into the token is the *seed*.

A. *One time secret generation*

A primitive polynomial modulo 2 with degree n=160 is employed to generate a maximum period LFSR with period 2^{160}-1. The LFSR is loaded with the secret *seed*. Set the branching factor of the *MTree* to t= 100 and height h = 3. Since *MTree* is a complete tree, the number of elements in each node is t = 100. From (3), the total number of leaf elements in *MTree* = $100^3 = 10^6$ elements. The leaf elements in *MTree* are the set of all one time secrets s_2 that can be used by user U for authentication. The generation of the one time secret s_2 by using one way hash functions is discussed in Algorithm *Genhash*.

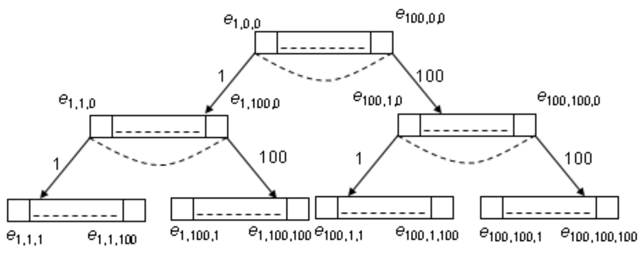

Figure 3. Ordering elements in *MTree*

Figure 3, illustrates the ordering of the elements in the *MTree*. For e.g. the first element in the first node at h = 3 is represented as $e_{1,1,1}$ and the last element in the last node at h = 3 is represented as $e_{100,100,100}$. Since only the leaf nodes are used as the dynamic secret s_2, the input is a 3 tuple (aa, bb, cc) representing the position of the s_2 amongst the leaf nodes. Further $1 \leq aa,\ bb,\ cc \leq t$. The use of *MTree* over a binary tree reduces the height of the tree from $\log_2$ (*nodes*) to $\log_t$ (*nodes*); *nodes* the total number of nodes in the tree. The large branching in *MTree* allows us to restrict its height and save computational time. The value of h is restricted to 3, since its leaf nodes hold a sufficiently large number of elements. Using a *MTree* with $h>3$, increases the computational complexity because of the additional use of the hash function. Hence, increasing the value of t is a lucid solution to increase the key space of the dynamic secret s_2.

Algorithm *PickDynamicSecret* picks leaf nodes from *MTree* in non-ascending index order and is used to synchronize user U with server S. Separate copies of the algorithm are run at both the user and server end with the same initial value of (aa, bb, cc). The following steps are carried out in the Algorithm *PickDynamicSecret*.

```
Algorithm PickDynamicSecret (aa, bb, cc)
Input: Current value of (aa, bb, cc)
Output: Next value of (aa, bb, cc)
Step 1. if (cc>1)
            cc ← cc – 1
        else
            cc ← t
            if (bb>1)
                bb ← bb – 1
            else
                bb ← t
                if (aa>1)
                    aa ← aa – 1
                else
                    print (change seed)
Step 2. if (aa>0)
            return (aa, bb, cc)
```

Algorithm *Genhash* is used to create the one time secret $e_{aa,bb,cc}$. The secret $e_{aa,bb,cc}$ is dynamic because none of the leaf nodes in the *MTree* are revisited by nature of indexing *MTree* in Algorithm *PickDynamicSecret*. The value of t is suitably chosen to obtain a *MTtree* with the required key space.

```
Algorithm Genhash (aa, bb, cc)
Input: Current value of (aa, bb, cc)
Output: Dynamic secret $e_{aa,bb,cc}$
Step 1. Use the maximum period LFSR of length $2^{160}-1$ loaded with the secret seed.
Step 2. Clock the LFSR aa times and then store the internal state of the LFSR onto $e_{aa,0,0}$.
Step 3. To find $e_{aa,bb,cc}$ do the following:
        Compute $e_{aa,bb,0} \leftarrow ( H(e_{aa,0,0} + bb) ) \bmod 2^n$
        Compute $e_{aa,bb,cc} \leftarrow ( H(e_{aa,bb,0} + cc) ) \bmod 2^n$
Step 4. return $e_{aa,bb,cc}$
```

B. Authentication Steps

The fixed secret s_1 and the static secret *seed* are pre-shared between the authentication server S and the user U. The *seed* is embedded into the tamper resistant token given to the user. The server first authenticates secret s_1 and then proceeds to authenticate s_2. The initial value of (aa, bb, cc) is (t, t, t). The authentication of the dynamic secret s_2 is synchronized at user side and server side to pick the leaf element of the *MTree* whose index position is given by Algorithm *PickDynamicSecret*. Once the tuple (aa, bb, cc) is retrieved, the corresponding secret s_2 is calculated by invoking Algorithm *Genhash.* For every invocation of Algorithm *PickDynamicSecret*, the value of the tuple (aa, bb, cc) is updated to the value returned by it. i.e. the new tuple returned by Algorithm *PickDynamicSecret* is the updated value for (aa, bb, cc) at both the user and server side. Algorithm *Auth* illustrates the sequence of steps involved. The initial value of (aa, bb, cc) for Algorithm *Auth* is (t, t, t).

```
Algorithm Auth (aa, bb, cc)
Input: Current value of (aa, bb, cc)
Output: Dynamic secret $e_{aa,bb,cc}$
Step 1. (aa, bb, cc) ← PickDynamicSecret (aa, bb, cc)
Step 2. $e_{aa,bb,cc}$ ← Genhash (aa, bb, cc)
Step 3. Store the updated value of (aa, bb, cc)
Step 4. Print the dynamic secret $e_{aa,bb,cc}$
```

The time complexity of the algorithm is the time involved in clocking the LFSR atmost t times, the time for two invocations of hash function H, two modular additions and the time involved in determining the position of the element to be picked from the *MTree*. The space complexity of the algorithm is the memory required to store the value of *seed* and the present value of the tuple (aa, bb, cc).

It is possible to group adjacent leaf nodes of the *MTree* into blocks, each block consisting of *window_size* number of leaf nodes. The value of *window_size* is set to match the maximum number of reasonable tries a user U may require to successfully authenticate with the server S in the presence of network faults or other unforeseen errors. Only one successful authentication occurs per block. The next round of authentication picks the next adjacent block of leaf nodes in the *MTree* by reducing the index value of (aa, bb, cc) suitably. The *window_size* of the blocks of the *MTree* and the offset position to retrieve the next adjacent block are preset at both the user side and server side.

IV. SECURITY CONSIDERATIONS

The security model considers that server S authenticates s_1 followed by s_2. If authentication of s_1 fails, the dynamic secret s_2 on the server side is not looked up. The following two cases are of interest:

Case 1. If an attacker supplies the correct key s_1 and an incorrect guess s_2, the server first authenticates the user supplied s_1. When the server finds the supplied credential s_1 to be true, the server generates s_2 and checks it with the user supplied s_2. After *window_size* attempts, the server declares the secret s_1 of user U to be compromised and suspends the login until both secrets are reset.

Case 2. If an attacker supplies an incorrect key s_1 and a correct guess s_2 at least *window_size* number of times(in this case the attacker has temporarily access to the token supplied to user U), the server first authenticates the user supplied s_1. When the server finds the supplied credential s_1 to be false no further steps are taken and the login attempt is declared as failed. Next, when the actual user U attempts to login, the first level of authentication succeed but the second level fails since only one successful authentication can be carrier out per block and now the blocks on the user and server side are out of synchronization. As a result, the server declares the secret s_2 of user U to be compromised and suspends the login until both secrets are reset.

V. CONCLUSION

A time and memory efficient two factor authentication protocol involving a multi way tree and no clock synchronization is proposed. Attack attempts on s_1 or s_2 but not both, are detected with a high degree of certainty.

REFERENCES

[1] N. Haller, C. Metz, P. Nesser, M. Straw, "A One-Time Password System", Request for Comments: 2289.J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] Markus G. Kuhn, "Electromagnetic Eavesdropping Risks of Flat-Panel Displays", Presented at the Fourth Workshop on Privacy Enhancing Technologies, 26–28 May 2004, Toronto, Canada.

[3] Neil Haller. "The S/KEY one-time password system". In Proceedings of the ISOC Symposium on Network and Distributed System Security, pages 151-157, San Diego, CA.

[4] N. Haller, "The S/KEY One-Time password system", Request for Comments: 1760.

[5] Leslie Lamport, "Password authentication with insecure communication", Communications of the ACM, v.24 n.11, p.770-772, Nov. 1981.

[6] "RSA SecurID Authenticators, Accelerate your business with the gold standard in two-factor authentication", RSA data sheet.

[7] "Are passwords really free? A closer look at the hidden costs of password security", RSA white paper.

[8] Alex Biryukov, Joseph Lano and Bart Preneel, "Cryptanalysis of the Alleged SecurID Hash Function", 10th Annual International Workshop, SAC 2003, Ottawa, Canada, August 14-15, 2003.

[9] S. Contini, Y.L. Yin, "Improved Cryptanalysis of SecurID", Cryptology ePrint Archive, Report 2003/205, September 2003.

[10] Schneier, Bruce. *Applied Cryptography*, John Wiley & Sons, 1994.

[11] Ralph C. Merkle: "One Way Hash Functions and DES". CRYPTO 1989: 428-446.

[12] H. Dobbertin, A. Bosselaers, B. Preneel, ``RIPEMD-160, a strengthened version of RIPEMD," Fast Software Encryption, LNCS 1039, D. Gollmann, Ed., Springer-Verlag, 1996, pp. 71-82.

[13] Antoon Bosselaers, René Govaerts, Joos Vandewalle: "Fast Hashing on the Pentium". CRYPTO 1996: 298-312.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. Chapter 18: B-Trees, pp.434–454.