Security Token for Web Bank Applications Using a Linear and Congruential Random Number Generator

Luis Orantes[1], Marco Ramírez[2], Pablo Manrique[2], Victor Ponce[2], Aniceto Orantes[3], Victor Salazar[3], Antonio Montes[3], Carlos Hernández[4], Eric Gómez[5]

[1]Center for Research in Computing, Av. Juan de Dios Bátiz, Mexico City 07738, Mexico
lorantesg1101@alumno.ipn.mx
[2]Center for Research in Computing, Av. Juan de Dios Bátiz, Mexico City 07738, Mexico
{mars, pmanriq, vponce}@cic.ipn.mx
[3]HighBits, Av. Central Poniente #847 Int. 3, Tuxtla Gutiérrez, Chiapas 29000, Mexico
{aorantes, vsalazar, antonio}@highbits.com
[4]Unidad Profesional Interdisciplinaria en Ingeniería y Tecnologías Avanzadas, Av. Instituto Politécnico Nacional 2580, Mexico City 07340, Mexico
carlos@highsecret.com
[5]Escuela Superior de Ingeniería Mecánica y Eléctrica Unidad Zacatenco, Av. Instituto Politécnico Nacional S./N. Unidad Profesional Adolfo López Mateos. Mexico City 07738, Mexico
ergomez@ipn.mx

Abstract. This paper presents a new algorithm for using with an one-time password security token; the objective is to provide security for the authentication of customers using bank websites even in the cases when the user has been the victim of a phishing or spyware attack and their bank account secret password has been stolen. For the token's performance, the algorithm make use of a Linear and Congruential Random Number Generator (LCG) (for a better understanding of the presented algorithm a short introduction to this arena is given), and an exhaustive algorithm for the validation of the one-time password keys is presented. This paper shows that the present algorithm is easy to implement and safer than a competing algorithm widely used in today's security tokens.

Keywords: Security token, cryptography, random number generators.

1 Introduction

Bank institutions have modernized their operations by allowing their customers to perform almost any account transaction using the Internet. One of these operations has been to transfer funds electronically to other bank accounts, bringing with this a profound danger. The user may be a victim of phishing or may have, without knowing it, installed spyware on their computer. Thieves use the user's stolen passwords to empty the user's bank account and transferring the funds to a ghost bank account for later withdrawal.

Advances in Computer Science and Engineering
Research in Computing Science 58, 2012 pp. 363–372
Paper Recived 20-08-2012 and Acepted 07-11-2012

One solution that has been given for this problem is the use of one-time password generators (security tokens), which create a password that is valid for bank access just once. With this technique, if the password is stolen it is rendered useless for accessing the bank website. In addition to the conventional password the bank website will request the one-time password which if stolen it's useless because it will be already expired just right after the legitimate user introduced it to the bank website. In other words, the token system proves that the user is who claims to be and acts as an electronic key to access the bank website services, offering security to the user even in cases where their password has been stolen.

A security token is a hardware device usually with a LCD screen (this kind of security token doesn't need to be connected to the computer when used) or provided with a USB plug (this kind of security token needs to be connected to the computer when used). Regarding the kind of approach they are used, security tokens can be classified into several categories with some of the most common approaches: 1) one-time passwords, 2) time-synchronized passwords and 3) challenge/response passwords. In this paper, the algorithm presented belongs to the one-time password category.

It has been done plenty of research about token security but in this paper just one reference will be done, just to the most common security token used nowadays. This security token is the secureID token developed by RSA Security which uses a 64 bit secret key for a hash function called Alleged SecureID Hash Function (ASHF). In [14] it has been shown that the core of this security token can be broken in a few milliseconds and they conclude that it doesn't provide the security demanded by institutions nowadays including banks. In contrast, the algorithm presented in this paper offer much higher security just as it is (with a 64 bits secret key) but it may be virtually unbreakable resizing the presented algorithm to a larger key size as explained later.

The algorithm presented in this paper uses a Pseudo Random Number Generator (RNG) as the encryption mechanism which is necessary for the generation of the one-time passwords. The following section has a brief explanation of what a RNG is because randomness is the core of the presented algorithm and a basic background on this topic is required for a better understanding of it.

2 Random Number Generators (RNG)

Computers can't be random. What computers can do is to simulate a random process by using a RNG. A RNG is an equation, which can generate a sequence of pseudorandom numbers. This sequence of pseudorandom numbers is finite and, after a certain quantity of pseudorandom numbers created, the sequence is repeated again in a cyclic way. The length of the cycle is called the RNG period and this is given, in the best cases, by the number of bits of the RNG and it's limited by the bit number of

the mathematical operations that can be computed. In ordinary PCs it is only 32 bits, nevertheless it is possible to simulate 64 bits in C++ simply by defining a long long variable type or even larger number of bits using a long int library.

There are several random number generators; the generator is selected depending on the type of a specific. There are several kinds of applications for a RNG; the most typical applications are: simulations (e.g., of physical systems to be simulated with the Monte Carlo method), cryptography and procedural generation. For the selection of the appropriate random number generator it should be taken on account not just the quality of the random numbers the generator creates but also the complexity of the generator.

For instance, for the particular application presented, it should be taken on account that this algorithm may be implemented in hardware devices (e.g. a microcontroller). These kind of devices may have limited resources such as computing capabilities and battery life. Another factor to take on account in the generator selection is energy consumption because we want the token to last as much as possible (at least a couple of years). If the generator is too complex (i.e. computationally speaking too expensive) the token life would last just a few months. A token with a short life span would be too impractical for being used in real life applications. Also it is necessary the token to generate the one-time passwords instantaneously and for achieving this goal a random number generator quick to compute is mandatory.

On one hand there are very quick generators such as xorshift [1] [2] [3] which in some cases may generate a full period but they generate very low quality random sequences. The Linear feedback shift register RNG (LSFR) [4] is a popular generator which in the past has been implemented in hardware [5] but it doesn't generate a full period. This is a very popular generator which has been implemented in several applications; important LFSR-based stream ciphers include A5/1 and A5/2, used in GSM cell phones, E0, used in Bluetooth, and the shrinking generator. Nevertheless its drawbacks were reveled when the A5/2 cipher has been broken and both A5/1 and E0 have serious weaknesses [6] [7].

On the other hand there are very high quality random number generators such as Blum Blum Shub [8] [9], Yarrow algorithm [10] (incorporated in Mac OS X and FreeBSD), Fortuna [11] [12] and CryptGenRandom [13] (incorporated in Windows) that have the inconvenient of being computationally speaking too expensive for the purpose of this algorithm.

2.1 The selected random number generator

The selected random number generator for being incorporated in the algorithm presented in this paper is the Linear and Congruential Generator". This RNG was selected by its satisfactory quality; also it has the advantage that it computes the random

sequences very quickly. Because of its low complexity this RNG is suitable for being implemented in hardware applications with limited resources (e.g. a microcontroller).

A sequence of random numbers is obtained by evaluating the following equation:

$$Z_i = (aZ_{i-} + \cdot) \bmod m \quad (1)$$

This RNG requires a seed Z_0 which is the initial state of the RNG; this can be seen as the index or initial point of the random table and it will be the first value of the random table. The next value of the random table is calculated by replacing Z_{i-1} with the new obtained value and this process is repeated again. In other words, Z_{i-1} is the value of the previously computed random number.

a, *c* and *m* are constants, nevertheless these values of these constants need to be chosen carefully because the quality of the random table depends on them. There are values for these constants that generate a poor quality random table and others that don't generate any random table at all. The value of m gives the cycle size of the random table, that is to say, from which number the random table would repeat again. It is desirable to have as many random numbers as possible; therefore the value of m usually is as big as the maximum value it can be computed.

Multiple researches have been conducted to find out what the values of these constants are the best. There are several approaches for determining the quality of the pseudorandom number sequence generated by a given constant values, for instance in [15] this quality is examined by scatter plots and spectral test but there are many other techniques for determining the quality of a pseudorandom sequence and if this quality is acceptable enough. The generator, "Linear and Conguential Generator" is especially very sensitive to the choice of these constants values and in the past have been poor choices for the values of these constant with not good results [16].

These constants need to meet some conditions; for instance to guarantee a generation of a full period for any seed values when having a non-zero value for *c*; they need to meet the following conditions [15]:

1. *c* and m must be relative primes,
2. *a-1* must be divisible by all prime factors of m,
3. *a-1* must be a multiple of 4 if m is a multiple of 4.

It is recommended for a 64-bit variable such as an ordinary PC can simulate by defining a long long variable type in C++ to be the next:

a = 6364136223846793005
c = 1442695040888963407
$m = 2^{64}$

3 Algorithm for the generation of one-time passwords on the token side

Equation 1 is capable of generating a pseudo-random table; nevertheless it is necessary to have a unique random table for every token that is constructed. It is desirable to have a different sequence of numbers for every token. It is possible to achieve this by encrypting equation 1 by performing a XOR operation of the generated random number with a secret key *K*. After this, it is necessary to resize the encrypted sequence to values that can fit within the eight digits of a LCD display (i.e. values that range from 0..99,999,999 = 9^8) This is done by performing a modulus operation of the calculated Z_i value with 9^8+1. With this the one-time passwords will range from 0 to 9^8. The equation then becomes:

$$Z_i = [aZ_{i-1} + c] \bmod m \otimes \quad (2)$$

$$OneTimePassword = Zi \bmod (9^8 + 1) \quad (3)$$

The security token needs to be initialized, it is necessary to have a seed value for this and the same value of the secret key K as the seed for the token which will be the Z_i value. This value will be stored in the token's memory and it will be used for computing the next one-time password by the security token and also at the server side to validate a one-time password. The first 1,000 one-time passwords are generated at the security token side for being wasted (this is done for allowing detection of 1,000 expired one-time password as explained later).

On the token side the value of *K* is obtained and is used as seed Z_0 for evaluating Equation 2 and 3 and generating a one-time password. This equation is evaluated again to obtain another one-time password. In this way, every time the "generate a one-time password" button is pushed a pseudorandom encrypted sequence is obtained. See Table 1 for an example of this.

Table 1. Example of a sequence of numbers generated by the token

Counter	One-time password
1	82475249
2	82040631
3	72383201
4	68714439
5	32340945
6	88383319
7	94725313
8	10436071

4 Algorithm for the one-time password validation at the server side

In order to validate a one-time password on the server side, it is necessary to decrypt the one-time password introduced by the user. However, a modulus operation can't be reversed (the resulting double modulus operation used in Equation 3 is even more irreversible). Therefore it is not possible to decrypt the one-time password introduced by the user as usual (i.e. performing the inverse operations); for this reason it is decrypted using an exhaustive algorithm.

At the server side, the one-time password is validated by computing N_{exp} (it's computed for 1,000) expired one-time passwords generated from the actual value of Z_i that is stored in the server using the algorithm previously explained (refer to section 3). These are previous valid one-time passwords that were introduced by the user but they already expired; this is done to give feedback to the user and the user may distinguish between an invalid or an expired one-time password (valid but expired; it was already introduced by the user to the bank website before). If this is found within the N_{exp} expired it comes to inform the user that introduced one-time password was valid in the past but it already expired (i.e. it won't be accepted as valid anymore).

After this, N_{val} (10,000) valid one-time passwords are generated using the same previous approach. These are the possible valid keys for the actual state of the token. If the one-time password introduced by the user is within this range the key is accepted as valid. The value of Z_i for the key that matched minus N_{exp} expired is stored in the server. This value will be necessary for computing the N_{exp} and N_{val} one-time passwords for a future validation. If no match is found it comes to reject the introduced one-time password by the user and the Z_i value stored at the server side remains unchanged.

One advantage of the algorithm presented in this paper is that a range of only N_{val} valid keys are accepted. One-time passwords that are not valid in one moment become valid in another. This is according to the actual value of Z_i stored at the present moment at the server side. This allows us to have a very low probability that a one-time password is accepted by the server as valid, which redounds to a highly secure system.

It is taken into account that the user may waste one-time passwords by generating and not using them (i.e. the user didn't introduced to the bank website a one-time password generated by the token); for this reason there is a range N valid. This range gives us a tolerance in the number of acceptable one-time passwords; it is necessary this value to be as small as possible because this will give us a lower probability that an attacker, in a random way, may guess a one-time password. This further contri-

butes to a higher level of security. Given that these one-time passwords are of eight numeric digits and have a range N_{val} (of 10,000 valid one-time passwords). This results in a probability of almost one in 100 million that a given one-time password will be accepted as valid. On the other hand, if the user wastes more than 10,000 keys by generating one-time passwords and not introducing it at the server, a desynchronization occurs and the token becomes useless; therefore the value for the range of N_{val} should be carefully considered.

Noteworthy this maximum value for wasted one-time passwords is reset between validations, which means that if the user has a considerable amount of one-time passwords wasted the range N_{val} is reset to 10,000 at the server side once a one-time password is accepted, giving the user the maximum number of one-time passwords that can be wasted again.

One advantage of this algorithm is that it isn't necessary to store the generated one-time passwords in the server. only the value of Z_i is stored in the server. For this reason, this algorithm doesn't waste space in the server for storing expired one-time passwords nor resources to determine if the one-time password introduced by the user is part of the expired ones. Perhaps this is not significant with one user but it's taken into account that a bank institution may have millions of clients then the saved space and resources becomes significant. The algorithm also has the advantage that once a one-time password has been accepted and validated, all the previous ones are automatically expired, even in cases where they haven't been introduced to the server.

5 Number of maximum generated one-time passwords

In spite of the fact, that this algorithm is able to compute eight digits range, it is not recommended to use the full range. It is possible that an attacker may be storing the one-time passwords that have been generated by the token as they are introduced to the website by the user. This leads to the hypothetical possibility that the probability of guessing a one-time password in an arbitrary way increases. This is because the attacker knows which one-time passwords already were used in the past and he wouldn't try them again. The recommendation is to cancel the token after a million one-time passwords are remaining within it. Thereby, the probability of guessing a one-time password goes from approximately 1-in-100 million to 1-in-one million. This calculation was done by assuming that there was an attacker storing the entire history of the generated one-time passwords by the token for years. This scenario is very impractical. This is a very theoretical scenario but this is done as an extra security measure.

6 Attacking the algorithm

The way to attack the presented algorithm is to have an attacker storing the entire one-time password introduced by the user in the bank web site. Let´s say the attacker has two one-time passwords collected; to attack the algorithm a full search through the 64 bits generated random sequence is performed. This is done for every possible key *K* to find which key produce a sequence that matches those the attacker has. It may occur that there is more than one single *K* than match the sequence; if this is the case the attacker needs to wait for another one-time password. This will help to reduce the number of candidate secret keys *K* which match the stored sequence and could be the secret key. This process needs to be repeated again and again until it's found that only one key *K* match the sequence of one-time passwords the attacker has collected. If this is the case the secret key *K* has been for the token under attack. The following one-time passwords can be predicted for sure, therefore breaking the token security.

Performing this attack as the algorithm was presented in this paper (i.e. having a 64 bits Linear and Congruential Random Number Generator) is already computationally speaking a hard code to break with a PC because the attacker needs to try with 2^{64} one-time passwords times the 2^{64} possible keys resulting in 2^{128} one-time passwords in total. Nevertheless it is possible to strengthen the algorithm and protect it from this kind of attack by using a larger number of bits for the RNG; (let's say 1024 bits for instance). The algorithm presented in this paper was of 64 bits; this is because 64 bits operations can be computed with ordinary PC instructions. As explained previously it is possible to simulate operations of larger number of bits using a long integer library (this library is commonly used for cryptographic applications). With this it is possible to assure that the algorithm is totally unbreakable.

7 Open Research Issues

Equation 2 is capable to generate multiple derived random sequences for a given values for the constants *a*, *c* and *m*. A research needs to be conducted for determining if the derived random sequences are as random as the original one, or at least random enough. Another issue that is left for future research is related with the constants of the RNG. In this paper the algorithm was presented assuming a 64 bits RNG and it was also proposed to enlarge the bit number of the RNG to make the algorithm stronger. Nevertheless, a research needs to be conducted to find out the best constants values for *a* and *c* that generate the best random sequence for a 1024 bit *m* size.

To keep in touch with the research advances of this project please visit http://www.highsecret.com where related information will be posted continuously.

Acknowledgments. The first author acknowledges support from the Mexican Council of Science and Technology (CONACYT) to pursue MSc studies at CIC-IPN. The second author acknowledges National Polytechnic Institute of Mexico.

Bibliography

1. Marsaglia, George (July 2003). "Xorshift RNGs". *Journal of Statistical Software*
2. Brent, Richard P. (August 2004). "Note on Marsaglia's Xorshift Random Number Generators". *Journal of Statistical Software*
3. Panneton, François (October 2005). "On the xorshift random number generators". *ACM Transactions on Modeling and Computer Simulation (TOMACS)*
4. M. Goresky and A. Klapper, Algebraic Shift Register Sequences, Cambridge University Press, 2012
5. Linear Feedback Shift Registers in Virtex Devices, Maria George and Peter Alfke, *Xilinx press*
6. Barkam, Elad; Biham, Eli; Keller, Nathan (2008), *Journal of Cryptology*
7. Lu, Yi; Willi Meier; Serge Vaudenay (2005). "The Conditional Correlation Attack: A Practical Attack on Bluetooth Encryption"
8. Lenore Blum, Manuel Blum, and Michael Shub. "A Simple Unpredictable Pseudo-Random Number Generator", *SIAM Journal on Computing*
9. Lenore Blum, Manuel Blum, and Michael Shub. "Comparison of two pseudo-random number generators", *Advances in Cryptology: Proceedings of Crypto*
10. Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator, J. Kelsey, B. Schneier, and N. Ferguson, *Sixth Annual Workshop on Selected Areas in Cryptography, Springer Verlag*
11. Niels Ferguson and Bruce Schneier, *Practical Cryptography*, published by Wiley
12. John Viega, "Practical Random Number Generation in Software," acsac, pp. 129, 19th Annual Computer Security Applications Conference
13. Dorrendorf, Leo; Zvi Gutterman, Benny Pinkas. "Cryptanalysis of the Random Number Generator of the Windows Operating System"
14. Cryptanalysis of the Alleged SecurID Hash Function, Biryukov Alex
15. A Collection of Selected Pseudorandom Number Generators, Kart Entacher
16. Press, William H., et al. (1992). Numerical Recipes *in FORTRAN 77: The Art of Scientific Computing* (2nd ed.).