

An Effective Defense Against First Party Attacks in Public-Key Algorithms

**Stephen M. Matyas, Jr. and Allen Roginsky
IBM Corporation
3039 Cornwallis Road
RTP, NC 27709**

smatyas@us.ibm.com
roginsky@us.ibm.com

An Effective Defense Against First Party Attacks in Public-Key Algorithms

ABSTRACT

This paper describes a method for assuring that user-generated public and private key pairs are cryptographically strong. This assurance is achieved by limiting the number of attempts a user can make while generating the keys. Since it takes many billions of attempts to generate so-called “weak” keys, with any significant probability of success, our method precludes users from cheating.

The described method has a potential positive impact on several evolving cryptographic standards, where the strength of the keys used with public key cryptography are a matter of major concern. It has no negative impact on key generation performance. The method is simple and straightforward, and it can be easily performed with just a few computational steps.

KEYWORDS

- Encryption
- Hash function
- Public key

1. Introduction

Two fundamentally different encryption techniques are known to cryptographers. They are the symmetric key algorithms and the public key methods. These techniques do not compete; instead, they complement each other.

Whenever a symmetric key cryptosystem is employed, two parties, usually named Alice and Bob, share a secret key which they use for both encryption and decryption. The best-known symmetric key algorithm is the Digital Encryption Standard (DES), described in [9]. It was approved by the American National Standards Institute (ANSI) 1981 as a standard of choice for the US financial industry. To use this algorithm, Alice and Bob must share a secret key K which of party uses for an encryption and another for the decryption of data. DES is a block cipher that operates on 64-bit-long blocks of data. It is called a symmetric-key algorithm because both parties use the same key. The key K has 56 bits of secret entropy.

Over the years, DES was meticulously studied by cryptanalysts all over the world. This analysis revealed no cryptographic weaknesses. However, with the recent advances in computer technology it became possible to attack DES by “brute force”. This means that the entire space of all possible keys could be searched until the correct key was found. The most recent well-publicized brute-force attack against DES was mastered during the annual RSA Data Security Conference in January 1999. The secret key was discovered in 22 hours. This vulnerability is unacceptable for many real-life applications.

Various other symmetric key algorithms exist. We will mention the Triple DES method, where a block of data is first encrypted under one DES key, then decrypted using the second key and, last, encrypted with the third key. This procedure significantly increases the cryptographic strength of the algorithm, at the expense of the diminished performance.

In 2000 the National Institute for Standards and Technology (NIST) will select a new algorithm which will provide a long-term replacement for DES and Triple DES.

All symmetric key algorithms must rely on the existence of a complicated infrastructure that ensures that secret keys are distributed to pairs of users that need to know them. The integrity of these secret keys must also be guaranteed. In a network of n users there will therefore simultaneously exist $n(n-1)/2$ cryptographic keys. Every time a key changes, a complicated mechanism must be used to ensure that a replacement key is delivered to Alice and Bob. Even though some optimizations exist that allow a network to function with fewer keys, the problem of managing and distributing this large number of keys remains difficult to solve.

In 1976 W. Diffie and M. Hellman published a paper [11] where a solution to the key generation and distribution problem was presented. Much more importantly, this paper lays the groundwork for a creation of a new field: public key cryptography. The authors suggested that each user will be provided with two different keys, one known to the public and one kept private. They demonstrated that without the knowledge of each other's private keys, Alice and Bob could generate a key that could be used for DES or any other symmetric key algorithm. This key could not be discovered by anyone who did not know Alice's and Bob's private keys, even if a potential attacker intercepted the communications between Alice and Bob. Indeed, this communication did not have to be secret at all.

The reason that the Diffie-Hellman protocol worked was that a certain mathematical problem could not be solved without performing an extremely large number of calculations. This number is so large that finding the secret symmetric key that Alice and Bob generate is computationally impossible, if the underlying parameters (public keys) are chosen prudently. The difficult mathematical problem that makes it possible is that of solving a discrete logarithm problem in a particular finite group.

While this protocol also requires the existence of a trusted third party (TTP) that authenticates the users, the keys themselves can be generated independently by Alice and Bob. Furthermore, the mechanism of public key cryptography is ideally suited for other applications, such as digital signatures, and also for data encryption.

It turned out, that other mathematical problems can serve as underlying tools for public key cryptosystems. The RSA algorithm, named after its inventors Rivest, Shamir and Aldeman [1], utilizes the difficulty of factoring large integers into their prime factors. Yet another important example of the use of public key technology is that of an elliptic curve cryptosystem. It was first introduced by N. Koblitz in [2] and V. Miller in [3]. This system is also based on the computational difficulty of solving a discrete logarithm problem in a group but the group is different from that proposed in [11]. It is a group of points on an elliptic curve over a finite field.

To make these public key cryptosystems solid and operational, certain requirements must be imposed on public and private keys that Alice and Bob own. The sizes of the finite groups must be sufficiently large, the prime factors in the RSA algorithm must satisfy many specific constraints (for example, if p is an RSA prime, then both $p-1$ and $p+1$ must have large prime divisors – see [7]), the choice of an elliptic curve must be made very carefully, to avoid the so-called anomalous curves and to satisfy the so-called MOV condition [8].

If some of the conditions on public and private keys are not met, this might result in a weakening the security of the generated key or of that of a digital signature. A key that does not satisfy the conditions that a particular algorithm (Diffie-Hellman, RSA or Elliptic Curves) calls for is referred to as a “weak” key. The notion of weak keys is particularly important when the RSA method is employed.

Under normal circumstances, the probability of accidentally choosing a weak key is extremely low and the entire scenario may look to be too improbable to have occurred in real life. However, there is the possibility that a cheating party will deliberately try to generate a weak signing key, and then repudiate its signed messages by claiming that its signing key was “broken” by an attacker. This is called “the first party attack.” The first party attack presents a significant problem to designers of public key cryptosystems. For this reason, the ANSI standard X9.31 [7] requires that the prime numbers p and q used in the generation of the public modulus satisfy certain additional properties, which ensure that the generated keys are not too weak for the purposes of the RSA digital signature algorithm.

The ANSI X9F1 committee was concerned that a first-party attacker would not follow the prescribed prime-generation procedure but would instead purposely construct bad primes. Therefore, the committee put a requirement in the X9.31 standard that the primes, p and q , must be generated from seed values (6 in number) and that the seed values must be saved in order to later prove that the prime numbers were generated in compliance with the prescribed procedure. The X9.31 protocol relies on the use of one-way functions to prevent an attacker from working backwards to determine the seed values that lead to the generation of a particular prime pair. Hence it forces an attacker to select seeds values and use a forward construction process. Although this is an improvement (i.e., it provides some defense against the first party attack), the protocol does not guarantee that an attacker cannot find “weak” primes. A cheating party (often referred under these circumstances as the “first party”, because it is the party responsible for generating p and q and for signing messages with the generated signature key) can try many seeds until possibly it finds one that leads to the creation of “weak” primes. That is, it does not stop an attacker from performing an exhaustive attack using the “forward construction process.” Such an exhaustive attack need not be performed in real time and, therefore, allows for billions of primes to be tested. Later the first party can say that this was the very first seed that it selected, and was just unlucky.

In this paper we show a way to eliminate the opportunities for the first party attacks described above. We believe that instead of introducing many complicated requirements that would ensure that no key selected by the first party attacker is weak, it is better to limit the possibilities of trying many candidate keys. Of course, a selected key could still be “weak,” but this could occur only by pure chance. The ability to perpetrate a meaningful attack would have been removed.

Another point worthy of mentioning is that the seeds used in the generation of private keys should be distributed uniformly over the space of all possible seeds. The current standard (X9.31) satisfies this requirement, and it was therefore deemed essential that our solution method also guarantee a uniform distribution of seeds.

The rest of the paper is organized as follows.

In Section 2 we outline the general ideas and methodologies behind our algorithms. In the following three sections we will show how to use the existing mechanisms and the infrastructure of public key technology to create the keys with a guarantee they were not created by a first party attacker. In Section 3, the Diffie-Hellman ideas will be used, in Section 4 we will use the RSA method and in Section 5 – an Elliptic Curve Cryptosystem. Note that any protocol can be used to create keys for any other public key algorithm. Thus, the Diffie-Hellman method can help us generate the keys to be used with the elliptic curves, the elliptic curve protocol can, in turn, be used to generate the RSA keys, and so on.

In Section 6 we briefly talk about possible alternatives to the proposed algorithms. Section 7 addresses the security of the infrastructure needed to operate these protocols.

2. Solution Method

Here we describe a protocol between a user and a Trusted Third Party for generating a secret seed value. In this paper, the TTP will be a Certification Authority (CA) that participates in the generation of the seed. The CA is already used in cryptography and is a critical part of the public key infrastructure (see [5]). Hence involving the CA in the prevention of a first party attack does not introduce a new entity and thus does not significantly increase the overhead. We will assume, as it is usually done, that the CA is an honest partner in this protocol.

We describe three possible protocols that are almost equivalent, but rely on the strength of different public key algorithms, as follows: the discreet logarithm (DL) problem in a multiplicative group of a finite field, the RSA algorithm, and the elliptic curve algorithm. It is noteworthy that the choice of one or these three methods has nothing to do with the further use of the key that the method helps generate. Thus, an RSA algorithm can be used to generate a key later used in an elliptic curve cryptosystem, a DL approach can be used to create an RSA key, and so on. However, one should note that the elliptic curve logarithm problem is harder to solve for the given size of the security parameters, than either the problem of finding the RSA key or the discreet logarithm problem in a multiplicative group of a finite field.

The main idea behind our approach is to require that the user generates the public and private key pair from a secret seed value *Xseed*, where *Xseed* is computed from a portion of the seed selected by the user, denoted x , and a portion of the seed selected by a CA, denoted z , who cooperate using a prescribed key generation protocol. The CA also computes and returns a special value that the user can store for purposes of an audit, which enables the user to prove that the composite value *Xseed* was generated in accordance with the prescribed key generation protocol.

The prescribed key generation protocol is such that the user cannot create a seed unilaterally and if it tries many different times to generate an x (e.g., billions of times),

hoping that the resulting seed value $Xseed$ will lead to a weak key, the CA will undoubtedly catch the cheating party or at least detect that some party is attempting to cheat.

One significant advantage that each of the protocols presented below offers is that the random seeds that it creates are distributed uniformly over the space of all possible seeds. This means that the attacker will not be able to exploit any bias among the keys generated from these seeds. The requirement for uniformly distributed random seeds affects the design of the protocols somewhat. For example, by simply computing $Xseed=x+z$ in Protocol 1, we would end up with a so-called “triangle” distribution of seeds.

To further increase the security and the independence of these protocols the CA may use a separate (secret, public) key pair for these protocols. This way even if the CA accidentally signs something related with its “main” public key, it will in no way impair the strength of our protocols. Similarly, our protocols have no adverse affect on the CA operations and on the security of the users’ private keys.

We will now describe the three protocols in greater detail.

3. Generating Keys Relying on a “Conventional” Discrete Logarithm Problem

This algorithm is based, like the original Diffie-Hellman idea, on the difficulty of finding an integer y , given a large prime p , an integer g between 2 and $p-1$ and the value of $g^y \pmod{p}$. The input to this algorithm includes the following values: L - the length in bits of a seed to be generated; J - the bit length of the prime number that guarantees the security of this method. (The size of J must be such that it would be considered infeasible for an attacker to solve the corresponding discrete logarithm problem); p_0 - a publicly known prime number of the length at least J bits; g - a publicly known generator used to secure communications between the user and the Certification Authority. It also utilizes a cryptographic hash function H (see [10] and also [4]) and is depended on the CA’s public and private key pair used for signing (PKca, SKca).

The output is a seed $Xseed$ that can be used to generate keys for a public key algorithm..

The procedure to generate $Xseed$ is as follows.

The user randomly generates an integer x between 1 and 2^L-1 . The user then computes $y = g^x \pmod{p_0}$ and sends y to the CA. The Certification Authority generates uniformly randomly a (non-secret) integer z between 0 and 2^L-1 , computes $w=H(y * g^z \pmod{p_0})$ and signs it with its private key SKca. This signed value is called $Sigw$. The CA next

increments a counter to track the number of times this user asked for assistance establishing seeds. This can be done per user or for the entire CA. If the count is larger than some reasonable threshold for a particular user (or for the CA if the counters are not kept per user) then the Certification Authority might suspect that a user is attempting a first party attack and the CA will send a warning to a system administrator.

The CA now sends z and $Sigw$ to the user. The user computes $Xseed = x + z \pmod{2^L}$. The values of $Xseed$ are uniformly distributed over the set of integers between 0 and $2^L - 1$, since z is drawn uniformly randomly from the set of integers between 0 and $2^L - 1$. The user then computes $w = H(g^{x+z} \pmod{p_0})$ and verifies the signature on w ($Sigw$) using the public verification key of the Certification Authority, PKca. This ensures the user that an audit can be passed.

The user now saves $Sigw$ and $Xseed$ and proceeds to generate the public keys using $Xseed$ as seed. If more than one seed is needed then a similar procedure can be used to generate each required seed.

If there is an audit, the user will have to prove that the proper procedure was followed. The user does the following:

Compute $w1 = H(g^{Xseed} \pmod{p_0})$ and compute $w2 = H(g^{Xseed'} \pmod{p_0})$ where $Xseed' = Xseed + 2^L$. Since $0 < x + z < p_0$, (the latter inequality holds since $J > L + 1$), there is no "false" value Xs between 0 and $2^L - 1$ such that either $g^{Xs} \pmod{p_0}$ or $g^{Xs+2^L} \pmod{p_0}$ equals $g^{x+z} \pmod{p_0}$. Next the user validates $w1$ and $w2$ using $Sigw$ and the public verification key of the CA, PKca. If either $w1$ or $w2$ is valid, then the user did everything according to the rule.

Note. Only one of the values $w1$, $w2$ is correct. The other one is not, so by making it valid for the purposes of the audit, we introduce a chance that a cheating user can pass the audit while using an incorrect seed. However, the probability of this happening is extremely low.

4. Generating Keys Using the RSA Method

The input to this procedure includes L - the length in bits of a seed to be generated; N - a publicly known composite number of the length $L + 1$ bits ($N = p * q$, where p and q are large unknown primes. The secret parameters p and q are generated by the user and are certified by a trusted third party); e - an RSA public exponent and also H , PKca and SKca that are defined as in the previous section.

The output is a seed $Xseed$ that can be used to generate keys for public key cryptography.

The procedure to generate $Xseed$ is as follows.

First, the user randomly generates an integer x between 1 and $N-1$, such that it is mutually prime with N . Next the user computes $y = x^e \pmod{N}$ and sends it to the CA. The CA randomly generates a (non-secret) integer z between 1 and $N-1$, computes $w = H(y * z^e \pmod{N})$ and signs it with SKca. This signed value is *Sigw*. Note that if z takes all integer values between 1 and $N-1$, then, since x is mutually prime with N , $x * z \pmod{N}$ also takes all possible values between 1 and $N-1$. Each of these values will occur exactly once while z is changing between 1 and $N-1$, so the likelihood for $x * z \pmod{N}$ to take any given value is the same and is equal to $1/(N-1)$, hence resulting in a uniform distribution. Next the Certification Authority increases by 1 the count for the number of times the user asked for assistance establishing the seeds. The CA then sends z and *Sigw* to the user.

The user now computes $Xseed = x * z \pmod{N}$. If $Xseed \geq 2^L$ then the user must retry using a new x . Note also that since the probability of returning to Step 1 here is less than 0.5, one can expect to succeed in moving to the next step only after a small number of attempts. The user then computes $w = H((x * z)^e \pmod{N})$ and verifies the signature on w (*Sigw*) using the public verification key of the CA, PKca. The user now saves *Sigw* and *Xseed* and proceeds to generate the public keys using *Xseed* as seed.

If there is an audit, the user will have to prove that the proper procedure was followed. The user will compute $w = H((Xseed)^e \pmod{N})$ and then validate it using *Sigw* and the public verification key of the CA, PKca. If the signature is valid, the audit is passed successfully.

5. Generating Keys Using Elliptic Curves

The security of data exchange between the user and the CA is based here on the strength of the discrete logarithm problem for a group of points on an elliptic curve over a finite field. The reader can find a lot of information on this problem and on the use of the elliptic curve in cryptography in general in [6].

The following parameters must be provided as an input to this procedure: L - the length in bits of a seed to be generated; J - the bit length of the prime number n that guarantees the security of this method. (The size of J must be such that it would be considered infeasible for an attacker to solve the corresponding elliptic curve discrete logarithm problem. In any case, $J > L + 1$; this assures that $x + z < n$); E - an elliptic curve over a finite field F_q . This is a set of points that satisfy a certain algebraic relationship (see [6].)

The number of points on this curve is nh , where n is a large prime (has at least J bits) and h is a small cofactor. G - a base point. It is a point of order n on E . The other input parameters H , PKca and SKca are defined like in previous two sections.

The output is a seed $Xseed$ that can be used to generate keys for public key cryptography.

The procedure to generate $Xseed$ is as follows.

The user randomly generates an integer x between 1 and $2^L - 1$. The user then computes a point $Y = xG$ on the elliptic curve E and sends it to the Certification Authority. The CA randomly generates a (non-secret) integer z between 0 and $2^L - 1$. The CA then computes a point $P = Y + zG$ on E . The coordinates of P are denoted x_p and y_p . The CA computes $w = H(x_p)$ and signs it with SK_{ca} . As in the previous two algorithms, this signed value is called $Sigw$. The Certification Authority increases by 1 the count for the number of times the user asked for assistance establishing the seeds. The CA then sends z and $Sigw$ to the user.

The user now computes $w = H(x_p)$, where x_p is the x-coordinate of the point $(x + z)G$ on E . The user then verifies the signature on w ($Sigw$) using the public verification key of the CA, PK_{ca} . The user now saves $Sigw$ and $Xseed$ and proceeds to generate the public keys using $Xseed$ as seed. This ensures that the user can pass an audit.

If an audit is performed, the user will compute $w1$ and $w2$, the hash values of the x-coordinates of the points $(Xseed)G$ and $(Xseed + 2^L)G$, correspondingly, on E , and then proceed as shown in Section 3.

6. Alternative Procedures

Each of the three protocols described above could be used in a slightly different manner. The certificate that the CA issues would contain either the value w or both w and $Sigw$. The advantage of this alternative algorithm is that the user does not have to store $Sigw$. A possible disadvantage is that it might require a slight change to the existing standard protocol between the user and the Certification Authority.

7. CA Security

One should notice that the proposed scheme requires the CA to sign many messages, in addition to its usual task of signing the certificates. This raises a question of whether it will be possible to learn more about the CA's private key or to use it as an oracle by making it sign various messages whose context the CA can not fully control.

We do not see it as a problem. The CA's private key is supposed to be absolutely secure so even if it signs many sets of data, no useful information should be derived from these signatures. In addition, it is possible to require the CA to have another (private, public) key pair to be used only to perform the operations outlined in this TR.

8. Conclusion

We presented an algorithm that assures that a seed used for the user-generated public and private key pair is strong. This assurance comes from the fact that a first party attack based on the multiple tries to find a weak key pair has been made impossible. The resulting seeds are distributed uniformly over the space of all possible keys. The principle innovation is to the Certification Authority limits the number of attempts the user can make in generating a public and private key pair. The security of the proposed method relies on the difficulty of solving one of the following problems: (1) the discrete logarithm problem in a multiplicative group of a finite field, (2) the integer factorization problem, or (3) the discrete logarithm problem in a group of points in an elliptic curve over a finite field. Regardless of what method was chosen to assure the security of our method, the generated keys can be used in any algorithm suitable for public key cryptography.

References

- [1] Rivest, R.L., Shamir, A., and Adleman, L.M., "*Cryptographic Communications System and Method*," US Patent #4,405,829, 20 Sep., 1983.
- [2] Koblitz, N., "*Elliptic Curve Cryptosystems*," *Mathematics of Computation*, 48 (1987), pp. 203-209.
- [3] Miller, V., "*Uses of Elliptic Curves in Cryptography*," *Advances in Cryptology - CRYPTO 85, Lecture Notes in Computer Science*, 218 (1986), pp. 417-426, Springer-Verlag.
- [4] Schneier, B., "*Applied Cryptography*," 2nd edition, John Wiley & Sons Inc, 1996.
- [5] Menezes, A.J., van Oorschot, P.C., and Vanstone, S.A., "*Handbook of Applied Cryptography*," CRC Press, 1997.
- [6] Menezes, A.J., "*Elliptic Curve Public Key Cryptosystems*," Kluwer Academic Publishers, Fourth Printing, 1997.
- [7] ANSI Standard X9.31-1998, Digital Signatures Using Reversible Public Key Cryptography For The Financial Services Industry (rDSA).
- [8] ANSI Standard X9.62-1998, The Elliptic Curve Digital Signature Algorithm.

- [9] National Bureau of Standards, NBS FIPS PUB 46, "*Data Encryption Standard*," National Bureau of Standards, U.S. Department of Commerce, Jan. 1977.
- [10] ANSI Standard X9.30-1993, Part 2: "*Public Key Cryptography Using Irreversible Algorithms for the Financial Services Industry: The Secure Hash Algorithm 1 (SHA-1)*" (Revised). 1993.
- [11] Diffie, W. and Hellman, M.E., "*New Directions in Cryptography*," IEEE Transactions on Information Theory, v. IT-22, n. 6, Nov 1976, pp. 644-654.