Breaking the Rabin-Williams digital signature system implementation in the Crypto++ library

Evgeny Sidorov,
Yandex LLC
e-sidorov@yandex-team.com

April 16, 2015

Abstract

This paper describes a bug in the implementation of the Rabin-Williams digital signature in the Crypto++ framework. The bug is in the misuse of blinding technique that is aimed at preventing timing attacks on the digital signature system implementation, but eventually results in an opportunity to find the private key having only two different signatures of the same message. The CVE identifier of the issue is CVE-2015-2141.

1 Introduction

The Rabin-Williams digital signature system (RW) is a signature system based on the difficulty of the integer factorization problem. Originally this signature system was proposed by Rabin in [9] and was improved by Williams in [10]. The RW digital signature is similar to the RSA system, but it has a number of advantages. One of them is the use of a small exponent (usually it's 2) that makes the verification procedure blazingly fast.

The RW signature system variant was standardized by IEEE in [2] and ISO [3], but nowadays there are few implementations of it in popular cryptographic frameworks though a number of open source reference implementations can be easily found in the Internet.

Crypto++ ([1]) is a popular cryptographic framework that contains an implementation of the Rabin-Williams signature system as well as a number of padding schemes used with it. The implementation corresponds the description of the RW system given in [2], but with a small difference. The authors decided to enforce the code with a blinding technique aimed at preventing timing attacks. The technique works for the RSA signature system but in case of the RW system it results in an opportunity to find the private key having at least two signatures for the same message.

In a nutshell, the class `RWSS<P1363_EMSA2,H>::Signer` (where `H` is a hash function) being applied to a fixed message several times produces a sequence of signatures $s_1, s_2, s_3, \ldots$ so that

$$s_i^2 \equiv s_j^2 \ (mod\ N)$$

where $N = p \cdot q$ is a RW modulus and s_i, s_j are likely to be different. Simply computing $GCD(s_i - s_j, N)$ an attacker can easily reveal a non-trivial factor of N and thus can recover the private key.

2 Modular square roots

This section contains some well-known number theory facts and definitions that will be used later on. All these facts can be found in [8].

Let $\mathbb{Z}_n^*$ be the multiplicative group of integers modulo n. Let $QR(n)$ be the subgroup of quadratic residues modulo n, and $QNR(n)$ is the set of all non-residues modulo n.

Definition Let p and q be two distinct primes such that $p = 3\ (mod\ 4)$ and $q = 3\ (mod\ 4)$, and let $n = p \cdot q$. Such integers n are called *Blum integers*. Let $a \in QR(n)$ be a quadratic residue modulo n. The unique square root of a in $QR(n)$ is called the *principal square root* of a modulo n.

Theorem 2.1 *If $n = p \cdot q$, p and q are two distinct primes each congruent to 3 modulo 4, then the function $f : QR(n) \to QR(n)$ defined by $f(x) = x^2\ (mod\ n)$ is a permutation. The inverse function of f is:*

$$f^{-1}(x) = x^{((p-1)\cdot(q-1)+4)/8}\ (mod\ n)$$

To inverse the function f there's no need to use the formula above. Square roots of x modulo p and q can be found separately and then turned into one modulo n using the well-known CRT (Chinese Remainder Theorem) theorem.

If $p \equiv 3\ (mod\ 8)$ and $q \equiv 7\ (mod\ 8)$, then Legendre symbols of (-1) and 2 are

$$\left(\frac{-1}{p}\right) = -1,\ \left(\frac{-1}{q}\right) = -1,\ \left(\frac{2}{p}\right) = -1,\ \left(\frac{2}{q}\right) = 1.$$

Using these values and the fact that for an arbitrary integer x its Jacobi symbol is $\left(\frac{x}{p\cdot q}\right) = \left(\frac{x}{p}\right) \cdot \left(\frac{x}{q}\right)$ we can *tweak* any element of $\mathbb{Z}^*_{pq}$ so that the Jacobi symbol of the tweaked value is 1.

Definition Let $n = p \cdot q$ be a product of two distinct primes $p = 3\ (mod\ 8)$ and $q = 7\ (mod\ 8)$. Let $h \in \mathbb{Z}^*_n$. Triplet (e, f, s) where $e \in \{-1, 1\}$, $f \in \{1, 2\}$ and $e \cdot f \cdot s^2 = h\ (mod\ n)$ is called *tweaked square root* of h modulo n.

If (e, f, s) is a tweaked square root of h modulo n and $s \in QR(n)$, then the vector (e, f, s) is called *principal tweaked square root* of h modulo n.

If (e, f, s) is a principal tweaked square root of h modulo n, then the vector $(e, f, min\{n - s, s\})$ is called $|$*principal*$|$ *tweaked square root* of h modulo n.

The following theorem shows how tweaked values are used to build a digital signature system.

Theorem 2.2 *Let p and q be distinct primes each congruent 3 modulo 4, and let $n = p \cdot q$. Let x be an integer having Jacobi symbol $\left(\frac{x}{n}\right) = 1$, let $d = ((p - 1) \cdot (q - 1) + 4)/8$. Then*

$$x^{2d} = \begin{cases} x, & \text{if } x \in QR(n) \\ n - x, & \text{if } x \in QNR(n) \end{cases} \tag{1}$$

And the following two results are used to break the implementation of the Rabin-Williams digital signature system.

Theorem 2.3 *Let x, y, n be integers. If $x^2 \equiv y^2\ (mod\ n)$ but $x \not\equiv \pm y\ (mod\ n)$, then $GCD(x - y, n)$ is non-trivial factor of n.*

Theorem 2.4 *Let $n = p \cdot q$ a product of two distinct primes. If x, y are random elements of $\mathbb{Z}^*_n$ so that $x^2 \equiv y^2\ (mod\ n)$, then the probability that $GCD(x - y, n)$ is a non-trivial factor of n equals $\frac{1}{2}$.*

3 Rabin-Williams digital signature system

This section describes the Rabin-Williams signature system and `EMSA2` encoding scheme in the way they are defined in [2]. Although there are some variants of this digital signature system defined in [2], the only one variant is implemented in the `Crypto++` library and the only this variant will be described here.

3.1 Public and private keys

The public key in the RW digital signature system is $N = p \cdot q$ a product of two large primes p and q such that

$$p \equiv 3 \ (mod\ 8) \text{ and } q \equiv 7 \ (mod\ 8).$$

The exponent is the fixed value 2 and so there is no need to include it into the public key.

The private key of the RW system is a tuple (p, q, c) where p and q are as stated above and $c \equiv q^{-1} \ (mod\ p)$.

3.2 The signing algorithm

Input:

1. The signer's RW private key (p, q, c)
2. The message representative, which is an integer f such that $0 \leq f < N$ and $f \equiv 12 \ (mod\ 16)$

Output: The signature which is an integer s such that $0 \leq s < \frac{N}{2}$.

Operation:

```
if the Jacobi symbol $\left(\frac{f}{N}\right) = 1$ then
    $u \leftarrow f$
else
    $u \leftarrow \frac{f}{2}$
end if
$j_1 \leftarrow exp(u,\ (p+1)/4) \ (mod\ p)$
$j_2 \leftarrow exp(u,\ (q+1)/4) \ (mod\ q)$
$h \leftarrow c \cdot (j_1 - j_2) \ (mod\ p)$
$s \leftarrow min(j,\ N - j)$
return $s$
```

The result value s is the third component of the |principal| tweaked square root of f modulo N, but in this particular case there is no need to store tweaks as they can be easily recovered knowing that $f \equiv 12 \ (mod\ 16)$. The recovering process is shown in the verification algorithm.

3.3 The verification algorithm

Input:

1. The signer's public key N.

2. The signature to be verified, which is an integer s.

Output: The message representative, which is an integer f such that $0 \leq f < N$ and $f = 12\ (mod\ 16)$ or “invalid”.

Operation:

```
if $s \notin [0, \frac{N-1}{2}]$ then return “invalid”
end if
$t_1 \leftarrow s^2\ (mod\ N)$
$t_2 \leftarrow N - t_1$
if $t_1 \equiv 12\ (mod\ 16)$ then
    $f \leftarrow t_1$
else if $t_1 \equiv 6\ (mod\ 8)$ then
    $f \leftarrow 2 \cdot t_1$
else if $t_2 \equiv 12\ (mod\ 16)$ then
    $f \leftarrow t_2$
else if $t_2 \equiv 6\ (mod\ 8)$ then
    $f \leftarrow 2 \cdot t_2$
else return “invalid”
end if
return $f$
```

3.4 EMSA2 encoding scheme

Before applying the signing procedure the message representative should be computed. As it was mentioned above the message representative must be congruent to 12 modulo 16 and to achieve this a proper message encoding scheme should be used. There are several different approaches to computing message representatives. The standard [2] defines only two of them - `EMSA2` and `EMSA4`.

Although both of these schemes are implemented in the `Crypto++` framework, only `EMSA2` is mentioned in its documentation in relation with the RW digital signature, and only this scheme is affected by the issue as it doesn't add any randomness during computation of a message representative.

To generate the representative of a message m (here we assume that the length of the message is greater than 0) according to the `EMSA2` scheme the following steps should be performed:

1. Compute the hash value `H` of the message m with the selected hash function

2. Make an octet string in the following way

`0x6b||0xbb...0xbb||0xba||H||HashID||0xcc`

where `HashID` is a one byte that defines the used hash function. A complete list of the `HashID` values can be found in [2].

The number of `0xbb...0xbb` octets is $len(N) - len(H) - 4$, where $len(N)$ and $len(H)$ are the number of octets in binary representation of N and the number of octets in hash value H respectively. The `EMSA2` padding scheme doesn't include any randomness into a message representative.

4 Timing attacks on modular exponentiation

Many public-key algorithms use modular exponentiation and usually this operation is performed with the following algorithm (can also be found in [7]).

Input: Integers x, y, N. Binary representation of x consists of w bits.
Output: Integer $R = y^x$ $(mod\ N)$.
Operation:

```
$s_0 \leftarrow 1$, $k \leftarrow 0$
while $k \leq (w - 1)$ do
    if (bit $k$ of $x$ is 1) then
        $R_k \leftarrow (s_k \cdot y)$ $(mod\ N)$
    else
        $R_k \leftarrow s_k$
    end if
    $s_{k+1} \leftarrow R_k^2$ $(mod\ N)$
    $k \leftarrow k + 1$
end while
return $R_{w-1}$
```

Observing execution time of the algorithm above an attacker can reveal information about x because the execution time depends on the number of ones in the binary representation of x. Additional information about such attacks and their countermeasures can be found in [7].

One approach to mitigate these attacks is to use the so-called *blinding* technique. The main idea of the method is to add randomness before doing exponentiation and after the operation has been performed - remove the random parameter.

For the RSA signature system this can be done in the following way. Let N is an RSA modulus, m is message representation and e, d are public and private exponents respectively. To perform blinding a random integer r is generated and $r^{-d}\ (mod\ N)$ is calculated. The signing algorithm consists of three steps:

1. Blinding step: calculate $m \cdot r\ (mod\ N)$
2. Modular exponentiation: calculate $(m \cdot r)^d\ (mod\ N)$
3. Unblind the result: $(m \cdot r)^d \cdot r^{-d}\ (mod\ N) = m^d\ (mod\ N)$

The approach above works fine for the RSA public key system, but it cannot be directly applied to the Rabin-Williams digital signature.

5 The Rabin-Williams digital signature system implementation

As it was mentioned before, the vulnerable class of the `Crypto++` library is `RWSS<P1363_EMSA2,H>::Signer`. In its internals the `EMSA2` message representative is computed and the `InvertibleRWFunction::CalculateInverse` function is called. The whole `C++` listing of this function is:

```
Integer InvertibleRWFunction::CalculateInverse(
   RandomNumberGenerator &rng, const Integer &x) const
{
    DoQuickSanityCheck();
    ModularArithmetic modn(m_n);
    Integer r, rInv;
    do
    {
        r.Randomize(rng, Integer::One(), m_n - Integer::One());
        rInv = modn.MultiplicativeInverse(r);
    }
    while (rInv.IsZero());
    Integer re = modn.Square(r);

    re = modn.Multiply(re, x);
    Integer cp=re%m_p, cq=re%m_q;
    if (Jacobi(cp, m_p) * Jacobi(cq, m_q) != 1)
    {
        cp = cp.IsOdd() ? (cp+m_p) >> 1 : cp >> 1;
        cq = cq.IsOdd() ? (cq+m_q) >> 1 : cq >> 1;
    }

    #pragma omp parallel
        #pragma omp sections
        {
            #pragma omp section
               cp = ModularSquareRoot(cp, m_p);
            #pragma omp section
               cq = ModularSquareRoot(cq, m_q);
        }
31
32      Integer y = CRT(cq, m_q, cp, m_p, m_u);
33      y = modn.Multiply(y, rInv);
34      y = STDMIN(y, m_n-y);
35      if (ApplyFunction(y) != x)
36          throw Exception(Exception::OTHER_ERROR,
37              "InvertibleRWFunction:␣computational␣error␣during␣private␣key␣
        operation");
38      return y;
39  }
```

In the above function the argument `x` is a message representative generated from a message using the `EMSA2` scheme. Let's denote this value as x and from encoding algorithm we know that $x \equiv 12 \ (mod \ 16)$. Let (e, f, s) also be the principal tweaked square root of x modulo n, so by the definition:

$$e \cdot f \cdot s^2 \equiv x \ (mod \ n)$$

.

In the lines 7 to 13 a random integer is generated and stored in the `r` variable, then its multiplicative inverse modulo n is calculated and stored in the `rInv` variable. Let denote these values by r and r^{-1} respectively. The value in `r` is also squared modulo n and stored in the `re` variable, let denote it by r^2.

In the 15th line the blinding is applied to `x` and as a result `re` contains the $x \cdot r^2 \ (mod \ n)$ value.

Then using the fact that $\mathbb{Z}_n^* \cong \mathbb{Z}_p^* \times \mathbb{Z}_q^*$ all calculations are made in the two multiplicative groups $\mathbb{Z}_p^*$ and $\mathbb{Z}_q^*$. In those subgroups the value $x \cdot r^2 \ (mod \ n)$ is multiplied by f^{-1} and then an operation similar to square root extraction is performed. Then two results modulo p and modulo q are combined into one modulo n.

The multiplication by f^{-1} ensures that the Jacobi symbol

$$\left(\frac{x \cdot r^2 \cdot f^{-1}}{n}\right) = 1$$

and the theorem 2.1 can be applied. According to the theorem and the fact that $r^2 \in QR(n)$ as a result in the 32nd line we get

$$e \cdot s \cdot r_p \ (mod \ n)$$

where e and s are parts of the principal square root vector of x and r_p is the principal square root of r^2 modulo n.

Then the 33rd line is the unblinding step and the previous value is multiplied by r^{-1}, so it becomes $e \cdot s \cdot r_p \cdot r^{-1} \ (mod \ n)$. Actually the step doesn't remove the blinding value as $r_p \cdot r^{-1} \ (mod \ n)$ doesn't always equal 1. In fact

$r_p \cdot r^{-1} \equiv 1 \ (mod\ n)$ if and only if r was chosen to be equal r_p and this holds only in 1/4 of all random choices of r.

The 34th line adds a multiplier ϵ that is either -1 or 1 depending on the result of the `STDMIN` function. So after this step we have the following value

$$\epsilon \cdot e \cdot s \cdot r_p \cdot r^{-1} \ (mod\ n),$$

and this is a result of the function. The square modulo n of this value is

$$(\epsilon \cdot e \cdot s \cdot r_p \cdot r^{-1})^2 \equiv s^2 \cdot (r^2 \cdot r^{-2}) \equiv s^2 \equiv e \cdot f^{-1} \cdot x \ (mod\ n),$$

where the last equivalence holds according to the theorem 2.1. The $e \cdot f^{-1} \cdot x \ (mod\ n)$ won't change if we apply signing function once again to the same message as the `EMSA2` scheme doesn't add any randomness to the message representative. By applying the signing function to a fixed message we'll get the sequence of values $l_1, l_2, \ldots$ in which every $l_i = \epsilon^{(i)} \cdot e \cdot s \cdot r_p^{(i)} \cdot (r^{(i)})^{-1} \ (mod\ n)$ where e and s are fixed, $r_p^{(i)}$ and $(r^{(i)})^{-1}$ are calculated from a random blinding value $r^{(i)}$, and $\epsilon^{(i)} \in \{-1, 1\}$ is the sign added by the `STDMIN` function.

For any two elements of the sequence $l_1, l_2, \ldots$ holds that $l_i^2 \equiv l_j^2 \ (mod\ n)$ and according to the theorems 2.3 and 2.4 calculating of $GCD(l_i - l_j, n)$ gives us a non-trivial factor of n with 1/2 probability that allows to recover the private key.

The following `C++` code shows how to attack the implementation using classes from the `Crypto++` library.

```cpp
/* privKey1 equals privKey2 */
/* and the same should be held for pubKey1 and pubKey2 */
RWSS<P1363_EMSA2, SHA256>::Signer rwSigner1(privKey1);
RWSS<P1363_EMSA2, SHA256>::Verifier rwVerifier1(pubKey1);

RWSS<P1363_EMSA2, SHA256>::Signer rwSigner2(privKey2);
RWSS<P1363_EMSA2, SHA256>::Verifier rwVerifier2(pubKey2);

s_len1 = rwSigner1.SignMessage(rng2, (const byte*)msg.c_str(),
                 msg.length(), signature1);
Integer sng1(signature1, s_len1);

s_len2 = rwSigner2.SignMessage(rng2, (const byte*)msg.c_str(),
                 msg.length(), signature2);

Integer sng2(signature2, s_len2);

Integer gcd = Integer::Gcd((sng1 - sng2), params1.GetModulus());
```

6 How to fix the issue

To fix the bug one should ensure that the value used for blinding is a quadratic residue modulo p and modulo q. This condition guarantees that the blinding value will be removed at the unblinding step and won't affect the result of the signing procedure.

In terms of the code the `while` loop should be implemented as follows.

```
do
{
   r.Randomize(rng, Integer::One(), m_n - Integer::One());
   rInv = modn.MultiplicativeInverse(r);
}
while (rInv.IsZero() ||
      (Jacobi(r % m_p, m_p) == -1) || (Jacobi(r % m_q, m_q) == -1));
```

This approach definitely reduces the performance of the signing function, but it seems to be the easiest way to fix the bug. Actually the function itself is quite slow and can be reimplemented without calculations of the Legendre and Jacobi symbols. The faster algorithm can be found in [6].

7 Conclusions

The authors of `Crypto++` aimed at improving the security of the Rabin-Williams signature system implementation but eventually made the system completely insecure. Some researchers call such bugs "bugdoors" as on the one hand they are usual backdoors (allow to break the system for those who know about them), but on the other hand such issues remain bugs that are likely to be made unconsciously. This case proves once again that "open" doesn't mean "secure" and all popular crypto frameworks should be carefully audited, especially if they are used in other security systems.

8 Acknowledgements

I am grateful to Martijn Grooten and the Yandex Product Security Team members for their encouragement, helpful comments and useful suggestions.

References

[1] Crypto++ Library 5.6.2 - a Free C++ Class Library of Cryptographic Schemes. URL: `http://cryptopp.com/`.

[2] IEEE Standard Specifications for Public-Key Cryptography - Amendment 1: Additional Techniques. *IEEE Std. P1363a-2004*, 2004. URL: `http://standards.ieee.org/findstds/standard/1363a-2004.html`.

[3] Information technology - Security techniques - Digital signatures with appendix - Part 2: Integer factorization based mechanisms. *ISO/IEC 14888-2:2008*, 2008. URL: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=44227`.

[4] M. Bellare and P. Rogaway. The Exact Security of Digital Signatures-how to Sign with RSA and Rabin. In *Proceedings of the 15th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'96, pages 399–416, Berlin, Heidelberg, 1996. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1754495.1754541`.

[5] D. J. Bernstein. Proving Tight Security for Rabin-Williams Signatures. In N. P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2008.

[6] D. J. Bernstein. RSA signatures and Rabin-Williams signatures: the state of the art. 2008. URL: `http://cr.yp.to/papers.html#rwsota`.

[7] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, 1996. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=646761.706156`.

[8] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1996. URL: `http://cacr.uwaterloo.ca/hac/`.

[9] M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical Report 212, MIT Laboratory for Computer Science, 1979. URL: `http://ncstrl.mit.edu/Dienst/UI/2.0/Describe/ncstrl.mit_lcs/MIT/LCS/TR-212`.

[10] H. Williams. A modification of the RSA public-key encryption procedure (Corresp.). *Information Theory, IEEE Transactions on*, 26(6):726–729, Nov 1980. `doi:10.1109/TIT.1980.1056264`.