A-012

Experiment about ElGamal signature on GnuPG

Kazuki Tsubo[†] Thomas Zeugmann[†]

1. Introduction

Nowadays, cryptography is sought for in a variety of applications. These applications include even the normal user's daily usage of the Internet. For example, we wish to protect private information such as credit card numbers and personal addresses, data stored on a hard disk, our email, and so on. Clearly, any company or governmental institution has even much more data that should be protected.

So, the problem is which cryptographic software one should use. On the one hand, looking at the progress made in the last decades, one is tempted to think that there is more and more secure cryptography around. On the other hand, if no source code is available, how can we be sure that the cryptography implemented is really reliable and secure? Looking at the relevant literature is rapidly resulting in the insight that the one cannot be too careful. The border between good and bad cryptography is very thin, see e.g., [1, 2, 4]. Often, important details are overlooked when cryptographic algorithms or cryptographic protocols are implemented. As a results, the implemented versions may become vulnerable to specifically chosen attacks.

2. The ElGamal Signature Scheme

GnuPG v1.2.3 allowed for the usage of the ElGamal signature scheme [3] used for both signature and encryption. Therefore, we first describe this scheme here.

Let H be a hash function. By default, the SHA-1 hash function is used in GnuPG v1.2.3. Furthermore, let p be a large prime, and let g be a randomly chosen generator of $\mathbb{Z}_p{}^*$. These numbers be shared as public information.

For the key generation scheme, choose randomly a number x which satisfies $1 < x < p-1$. This number is the *private key*. Next GnuPG computes $y \equiv g^x \bmod p$. The *public key* is (p, g, y). Note that the private key x is also used for making the signature scheme.

The ElGamal signature scheme uses the public key when performing encryption and verifying a signature.

3. Key Generation

We need a prime p such that $p-1$ has only large prime factors. Usually, we try $p = 2q+1$. Here q has to be necessarily a large prime number. So, one chooses a large prime number q, computes $p = 2q+1$ and checks whether or not $p = 2q+1$ is prime. If it is, we have found a desired p. But if p is not a prime, we have to retry with a different q.

This may be time consuming. Thus, the actual implementation in GnuPG v1.2.3 uses a different approach. Here the so-called Wiener table is used to ensure that $p-1$ has only large prime factors. More specifically, p is chosen in **generate_elg_prime** of the file **cipher/primegen.c**. And a generator g is computed by the function **generate** of the file **cipher/elgamal.c**.

Once p is selected, a generator g is found by testing successively potential generators, starting with the number 3. If 3 is not a generator, the next number tested is 4, and so on. By construction, all factors of $(p-1)/2$ have at least $q_{bit} \geq 119$ bits, and $g > 2$.

However, as we shall see below, the most significant deviation from the theoretical requirements occurs in GnuPG v1.2.3 when it comes to selecting x. In the actual implementation, the private exponent x is not chosen as a random number modulo $p-1$, because decryption will be much faster when the bit-length of x is less than $3q_{bit}/2$.

3.1 Signature

Next, we describe the signature scheme.

Signature: The *signature* of a message already formatted as an integer m modulo p, is the pair (a, b) where:

$$\begin{aligned} a &\equiv g^k \bmod p\ ; && (1)\\ b &\equiv (m - ax)k^{-1} \bmod p-1\ . && (2) \end{aligned}$$

Here $k \in \mathbb{Z}$ is a "random" number being coprime with $p-1$. Furthermore, the implementation ensures that the bit-length of k is not less than $3q_{bit}/2$bits.

Verification: For verifying a signature, the public key is used. So, let a signature (a, b) be given. GnuPG verifies a signature (a, b) by checking if

$$0 < a < p\ ; \qquad (3)$$

$$g^m \equiv y^a a^b \bmod p\ . \qquad (4)$$

If (3) and (4) are fulfilled, the verifier accepts the signature (a, b).

[†] Graduate School of Information Science and Technology, Hokkaido University.

First, note that such a signature verification does not prevent malleability: if (a, b) is a valid signature of m then $(a, b+u(p-1))$ is another valid signature of m for all $u \in \mathbb{Z}$, because there is no range check over b.

Second, the correctness of this verification is obtained as follows. By construction (cf. Equation (2)):

$$m \equiv ax + bk \bmod p-1 \ . \tag{5}$$

Recalling that $y \equiv g^x \bmod p$ and using (1), by Fermat's little theorem we obtain:

$$\begin{aligned} g^m &\equiv g^{ax} g^{bk} \bmod p \\ &\equiv (g^x)^a (g^k)^b \bmod p \\ &\equiv y^a a^b \bmod p \ . \end{aligned}$$

4. Implementing the Attack

Next, we describe how we did implement the attack found by [4] which is based on using the weakness of (5). This weakness is caused by the fact that k and x are unusually small.

As INPUT we have a public key (p, g, y) and a signature (a, b) of the *message* m having been signed by using (p, g, y).

The OUTPUT is then the *private key* x.

We wrote the code in the following steps. First, we installed the GnuMP library, since we need multiprecision integer arithmetic (called MPI). Second, we installed Shoup's [5] NTL library which is used to solve the closest vector problem (abbr. CVP).

4.1 Preparation

When initializing GnuPG v1.2.3, it can output two types of files. The first one is an octet stream (binary) file having the extension gpg. The second one is in radix-64 format (i.e., in ASCII). The radix-64 output is used to publicize the public key, e.g., on a web-page. For example, the hexadecimal description of a public key is shown in Figure 1 and the radix-64 output is displayed in Figure 2. Note that the expressions in Figures 1 and 2 describe the same public key. However, the hexadecimal expression uses 8 bits per symbol, while the radix-64 encoding uses only 6 bits per symbol. Thus, in radix-64 two bits are not visible.

```
99 02 0d 04 4a 41 f7 55 14 04 00 e4 f7 88 de 44
...
```

Figure 1: Hexadecimal format sample

So, in order to implement the attack, one has to perform a hex-to-ascii transformation (see Figure 3), where "99 02 0d 04" expresses "10011001 00000010 00001101 00000100" as bit stream. Now "100110" stands for m, "010000" for Q, and so on. To continue and for verification purposes, two more steps are necessary. The gpg file is processed by using the od program:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.2.3 (GNU/Linux)

mQENBEpB91UUBADk94...
...
...
=4HR1
-----END PGP PUBLIC KEY BLOCK-----
```

Figure 2: ASCII output sample

```
+--first octet--+-second octet--+--third octet--+
|7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0|
+-----------+---+-------+-------+---+-----------+
|5 4 3 2 1 0|5 4 3 2 1 0|5 4 3 2 1 0|5 4 3 2 1 0|
+--1.index--+--2.index--+--3.index--+--4.index--+
```

Figure 3: Correspondence of bits

od OUTPUT.gpg -tx1 > united-OUTPUT.txt.

Then we decode the radix-64 encoding. Radix-64 encoding is easy to decode because it is an extended encoding of BASE 64 that is defined `RFC 2045 - MIME`. The only difference is that CRC octets are appended to the tail in Radix-64 encoding. We then wrote a program called Radix-64.c to decode the ASCII output. This program returns the same output as obtained by the od call described above.

Omitting details, now we have a file looking as shown in Figure 4 which can be used as input to MPI.

The following code transforms this input to the large integer type *mpz_t x* of GnuMP:

```
i = (bitlength + 7)/8;
while(i>= 0&&get(input)){
   x+=input*256^i;
   i--;
}
```

After these steps, we execute the attack to compute the private key x from the public key (p, g, y) and the signature (a, b) of a signed message m.

04 00	f1 9a 6d ee b5 ...

Figure 4: Input to MPI

4.2 Executing the Attack

In the signature scheme in GnuPG v1.2.3, the signature (a, b) satisfies $ak + bx = m \bmod p - 1$. Here m is actually the value $f(message)$, where f is a hash function. The default hash function is SHA-1, but one can also choose MD5, SHA-256, SHA-384, and so on. The hash function is used to ensure that the length of *message* is undecidable. But m may be to short to prevent a simple attack. Therefore, the signature scheme adds the prefix octets that contains default prefix octets as defined by the Abstract Syntax Notation One(ASN.1) and many zeros like 10....0 to make m long enough (see Figure 5) (where SHA-1 is used as hush function, and the constant is "30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14"). By doing this, the programmers of GnuPG expected that the signature is safe. But this trick does not help, since k and x are unusually small. The attack proceeds a follows. Search (k', x') satisfying $ak' + bx' = m \bmod p - 1$. This is done by using the extended Euclidean algorithm. Next, we compute two lattice vectors l_1 and l_2, where u has to satisfy $au + b\left(\frac{gcd(a,p)}{gcd(a,b,p)}\right) = 0 \bmod p - 1$,

10 00 ... 00	Constant	f(message)

Figure 5: Octet stream of m

$$l_1 = \left(\frac{p}{gcd(a,p)}, 0\right), l_2 = \left(u, \frac{gcd(a,p)}{gcd(a,b,p)}\right).$$

Then l_1, l_2 forms a two-dimensional lattice L.

4.3 The Lattice Attack

The correspondence between the size of p and the threshold is given by the so-called Wiener table. Note that $4q_{bit}$ is always less than the bit-length of p.

Bit-length of p	512	1024	1536	2048
q_{bit}	119	165	198	225
Bit-length of p	2560	3072	3584	4096
q_{bit}	249	269	288	305

Figure 6: The Wiener table

Now, we have the lattice L and the target vector $t = (k' - 2^{3_{q_{bit}}/2-1}, x' - 2^{3_{q_{bit}}/2-1})$. We compute the lattice vector l that is closest to t in the lattice L. This is done by using Shoup's [5] NTL library. Finally, let $l = (l_x, l_y)$, then the private key x is $x' - l_y$.

5. Result

Our results are displayed in Figure 7. Each experiment has been run 100 times. As we see, all keys have been broken quickly.

Key length	768	1024	1280	1536	1792
Time(s)	0.030	0.069	0.132	0.224	0.359

Key length	2048	2304	2560	2816	3072
Time(s)	0.545	0.803	0.938	1.204	1.516

Key length	3328	3584	3840	4096
Time(s)	1.977	2.359	2.731	3.244

Figure 7: Results of our experiments

In GnuPG version 1.2.3, the maximum key length is 4096 bits. However, due to the weak implementation even a longer key length would be of no help.

6. Conclusions

As we have seen, using some standard software packages and implementing the rest ourselves allowed for breaking the ElGamal signature scheme as implemented in GnuPG version 1.2.3 using the attack discovered by Nguyen [4]. This clearly shows that one can never be too careful when using cryptographic software. In particular, the source must be available, since otherwise no verification is possible.

References

[1] D. Bleichenbacher. Generating ElGamal signatures without knowing the secret key. In *Advances in Cryptology—EUROCRYPT 96*, volume 1070 of *Lecture Notes in Computer Science*, pages 10–18. Springer-Verlag, 1996.

[2] D. Bleichenbacher. Breaking a cryptographic protocol with pseudoprimes. In *Public Key Cryptography - PKC 2005, 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005, Proceedings*, volume 3386 of *Lecture Notes in Computer Science*, pages 9–15. Springer, 2005.

[3] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inform. Theory*, IT-31(4):469–472, 1985.

[4] P. Q. Nguyen. Can we trust cryptographic software? Cryptographic flaws in gnu privacy guard v1.2.3. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, volume 3027 of *Lecture Notes in Computer Science*, pages 555–570. Springer, 2004.

[5] V. Shoup. Number theory C++ Library (NTL) version 5.3.1. Available at `http://www.shoup.net/ntl/`.