Chapter 10: An Elliptic Curve Asymmetric Backdoor in OpenSSL RSA Key Generation*

Adam L. Young and Moti M. Yung

Abstract

In this chapter we present an experimental implementation of an asymmetric backdoor in RSA key generation. The implementation is written in ANSI C. We codified what it means for an asymmetric backdoor to be secure (for the designer) in our definition of a secretly embedded trapdoor with universal protection (SETUP). The main properties of a SETUP are: (1) the complete code for the backdoor does not enable anyone except the designer to use the backdoor, and (2) the key pairs that are output by the backdoor RSA key generator appear to all probabilistic polynomial time algorithms like normal (no backdoor) RSA key pairs. We introduced the notion of a SETUP at Crypto '96 [15] and there has been significant advances in the area since then. This chapter and the corresponding appendix constitutes *Fundamental Research* in cryptovirology and expands on our elliptic curve backdoor in RSA key generation that we presented at the *Selected Areas in Cryptography* conference in 2005. In particular, the design employs several algorithmic improvements that enable the key generator to run faster. This chapter provides a walk-through of the experimental implementation. The backdoor is based on OpenSSL and the code for it appears in the appendix that is associated with this chapter. For over 10 years we have advocated that the industry change the way RSA keys are generated. We devised and presented heuristic methods that completely foil this entire class of backdoors in RSA key generation [15, 12]. The approach in [12] is reminiscent of the NIST FIPS 186-2 DSA parameter generation method.

*If this file was obtained from a publicly accessible website other than the website www.cryptovirology.com then (1) the entity or entities that made it available are in violation of our copyright and (2) the contents of this file should therefore not be trusted. Please obtain the latest version directly from the official Cryptovirology Labs website at: http://www.cryptovirology.com.

1 Introduction

For years there have been efforts by cryptologists to design undetectable backdoors in the RSA key pair generation algorithm. Research papers can be traced back to 1993 that propose, cryptanalyze, revise, and critique backdoors in RSA key generation. Based on our knowledge of the subject, the following appears to hold: no experimental implementation of an asymmetric backdoor in RSA key generation has been made publicly available and none have been scrutinized by the research community.

It is not clear why this is the case. Perhaps researchers have not made implementations available because, of the few academics actively working on this problem and doing experiments, none of them has been content enough with the design to make the source code available.

Kleptographers suffer from a design restriction that cryptographers take for granted: A kleptographer cannot change the I/O specifications of the target cryptographic algorithm. The kleptographer must design his or her backdoor algorithm *within* the cryptographic algorithm, maintain the same I/O specifications of the targeted algorithm, and preserve the security guarantees insofar as possible. To make matters worse, timing analysis and power analysis have the potential to distinguish backdoor key generators from "honest" key generators that have no backdoor.

One may therefore draw the conclusion that designing a secure asymmetric backdoor is a difficult endeavor. We believe that there is still much research to be done on this subject and we also believe that a freely-accessible experimental implementation may help drive progress on this academic problem.

This chapter presents an experimental asymmetric backdoor in RSA key generation that we designed. Our initial benchmarks indicate that the running time is favorable enough to not arouse the suspicion of a casual user of the backdoor RSA key generator.

There have been efforts to mitigate the threat posed by backdoors. In response to allegations that primes in the Digital Signature Algorithm (DSA) could exhibit a backdoor, NIST published a DSA parameter generation algorithm that in theory makes it difficult to plant a backdoor in DSA (see Appendix 2 of [10]). Also, we briefly outlined a heuristic algorithm for foiling asymmetric backdoors in RSA key generation in the same paper that we introduced the notion of an asymmetric backdoor [15]. This was over 10 years ago. To make it even easier for standardization bodies to adopt our heuristic, we presented a detailed method for mitigating this threat in *CryptoBytes*, the technical newsletter of RSA Labs [12]. This latter method is specifically tailored after the aforementioned NIST DSA parameter generation method. We believe that there has been ample warning, technical guidance, and time to permit standardization bodies to reduce the threat posed by backdoors in RSA key generation.

As part of our research on asymmetric backdoors, we obtained a copy of the OpenSSL source code and used it as a test bed for our backdoor designs. We found and reported two weaknesses and an inefficiency while we worked with the code (a process that spanned many years). These were subsequently fixed by the OpenSSL development team, thereby making OpenSSL an even better cryptographic library. So, in retrospect it is ironic that while researching a robust backdoor in OpenSSL we ended up strengthening OpenSSL.

The vulnerability was as follows. We discovered that OpenSSL chose witnesses for the Miller-Rabin probabilistic primality test using a biased distribution. We therefore found a bug in OpenSSL RSA key generation since OpenSSL RSA key generation relies on Miller-Rabin. We suggested that an iterative acceptance/rejection algorithm be used so that witnesses would be selected using a uniform distribution. In response, the OpenSSL development team implemented our proposed fix by adding the functions `BN_rand_range` and `BN_pseudo_rand_range` to OpenSSL. These were in turn used to repair the bug we found in OpenSSL RSA key generation.[1]

The inefficiency was as follows. We discovered that the predicate for testing whether or not a point is on an elliptic curve used more multiplications than necessary. We suggested that Horner's Algorithm be used to evaluate the cubic polynomial in the Weierstrass equation.[2]

The most recent weakness was found with the help of an anonymous referee during a paper submission of ours. Given that our backdoor is built into OpenSSL, part of the OpenSSL algorithm was in the submission and this resulted in the discovery of yet another bug in OpenSSL RSA key generation.

The weakness was a 32-bit unsigned overflow in the trial-division function `probable_prime`() in OpenSSL. The bug is no longer there since Bodo Möller made our bug fix to OpenSSL.[3] Given the `goto loop` statement it was possible for `mods[i]` + `delta` to overflow before `delta` overflows.

In particular, the constant `NUMPRIMES` is 2048 and the array `primes`

[1]Specifically, the OpenSSL distributions 0.9.6a/0.9.6b/0.9.6c and 0.9.7 (and later) have been fixed.

[2]The OpenSSL development team responded by fixing this inefficiency in OpenSSL distribution 0.9.8. See `http : //cvs.openssl.org/chngview?cn = 12445`.

[3]Our fix was made by Bodo, check-in number 15563 to Branch: OpenSSL_0_9_7. See http://cvs.openssl.org/chngview?cn=15563.

stores the 2048 primes starting from the value `primes`$[0] = 2$ up to the value `primes`[`NUMPRIMES` − `1`] = `17863`, inclusive. The array `mods` has the same number of elements as the array `primes`. Define `MAXDELTA` to be $2^{32} - 1 -$ `primes`[`NUMPRIMES` − `1`]. Our OpenSSL fix uses this as an upper limit on `delta` to ensure that the 32-bit word (`mods`[`i`] + `delta`) does not overflow.

We reported these findings immediately upon finding each of them. In all cases the fixes were made to OpenSSL. So, from this perspective our research has had a very direct and positive impact on OpenSSL.

It behooves us to remind software developers that it is *imperative* to verify the integrity and authenticity of a cryptographic library prior to using it. The OpenSSL development team provides MD5 digests and ASC signatures on each tarball that they make available. Developers that use OpenSSL should always check the authenticity of the OpenSSL source code using `pgp` or `gpg` prior to using it. The public key of the appropriate OpenSSL team member is needed to verify the signature on OpenSSL. The OpenSSL FAQ explains how to verify the authenticity of OpenSSL distributions.

We encourage the international cryptologic research community to study our design, to break it, and to fix it. The story is far from over and it would be silly for us to think that this design is the be all and end all of asymmetric backdoors in RSA key generation. We encourage developers and researchers to send comments, suggestions, and constructive criticisms to: `feedback@cryptovirology.com`.

2 Background on RSA Key Generation Backdoors

Ross Anderson presented a backdoor in RSA key generation in 1993 [1]. This design would later fall into the category of backdoors known as *symmetric backdoors*. A symmetric backdoor is a backdoor that has the following property: if the code that contains the backdoor is published then the backdoor can be used by anyone. This contrasts with an *asymmetric backdoor* that even if published can still only be used by the person that plants it. Burt Kaliski cryptanalyzed Anderson's backdoor construction [4].

We introduced the notion of an asymmetric backdoor in our Crypto '96 paper [15]. Our goal was to devise a Trojan horse for the RSA key generation algorithm in such a way that the Trojan is highly robust against reverse-engineering.

The paper shows how to use public key cryptography to undermine public key cryptography itself by having the designer plant his or her public key within an RSA key generator and use it to securely compromise the coin flips that are used to generate RSA primes. The cryptotrojan encodes the asymmetric encryption of a randomly generated seed in the upper order bits of the RSA modulus that is being generated and uses the seed to generate one of the RSA primes (the seed is passed through a cryptographic hash function [16]). So, from the designer's perspective, an RSA modulus is an RSA public modulus *and* an asymmetric ciphertext that permits said modulus to be factored. Only the designer can decipher the encoding since only the designer knows the needed private decryption key.

The Crypto '96 paper does not utilize the terminology "asymmetric backdoor". Rather, it refers to the designer's public key as a secretly embedded trapdoor. This trapdoor provides universal protection since if a reverse-engineer obtains the code for the backdoor he will not be able to use the backdoor since the designer's private decryption key is needed. In an attempt to formalize what it means for an asymmetric backdoor to be secure for the designer, we introduced the definition of a SETUP (secretly embedded trapdoor with universal protection).

Over the years we have come to the conclusion that the term asymmetric backdoor is more intuitive than the previous terminology that we've used (SETUP attack, kleptography, cryptotrojan). So, at this time we still employ the definition of a SETUP attack, but for simplicity we refer to this class of backdoors as asymmetric backdoors. In retrospect, our Crypto '96 paper partitioned backdoor attacks into those that are symmetric (i.e., the previous backdoor designs) and those that are asymmetric.

The Crypto '96 result was later strengthened and the notion of a weak, regular and strong SETUP was introduced [16]. A regular SETUP ensures that dishonest public keys (i.e, public keys that exhibit a backdoor) are computationally indistinguishable from honest public keys (i.e., public keys with no backdoor). A strong SETUP goes beyond this by ensuring that backdoor key *pairs* are computationally indistinguishable from honest key pairs. This is an important requirement since a suspicious user (distinguisher) may have explicit access to both the public key and corresponding private key that are produced by a black-box key pair generator. This contrasts with the case that a key pair is generated in a smartcard and the private key is non-exportable.

The results [18, 3] revealed a weakness in the previous SETUP attacks on RSA key generation. More specifically, they showed a weakness that adversely affects the indistinguishability argument regarding honest vs. dishonest public keys. These and subsequent works patched the issue. It is necessary to very concretely define the honest RSA key generation algorithm. For instance, it must be specified whether each RSA prime is guaranteed to be a k-bit binary integer, whether the two uppermost bits in the binary representation are always unity, and so on.

A paper by Slakmon and Crépeau [3] introduced a useful new primitive for designing asymmetric backdoor attacks. This paper presents a symmetric backdoor in RSA key generation in which the designer utilizes Coppersmith's factoring algorithm [2] to recover the RSA primes. This significantly lowers the required bandwidth of the subliminal channel in RSA composites.

The backdoors in [18, 17, 13] utilize the random oracle model and substantiate the properties of a SETUP using reduction arguments. The paper [13] builds a backdoor in RSA using the Rabin cryptosystem [11] whereas [14] relies on Elliptic Curve Diffie-Hellman.

A challenge that remained after these works was to construct an asymmetric backdoor in 1024-bit RSA moduli. If the asymmetric backdoor is a Rabin or RSA public key then the backdoor has a security parameter that is about half that of the RSA key pairs being generated. In 1996 it was not unreasonable to use a 512-bit key as an asymmetric backdoor (but it was cutting it close even then). These days a security parameter of 512 for factoring-based cryptosystems is entirely unacceptable.

The backdoor that we presented at the *Selected Areas in Cryptography* '05 conference solved this problem [14]. The challenge was to embed a compact asymmetric ciphertext[4] in the upper order bits of the modulus being generated while maintaining the indistinguishability property of a SETUP. A straightforward application of ECDH would not suffice. To see this, suppose that a binary curve over GF(2^m) is used and that points are represented in compressed form. Only about half of the strings in $\{0, 1\}^{m+1}$ are compressed points on the curve. This permits a trivial poly-time distinguisher.

We solved this problem by utilizing a twisted pair of elliptic curves over GF(2^m). Kaliski utilized twists to construct a provably secure pseudorandom bit generator [5, 6]. Möller later used twisted binary curves to construct a public key stegosystem [9].

The asymmetric backdoor that is the subject of this chapter is based on [14] and it utilizes twisted binary curves. It also employs Möller's fast EC scalar multiplication algorithm [8]. We utilize this method, called the wNAF-splitting method, because it is ideal for our needs and because it is already built into OpenSSL.

This chapter is far from a self-contained scientific explanation of the backdoor we discuss. To obtain an in-depth understanding of our design it is necessary to consult the cited scientific literature. Our primary goal is to

[4]Or Diffie-Hellman key exchange value.

provide a walk-through of our scientific experiment.

We remark that it is likely that *something* is wrong with our implementation. In the grand scheme of things, there has been very little theoretical work on asymmetric backdoors in RSA key generation, as compared to, say, developing secure public key cryptosystems. This implies the possibility of a theoretical oversight. Also, our experimental code is the only one that we are aware of that is currently in the public domain. At this time it has undergone no scrutiny whatsoever by the software development community. This implies the possibility of an implementation error. So, our readers have been warned.

In regards to the aforementioned theoretical work, we give a listing of those research scientists that we are aware of that have published academic papers on backdoors in RSA key generation. From the United States there is ourselves and Burt Kaliski from RSA Labs. From the University of Cambridge in England there is Ross Anderson. From McGill University in Canada there is Claude Crépeau and Alain Slakmon. In Poland there is Daniel Kucner from Wrocław University and Mirosław Kutyłowski from the Wrocław University of Technology [7]. This list does not include researchers that have investigated other types of backdoors. We apologize if we missed any research scientists in this list.

3 Creating the Program

The program `gf2mklepto` consists of the ANSI C source files `gf2mklepto.c` and `symklepto.c`. It also consists of the header files `gf2mklepto.h` and `ecpubkeys.h`. The latter header file is created by our program and it contains the asymmetric backdoor. Due to the low-level routines that are used, the following native OpenSSL header files are also needed for compilation: `ec_lcl.h`, `bn_lcl.h`, and `bn_prime.h`.

We constructed a simple makefile to create `gf2mklepto`. The program was compiled using `gcc`. We used the Minimalist GNU for Windows development suite (MinGW) and employed the OpenSSL crypto library for its underlying crypto routines. In particular we used OpenSSL version 0.9.8d. Our program also utilizes the Microsoft Cryptographic API (MS CAPI) to seed the OpenSSL random number generator. Compiling the program results in the MS DOS command-line program `gf2mklepto.exe`.

Acknowledgments: We thank the OpenSSL development team for producing and maintaining their superb cryptographic library. Our code base relies on the OpenSSL DLL `libeay32.dll`. Also, our code base uses func-

tions that are internal to OpenSSL and that have no public interface. For example, we use the OpenSSL function `compute_wNAF` that is defined as a `static` function in the OpenSSL source code. We claim no copyright over these functions and acknowledge OpenSSL for their creation.

4 Running the Program

This walk-through will not focus on the technical details of the underlying asymmetric backdoor. The reader is referred to the recent literature for this [14]. Instead, we will concentrate on the input/output aspects of using the asymmetric backdoor.

The program starts by printing out copyright information and some bibliographic references. It seeds the OpenSSL pseudorandom number generator as follows. It obtains 512 bytes from the Microsoft Cryptographic API call `CryptGenRandom` and then passes them to the OpenSSL function `RAND_seed`. The program then calls `RAND_status`. This OpenSSL function returns 1 if OpenSSL believes that it has been seeded with a sufficient number of bytes and it returns 0 otherwise.

```
"gf2mklepto" Copyright (c) 2005-2006 by
Moti Yung and Adam L. Young. All rights reserved.
This program is based on the following:
(1) A. Young, M. Yung, "The Dark Side of Black-Box
  Cryptography, or: Should we trust Capstone?" Advances in
  Cryptology---Crypto '96, N. Koblitz (Ed.), LNCS 1109,
  pp. 89-103, 1996.
(2) A. Young, M. Yung, "Malicious Cryptography: Exposing
  Cryptovirology", John Wiley & Sons, Feb., 2004.
(3) A. Young, M. Yung, "A Space Efficient Backdoor in RSA and
  its Applications," Selected Areas in Cryptography---SAC '05,
  B. Preneel, S. Tavares (Eds.), LNCS 3897, pp. 128-143, 2005.

RAND_status() returned 1.
SETUP functions:
Type (a) to generate new EC key pairs for the SETUP attack.
Type (b) to generate and store an RSA key pair with a SETUP.
Type (c) to test RSA encryption/decryption with the key pair.
Type (d) to read in the RSA public modulus and factor it.
Type (e) to execute a support function.

Enter command (a-e) :
```

This chapter will only cover commands (a) through (d). Command (e) is used for software development and testing purposes. Some of the sub-functions in (e) are deprecated.

4.1 Generate New Backdoor EC Key Pairs

The backdoor uses a predefined pair of twisted binary curves. Command (a) generates a maximal order base point for each of these two curves. It then generates EC key pairs using these base points. The base points and public keys are stored in a format that enables the *wNAF splitting* method to be used [8]. This is a method for performing fast scalar multiplication.

```
Enter command (a-e) : a
WARNING: Overwriting all EC private keys in "privkeys.txt".
Generating new EC key pairs for the Young-Yung SETUP attack.

Group parameters for binary curve E_{0,b} in twist is:
Weierstrass coefficient a = 0
Weierstrass coefficient b = 197D4C3C909B4C8EAC18BB296C11BFB18C80
B37C0C62AFD8E5F00104C46EEAF0B
irreducible trinomial p = 20000000000000000000000000000000000000
000000000000000000000001001
prime r_0 = 800000000000000000000000000000005EB3E3179500E2B5D2F8
EA6DCC363C1F
cofactor = 4
maximum order = cofactor * r_0 = 2000000000000000000000000000000
017ACF8C5E54038AD74BE3A9B730D8F07C
Commencing EC key generation for curve E_{0,b}...
Generating random base point G_0 with maximum order...
Computing wNAF splitting values for G_0.
Generating EC private key x_0 < r_0 randomly...
EC private key x_0 is 35902D6105480EF41E4E88B7B07ED596A5741DC74E
CF3C4244FFCD3F8D49C380
Computing public key Y_0 = x_0 * cofactor * G_0.
Computing wNAF splitting values for Y_0...

Group parameters for binary curve E_{1,b} in twist is:
Weierstrass coefficient a = 1
Weierstrass coefficient b = 197D4C3C909B4C8EAC18BB296C11BFB18C80
B37C0C62AFD8E5F00104C46EEAF0B
irreducible trinomial p = 20000000000000000000000000000000000000
000000000000000000000001001
prime r_1 = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF429839D0D5FE3A945A0E
```

```
2B24679387C3
cofactor = 2
maximum order = cofactor * r_1 = 1FFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FE853073A1ABFC7528B41C5648CF270F86
Commencing EC key generation for curve E_{1,b}...
Generating random base point G_1 with maximum order...
Computing wNAF splitting values for G_1.
Generating EC private key x_1 < r_1 randomly...
EC private key x_1 is C2DEFD956EF2DB97BA3C7C240048B2A6CCBF68CC09
ACA824952EE6E9F3FE37FA
Computing public key Y_1 = x_1 * cofactor * G_1.
Computing wNAF splitting values for Y_1...

EC private keys successfully written to "privkeys.txt".
EC public  keys successfully written to "pubkeys.txt".
Wrote EC public keys (wNAF splitting values) to the
file named "ecpubkeys.h".

To use the new EC key pairs RECOMPILATION is needed.
Number of open files = 0.
-----Memory Leaks displayed below--(shouldn't be any)---
-----Memory Leaks displayed above-----------------------
```

The two EC private keys, one for each curve in the twist, are saved in the file `privkeys.txt`. The corresponding public keys are saved in `pubkeys.txt`. Also, the wNAF splitting values are stored in `ecpubkeys.h` as ANSI C data structures. To use the backdoor, the program must be recompiled using this new ANSI C header file.

4.2 Generate an RSA Key Pair with a SETUP

After recompilation, the asymmetric backdoor is ready for use. Command (b) generates a new RSA key pair. However, the resulting RSA public key contains a SETUP.

```
Enter command (a-e) : b
The default RSA public exponent is 0x10001
Enter e in hex or type "z" for default: z
Using default e.
The default modulus size is 1024 bits.
Enter size or type "z" for default: z
Using default modulus size.
```

```
NOTE: The program is currently configured in such a way that it
does NOT attempt to mimic the distribution of OpenSSL RSA
primes. There are functions in the code base that attempt
to do this.
Randomly choosing which curve to do rogue key exchange on...
Rogue ECDH exchange will use curve E_{1,b}.
Loading wNAF splitting values for curve E_{1,b}.
Choosing K < 2*r_1 randomly.
Computing k = K mod r_1.
Computing P = KG_1.
Compressed P to get s_pub = 22288BA92B806552E4635C2132BACE4E35C4
51FCA8538B7AB4A054D5BA2D52392
s_pub is 258 bits in length.
Now setting P = kY_1.
Compressed P to get s_priv = 374B7EC194C4D7A56D60E51FB684207A1CC
BFB0B64781F1173C9A960AAF290E5A
s_priv is 258 bits in length.
s_pub is the ECDH key exchange value generated by this
RSA key generation algorithm. s_priv is the corresponding
ECDH shared secret that is shared between this key generation
algorithm and the malicious designer.
s_pub is encoded into the upper order bits of the candidate
RSA modulus n. s_priv is used to derive the RSA prime p1.
e  = 10001
n  = D688A22EA4AE01954B918D7084CAEB3938D71147F2A14E2DEAD2815356E
8B548E4AA6F1DD26A96FDDABF28432316553DAEC82BAD80C802BBDFDA5C9C3E1
311F1C5EB14447F2C5AB3FA48EBB46979CD49356D5DC3139EA92290828AD7217
ECF30F710ED2003E10784C3884D32FDCB0F4821301958469C14F0D33308A4F69
FF771
p1 = FFBF56B35395C8403A1D0ADDD1B86B19E37E27290D22E2069CE00D9B6A1
FE4C84341B08356EC875E6641CBA512EDADB133B6DCC0B2A3965BAD94363F282
F6A9B
q1 = D6BEDFEAF2888FD3DAC5C93B5A35D3A5F8C7DE77BFFC41A84172A6C39F2
D644F2AF7BCB1588648E8EEB590A293569C6ADB9C880108299F3AA3C2DACBCC0
C50E3
Wrote RSA public key to "rsapubkey.txt".
Wrote RSA private key to "rsaprivkey.txt".
Number of open files = 0.
-----Memory Leaks displayed below--(shouldn't be any)---
-----Memory Leaks displayed above-----------------------
```

Command (b) writes the new RSA public key to the file `rsapubkey.txt`.

It writes the corresponding RSA private key to file `rsaprivkey.txt`.

4.3 Test RSA Encryption and Decryption

Command (c) is used to perform a simple verification of the RSA key pair. It loads the RSA public key from the file `rsapubkey.txt` and the RSA private key from the file `rsaprivkey.txt`.

```
Enter command (a-e) : c
Loading RSA public key from file "rsapubkey.txt".
n = D688A22EA4AE01954B918D7084CAEB3938D71147F2A14E2DEAD2815356E8
B548E4AA6F1DD26A96FDDABF28432316553DAEC82BAD80C802BBDFDA5C9C3E13
11F1C5EB14447F2C5AB3FA48EBB46979CD49356D5DC3139EA92290828AD7217E
CF30F710ED2003E10784C3884D32FDCB0F4821301958469C14F0D33308A4F69F
F771
e = 10001
Loading RSA private key from file "rsaprivkey.txt".
p1  = FFBF56B35395C8403A1D0ADDD1B86B19E37E27290D22E2069CE00D9B6A
1FE4C84341B08356EC875E6641CBA512EDADB133B6DCC0B2A3965BAD94363F28
2F6A9B
q1  = D6BEDFEAF2888FD3DAC5C93B5A35D3A5F8C7DE77BFFC41A84172A6C39F
2D644F2AF7BCB1588648E8EEB590A293569C6ADB9C880108299F3AA3C2DACBCC
0C50E3
d = 7489241052447377B5E50AFFE42296442F2C24A70095BEF2126CE6F36E72
5A878E2F46CCDC502A551B4E5B809CBEB4EF1CD27F67705D359EF8AA95440A34
31BA3444DEE23C7BEFD10BBBB9770AA79475E78807C816A38ECA9A3082401A64
4C83F2619D8B2801F6DFD7FFC3DF135C921EAE554583C1CE4E65CBC4C16B5781
3B5
Choosing m_1 randomly from Z_n^*...
plaintext m_1 = B18CB6D0A7C5C717423D1A8C312FAF21AF2BD757751560FD
4A8E92BF4B4BA03AC8505C8399B67DAAC0DDEB7CC5E87CDAF008E48C29442C99
6A07F4E82977A1297BB89BD9A9B17BC40A1BDFE2F440A72F86F69509D6B927B8
5FA1F7945B0950FDE89EF9F49814A0427C48512DE8A16463B4D805864825FA9A
2A49E015708C19C8
ciphertext c = m_1^e mod n = 68619D243521DDE287FADA398A0B0DE576F
1E9940D303FFA7BABA264AB9AE92869DADEA8E639C95D7993EDA2F5047CAB411
C5ADBD270563C6E13293C2DA0E95E4F61D6289FCB608D9E294894ED06FE2D2B6
E9020E3BCA323390F8D55785B79C04108ADAE96B8E8CC16194D4B3912380C4DE
C794963D9F5D35C62E3BE6969653D
Now computing m_2 = c^d mod n...
plaintext m_2 = B18CB6D0A7C5C717423D1A8C312FAF21AF2BD757751560FD
4A8E92BF4B4BA03AC8505C8399B67DAAC0DDEB7CC5E87CDAF008E48C29442C99
6A07F4E82977A1297BB89BD9A9B17BC40A1BDFE2F440A72F86F69509D6B927B8
```

```
5FA1F7945B0950FDE89EF9F49814A0427C48512DE8A16463B4D805864825FA9A
2A49E015708C19C8
RSA decryption succeeded since m_2 = m_1.
Number of open files = 0.
-----Memory Leaks displayed below--(shouldn't be any)---
-----Memory Leaks displayed above-----------------------
```

Command (c) generates a random plaintext m_1 and then RSA encrypts it to get the ciphertext c. The value c is decrypted using the RSA private key. The command indicates whether or not RSA decryption succeeds.

4.4 Factor the RSA Modulus Using the EC Private Keys

Command (d) is used to factor an RSA modulus that was created using this program. It loads the modulus by reading in the RSA public key from the file `rsapubkey.txt`. The two EC private keys are loaded in from the file `privkeys.txt`.

If the EC public keys that are embedded within `gf2mklepto.exe` do not correspond to the private keys in `privkeys.txt` then the factoring attempt will fail. This type of failure can happen during experimentation when one forgets to recompile the program after command (a), for example. It is very easy to make this mistake while experimenting. It happened to us many times.

```
Enter command (a-e) : d
Loading RSA public key from file "rsapubkey.txt".
n = D688A22EA4AE01954B918D7084CAEB3938D71147F2A14E2DEAD2815356E8
B548E4AA6F1DD26A96FDDABF28432316553DAEC82BAD80C802BBDFDA5C9C3E13
11F1C5EB14447F2C5AB3FA48EBB46979CD49356D5DC3139EA92290828AD7217E
CF30F710ED2003E10784C3884D32FDCB0F4821301958469C14F0D33308A4F69F
F771
e = 10001
Loading EC private keys from file "privkeys.txt".
x0 = 35902D6105480EF41E4E88B7B07ED596A5741DC74ECF3C4244FFCD3F8D4
9C380
x1 = C2DEFD956EF2DB97BA3C7C240048B2A6CCBF68CC09ACA824952EE6E9F3F
E37FA
Attempting decompression of s_pub as if it is a
compressed point on curve E_{0,b}...
Decompression failed. Attempting decompression of s_pub
as if it is a compressed point on curve E_{1,b}...
```

```
Determined that s_pub is on curve E_{1,b}.
s_pub successfully decompressed.
Recovered shared ECDH secret s_priv.
Now using s_priv to try to factor n...
succeed = 1.
p1 = FFBF56B35395C8403A1D0ADDD1B86B19E37E27290D22E2069CE00D9B6A1
FE4C84341B08356EC875E6641CBA512EDADB133B6DCC0B2A3965BAD94363F282
F6A9B
Number of open files = 0.
-----Memory Leaks displayed below--(shouldn't be any)---
-----Memory Leaks displayed above-----------------------
```

The program uses trial-and-error to determine which of the two curves were used to conduct the rogue elliptic curve Diffie-Hellman key exchange. This trial-and-error method is implemented by using point decompression. Recall that the decision as to which curve is used is made randomly during RSA key generation.

Once the correct curve in the twist is determined, the corresponding EC private key is used to recover the shared elliptic curve Diffie-Hellman secret. This shared secret is then used to recover the RSA prime `p1`.

5 Conclusion

We reviewed the academic literature on backdoors in RSA and in particular we covered the notion of a SETUP attack. In addition we noted some of the research scientists from around the world that are actively working on the problem of designing backdoors in RSA key generation. We emphasize yet again the importance of having the industry move away from the way RSA key pairs are currently generated. By simply choosing two RSA primes randomly, the way is paved for this class of clandestine asymmetric backdoors. To guide developers and standardization bodies, we provided pointers to research papers that detail countermeasures against this class of backdoors.

We presented an experimental implementation of our asymmetric backdoor in RSA key generation that employs a twisted pair of binary curves, a design that is based on [14]. A step-by-step demonstration of the experimental program was given and the resulting output was provided.

References

[1] Ross Anderson. A practical RSA trapdoor. *Electronics Letters*, 29(11):995, 27 May 1993.

[2] Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In Ueli Maurer, editor, *Advances in Cryptology—Eurocrypt '96*, pages 178–189. Springer, 1996. Lecture Notes in Computer Science No. 1233.

[3] Claude Crépeau and Alain Slakmon. Simple backdoors for RSA key generation. In Marc Joye, editor, *Topics in Cryptology CT-RSA, The Cryptographers' Track at the RSA Conference*, pages 403–416. Springer, 2003. Lecture Notes in Computer Science No. 2612.

[4] Burton S. Kaliski. Anderson's RSA trapdoor can be broken. *Electronics Letters*, 29(15):1387–1388, 1993.

[5] Burton S. Kaliski, Jr. A pseudo-random bit generator based on elliptic logarithms. In A. M. Odlyzko, editor, *Advances in Cryptology—Crypto '86*, pages 84–103. Springer-Verlag, 1987. Lecture Notes in Computer Science No. 263.

[6] Burton S. Kaliski, Jr. One-way permutations on elliptic curves. *Journal of Cryptology*, 3:187–199, 1991.

[7] D. Kucner and M. Kutylowski. Stochastic kleptography detection. In Public-Key Cryptography and Computational Number Theory, pages 137–149. Walter de Gruyter, 2001.

[8] Bodo Möller. Improved techniques for fast exponentiation. In P. J. Lee and C. H. Lim, editors, *Information Security and Cryptology—ICISC '02*, pages 298–312. Springer, 2002. Lecture Notes in Computer Science No. 2587.

[9] Bodo Möller. A public-key encryption scheme with pseudo-random ciphertexts. In *Proceedings of the 9th European Symposium on Research in Computer Security*, pages 335–351. Springer-Verlag, 2004.

[10] National Institute of Standards and Technology (NIST). FIPS Publication 186-2: Digital Signature Standard. *Federal Register*, January 27, 2000.

[11] Michael Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1979.

[12] Adam Young. Mitigating insider threats to RSA key generation. *CryptoBytes*, 7(1):1–15, 2004.

[13] Adam Young and Moti Yung. Malicious cryptography: Kleptographic aspects. In Alfred Menezes, editor, *Topics in Cryptology CT-RSA, The Cryptographers' Track at the RSA Conference*, pages 7–18. Springer, 2005. Lecture Notes in Computer Science No. 3376.

[14] Adam Young and Moti Yung. A space efficient backdoor in RSA and its applications. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography—SAC '05*, pages 128–143. Springer, 2005. Lecture Notes in Computer Science No. 3897.

[15] Adam L. Young and Moti M. Yung. The dark side of black-box cryptography, or: Should we trust capstone? In Neal Koblitz, editor, *Advances in Cryptology—Crypto '96*, pages 89–103. Springer-Verlag, 1996. Lecture Notes in Computer Science No. 1109.

[16] Adam L. Young and Moti M. Yung. Kleptography: Using cryptography against cryptography. In Walter Fumy, editor, *Advances in Cryptology—Eurocrypt '97*, pages 62–74. Springer-Verlag, 1997. Lecture Notes in Computer Science No. 1233.

[17] Adam L. Young and Moti M. Yung. *Malicious Cryptography: Exposing Cryptovirology*. Wiley, February 2004.

[18] Adam Lucas Young. *Kleptography: Using Cryptography Against Cryptography*. PhD thesis, Columbia University Graduate School of Arts & Sciences, 2002. Thesis Advisor: Zvi Galil (and Moti M. Yung).