Incorrectly Generated RSA Keys

A study of a potentially catastrophic cryptographic software bug

Daniel Shumow

Microsoft Research, Redmond WA 98052, USA

Abstract. When generating primes p and q for an RSA key, the algorithm specifies that they should be checked to see that $p-1$ and $q-1$ are relatively prime to the public exponent e, and regenerated if this is not the case. If this is not done, then the calculation of the decrypt exponent will fail. However, what if a software bug allows the generation of public parameters N and e of an RSA key with this property and then it is subsequently used for encryption? Though this may seem like a purely academic question, a software bug in the RSA key generation implementation in the CNG API of a preview release of the Windows 10 operating system makes this question of more than purely hypothetical value. Without a well defined decrypt exponent, plaintexts encrypted to such keys will be undecryptable thus potentially losing user data, a serious software defect. Though the decrypt exponent is no longer well defined, it is in fact possible to recover the plaintext, or a small number of potential plaintexts if the prime factors p and q of the public modulus N are known. This paper presents an analysis of what steps fail in the RSA algorithm and use this to give a plaintext recovery algorithm. The runtime of the algorithm scales linearly in the magnitude of the public exponent, in practice this is manageable as there are only a few small public exponents that are used. This algorithm has been implemented in a publicly available python script. We further discuss the software bug that lead to this and derive lessons that can be used while testing randomized functions in cryptographic software. Specifically, we derive an explicit formula that describes the trade off between number of iterations of tests of a randomized cryptographic functions and the potential number of users affected by a bug dependent on the random values.

Keywords: public key cryptography · cryptographic software · software bugs.

1 Introduction

The RSA asymmetric cryptosystem [10] is the first and most widely used asymmetric encryption algorithm. The algorithm is still widely used today despite the rising popularity of elliptic curve cryptography (ECC). Even though ECC is gaining in popularity, ECC provides only a key establishment algorithm Elliptic

Curve Diffie-Hellman (ECDH) and a digital signatures Elliptic Curve Digital Signature Algorithm (ECDSA) but no asymmetric encryption algorithm. As such, RSA remains the gold standard for asymmetric encryption and is ubiquitously used as such.

RSA keys consist of a public key and private key. The private key is a pair of primes p and q, to be kept secret. The public key is the product of the primes $N = pq$ and a public exponent e such that e shares no factors in common with $p-1$ or $q-1$. In the language of elementary number theory, e is relatively prime or coprime to $p-1$ and $q-1$. Encryption takes a a positive integer x such that $1 < x < N$ and computes a cipher text $c = x^e \bmod N$. Decryption is computed as follows, the decrypt exponent d is defined such that $ed = 1 \bmod (p-1)(q-1)$. That is that d is the *multiplicative inverse* of e modulo $(p-1)(q-1)$ and is defined provided that e is coprime to $(p-1)(q-1)$. Decryption continues by computing $c^d \bmod N$ which returns x. The proof of this is straight forward, simple and elegant relying only on basic results from the field of elementary number theory. This proof is omitted here for space but there is no shortage of literature explaining this, such as the original paper [10].

The correctness of the RSA algorithm, as stated, relies on the fact that the public exponent e is relatively prime to $p-1$ and $q-1$. As such, when an RSA key is generated the public exponent e is selected first and as the primes p and q are generated they are checked, by the extended euclidean division algorithm, that $p-1$ and $q-1$ are coprime to e and regenerated until they are not. This leads to the question, what will happen if this regeneration step is not performed? As noted previously, the decrypt exponent will not be defined and decryption can not be performed. However, note that the decrypt exponent is not needed for encryption so the public parameters can be generated and used without it. What happens if such an RSA key is generated and encrypts important data? This may seem like purely academic exercise, however this situation actually occurred in a prerelease build of the Windows 10 operating system.

This paper presents a solution to the problem of recovering data that has been encrypted by an RSA key where the public exponent is not coprime to $p-1$ or $q-1$. By a careful analysis of what exactly happens when encrypting with such a key, the underlying mathematical structure of the problem can be used to yield a solution. This solution runs in time $O(e)$, which is exponential in the length of e, but in actuality e is chosen to be relatively small for performance purposes. Most often e is only 17bits long, which provides a search space that is easily handled with modern computers. Mathematically speaking, the number of potential plaintexts is exactly e, however in practice valid plaintexts will have some structure, and this can be used to aid in the plaintext search. Specifically, the common RSA encryption padding schemes, used to map messages to a positive integer modulo N can be used to greatly narrow down the number of potential plaintexts to just a few, and in some cases uniquely to the correct

solution.

The discussion of this problem begins by analyzing why exactly RSA asymmetric encryption fails when e is not coprime to $(p-1)(q-1)$, beyond that the decrypt exponent is not defined. Furthermore key generation is analyzed so that the probability that a random key has e not coprime to $p-1$ or $q-1$ in certain instances. This mathematical analysis is used to derive a plaintext search, which runs in time $O(e)$, and by utilizing properties of padding algorithms greatly narrows down the number of potential plaintexts. This algorithm has been implemented in Python and is publicly available. There is a discussion of the bug that caused this actual problem in Win10 which is provided along with references to public forum posts identifying users who have actually hit it. Finally, this issue is used to derive lessons for testing cryptographic functions.

2 Incorrectly Generated RSA Keys

In this paper *incorrectly generated RSA keys* refers to RSA keys where the multiplicative group order is not relatively prime to the public exponent and hence the usual decrypt exponent is not well defined. More concretely, keys with modulus $N = p \cdot q$, with p and q prime, and public exponent e is *incorrectly generated* if e is not relatively prime to $\varphi(N) = (p-1)(q-1)$, where φ denotes Euler's totient function which is defined as the count of positive integers less than N that are coprime to N. In this discussion, e is assumed to be prime. This is a weak and uncontroversial assumption, as for practical cryptographic purposes one desires a prime exponent. In practice the vast majority of RSA keys use the Fermat prime $F_4 = 2^{2^4} + 1 = 65,537$, and most of the remaining public exponents are 3. Any other public exponents are relatively uncommon. Furthermore, CNG, where the incorrect RSA key generation bug occurs, uses F_4 as the public exponent in almost every case. For the sake of straight forward presentation, the analysis will proceed with the further assumption that $(p-1)(q-1)$ is not divisible by any higher powers of e. This assumption will be justified by a subsequent analysis of the likelihood of either of these respective cases. After addressing the simpler case of prime e, we provide a short discussion of how to extend to cases of composite e or when higher powers of e divide $\varphi(N)$.

To see what goes wrong when using such incorrectly generated RSA keys, consider the group of units of the integers modulo N, $(\mathbb{Z}/N\mathbb{Z})^\times$. This is the group in which the arithmetic of the RSA algorithm is implicitly performed. If $e|\varphi(N)$ let E be the subgroup of e order elements of $(\mathbb{Z}/N\mathbb{Z})^\times$, then there is an isomorphism

$$(\mathbb{Z}/N\mathbb{Z})^\times \cong G \times E \tag{1}$$

with $|E| = e$ and $|G| = (p-1)(q-1)/e$, by the assumption that e is prime. This isomorphism is given by decomposing any integer $x \in (\mathbb{Z}/N\mathbb{Z})^\times$ as $x = g \cdot \ell$.

where $g \in G$ and $\ell \in E$. So the exponentiation step of public key encryption with N and e has the following effect:

$$y = x^e \equiv (g \cdot \ell)^e \equiv g^e \cdot \ell^e \equiv g^e \bmod N. \tag{2}$$

So, public key encryption is not one-to-one, as the factor ℓ is lost, in an information theoretic sense. Thus the set of possible preimages under public exponentiation of y is

$$P = \{g \cdot \ell | \ell \in E\}. \tag{3}$$

So the set of possible plaintexts P has order e.

If the public exponent were drawn from the full set of possible exponents it would be infeasible to enumerate this entire set. Luckily, as noted above the most common $e = 65537$, which is only 17 bits long which is a trivial search space for modern computational devices. This gives hope that the plaintext that has been lost to this exponentiation can be recovered. Then the problem reduces down to determining which is the correct plaintext from the set of possibilities P, this is discussed in the following section.

One may speculate that the most likely way that an RSA key is incorrectly generated is if the check that to ensure that $\varphi(N)$ is coprime to e is simply not performed during key generation. As such, it is worthwhile evaluating how often this condition will occur for randomly generated RSA moduli by using the common but heuristic notion of probability of divisibility. As RSA moduli are the product of two primes p and q, the probability that $\varphi(N) = (p-1)(q-1)$ is divisible by e is the probability that $p-1$ or $q-1$ but not both is divisible by e. The probability that a random integer is divisible by e is $1/e$, and that is not is $(1-1/e)$. So, the probability that one prime satisfies this congruence, but not the other is $(1/e)(1-1/e)$, and there are two primes so the probability p or q satisfy this congruence is $(2/e)(1-1/e)$ or a little less than one in $32,000$. Similarly, the probability that $\varphi(N)$ is divisible by e^2 is the probability that both $p-1$ and $q-1$ are divisible by e and both are not divisible by e^2 or that only one of the values is divisible by e^2 and the other is not divisible by e. This leads to the more involved probability.

$$\frac{1}{e^2}\left(1-\frac{1}{e}\right)^2 + \frac{2}{e^2}\left(1-\frac{1}{e}\right).$$

For $e = 65537$, this evaluates out to less than one in 1.43 billion. These relative probabilities justify the assumption that no higher powers of e divide $\varphi(N)$. RSA Key generation is relatively expensive and RSA keys are usually persisted once they are generated, so there are relatively few RSA keys generated per user. The following thought experiment is illustrative of these relative probabilities. The population of the earth is over 7 billion people, if each person were to generate only one 20 keys with this flawed generation method the expected number of keys with $\varphi(N)$ divisible by e^2 would be about 100, but there would be over four million keys with $\varphi(N)$ divisible by e.

3 Recovering Plaintexts Encrypted with Incorrectly Generated RSA Keys

This section presents an algorithm for recovering potential plaintexts encrypted to an incorrectly generated RSA public key. The previous section shows that there are e potential plaintexts that can encrypt to a ciphertext c with such keys. If plaintexts had no additional structure, then it would be up to users to sift through these potential plaintexts to find potentially valuable lost data. This is not an insurmountable task, however in practice plaintexts will be padded before encryption and this can be leveraged to greatly reduce the number of potentially valid plaintexts. The two most popular asymmetric padding schemes for RSA are PKCS1v1.5 and OAEP. The more modern OAEP has provable properties and uses a hash function to generate the padding and this virtually ensures that only the valid plaintext will be recovered. The older scheme, PKCS1v1.5 has enough structure that it will greatly limit the number of potential plaintexts, though there will often be more than one potentially valid plaintext that users will have to evaluate to determine which one correct plaintext. First the overall algorithms for solving for potential plaintexts are presented and briefly analyzed. Then there is an evaluation of both padding schemes to determine the precise probabilities that they will be satisfied by a random string, and this yields an estimate of the number of candidate plaintexts. This algorithm has been implemented in a python script and recovers potential plaintexts in seconds for 4096-bit keys. This section also includes an interesting relationship between this problem and zero-knowledge proofs.

3.1 Plaintext Candidate Recovery Algorithm

The analysis in the previous section shows that by enumerating over the subgroup E of e-order elements in $(\mathbb{Z}/N\mathbb{Z})^\times$ will give all the potentially lost factors of the plaintext. By the assumption that e is prime and that e^2 does not divide $\varphi(N)$, the subgroup E will have exactly e elements. As a prime order group it is cyclic, and to generate all elements of E it is necessary to find a generator for this group. A consequence of elementary number theory is that for any prime p the integers mod p have a primitive element, or multiplicative generator (see chapter 8 in [3].) By our assumptions, e must necessarily divide only one of $p-1$ or $q-1$. By the Chinese Remainder Theorem, we can consider $(\mathbb{Z}/N\mathbb{Z})^\times$ as a product of $(\mathbb{Z}/p\mathbb{Z})^\times$ and $(\mathbb{Z}/q\mathbb{Z})^\times$. And by the aforementioned divisibility criteria, E reduces to a trivial subgroup modulo one of the primes and is isomorphic to a subgroup modulo the other prime. Without loss of generality, say that $e|p-1$, then the problem of finding a generator of the e-order subgroup of $(\mathbb{Z}/p\mathbb{Z})^\times$, reduces to finding an element g mod p such that $g^{(p-1)/e} \not\equiv 1 \bmod p$. Such a g will necessarily be a primitive root of p and $\tilde{g} = g^{(p-1)/e} \bmod p$ will be a generator of the e-order subgroup of $(\mathbb{Z}/p\mathbb{Z})^\times$. Then this element $\tilde{g}$ can be mapped via the Chinese Remainder Theorem into $(\mathbb{Z}/N\mathbb{Z})^\times$ and will be a multiplicative generator g_E of E. This algorithm is well known and used extensively

in modern cryptography, and in practice runs very quickly. This is a very technical discussion of the approach to finding a generator of E, but serves to show the correctness of the algorithm. In fact it is not necessary to determine exactly which prime has an e-order multiplicative subgroup and map the generator into the integers modulo N. Rather, all the work can simply be done modulo N. Specifically, selecting candidate generators g modulo N and testing that

$$g_E = g^{(p-1)(q-1)/e} \not\equiv 1 \bmod N$$

reduces to working modulo p and q and directly yields the generator g_E of E. In practice this algorithm will have to check only a handful of values g and can just begin sequentially searching at $g = 2$. This is algorithm is explicitly specified in Algorithm 1. For a proof of correctness and precise analysis of runtime see Theorem 2.2.7 in [4].

Algorithm 1 Find multiplicative generator g_E of the subgroup of e order elements $E < (\mathbb{Z}/N\mathbb{Z})^\times$.

```
φ̃ ← (p − 1)(q − 1)/e
g ← 1
repeat
   g ← g + 1
   g_E ← g^φ̃ mod N
until g_E ≠ 1
return g_E
```

With a generator g_E of E found by Algorithm 1, it is straight forward to enumerate over all elements $\ell \in E$. Algorithm 2 implements this plaintext search algorithm, with an abstracted padding check, which will be discussed in more depth in section 3.2.

The correctness of algorithm 2 is straightforward to see. By the results of section 5 if x is the plaintext, then $x = a \cdot \ell$ for $a \in G$ and $\ell \in E$. The ciphertext is of the form

$$c = x^e \equiv (a \cdot \ell)^e \equiv a^e \cdot \ell^e \equiv a^e \bmod N.$$

As $ed \equiv 1 \bmod \frac{(p-1)(q-1)}{e}$ it follows that

$$c^d \equiv (a^e)^d \equiv a^{ed} \equiv a \bmod N,$$

because $|G| = \tilde{\varphi} = (p-1)(q-1)/e$. So the algorithm correctly recovers the non lost factor a of x and it is required to iterate over the lost factor ℓ. At the i^{th} iteration of the loop, the value $\ell = g_E^i$, and as g_E is a multiplicative generator of E this will iterate over all elements $\ell \in E$. Thus the value of the encrypted plaintext x will be enumerated by this loop. Though the padding check is abstracted away in this description of the algorithm, this check ensures that only x with valid padding are added to the set of possible plaintexts P. Therefore, at

Algorithm 2 Find set of potential plaintexts that encrypt to ciphertext c with incorrectly generated RSA key $N = p \cdot q$ and public exponent e, given prime factor p and q.

$\tilde{\varphi} \leftarrow (p-1)(q-1)/e$
$d \leftarrow e^{-1} \bmod \tilde{\varphi}$
$a \leftarrow c^d \bmod N$
Use Algorithm 1 to find multiplicative generator g_E of e-order elements of $(\mathbb{Z}/N\mathbb{Z})^\times$.

$P \leftarrow \{\}$
$ell \leftarrow 1 \bmod N$
for all $i = 0 \cdots e-1$ **do**
  $x \leftarrow a \cdot \ell \bmod N$
  if x is a correctly padded plaintext. **then**
    $P \leftarrow P \cup \{x\}$
  end if
  $\ell \leftarrow \ell \cdot g_E \bmod N$
end for
return Set P of potential plaintexts.

the end of this algorithm the set P contains only potential plaintexts with valid padding, including the original plaintext x. The details of evaluating padding checks are investigated in section 3.2.

The complexity of Algorithm 2 is dominated by the loop enumerating the elements of E. It is straightforward to see that the algorithm performs the equivalent of two private key operations and exactly e padding checks and the algebraic complexity is $O(e)$ $(\mathbb{Z}/N\mathbb{Z})$ operations. Thus algorithm 2 has complexity of $O(e)$, ignoring the private key operations and scaling of underlying arithmetic. Ignoring the operations that scale with key size serves to describe the complexity of the search for lost plaintext itself.

A Brief Discussion of General Exponents It is also possible to relax the assumptions and show that even if $e^2|\varphi(N)$ or e is composite this approach can be used to solve for lost plaintexts. First, note that if any higher powers of e divide $\varphi(N)$ then it is possible that both $p-1$ and $q-1$ are divisible by e, in which case the subgroup E will not by cyclic, but will be generated by at most two elements (corresponding to the subgroup modulo each prime.) This is similar to how $(\mathbb{Z}/N\mathbb{Z})^\times$ will have 4 square roots of unity. In this case, iterating to find all e order elements will require $O(e^2)$ steps and the enumeration process is slightly more complicated though not much more. In the case that e is composite, then it may not be the case that $e|\varphi(N)$ but instead that they are not relatively prime. In this case, then the issue comes down to enumerating the $e' = \gcd(\varphi(N), e)$ order elements. And the group E' of e' order elements may be more complicated than just cyclic or the product of two cyclic groups of the same order. Enumerating the element of this group is not insurmountable and is not significantly more complicated than the algorithm here, but beyond the scope of this paper.

3.2 Padding Schemes and Recovering Plaintexts

To actually solve for a lost plaintext in practice, it is necessary to look at the padding schemes that are used to padded out plaintexts before they are encrypted with RSA. There are two commonly used standardized RSA plaintext padding algorithms, PKCS1 v1.5 [7] and OAEP [8]. In fact, these are the only two padding algorithms supported by the CNG API, where the software bug occurred. Both of these padding schemes are evaluated here.

RSA OAEP The RSA-OAEP scheme is the most modern RSA padding scheme and it is built on a theoretical foundation[1] and is provably **Ind-CCA2** in the random oracle model [9]. This property limits the malleability of decrypted plaintexts and as a direct consequence when RSA-OAEP is used will uniquely identify the correct plaintext with very high probability algorithm 2. The following description of RSA-OAEP shows how the construction limits the probability of false positives in the plaintext search.

RSA-OAEP padding takes a message M of length m, an optional label L which may be the empty string, and a hash function H. Let n be the length of the modulus N in bytes, and h be the digest length of H. The function also defines a *mask generation function* $MGF(s, k)$ that takes a variable length seed s and utilizes H to generate a k byte string, to be XORed as a mask similar to a stream cipher. The MGF which will not be described in detail here, works by iterating H applied to the seed and a counter, and concatenating digests to generate a sufficient number of bytes. The length of the message must satisfy $m \leq n - 2h - 2$. First, the message is padded out to an $n - h - 1$ byte string by concatenating values:

$$D = H(L) \| PS \| \text{0x01} \| M \tag{4}$$

where PS is a string of $n - m - 2 * h - 2$ zero bytes. Then a random h byte seed S is generated and input to MGF and XORed with D to produce:

$$D' = MGF(S, n - h - 1) \oplus D. \tag{5}$$

Now D' is passed as a seed to the MGF to mask out the seed S as

$$S' = MGF(D', h) \oplus S \tag{6}$$

and the plaintext is the concatenation:

$$x = \text{0x00} \| S' \| D'. \tag{7}$$

This is interpreted as a big-endian integer and encrypted in the usual way as $c = x^e \bmod N$.

The probability that a random n byte candidate x passes the OAEP padding validation is determined by evaluating the the probability that each padding step is satisfied in reverse order. The first thing checked is that as in equation 7 the

most significant byte of x is 0x00, which will occur with probability 256^{-1}. As mentioned before, the details of the construction of MGF are omitted, and the output of MGF is assumed to be indistinguishable from a random string. The candidate value D is recovered by parsing out S' and D' values and combing equations 6 and 5 at once, by calculating:

$$D = MGF(MGF(D', h) \oplus S', n - h - 1) \oplus D'.$$

The probability that D is accepted as a properly padded string is the probability that the first h bytes of D equal$H(L)$, which is simply 256^{-h}. The rest of the string D is properly formatted if it is all 0x00 bytes at the beginning followed by a single 0x01 byte. There are $n - 2h - 1$ bytes in the rest of D and there are $256^{n-2h-1-k}$ byte strings with a k byte prefix of the form 0x00 $\cdots$ 0x00 $\|$ 0x01 and summing this from $1 \leq k < n - 2h - 1$ gives $(256^{n-2h-1} - 1)/(256 - 1)$ total valid strings out of 256^{n-2h-1} total possible strings. Combining this with the other probabilities shows that the probability of any random string satisfying this is

$$\frac{1}{256} \cdot \frac{1}{256^h} \cdot \frac{256^{n-2h-1} - 1}{255 \cdot 256^{n-2h-1}} = \frac{256^{n-2h-1} - 1}{256^{h+1} \cdot 255 \cdot 256^{n-2h-1}} \approx \frac{1}{256^{h+1} \cdot 255}.$$

So the expected number of false positives in a set of size e, such as iterating over E is $e/256^{h+2}$, and for $e = F_4$ this is approximately 256^{-h} which is essentially none. Thus when using OAEP Algorithm 2 is expected to find a unique solution.

PKCS1 v1.5 The RSA PCKS1 v1.5 scheme predates OAEP and its design was more adhoc and does not have the same desirable provable properties as OAEP. In particular the plaintext that is output from decryption is more malleable than OAEP. Indeed, this malleability historically made this scheme susceptible to timing attacks [2] and programmers implementing PKCS1 v1.5 need to be careful to avoid serious security bugs related to this. In the case of trying to recover plaintexts lost to incorrectly generated RSA keys this malleability makes it more likely that there are false positives. In fact, it is so much more likely that we expect that we will frequently see at least one false positive. However, the expected number of plaintexts is small and easily reviewed by a human.

Compared to the scheme for OAEP, the PCKS1 v1.5 padding scheme is quite simple. To pad a PKCS1 v1.5 for an n byte public key, the padding algorithm takes a message M of m bytes where $m \leq n - 11$. First generate $n - m - 3$ padding string PS of random nonzero bytes, note that the length requirement on M ensures that PS is at least 8 bytes. Then the padded bytes to encrypt are

$$x = 0\text{x}00 \| 0\text{x}02 \| PS \| 0x00 \| M \tag{8}$$

interpreted as a big-endian integer and encrypted with RSA as usual.

Due to the simplicity of the PKCS1 v1.5 scheme, there are far fewer opportunities to detect invalid plaintexts. The first and most likely way to detect invalid

plaintexts is if the most significant two bytes are not 0x00‖0x02, the probability that a random plaintext candidate satisfies this is 256^{-2}. The only other way to catch an invalid PKCS1 v1.5 padded plaintext is if one of the first 8 padding bytes is zero. First observer that are 256^7 strings with a zero at each first 8 bytes, so there are $8 \cdot 256^7 = 2^{59}$ possible invalid values for the first 8 bytes. Thus there are $2^{64} - 2^{59} = 2^{59}(2^5 - 1)$ possible valid values for the first 8 bytes. Therefore the probability that a given random string has a valid value for the first 8 bytes of the padding string is

$$\frac{2^{59}(2^5 - 1)}{2^{64}} = \frac{2^5 - 1}{2^5} = \frac{31}{32}.$$

Combing the two probabilities gives

$$\frac{1}{2^{16}} \cdot \frac{31}{32} = \frac{31}{2^{21}}.$$

The expected number of plaintexts that satisfy this is $31e/2^{21}$ and when $e = F_4 = 2^16+1$, the expected number of plaintexts is $31(2^16+1)/2^{21} = 31/32+31/2^21 \approx 0.968$. Thus most plaintext searches will yield at least one false positive.

3.3 Software Implementation of Plaintext Recovery Algorithm

A python script implementing this algorithm has been made available under the MIT license on GitHub at `https://github.com/danshumow-msft/FixBadRsaEncryption`. This python script is fully self contained and has no external dependencies other than a Python interpreter version 3.4 or greater. For an incorrectly generated RSA key with a 4096-bit modulus and $e = 65537$, this script runs in under 30 seconds on a Laptop with a Intel Core i7 2.6Ghz processor and 16GB of RAM.

3.4 Connection to Zero Knowledge Proofs.

The algorithm for recovering plaintexts has an interesting connection to a zero knowledge proof of quadratic reciprocity. Specifically, this approach to recovering plaintexts is similar to, and in some ways a generalization of, a zero knowledge protocol for proving the ability to compute quadratic residues from [5]. To explain this connection the protocol is presented briefly here.

Suppose that N is an RSA modulus, and that Alice knows the factors p and q. Then Alice can compute square roots modulo N and may prove this to Bob as follows:

1. Bob picks $x \bmod N$ and sends $r = x^2 \bmod N$ to Alice.
2. Alice picks a random $y \bmod N$ and sends $a = y^2 \bmod N$ to Bob.
3. Bob picks a random $b \in \{0, 1\}$ and sends b to Alice.
4. Alice sets $c = y$ if $b = 0$ or computes x from r and sets $c = x \cdot r \bmod N$ if $b = 1$, and sends c to Bob.

5. Bob checks that $a \equiv c^2 \bmod N$ if $b = 0$ or that $c^2 \equiv a \cdot r \bmod N$.

If Alice cannot compute quadratic residues, then Bob will catch her with at least one-half probability. Implicit in this protocol is the fact that Alice can compute square roots modulo N. The e root finding algorithm performed in the plaintext search is a generalization of square root finding to any prime $e|\varphi(N)$. Furthermore, in an informal sense if Alice can recover a plaintext that has been encrypted to an incorrectly generated RSA key similarly can be used to create a zero knowledge proof that she can compute e-roots.

4 A Software Bug That Incorrectly Generates RSA Keys

The bug that CNG was incorrectly generating RSA keys was publicly reported in a prerelease version of Windows 10 on April 24, 2019 [6]. The bug reporter found that with Windows 10 version 1803 - OS Build 17134.706 that RSA keys were being incorrectly generated. By generating up to one hundred thousand 2048bit RSA keys a "bad" (incorrectly generated) RSA key was found. Furthermore, the discloser debugged down and determined that the the public exponent was not invertible modulo $\varphi(N)$ and determined that for this key the public exponent divided $q - 1$, correctly identifying that this is an invalid RSA key.

This public disclosure of the bug shows that this potentially serious issue was found by and was affecting Windows 10 users. Also, the occurrence prevalence of the bug shows that this is likely caused by some manner of incorrect or skipped check that the public exponent e is invertible modulo $\varphi(N)$. It is worth noting that this occurred after SymCrypt was used as the implementation of asymmetric cryptography, including RSA Key Generation, in CNG [13]. Though SymCrypt was not released on GitHub until after the Windows 10 1803 release with the reported issue. Reviewing the relevant code in the SymCrypt library, which occurs in rsakey.c for RSA key generation and primes.c for prime generation, shows that this code does not contain any obvious defects that would cause this. The conclusion would be that this bug was introduced in SymCrypt when CNG switched to using it for RSA Key Generation, but fixed before it was published on GitHub.

There have been other high profile issues with SymCrypt. Such as such as a bug in modular reduction, which would allow a carefully crafted DSA modulus to create a infinite loop [11]. This bug was caused because error returns were not properly checked while performing Montgomery multiplication. It is worth noting that a similar issue could cause this issue with RSA Key generation, such as if error returns were ignored while computing the decrypt exponent.

The original bug disclosure thread ends with a post confirming that the issue is no longer present in Windows 10 1903. However, Windows 10 1803 will be supported until 2021, due to the coronavirus pandemic. So any users running

into this issue with lost data may fix the issue with the scripts referenced in section 3.3.

5 A Lesson For Testing Randomized Cryptographic Functions

This bug that causes incorrectly generated RSA keys underscores the importance of detecting and fixing even seemly rare bugs in randomized cryptographic functions. This section presents another example of a similar bug in a randomized cryptographic function, specifically RSA PKCS1 v1.5 encryption padding. These two bugs are infrequent enough that they will not be found by typical developer unit tests which may exercise functions a few times to assert basic correctness However, they are frequent enough that they would affect a large number of users. This gap between what a developer may see and what users will see when software is released motivates an analysis of how many test iterations are required to detect bugs and limit the expected number of users affected by bugs.

The incorrect RSA key generation bug is reminiscent of a bug that the author once saw while developing the first version of the CNG cryptographic API in Windows Vista. This bug, which occurred in the RSA PKCS1 v1.5 encryption padding, had a similar failure rate occurring with a probability at least 2^{-13}. As described in section 3.2, during public key encryption with PKCS1 padding, there are at least 8 bytes of nonzero padding. The bug was that this padding was first filled with a Cryptographic PRNG, and then a loop went through each byte of padding and if there was a zero byte a new byte was generated and used instead of the zero byte. The bug was that this zero check should be performed in a loop until a nonzero byte was generated, not if a zero byte was generated. If two zero bytes were generated in a row for that index, which occurs with probability 2^{-16} the padding check would return a plaintext of the wrong size including all padding after the first zero byte along with the plaintext. There are at least 8 padding bytes in PKCS1 v1.5 padding, so this bug will occur in at least one in every 2^{13} encryption operations using this padding scheme.

Considering the incorrect RSA key generation bug and the incorrect padding bug shows that a bug that may seem rare to a developer will be rapidly magnified once software is released and affect a potentially large number of users. Suppose that a cryptographic function is directly or indirectly used by n users, and called on average c times per user. Then if a bug in a randomized cryptographic function has probability ϵ of occurring then the expected of times this bug will occur is $nc\epsilon$. A bug with probability ϵ is expected to occur with high probability after $1/\epsilon$ iterations. In other words, if a randomized function is run through t iterations of tests with fresh randomness then with high probability the test will expose bugs that have probability greater than $1/t$. So, this gives an upper bound on the probability that a bug will occur. Thus the number of

test iterations run bounds the expected number of users affected by a bug to nc/t.

This analysis is easily applied to the bug causing incorrectly generated keys, simply assume that every user generates at least one RSA Key (in actuality this number will be much higher.) Microsoft claims that there are over one billion Windows 10 devices, and as argued in sections and 4 the probability of this bug occurring is for one in every 32K keys. This yields that this bug threatened to affect over at least 32 thousand users.

This shows the importance of investigating thoroughly and fixing even seemly rare cryptographic bugs. Even though RSA key generation is relatively infrequent compared to other cryptographic functions a one in 32 thousand failure effects tens of thousands of users, and likely much more. Using the analysis in this section can give cryptographic developers an idea of how much testing is needed for randomized functions to catch bugs caused by randomness.

6 Conclusion

This paper analyzes the problem of recovering plaintexts encrypted to incorrectly generated RSA keys. This analysis reveals that if there are not proper checks that e is coprime to $\phi(N)$ then the probability of an incorrectly generated RSA key is approximately $2/e$, and for the most popular choice of $e = 65537$, this probability is slightly less than $1/32000$. The problem is analyzed for its mathematical structure and this is used to develop an algorithm to search for potential plaintexts that encrypt to a given ciphertext. This algorithm runs in time $O(e)$ which is tractable on current computers, due to the short public exponent that are typically used. Mathematically speaking, the problem has e solutions, but padding algorithms impose structure that can be used to greatly reduce the potential valid plaintexts. In the case of OAEP with virtual certainty we can expect only the correct plaintext. Though with PKCS1 v1.5 we may expect that there is almost always a false positive along with the correct answer, but the correct answer should be easily identifiable from this. This provides a useful solution to the problem, which actually occurred in a prerelease version of the Win10 operating system, as evidenced by public forum posts. Code implementing this algorithm is publicly available and runs in seconds for even large key sizes. The problem of recovering plaintexts encrypted to incorrectly generated RSA keys was also compared to a similar Zero Knowledge protocol for Quadratic Residues. Finally, this issue is used to derive a lesson for deriving the number of test iterations to run against a randomized cryptographic function relating to the number of users, and potentially affected users.

Acknowledgments The author would like to thank Erlend Graff, Greg Zaverucha and Brian LaMacchia for their help in the investigation of this issue and development of the plaintext recovery algorithm. Erlend Graff for his detailed investigation and bug report of this issue in Microsoft forums. Greg Zaverucha

for his work on the original investigation of the incorrectly generated RSA key and verifying that this approach works. Brian LaMacchia for helping turn this into an award winning CRYPTO '19 rump session talk. The author would also like to thank Dan Boneh for pointing out the connection between the plaintext recovery algorithm and zero knowledge proofs of quadratic residues.

References

1. Bellare M., Rogaway P.: Optimal Asymmetric Encryption – How to encrypt with RSA. In De Santis, A. (ed.) Eurocrypt 1994, pp. 92–111. LNCS, Vol. 950, Springer-Verlag (1995)
2. Bleichenbacher D.:Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Krawczyk H. (ed.) Advances in Cryptology – CRYPTO 1998, pp. 1–12 LNCS, Vol 1462, Springer, Berlin, Heidelberg (1998)
3. Burton, D.M.: Elementary Number Theory. 4th edn. International Series in Pure and Applied Mathematics. McGraw-Hill, USA (1998)
4. Crandall, R., Pomerance, C.: Prime Numbers A Computational Perspective. 2nd Ed. Springer, New York (2000)
5. Goldwasser, S., Micali, S., Rackoff, C.: The Knowledge Complexity of Interactive Proof Systems. SIAM J. Comput., **18**(1), 186–208 (1989)
6. Graff, E.: Bug in CNG RSA key generation? https://social.msdn.microsoft.com/Forums/windowsdesktop/en-US/3d581bdb-ccaa-43c7-bbaa-ae22fce06b32/bug-in-cng-rsa-key-generation Last Accessed 29 Aug 2020
7. Kaliski, B.: PKCS #1: RSA Encryption Version 1.5. RFC 2313. Mar. 1998.
8. Kaliski, B., Staddon, J.:PKCS #1: RSA Cryptography Specifications Version 2.0. RFC 2437. Oct. 1998.
9. Fujisaki, E., Okamoto, T., Pointcheval, D., and Stern, J.: RSA-OAEP is secure under the RSA assumption. In Kilian, (J.) (ed.) Advances in Cryptology–CRYPTO 2001, pp. 260-274. LNCS, vol. 2139, Springer, Berlin, Heidelberg (2001)
10. Rivest, R., Shamir A., Adleman, L.: A method for obtaining digital signatures and public key signatures. Comm. ACM, **2**(2), Feb. 1978.
11. 1804 - cryptoapi: SymCrypt modular inverse algorithm - Project Zero https://bugs.chromium.org/p/project-zero/issues/detail?id=1804 Last Accessed 30 Aug 2020
12. Microsoft by the Numbers, https://news.microsoft.com/bythenumbers/en/windowsdevices Last Accessed 29 Aug 2020
13. microsoft/SymCrypt: Cryptographic Library, https://github.com/Microsoft/SymCrypt Last Accessed 29 Aug 2020