”Politehnica” University of Bucharest

Faculty of Automatic Control and Computer Science,
Computer Science Department

Cryptography Scientific Report

Sanity Check of Public RSA Keys from Internet

Coordinator

Lector Dr. Mat. Emil Simion

Authors:

Adrian Stratulat, SCPD
Mihai Barbulescu AAC
Raluca Oncioiu SCPD
Vlad Traista-Popescu SRIC

Bucharest, 2015

Contents

1 **Project Description** **3**

2 **Theoretical Background** **4**

2.1 RSA Key Generation [1] 4

2.2 RSA encryption and decryption 5

2.3 Extended Euclidean algorithm 5

2.4 Square and multiply algorithm for modular exponentiation . . . 6

2.5 RSA integer factorization 6

2.6 Tests to check if a number is prime [2] 7

2.7 Random Numbers and Pseudorandom Numbers [3] 8

3 **Software application** **10**

3.1 Overall architecture 10

3.2 Big number C library 10

3.3 Factorization procedure 11

3.4 Getting the private key 12

4 **Results** **15**

5 **Conclusions** **16**

A **Python Code for Extended Euclidean Algorithm** **17**

B **C code for finding common RSA factor** **18**

Abstract

In this paper we present an application that makes use of an online public key crawler with the main purpose of making a sanity check session on approximately 42 million public keys discovered when scanning TCP port 443 for raw X.509 certificates, between June 6, 2012 and August 4, 2013.

The purpose of this project is to demonstrate that on the Internet, people are using keys with a common divisor. This is a potential cyber attack, because any man-in-the-middle that listens on the same channel with two keys having some common factors can discover what are the keys and then, continue with various attack mechanisms.

Our application is designed to execute the GCD (Greatest Common Divisor) factorization in parallel. It also has a statistics engine in order to derive results easily.

We show that the percentage of broken keys with 512 bits is 3.7%, while the percentage of broken keys with 1024 bits is 0.05%. The smaller value is due to the fact that factorization of large numbers include new prime numbers, unused in the small keys.

In the end, we derive our conclusions and make some suggestions when generating private/public RSA keys, like using keys of at least 2048 bits length and certificates that expire in more than a year.

1 Project Description

We report the results of a basic sanity check session using a large-scale measurement study of the HTTPS certificate ecosystem[4] performed by a research group from Department of Electrical Engineering and Computer Science, University of Michigan, which peformed by web crawling 110 Internet-wide scans over 14 months, by collecting and validating X.509 certificates, using the following methodology:

- discover hosts with HTTPS (443) port activated. One can easily achieve this by using `nmap` command, similar to the following execution:

```
mihai@blackhole: ~ $ nmap --script=ssl-cert.nse -p 443 www.google.
    com

Starting Nmap 6.40 ( http://nmap.org ) at 2015-06-05 20:43 EEST
Nmap scan report for www.google.com (173.194.112.18)
Host is up (0.039s latency).
Other addresses for www.google.com (not scanned): 173.194.112.19
    173.194.112.20 173.194.112.16 173.194.112.17
rDNS record for 173.194.112.18: fra07s27-in-f18.1e100.net
PORT   STATE SERVICE
443/tcp open https
| ssl-cert: Subject: commonName=www.google.com/organizationName=
    Google Inc/stateOrProvinceName=California/countryName=US
| Issuer: commonName=Google Internet Authority G2/organizationName=
    Google Inc/countryName=US
| Public Key type: rsa
| Public Key bits: 2048
| Not valid before: 2015-05-06T09:29:25+00:00
| Not valid after: 2015-08-03T23:00:00+00:00
| MD5:  3e35 9be7 db85 d15b 9806 b52e e236 0e68
|_SHA-1: 4b9d 33e6 4ef6 104e 2043 bf1e 0928 924f 6d41 337a

Nmap done: 1 IP address (1 host up) scanned in 0.27 seconds
```

- completing a TLS handshake with responsive addresses and collecting the presented certificate chains. This can be achieved in Linux command line by using the `openssl` suite:

```
mihai@blackhole: ~ $ openssl s_client -crlf -connect www.example.
    net:443
```

- parse and validate certificate. A full C example of how this can be done using OpenSSL library is described here [1]

[1] https://zakird.com/2013/10/13/certificate-parsing-with-openssl/

There are similar efforts that dump certificates in databases. One example is the investigation performed by the EFF SSL Observatory project, done by Electric Frontier Foundation (EFF) [2] , supported by NLnet Foundation and SingleHop LLC. The scanning is done with a similar procedure as the one described above, but it focused on Certificate Authorities trusted by Mozilla and Microsoft.

In the next sections we will focus on background theory on RSA, key factorization and number theory required and how we used all this information to achieve the factorization of the keys found.

2 Theoretical Background

RSA is an asymmetric cryptographic scheme developed by Rivest, Shamir and Adleman. It currently is one of the most used cryptographic systems worldwide alongside AES. This cryptosystem is mostly used in applications that involve:

- small data transfer such as key exchanges (e.g. it secures the Diffie Hellman Key Exchange)
- digital signatures and digital certificates

2.1 RSA Key Generation [1]

Since RSA is an asymmetric algorithm it uses two different keys: a public key and a private key. The public key is for general use and can be used for encrypting data whereas the private key is secret, it is not shared and it is generally used for decryption. An exception to this general rule is represented by the authentication process in SSH where the private key is used to encrypt a challenge string and the public key is used to decrypt the cyphertext.

The key generation process for an RSA entity is as follows:

- two large numbers p and q are generated randomly.
- the following value is computed $n = p * q$.
- the value $\phi(n) = (p-1)(q-1)$ is computed where $\phi(n)$ is Euler's totient function. This value is kept secret.
- The public exponent e is selected from the set $\{1, 2, \ldots, \phi(n) - 1\}$ with the condition that $gcd(e, \phi(n)) = 1$.
- The private key d is computed such that $d \cdot e \equiv 1 \bmod \phi(n)$.
- The public key is represented by the pair (n, e) and the private key is represented by d.

[2] https://www.eff.org/observatory

2.2 RSA encryption and decryption

The encryption and decryption computation is done in the integer ring $\mathbb{Z}n$ and it heavily features modular exponentiation.

In order to encrypt a message m the public key (n, e) is used as follows:

$c \equiv m^e \bmod n$

In order to decrypt a cyphertext c the private key is used:

$m \equiv c^d \bmod n$

A very important property that this algorithm must have is that since it uses two different keys it must be computationally hard to deduce the private key d just by knowing the public key (n, e).

Since most of the operations for computing the public/private keypair and for encrypting/decrypting involve computation with very large numbers, certain time optimizations can be done such as the extended Euclidean algorithm for the modular multiplicative inverse and the square and multiply algorithm for modular exponentiation.

2.3 Extended Euclidean algorithm

In the RSA key generation step the private key d was calculated as the inverse of the public exponent e modulus $\phi(n)$. This modular inverse is usually calculated using the extended Euclidean algorithm.

This algorithm computes the greatest common divisor of two numbers and it also computes the coefficients of Bezout's identity. The principle behind this algorithm is that being given two numbers a and b, these numbers are coprime only if two numbers x and y can be found that satisfy the following relationship:

$a \cdot x + b \cdot y = gcd(a, b)$

In the RSA case, the equation can be written as:

$e \cdot x + \phi(n) \cdot y = 1$ (e and $\phi(n)$ are coprime)

Solving this equation modulo $\phi(n)$ will lead us to:

$e \cdot x \equiv 1 \bmod \phi(n)$

In our case, x is represented by d the private key.

The algorithm is an extension of the standard Euclidean algorithm. The standard one was using a series of divisions where only the remainders were used repeatedly. The extended one uses both the quotients and the remainders. A small implementation of the algorithm can be found in appendix

2.4 Square and multiply algorithm for modular exponentiation

RSA has an asymmetric behaviour not only when it comes to the pair of keys it uses, but also because of the duration it takes to encrypt and decrypt a message. The encryption process lasts a shorter amount of time because since e is the public exponent, it can have a small value without compromising its security. On the other hand the decryption process lasts a considerable amount of time because the private key d has a very large value for security reasons.

The decryption process consists in a modular exponentiation operation. In a naive implementation an exponentiation x^n would have a complexity of $O(n)$ since the exponentiation is done by multiplying the same number n times. Using the square and multiply algorithm the complexity of the exponentiation is reduced to $O(log(n))$.

```
def squareAndMultiply(base, power, modulus):
  tmp = 1
  while power:
    if power % 2 == 0:
      power /= 2
      base = (base ** 2) % modulus
    else:
      power -= 1
      tmp = (tmp * base) % modulus
  return tmp
```

2.5 RSA integer factorization

The strong security power of RSA comes from the fact that it is computationally hard to factorize a large number (in our case n). In the early days of RSA (1994) it was estimated that factoring the RSA moduli of a 426 bit key would take trillion of years (it must be taken into account also that the computational power at that time does not match the current one).

However, over the course of time the factoring algorithms have been drastically improved and the computational power has increased which resulted in an increase of the RSA keys length. An evolution of the RSA keys length is show in the table below:

Decimal digits	Bit length	Date
100	330	April 1991
110	364	April 1992
120	397	June 1993
129	426	April 1994
140	463	February 1999
155	512	August 1999
200	664	May 2005

Table 1: RSA keys length evolution

In the present the RSA keys length are usually either 1024 bit or 2048 bit. In 2011, NIST has flagged the RSA 1024 as deprecated with the recommendation to transition to 2048 bit keys.

2.6 Tests to check if a number is prime [2]

Euler Theorem. If $(b, m) = 1$ then $b^{\phi(m)} \equiv 1 \mod m$.

Fermat Theorem. If p is prime number which doesn't divide to b then $b^{p-1} \equiv 1 \mod p$.

Wilson Theorem. The number p is prime if and only it's not divisible to any natural number less than $\left[\sqrt{p}\right]$.

For big values of p, the Wilson test and Erastotene test are not computationally efficient.

Miller-Rabin test. It consists in the following steps:

- We randomly choose different values in base b instead of a single value
- While we compute each modular exponentiation (power of two followed by modulo p operation), we establish if we have found a square root of 1 mod p. If we have found it, the test will say if the number p is composite. The error ratio of Miller-Rabin test for every odd number is $\frac{1}{2^s}$, where s is the total number of random values chosen for base b.

Fermat Test. It uses the reciprocal of Fermat Theorem. We present the algorithm below:

FERMAT(n, t)

Input: an odd number $n \geq 3$ and a security parameter $t \geq 1$.

Output: the decision.

Step 1. For $i = 1, \ldots, t$ do:

- choose a random number a with $2 \leq a \leq n - 2$.
- compute $r = a^{n-1} \mod n$.

- if $r \neq 1$ then we decide that n is composite.

Step 2. We decide: n is prime.

Observation. A composite number n which is considered *prime by Fermat test* is called pseudoprime Fermat number and numbers of type a are called *false Fermat numbers.*

Soloway-Strassen. It uses Euler's criteria, that is: if n is a prime number then:

$a^{\frac{n-1}{2}} \equiv (\frac{a}{n})$ for every integer a that satisfies (a,n) = 1, where $(\frac{a}{n})$ is the symbol of Legendre.

We present the algorithm below:

Solovay-Strassen(n, t)

Input: an odd number $n \geq 3$ and a parameter for security $t \geq 1$.

Output: the decision.

Step 1. For i = 1,...,t do:

- we randomly choose a with $2 \leq a \leq n-2$.
- compute $r = a^{\frac{n-1}{2}} \mod n$.
- $r \neq 1$ and $r \neq n-1$ then we decide n is composite.
- we compute the symbol of Legendre $s = \left(\frac{a}{n}\right)$.
- if $r \neq s$ then we decide n is composite.

Step 2. We decide: n is prime.

2.7 Random Numbers and Pseudorandom Numbers [3]

The correct way of speaking about random numbers is taking a sequence of random numbers. The computers generate random numbers using specific rules for the number of 0 and 1 values. For example, we know that a good random number must have an approximately equal number of 0 values with the number of 1 values. There are also statistical tests that show how much randomness is in a sequence of numbers. A perfect random number is the one that can be guessed only with brute force.

The factorization can be made harder for a cracker depending on the quality of RNG (Random Number Generators) and the system's entropy pool. Because some systems don't implement or use them correctly, sometimes the same p or q values are generated across machines.

Usually, we generate random numbers based on mouse's moves and/or clock ticks in the computer. The property of randomness has been introduced with the help of pseudorandom numbers.

Definition. Let be m, k ($m-1 \geq k > 0$) integer numbers. A (k,m) pseudorandom generator is a recursive function:

$f : \mathbb{Z}_2^k \longrightarrow \mathbb{Z}_2^m$

In applications, we obtain m from k with a polynomial algorithm. A pseudorandom generator must satisfy the following rules:

- It must be simple and fast.
- It must produce sequences of numbers with arbitrary lengths that don't contain repetitions.
- It must produce independent numbers, or with an ambiguous correlation.
- It must generate numbers with an uniform distribution.

Some examples of pseudo-random number generators are:

- Linear congruential generators: $x_{n+1} = ax_n + b(\mod m)$
- Ranrot generators: $TyepA : x_n = ((x_{n-j} + x_{n-k})(\mod 2^b)) >> r$
- Shift registers, such as linear feedback shift register (LFSR), Geffe generator, Stop-and-Go
- Mother-of-all generator:

```
MOTHER OF ALL GENERATORS
$n = 4$
**while** $n \leq MAX$
    $S = 2111111111 \cdot x_{n-4} + 1492 \cdot x_{n-3} + 1776 \cdot x_{n-2} + 5115 \cdot x_{n-1} + c$
    $x_n = S \bmod 2^{32}$
    $c = \frac{S}{2^{32}}$
    $n = n + 1$
```

- Blum - Micali generator: For g prime and p odd prime number, x_0 initial value. We generate $x_{i+1} = g^{x_i} \bmod p$. The output is 1 if $x_i < \frac{p-1}{2}$ and 0 otherwise.
- RSA generator: The RSA cryptosystem can be used to generate random numbers. Having $n = pq$ and public exponent e such that $gcd(e, (p-1)(q-1)) = 1$ and x_0 initial value, $x_0 < n$. We define: $x_{i+1} = x_i^e(\mod n)$. The output is $z_i = x_i \bmod 2$. We need to choose a very high value for n in order to be secure.

3 Software application

3.1 Overall architecture

Figure 1 describes the overall architecture of our checker. In this paper the focus is on the parallelized execution of GCD factorization and statistics engine. The web crawler is, at this moment, just the download of public databases (or data sets) that are discovered by third parties. Currently the statistics engine is focused on performance evaluation of GCD computation and counting broken keys. It can be enhanced with functionalities such as counting the number of broken keys which correspond to valid certificates (that are not expired) for instance. Other functionality that can be added could be a unit testing engine which compares the values obtained by the factorization engine against what `openssl`'s information for a RSA private key.

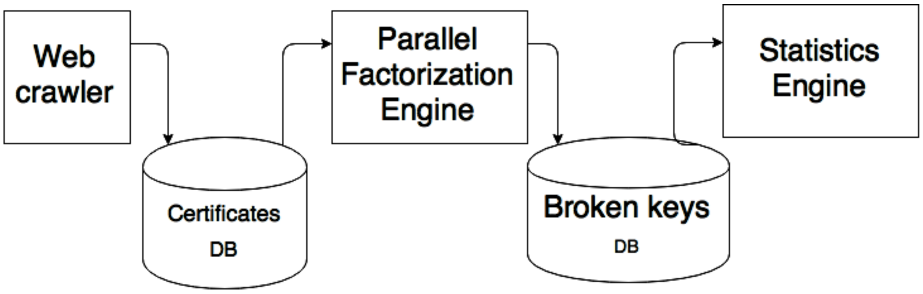

Figure 1: Overall Software Architecture of RSA Public Key Checker

3.2 Big number C library

For running the sanity check session we used the C language with the OpenMP support for easy multithreading enablement in order to use at maximum an AMD multi-core architecture. Because C does not have built-in support for big numbers and for our application using integers, the native CPU register size was not enough and that we also needed high-precision calculations that exceed the precision of the underlying CPU, we used arbitrary precision (bignum) library.

We decided to use GMP (GNU Multiple Precision Arithmetic Library) [5], as it has support for integer and rational numbers, can do computations in finite fields, aiming at speed and supporting numerical algorithms such as greatest common divisor, extented euclidean algorithm for inverse modulo n and other useful cryptographic computations.

3.3 Factorization procedure

The brute-force approach to find the prime factors of a number n is to check against all the prime numbers in the interval $[2, \sqrt{n}]$. Because this is not feasible for big numbers (larger than 2^{100}), another approach has to be chosen.

The approach we used was to compute the GCD[3] using Euclid's algorithm on all the possible pairs in a set of numbers. This way, instead of storing a large database of prime numbers, we only store the set of numbers to be checked.

```
1  for $i = 0$ to $M - 1$
2      for $j = i$ to $M$
3          $t = \text{gcd}(A[i], A[j])$
4          if $t \neq 1 \wedge t \neq A[i]$
5              print $i : A[i] : t$
6              print $j : A[j] : t$
```

Because sometimes we encounter identical keys, their GCD is equal with themselves but we have to discard this result because it does not give information about the private key.

As an optimisation, the outer loop can be run in independently, in parallel but because the print procedures are not atomic, the lines 5 and 6 should be protected by an mutex.

To demonstrate the weakness, we have to analyse the KeyGen procedure.

```
RSA Key Generation
1  Choose randomly two large primes $p$ and $q$
2  Compute $n = p \cdot q$.
3  Compute $\phi(n) = (p - 1)(q - 1)$.
4  Select the public exponent $e \in \{1, 2, \ldots, \phi(n) - 1\}$ such that $gcd(e, \phi(n)) = 1$
5  Compute the private key d such that $d \cdot e \equiv 1 \bmod \phi(n)$
6  return public key: $k_{PUB} = (n, e)$, private key: $k_{PR} = d$
```

The only decision step of the algorithm is the first, which depends on the quality of the RNG [4] and the system's entropy pool. Because some systems don't implement or use them correctly, sometimes the same p or q values are generated across machines.

If we know the public key and the p value, all the other information can be computed easily.

The full C snippet used in our research is described in appendix B.

[3] Greatest Common Denominator

[4] Random Number Generator

3.4 Getting the private key

In the above subsections we described how we manage to find a common factor for a given public key. Once this is achieved, we only have to apply the RSA algorithm so that we are able to obtain the private key from the public key and factor found.

The following output shows an excerpt from what the key-factoring parallel code generates. The line format is `ID:RSAPublicKey:Factor`

```
48:e8d8454ed5b58c09fbfb186880fac61dd6e7e4
79459caa85d6cd9754ce41f1f6c42841151f886277e4b1f8
2ff28cc04e57e9e5e551339fbb62a15a05b65e69f1:f4ee2329d4e7b
93c822be343eb5b29a1756cf39424bce3e443f95b97d7dfafdd
49:c15da3d3511cf0fd0bc2feac741d51d056394209c0b224
0d3ca62ca06d0673c87aa5666b46c3d86d679e7e1e0713
e4893e00159e9833e26684ab4e2fdd182261:c5ea75702937
f53ad151e8c16fc34b8caad05c0073d2cbe87e79d78953d24d77
85:af2b7afa0d08276b394f6f973546e94244e1f715
3dc183f620bb1b7ab90e5a43671fc4bdb841cf0580f56a6
a14f3b232b978d327d34443ca2731037a299c8869:cfc5674
b72e96f856755a70110ce9865003ebc5d413c7e7a305
fa852480e24a9
```

The schema provided by the certificates scaned by University of Michigan is that we have a `public_keys.csv` file with public keys description and unique identifier and a `certificates.csv` file with raw X.509 certificates with unique ID and the corresponding ID of the public key.

Thus, for a given factored public key, we need to find it first in `public_keys.csv`

```
mihai@blackhole: 2993AD0A1C27EE5D $ grep
    e8d8454ed5b58c09fbfb186880fac61dd6
e7e479459caa85d6cd9754ce41f1f6c42841151f886277e4b1f82ff28cc04e57e9e5e
551339fbb62a15a05b65e69f1 public_keys.csv
25509759,rsaEncryption,,65537,\xe8d8454ed5b58c09
fbfb186880fac61dd6e7e479459caa85d6cd9754ce41f1f6c42841151f8
86277e4b1f82ff28cc04e57e9e5e551339fbb62a15a05b65e69f1,,,,,512
```

We observe that the above key has 512 bits length, so obviously it might be vulnerable. For the public key with ID `25509759` we find the following corresponding certificate:

```
35158611,\x80d982577d2f07e6944d1260d084b7f5ecb009f5,0,,1,"C=US, ST=
    California, L=Irvine, O=Cisco-Linksys,
LLC, OU=Division, CN=Linksys/emailAddress=support@linksys.com","C=US, ST=
    California, L=Irvine, O=Cisco-
Linksys, LLC, OU=Division, CN=Linksys/emailAddress=support@linksys.com
    ",3,t,2010-12-08 03:45:55,2020-12-05
03:45:55,t,,f,f,f,f,,,,,,,,,,,,,,,,md5WithRSAEncryption
    ,0,25509759,2013-04-23 01:23:24.614823,
rsaEncryption,,,,,,,
```

Now we know that a key from Linksys has a vulnerability. Moreover, this certificate is still available and last seen in 2013, but expiring in 2020! This is a serious issue.

To determine the other factor from equation $n = pq$, as we already have the public key n and a factor p we used Python, which has built-in support for big numbers and can be easily used as scripting language for automating all these steps and also the PyCrypto library [6]

```
      n = int(hex_pub_key_from_csv, 16)
      q = int(hex_factor_from_dump, 16)
      p = n/q
```

Also, the `public_keys.csv` gave us information about the public exponent used, which is $e = 65537$.

Now we can generate a public-private key, in Python, using PyCrypto, which can be used either as OpenSSH authentication or digital signature:

```
from Crypto.PublicKey import RSA
from Crypto.Util import number
def genRSAkey(p,q):
  n = p*q
  e = 65537L
  phi = (p-1)*(q-1)
  d = number.inverse(e,phi)
  return RSA.construct((n,e,d,p,q))

if __name__ == "__main__":

       a = genRSAkey(11078508734463573664171622129l4
              94697477270872142202066316578327864502007082973L,
  11007864317016659601338120647504991277877248
  34944050409281159866302208851O1349L)

       priv_key = a
       pub_key = a.publickey()

       print priv_key.exportKey()
       print ’\n’
       print pub_key.exportKey(format=’OpenSSH’)
```

The above code has the following output,

```
-----BEGIN RSA PRIVATE KEY-----
MIIBOwIBAAJBAOjYRU7VtYwJ+/sYaID6xh3W5+R5RZyqhdbNl1TOQfH2xChBFR+I
Ynfksfgv8ozATlfp5eVRM5+7YqFaBbZeafECAwEAAQJBAL5KwZnEHjLslfqH4CBI
```

```
OlDgqXOXiMSYt/js3Jh6WHqdWnEygZE0jxbvLvcX8UdzG5q+VSGGe02ZDOl1eNGz
NIECIQD07iMp1Oe5PIIr4OPrWymhdWzzlCS84+RD+VuX19+v3QIhAPNeTdsPotbV
JWRdT476Wkii485WFsxdwVnCZ5JGcYslAiB18tfsJSMSCRmGmQL0LAp4PlwhUI4A
iJ48PbDmBocp4QIgA2L8kl/VVEgDDTJ/OO4i83/Qy75G8vEbD7S03EQ+sEECIQC4
xgdXNsEirFlWFsWA0bNNGU5/igaQpKldgRk4sQwqfA==
-----END RSA PRIVATE KEY-----

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAQQDo2EVO1bWMCfv7GGiA+
    sYd1ufkeUWcqoXWzZdUzkHx9sQoQRUfiGJ35LH4L/KMwE5X6eXlUTOfu2KhWgW2Xmnx
```

which then can be checked against what `openssl` discovers in the private key, after exporting the PEM format of the private key to a file `id_rsa.priv`, as it is described in the listing below:

```
mihai@blackhole: ~ $ openssl rsa -text -in id_rsa.priv
Private-Key: (512 bit)
modulus:
    00:e8:d8:45:4e:d5:b5:8c:09:fb:fb:18:68:80:fa:
    c6:1d:d6:e7:e4:79:45:9c:aa:85:d6:cd:97:54:ce:
    41:f1:f6:c4:28:41:15:1f:88:62:77:e4:b1:f8:2f:
    f2:8c:c0:4e:57:e9:e5:e5:51:33:9f:bb:62:a1:5a:
    05:b6:5e:69:f1
publicExponent: 65537 (0x10001)
privateExponent:
    00:be:4a:c1:99:c4:1e:32:ec:95:fa:87:e0:20:48:
    d2:50:e0:a9:73:97:88:c4:98:b7:f8:ec:dc:98:7a:
    58:7a:9d:5a:71:32:81:91:34:8f:16:ef:2e:f7:17:
    f1:47:73:1b:9a:be:55:21:86:7b:4d:99:0c:e9:75:
    78:d1:b3:34:81
prime1:
    00:f4:ee:23:29:d4:e7:b9:3c:82:2b:e3:43:eb:5b:
    29:a1:75:6c:f3:94:24:bc:e3:e4:43:f9:5b:97:d7:
    df:af:dd
prime2:
    00:f3:5e:4d:db:0f:a2:d6:d5:25:64:5d:4f:8e:fa:
    5a:48:a2:e3:ce:56:16:cc:5d:c1:59:c2:67:92:46:
    71:8b:25
exponent1:
    75:f2:d7:ec:25:23:12:09:19:86:99:02:f4:2c:0a:
    78:3e:5c:21:50:8e:00:88:9e:3c:3d:b0:e6:06:87:
    29:e1
exponent2:
    03:62:fc:92:5f:d5:54:48:03:0d:32:7f:38:ee:22:
    f3:7f:d0:cb:be:46:f2:f1:1b:0f:b4:b4:dc:44:3e:
    b0:41
coefficient:
    00:b8:c6:07:57:36:c1:22:ac:59:56:16:c5:80:d1:
    b3:4d:19:4e:7f:8a:06:90:a4:a9:5d:81:19:38:b1:
    0c:2a:7c
```

As it can be seen from the above output `prime1` is the factor discovered by our factorization engine, so the certificate (which is still available as we saw, not expired!) is broken.

4 Results

The following results should respect the principles that should lead to relevant statistical tests [7]

The total number of public keys provided by the web crawler and potential input for factorization: 44474713

- 512-bit keys, searching the entire database for matches
  - Number of found keys: 323338 (0.7% of the raw dataset)
  - Number of tested pairs: $323337 \cdot 323336/2$
  - Number of common divisors: 4717
  - Number of broken keys: 12209
  - Percent of broken keys: 3.7%
- 1024-bit keys, phase I (exhaustive search for matches on a set of $100k$ keys)
  - Number of keys provided as input: 100000
  - Number of tested pairs: $99999 \cdot 99998/2$
  - Number of common divisors: 2
  - Number of broken keys: 6
  - Percent of broken keys 0.006%
- 1024-bit keys, phase II (trying to match the 2 divisors from the previous set against the full dataset)
  - Number of keys in the database: 26869026 (60% of the raw dataset)
  - Number of tested pairs: $26869024 * 2$
  - Number of broken keys: 13617
  - Percent of broken keys 0.05%

  The search speed obtained (on a computer with AMD quad-core x86_64 CPU, running at 3.9GHz, with 6 GB RAM):

  - 512-bit: $720k$ GCD/s
  - 1024-bit: $355k$ GCD/s

5 Conclusions

In this report we presented the results of a sanity check session of X.509 certificates crawled between 2012 and 2013 and some relevant statistical results.

By using this results we encourage the generation of keys of at least 2048 bits length and also encourage the use of certificates that expire in an year or two. It is important to note from these results that once generated your public/private key pair you should do a minimal check against a database of public keys if a common factor exists. Thus you might be safe for at least two up to ten years (the maximal value of certificate expiration date that we found in our study).

A Python Code for Extended Euclidean Algorithm

```
import math

def extend_euclid_algorithm(r0, r1):

  if(r0 < r1):
    print "Failure: unable to process this request"
    return "ERROR"

  s = [1, 0]
  t = [0, 1]
  i = 1
  r = [r0, r1]
  q = [0]

  while True:
    i = i + 1

    s.append(0)
    t.append(0)
    r.append(0)
    q.append(0)

    r[i] = r[i - 2] % r[i - 1]
    q[i - 1] = (r[i - 2] - r[i])/r[i - 1]
    s[i] = s[i - 2] - q[i - 1] * s[i - 1]
    t[i] = t[i - 2] - q[i - 1] * t[i - 1]

    if r[i] == 0:
      break

  gcd_string = "gcd(%s, %s) = %s" % (str(r0), str(r1), r[i-1])
  s_string = "%s" % str(s[i-1])
  t_string = "%s" % str(t[i-1])
  print gcd_string + " = " + s_string + " * %s " % str(r0) + " +
    " + t_string + " * %s" % str(r1)
  print "s=%s, t=%s" % ( str(s[i-1]), str(long(t[i-1])) )


if __name__ == "__main__":

  extend_euclid_algorithm(67, 12)
  extend_euclid_algorithm(121950720983858964\
  21307279123243774582372328290469176220515\
  78738788423890327596202279187404726945545290\
  122578069128419633718865731073695091\
  0431684065046256L , 65537L)
```

The above code is runnable with both Python 2.7.x and Python 3.

B C code for finding common RSA factor

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>

#define SPACE_SIZE 20000

int main()
{
 FILE* f;
 mpz_t* p_b;
 char* line = NULL;
 size_t len = 0;

 p_b = malloc(SPACE_SIZE * sizeof(mpz_t));
 f = fopen("public_key_n2", "r");
 for(int i = 0; i < SPACE_SIZE; ++i){
  getline(&line, &len, f);
  mpz_init_set_str(p_b[i], line, 16);
 }
 fclose(f);
 free(line);

 #pragma omp parallel for schedule(static, 300)
 for(int i = 0; i < SPACE_SIZE -1; ++i){
  mpz_t gcd;
  mpz_init(gcd);

  for(int j = i+1; j < SPACE_SIZE; ++j){
   mpz_gcd(gcd, p_b[i], p_b[j]);
   if (mpz_cmp_si(gcd, 1)){
    #pragma omp critical
    {
     gmp_printf("%d:%Zx:%Zx\n\n", i, p_b[i], gcd);
     gmp_printf("%d:%Zx:%Zx\n\n", j, p_b[j], gcd);
    }
   }
  }
  mpz_clear(gcd);
 }
}
```

To compile the above code you need `libgmp-dev` and `openmp` libraries.

References

[1] C. Paar and J. Pelzl, *Understanding cryptography.* Springer, 2010.

[2] A. P. Vasile Preda, Emil Simion, "Symmetric cryptography." `http://andrei.clubcisco.ro/cursuri/f/f-sym/5master/aac-atcns/Criptografia%20asimetrica.pdf`.

[3] A. Atanasiu, "Random generators." http://www.galaxyng.com/adrian_-atanasiu/cursuri/cript/cr2_3.pdf.

[4] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the HTTPS certificate ecosystem," in *Proceedings of the 13th Internet Measurement Conference*, Oct. 2013.

[5] "Gmp library." `https://gmplib.org/`. [Online; accessed 11-Jun-2015].

[6] "Python pycrypto library." `https://www.dlitz.net/software/pycrypto/api/current/`. [Online; accessed 11-Jun-2015].

[7] E. Simion, "The Relevance of Statistical Tests in Cryptography," *Security and Privacy, IEEE*, vol. 13, no. 1, pp. 66–70, 2015.