Weaknesses of RSA and DSA keys used in networks

How bad entropy can affect your security system

Bruno Keith
ENSIMAG, Grenoble-INP
Grenoble, France
bruno.keith@ensimag.grenoble-inp.fr

Charles-Elie Simon
ENSIMAG, Grenoble-INP
Grenoble, France
charles-elie.simon@ensimag.grenoble-inp.fr

ABSTRACT

RSA and DSA keys can be vulnerable to attacks when a very large number of keys have been gathered. In this paper, we discuss such an attack on RSA by reproducing the attack and we provide a theoretical approach to this attack by approximating the number of keys necessary to factorize a given key with a given probability.

We also tried to determine if entropy holes at boot-time can be used to attack keys that are generated at this moment.

We largely based our work on "Mining your Ps and Qs: Detection of Widespread Weak Keys in Network Devices"[10], a research paper published in 2012 by Nadia Heninger, Zakir Durumeric, Eric Wustrow and J. Alex Halderman.

Keywords

RSA, entropy, PRNG

1. INTRODUCTION

RSA keys can be vulnerable to attacks when a very large number of keys have been gathered. Moreover if multiple keys have been generated with insufficient entropy, they can be an easy target for an attack based on gcd computation.

In this paper we conducted a theoretical approach to this attack: we provide a formula that allows someone to estimate the number of keys necessary to crack a given key using an attack using gcd computation based on the "Birthday Paradox"[4].

We also reproduced the attack on keys with small size generated using OpenSSL to check our results.

Then, we checked if certain applications that generate keys at boot-time could generate keys with insufficient entropy, we gathered keys from virtual machines that we mounted just to generate the keys.

Finally, we discuss counter-measures to such attacks and how we can improve the security regarding random numbers generation.

We largely based our work on "Mining your Ps and Qs: Detection of Widespread Weak Keys in Network Devices"[10], a research paper published in 2012 by Nadia Heninger, Zakir Durumeric, Eric Wustrow and J. Alex Halderman.

2. STATE OF THE ART

In 2012, Nadia Heninger, Zakir Durumeric, Eric Wustrow and J. Alex Halderman performed an Internet-wide scan of TLS and SSH servers and found that a lot of them used vulnerable RSA and DSA keys.

They were able to factorize the RSA keys in about 0.5% of the servers by computing the gcdx of all pairs of keys but they do not provide a theoretical approach to this attack. Their work showed that a lot of RSA and DSA keys used in networks are vulnerable because of weak entropy at generation or other problems in implementation.

2.1 RSA

RSA is a well known and widely used public-key cryptosystem. As in such a system, RSA uses two different keys, one public and another one kept private. We don't describe RSA encryption and decryption processes as they are beyond the scope of this article and instead focus only on the key generation process.

The RSA key generation step involves choosing two prime factors, usually noted p and q, and computing the modulus $n = pq$, which is public, used for encryption and decryption. The security of RSA relies on the fact that given a modulus n it is computationally infeasible to retrieve p and q for keys of a certain size. If an attacker retrieve those factors, it is very easy to compute the private key.

Our paper will focus on finding collisions on those prime factors.

2.2 Attack using GCDs on a large RSA key collection

In this section, we will discuss the attack that we conducted on large collections of keys of variable length. We focused our efforts on keys of size 32 to 96 bits because we did not have access to the sufficient infrastructure to conduct attacks on larger keys.

The main principle behind this attack is the following: given two moduli N_1 and N_2, if they share one prime factor, an attacker can obtain it by computing $\gcd(N_1, N_2)$ thus allowing allowing to efficiently factorize both keys and retrieve the two private keys.

To attack a key, an attacker can generate a large number of keys hoping that one of these keys will have one prime factor in common with the targeted key. It is somewhat similar to a birthday attack[3] where the likelihood of a collision, that is to say of a prime factor being repeated, allows a more efficient attack than to generate all possible keys, as in a classical brute force attack.

2.3 DSA

DSA is an algorithm used for digital signature. It works as the following: the public-key is composed of three domain parameters, two primes p and q and a generator g of the subgroup of order q mod p, as well of an integer y which is equals to $g^x mod\ p$ with x the private key. A DSA signature is a couple of integers r and s where r is $r = g^k mod\ p$ and s is $s = k^{-1}(H(m) + xr)\ mod\ q$, k is a randomly generated ephemeral private key and H(m) is the hash of the message to sign.

2.4 Attack on DSA when low entropy is involved

Since DSA uses an ephemeral randomly generated key to sign the message, it is really important that enough entropy is used at generation so that an attacker cannot predict k or that k does not take two identical values to sign two different messages. The following attacks enables an attacker to compute the private key x when one these situations arise.

In the event that an attacker knows the ephemeral key used to sign the message, he can retrieve the private key x as follows:

$$x = ((sk) - H(m))r^{-1} \mod q$$

In the event that two messages, m_a and m_b, have been signed with the same ephemeral key k, an attacker can retrieve this key k using the following computation where we note s_a the signature of m_a and s_b the signature of m_b:

$$s_a - s_b = k^{-1}(H(m_a) + xr) - k^{-1}(H(m_b) + xr)$$

which is equivalent to:

$$s_a - s_b = k^{-1}(H(m_a) - H(m_b))$$

k can then be computed as:

$$k = \frac{H(m_a) - H(m_b)}{s_a - s_b}$$

Once the attacker has retrieved k, he can compute x with the first formula.

2.5 Generating random numbers using Linux' random and urandom

2.5.1 Generating random numbers

According to the Linux `random(4)` manual page[11], `/dev/random` and `/dev/urandom` both are special files that let the user interact with the kernel random-number generator. It is important to remember that a CPU cannot create randomness by itself. However, device drivers, among others, constitute sources of noise (coming from the movement of the mouse, for instance) that are external to the CPU. They can be used by a computer to create true randomness. It should be noticed that embedded and headless devices benefit from less entropy sources than common personal computers. The kernel random-number generator thus collects the noise coming from its environment and stores it in an entropy pool. `/dev/random` and `/dev/urandom` are interfaces that use entropy from the entropy pool so as to provide data (understand bits) that's randomly generated.

The command

```
cat /proc/sys/kernel/random/poolsize
```

lets the user know the size (in bits) of the entropy pool (4096 bits, usually).

The command

```
cat /proc/sys/kernel/random/entropy_avail
```

lets the user know how much entropy (in bits) is currently available in the pool. This number can be interpreted as the number of truly randomly generated bits available for use. The displayed value is, however, only an estimation. Therefore, when the result is 4096, it means that the pool is full and that `/dev/random` and `/dev/urandom` can provide the user with 4096 truly randomly generated bits.

A user trying to read data from `/dev/random` when the entropy pool is not full enough ushers in the said user being blocked until enough environment noise is gathered by the kernel random-number generator to satisfy the request. `/dev/random` cannot provide more randomly generated bits than the value displayed by the command:

`cat /proc/sys/kernel/random/entropy_avail`.

On the contrary, when the entropy pool is not full enough, `/dev/urandom` uses a pseudorandom number generator (PRNG) to provide the requested bits (in a deterministic way): reading from `/dev/urandom` can never block the user, whereas reading from `/dev/random` can.

The Linux `random(4)` manual page underlines the fact that the kernel random-number generator was not designed to produce randomly generated bits at a high pace, but rather to focus on quality (which means low quantity in our context). Indeed, the truly randomly generated bits provided by `/dev/random` and `/dev/urandom` should be used to seed cryp-

tographic pseudo-random number generators (CPRNG). Processes should not read unnecessarily from `/dev/random` and `/dev/urandom` (that is to say squander entropy) as it may deprive other processes from the truly randomly generated material they need. Furthermore, this remark is especially true when it comes to processes reading from `/dev/urandom` for they may be actually provided with pseudo-randomly generated bits instead of truly randomly generated bits.

The authors of *Mining your Ps and Qs...*[10] highlighted that `/dev/urandom` is immensely more used than `/dev/random` by developpers for `/dev/random` is commonly considered to be too restrictive: such behaviour from programmers is not consistent with the way the kernel random-number generator was conceived, for it corresponds to a refusal to wait long enough to ensure that the produced data is cryptographically strong. Indeed, the very fact that, in the context of a lack of entropy, `/dev/urandom` keeps providing bits (in a deterministic way) without any warning can be misleading and can entail the production of cryptographically weak material without anyone noticing. It is also interesting to notice that most Linux distributions (see Linux `random(4)` manual page) save entropy in a file on shutdown, which means that entropy can be available when booting. However, as stated in *Mining your Ps and Qs...*[10], on the first boot, such a file is not available, which means that using `/dev/urandom` in this context can lead to a deterministic output, and therefore to cryptographically weak content, such as repeated keys, for instance.

2.5.2 Statistical testing of uniform random number generators

So as to assess the quality of the output provided by `/dev/urandom`, we decided to test it using a utility called `TestU01`[13] (version 1.2.3) which is, according to its website, "a software library [...] for the empirical statistical testing of uniform random number generator". We performed two batteries of tests provided by `TestU01`: `SmallCrush`, which is a "small and fast battery of test" and `Crush` which is more thorough. We used a ASUS N550JV laptop which features a Intel Core i7 4700HQ (2.40 GHz - 3.40 GHz) processor and 2 x 4096 Go of DDR3 1600 MHz SDRAM for the experiment. Our operating system was Ubuntu 14.04.2 LTS. `/dev/urandom` passed all the tests in `SmallCrush`. The execution of this battery of tests took approximately two minutes on our machine. However, it failed a test called `57 MatrixRank, 60 x 60` in `Crush`. All other tests were passed. The execution of `Crush` took more than three hours on our machine. Thus, it shows that a lot of work has been achieved to create quality artificial randomness using computers but there is still work to be done.

3. THEORETICAL AND EXPERIMENTAL APPROACH OF THE ATTACK ON GCDS

3.1 Theoretical approach of the attack on GCDs

In this section, we discuss the two parts of our theoretical approach to the attack on RSA based on GCDs computation, the first is focused on a collection of key without any focus on a given key where we distinguish two cases: one where the length of the key is an even number, another where the length of the key is an odd number. The other point is centered around breaking a given key using the same principle.

3.1.1 Attack on a collection of keys

To approximate the number of keys of a given size necessary to crack at least one of them, we used an approach similar to the birthday paradox.

Let n be the number of bits of a given modulus N and let $Q(n)$ be the number of possible values for the factors p and q of N.

For a key with an even number of bits n, $Q(n)$ can be described, for OpenSSL, as the following: it is the number of primes between $2^{\frac{n}{2}-1}+2^{\frac{n}{2}-2}$ and $2^{\frac{n}{2}}$. To generate a key of size n, OpenSSL will generate two primes p and q on $\frac{n}{2}$ bits by setting the two highest bits to 1 and the lowest bit as well and randomly generating the bits in between to ensure that the product pq is on n bits.

Let $\bar{P}(x,n)$ be the probability that for x generated keys of size n, all factors of all keys are distinct, i.e there are no collisions.

$\bar{P}(x,n)$ can be computed as the following :

$$\bar{P}(x,n) = \prod_{i=0}^{x-1} \frac{Q(n)-2i}{Q(n)} \frac{Q(n)-2i-1}{Q(n)}$$

where x is the number of generated keys.

Therefore the probability that in a given set of keys of same length, at least two of them share a common factor is :

$$P(x,n) = 1 - \bar{P}(x,n) = 1 - \prod_{i=0}^{x-1} \frac{Q(n)-2i}{Q(n)} \frac{Q(n)-2i-1}{Q(n)}$$

We tried to predict the number of keys to generate in order to have a 50% chance to be able to factorize at least one key (in reality two because if one key is factorized using the gcd attack, the other key with the common prime factor will be factorized as well). To approximate the number of prime numbers in the interval described above, we used the following approximation of the prime-counting function [6] noted $\pi(x)$:

$$\pi(x) \simeq \frac{x}{log(x)-1}$$

Therefore we can express $Q(n)$ as the following:

$$Q(n) = \pi(2^{\frac{n}{2}}) - \pi(2^{\frac{n}{2}-1}+2^{\frac{n}{2}-2})$$

Given n the length of our modulus N and p a probability, let x be the value so that $P(x,n) \simeq p$. We note:

$$X(p,n) = x$$

We were then able to compute the results shown in *Table 1.*

In the case where keys are encoded on an odd number of bits, the probability described before changes slightly. In

Key length n	X(0.5, n)	Approximation using (1)
32	22	22.7
34	31	31.1
36	42	42.7
38	58	58.8
40	81	81.1
42	112	111.9
44	154	154.7
46	214	214.0
48	296	296.2
50	410	410.4
52	569	569.2
54	790	789.9
56	1097	1096.9
58	1524	1524.3
60	2119	2119.4
62	2948	2948.5
64	4104	4104.1

Table 1: Number of keys to generate when p and q are of the same length

this case one of the prime factors will be on $\frac{n}{2}+1$ bits and the other on $\frac{n}{2}$ bits with $\frac{n}{2}$ being the result of the integer division. To simplify the writing of the approach, we suppose that p will be on $\frac{n}{2}+1$ bits.

We note $Q_p(n)$, the number of possible prime values for p and $Q_q(n)$ the number of possible prime values for q. We then note $P_p(x)$ (respectively $P_q(x)$) the probability that all p (respectively all q) are different for x generated keys. We have:

$$P_p(x,n) = \prod_{i=0}^{x-1} \frac{Q_p(n)-i}{Q_p(n)}$$

and

$$P_q(x,n) = \prod_{i=0}^{x-1} \frac{Q_q(n)-i}{Q_q(n)}$$

Therefore if we take our previous notation, $\bar{P}(x,n)$, which is the probability that all p and q are different, can be expressed as the following:

$$\bar{P}(x,n) = P_p(x,n) \times P_q(x,n)$$

since $P_p(x,n)$ and $P_q(x,n)$ represent the probability of independent events.

And therefore:

$$P(x,n) = 1 - P_p(x,n) \times P_q(x,n)$$

3.1.2 Attack on a given key

In this section, we focus on attacking a given key which most likely is the real-case scenario for an attacker. By using the same approach we provide formulas to approximate the number of keys to generate to have a certain probability of factorizing a given key. We first give the formulas for a key on an even number of bits and then for a key on an odd-number of bits.

Taking the same notation from the previous section, we have the following formula for $\bar{P}(x,n)$ which represents the probability, for x generated keys of size n, to **not** have a collision with the targeted key:

$$\bar{P}(x,n) = \prod_{i=0}^{x-1} \left(\frac{Q(n)-2}{Q(n)} \right)^2$$

and we also have:

$$P(x,n) = 1 - \bar{P}(x,n) = 1 - \prod_{i=0}^{x-1} \left(\frac{Q(n)-2}{Q(n)} \right)^2$$

In the case of a key on an odd number of bits, using the previous notation, the odds of not having a collision for x keys is:

$$\bar{P}(x,n) = \prod_{i=0}^{x-1} \frac{Q_p(n)-1}{Q_p(n)} \frac{Q_q(n)-1}{Q_q(n)}$$

and:

$$P(x,n) = 1 - \bar{P}(x,n) = 1 - \prod_{i=0}^{x-1} \frac{Q_p(n)-1}{Q_p(n)} \frac{Q_q(n)-1}{Q_q(n)}$$

3.2 Experimental approach of the attack on GCDs

We did an experiment which consisted of generating the predicted number of keys for a given length in order to have a 50% chance to be able to factorize at least two keys. Then we computed the gcd of all pair of keys to see if we could indeed factorize at least two keys in order to compare the outcomes against our predicted results using the theoretical approach.

For this experiment, we developed a program, mainly in *Python*, that given two arguments, one for the key size, the other for the number of keys, generates as many keys as requested and computes the gcds of all the keys and outputs the number of factorized keys.

To generate the keys, our program use system calls to OpenSSL with the required parameters to generate a RSA key. It works in three steps. The first is to tell OpenSSL to generate a RSA key without a passphrase and store it in a given file. The second one is to ask OpenSLL to extract the public key from the file previously created into a second temporary file. Finally the program asks OpenSSL to print information about the public key. We then parse that output to extract the modulus and put it in a file that will contain all modulus. During the final step we also extract the factor p from the key and store it in another file containing all p from each generated key.

The second step is to compute the gcd of each pair of keys. To do so we used a C++ program that parses the file passed as an argument containing one number per line in hexadecimal format and prints the results to two different files. One of the file contains all the keys that were successfully factorized and the second one contains all the corresponding factors. Our algorithm to compute pairwise key gcd was inspired by the algorithm provided by Nadia Heninger, Zakir Durumeric, Eric Wustrow and J. Alex Halderman. Their algorithm is based off an algorithm due to Bernstein[2]. In a nutshell their algorithm works by computing the gcd of a given key with the product of all the other keys, if one factor is repeated it will be the result of the gcd, however if both factors are repeated the result will be the key itself. On keys of size 1024 or 2048 bits, the odds of the latter case happening are very slim, therefore making the algorithm very suitable. But in our case the odds of both factors being repeated were too high so we had to implement a basic quadractic computation of the gcds using the GNU Multiple Precision Arithmetic Library[1] (GMP).

The final step is to extract all the vulnerable moduli and gcds computed in the previous steps to tell if we were able or not to factorize some keys.

3.3 Experimental results

We ran the experience as the following: for each size of key ranging from 32 to 64 bits with a step of 2, we generated 200 times the predicted number of keys to reach a 50% chance of factorizing at least two keys and we looked at how many times out of those 200 generations we were able to factorize at least two keys giving us a percentage. Finally we redid this ten times to be sure that the results were reproducible. Our reasoning behind doing it this way instead of just making 2000 generations was that we did not want the results to be influenced by bad entropy resulting from multiple successive generations.

At first the results obtained, in number of times we were able to factorize at least two keys, were far off what we predicted with our theoretical approach. We took some time to reevaluate our theoretical approach and after deciding that it wasn't faulty, we decided to take an in-depth look into OpenSSL prime number generation because the prime factors did not seem to be uniformly distributed. We found that OpenSSL generates prime number with the following algorithm:

Algorithm 1: OpenSSL prime number generation

```
do
    x = 11 | random bits | 1;
    while !Probable_Prime(x) do
        x += 2;
    end
while !Miller_Rabin_Primality_Test(x);
return x
```

where the Miller-Rabin primality test[5] is called multiple times. As we can see, with this algorithm, the distribution of prime numbers in a given interval will not be uniform. If we take for instance the interval $[11, 13, 15, 17, 19]$, it is trivial to see that, if all numbers have an equal chance of being generated, the odd of having 17 as the output of the OpenSSL prime generation are higher than those of the other primes in that interval. Since our theoretical approach is based on an uniform distribution of all prime numbers in a given interval, we changed the OpenSSL prime generation process in order to have a uniform distribution. We did that simply by removing the while loop of the algorithm described above.

After doing this, the results found were consistent with what we predicted and they are displayed in *Figure 1.*

3.4 Extrapolation on keys of greater size

Since we were able to confirm our theoretical approach with our experiment, we provide estimations on the number of keys to generate for keys of size 512, 1024, 2048, 3072 and 4096 bits which are widespread size of keys in todays security system so that an attacker can find at least one collision. Those estimations were made so that an attacker has a one in a million chance of being able to factorize at least two keys.

To do so we used a C program using the GMP and MPFR[9] library to have correct rounding when using large values.

Since the formula that we provide in Section 3.1 would require too many operations to be computed as it is we used the following approximation for the birthday paradox. Given a probability p of having a collision the number of drawn values k required to reach this probability can be computed with the following formula:

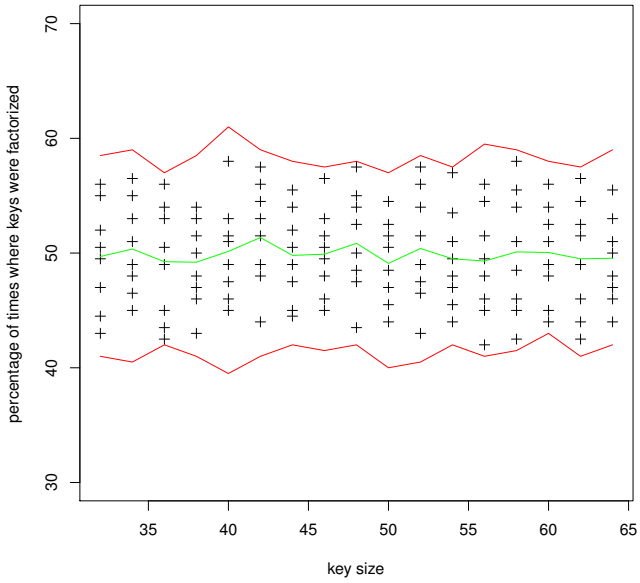

Figure 1: Experimental results of the gcd attack on RSA - the two red lines represent the lowest and highest measures and the green line the average of the ten measures

$$k = \sqrt{2N \ln \frac{1}{1-p}}$$

where N is the number of values that can be drawn. In our case, using the same notation that we used in the previous section, we can rewrite this formula:

$$X(p,n) = \frac{1}{2}\sqrt{2Q(n) \ln \frac{1}{1-p}} \tag{1}$$

We introduce a factor of $\frac{1}{2}$ before the square root because, in our case, we draw two values for each generated key, and $X(p,n)$ is a number of keys.

Running this formula in our program yields, for a probability of 0.000001% to find a collision, the following results that can be seen in Table 2.

Bounds for the approximation - We decided to give lower and upper bounds for our formula that can be computed more easily for they only involve polynomial terms. We remind the following polynomial approximation, where $x \in \mathbb{R}$ and is close to 0:

$$ln(1-x) = -\sum_{k=1}^{n} \frac{x^k}{k} + x^n \epsilon(x)$$

ϵ is a function so that $\lim_{x \to 0} \epsilon(x) = 0$.

We also remind that, for $u \in \mathbb{R}$ close to 0,

$$\frac{u^2}{2} \geq \frac{u^3}{3} + \frac{u^4}{4} + \frac{u^5}{5} + ... \tag{2}$$

Therefore, for $p \in \mathbb{R}$ sufficiently close to 0, we can find the following upper bound by truncating the polynomial approximation. The lower bound is found by applying the inequality 2 to our polynomial approximation. Thus, we get

$$-p - p^2 \leq ln(1-p) \leq -p - \frac{p^2}{2}$$

If $Q(n)$ is defined like before (and n is the length of the modulus), we then get

$$2Q(n)(p + \frac{p^2}{2}) \leq -2Q(n)ln(1-p) \leq 2Q(n)(p + p^2)$$

Finally, since all our terms are positive, we get

$$\frac{1}{2}\sqrt{2Q(n)(p + \frac{p^2}{2})} \leq \frac{1}{2}\sqrt{2Q(n) \ln \frac{1}{1-p}} \leq \frac{1}{2}\sqrt{2Q(n)(p + p^2)}$$

Unfortunately, we were not able to compute the values of the upper and lower bounds for our approximation (see equation 1) on big keys (see Table 2). We ran out of time to investigate this issue in order to fix it.

Key length in bits	Number of keys to generate
512	$9.0348792329384588.10^{33}$
1024	$2.1735426553424143.10^{72}$
2048	$1.7794731627727216.10^{149}$
3072	$1.6823291622045329.10^{226}$
4096	$1.6869942193575589.10^{303}$

Table 2: Number of keys to generate using OpenSSL - This table contains the estimated number of keys to generate to have a $1.e^{-6}$ (0.000001%) chance of being able to factorize at least one RSA key using (1).

4. STUDY OF ENTROPY HOLES

4.1 Experimental approach of key generation at boot-time

The authors of "Mining your Ps and Qs"[10] highlighted that what they call a "boot-time entropy hole" may usher in `/dev/urandom` being predictible at boot-time, especially on first boot. Indeed, at this moment, entropy hasn't previously been saved on shutdown, which explains why this issue at first boot is all the more critical. The consequences of such a vulnerability cannot be neglected, for long-term cryptographic keys can be generated by processes at this very moment. For instance, on distributions such as CentOS and Fedora, OpenSSH is installed by default. This means that the OpenSSH server keys are generated at first boot, and regenerated at boot time if for some reason they were deleted. Therefore, we decided to experiment with key generation at boot time with both these distributions in different versions: CentOS 6.5, CentOS 7.0, Fedora 19 and Fedora 20. Our choice was also motivated by the fact that these versions were available as boxes for Vagrant, which we will explain later. Our aim was to measure the available entropy at key generation for OpenSSH thanks to the command previously mentioned in this article, and to study the generated 2048-bit RSA private key. The file we are interested in can be found at `/etc/ssh/ssh_host_rsa_key` on all the mentioned versions, and is used by the sshd daemon for version 2 of the SSH protocol [12]. In CentOS 7.0, Fedora 19 and Fedora 20, the script launched at boot time to generate the keys for OpenSSH is located at `/usr/sbin/sshd-keygen`. We modified the function `do_rsa_keygen()` of this script so that the output of the command

```
cat /proc/sys/kernel/random/entropy_avail
```

would be saved in a file on the system at the moment of the generation of the RSA private key we wish to study. On CentOS 6.5, the script (which can be found at `/etc/init.d/sshd`) was different (because CentOS 6.5 doesn't feature systemd) but we modified it to achieve the same goal. We realized that, given the relatively short amount of time we had, we would not be able to study entropy and the generation of keys at first boot for it would imply to create our own custom distributions featuring the script modification previously mentioned. Instead, we decided to simply focus on boot-time and to assess whether we could show that this moment is critical entropy-wise or not. Indeed, generally speaking, the available entropy increases after boot-time. It is another reason why we decided to study entropy at boot-time, for it is the moment when cryptographically weak material is most likely to be produced.

To set up our experiment, we decided to use Vagrant. This software let us download and set up distributions easily and quickly. This also means that we experimented on virtual machines, instead of using regular setups. As explained, CentOS 6.5, CentOS 7.0, Fedora 19 and Fedora 20 are available as boxes for Vagrant (understand are available for immediate use with Vagrant). Among the different software compatible with Vagrant, we decided to use VirtualBox to host our guest operating systems. One of the biggest asset of Vagrant is that it lets the user boot the virtualized system with the following command (from the host machine):

```
vagrant up
```

After booting up the system, we used

```
vagrant ssh
```

to access the system and modify the script launched at boot time to generate the keys for OpenSSH. Finally, a single command lets the user shut down the system (from the host machine):

```
vagrant halt
```

Vagrant also lets the user configure the system through a single file. We modified it so that `/etc/ssh/ssh_host_rsa_key` would be deleted before shutdown. Thus, OpenSSH regenerated `ssh_host_rsa_key` at each new boot. We created a bash script which let us boot and shut down a system 30 times in a row using Vagrant. We performed this experiment using CentOS 6.5, CentOS 7.0, Fedora 19 and Fedora 20 as guests systems. Our host system for all the experiments used Ubuntu 14.04.2 LTS. The computer we used for the experiment is a ASUS N550JV laptop which features a Intel Core i7 4700HQ (2.40 GHz - 3.40 GHz) processor and 2 x 4096 Go of DDR3 1600 MHz SDRAM. For each of the four guest systems, we thus obtained 30 2048-bit private RSA keys generated at boot time by OpenSSH, that is to say 30 different copies of the file `ssh_host_rsa_key`, and the corresponding entropy measures at the time of key generation (just before key generation, to be accurate). We could access the files on the guest machines easily through shared folders between host and guest machines and designed our bash script with this information in mind.

4.2 Experimental results

Available entropy at key generation - The distribution of entropy we obtained during our experiments is shown in figure 2. We notice that whatever the distribution, all the measures are located between 120 bits and 200 bits, which is very low. Indeed, 120 bits (respectively 200 bits) only

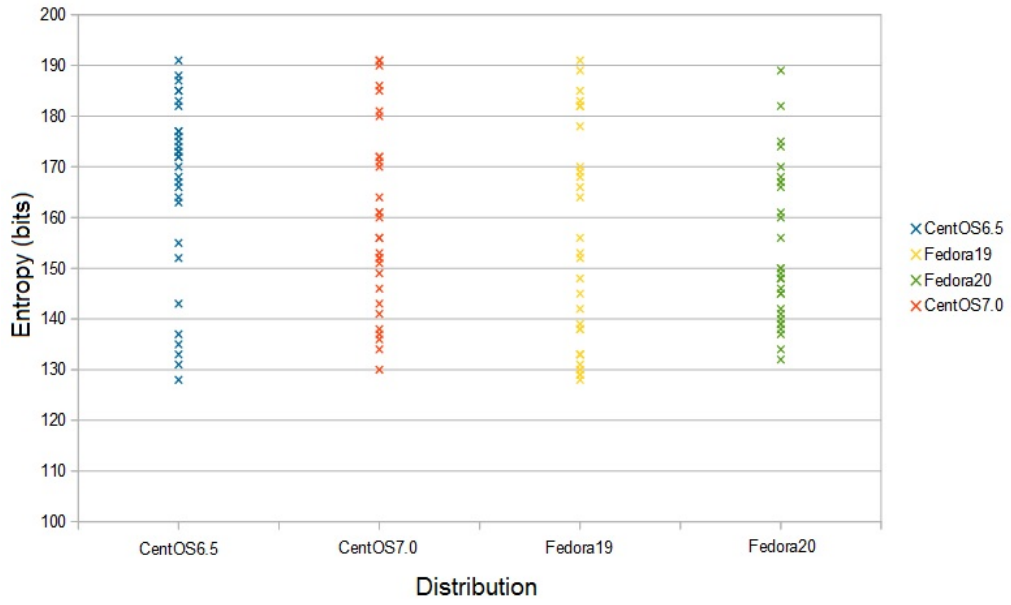

Figure 2: Entropy measures obtained through the experiment described in section 4.1 - For each distribution, we performed 30 measures.

represents 2.93% (respectively 4.88%) of the total capacity of the entropy pool (which is 4096 bits). Furthermore, OpenSSL uses 32 bits to seed its CPRNG, for instance. Besides, a key generator actually needs (only !) 128 bits from `/dev/random` (understand truly randomly generated bits) to generate a 3072-bit RSA private key. We remind that the file `/etc/ssh/ssh_host_rsa_key` is a 2048-bit private RSA key. Therefore, it appears that the amounts of available entropy right before key generation we have observed during our experiment are critical. At boot-time, OpenSSH seeds its internal PRNG from `/dev/urandom` [10]. Should several processes (in addition to the OpenSSH process that generates the keys under study at boot-time) request bits from `/dev/urandom` at boot-time, it is very likely that `/dev/urandom` will not be able to provide enough truly randomly generated bits and will therefore fill the gaps by feeding pseudo-randomly generated bits which can lead to content (e.g. generated keys) that may not be cryptographically secure.

We can assert that we did witness a boot-time entropy hole on each of the four distributions we studied in the conditions of our experiment. Nevertheless, we can assume that such low entropy levels can be explained by the fact that we performed our experiment through Vagrant and used virtual machines. We believe that the available entropy at boot-time would be higher on a regular setup (that is to say not using a virtual machine), but lack the proper arguments to prove this assumption for the time being.

We could draw a parallel with the situation of embedded or headless devices tackled in "Mining your Ps and Qs"[10] because it is likely that our guest systems launched through Vagrant do not benefit from devices such as the keyboard or the mouse as entropy sources for we only began to interact with the guest after bootup when we typed the command `vagrant ssh` in the terminal. In the case of the bash script we used to perform our experiment (see 4.1), we shut down our guest systems right after booting up and didn't even enter `vagrant ssh`: there were no interactions from the host with the guest systems whatsoever. Therefore, in our context, we can assume, while keeping in mind that we do not exactly know how Vagrant works, that devices such as the mouse or the keyboard cannot be considered as entropy sources for our guest systems, just like embedded or headless devices. Thus, we recreated a situation where certain entropy sources are not available and did observe entropy holes at boot time.

We also want to underline the fact that we could never measure the available entropy before key generation at first boot, since we always needed to boot a first time our guest systems to modify the script launched at boot time to generate the keys for OpenSSH. We are fairly certain that the number of truly randomly generated bits available would have been even lower than everything we have observed during our experiment, because most Linux distributions save entropy on shutdown (cf section 2.5.1).

Study of the generated keys We decided to study the keys we gathered to assess whether they are cryptographically reliable or not. Using the following command from OpenSSL,

```
openssl rsa -in rsa_private_key_file -text -noout
```

we were able to obtain the modulus and primes for each generated RSA private key we had gathered in hexadecimal. We used GMP to convert the hexadecimal values of the primes (we obtained 30 keys and therefore 60 primes per distribution) to binary. Our aim was to use TestU01 to evaluate the quality of the randomness of the primes gener-

Key length in bits	Uniform distribution	Non-uniform distribution
32	25	6
40	85	23
48	304	76
56	1155	231

Table 3: Average number of generation for a collision to be - This table contains the average number of keys to be generated successively to find a collision.

ated by OpenSSH at our disposal. However, even by gathering all the primes from all the distributions we studied in a same file, we were unable to run any battery of tests that TestU01 makes available, for the software needed more bits to perform even the simplest battery of tests called `SmallCrush`. TestU01 displayed that it read 253440 bits from our file, whereas, for instance, `SmallCrush` needs more than 51.320.000 bits to work. Besides, we used a bash script to find any duplicates among the primes. We didn't find any. We couldn't decide whether this result stemmed from the fact that OpenSSH could gather enough truly randomly generated bits from `/dev/urandom`, or if we couldn't show that OpenSSH used pseudo randomly generated bits from `/dev/urandom` because we didn't gather enough bits from this very device. Beforehand, even though we estimated that the entropy levels we dealt with at boot time were critical, we were not sure if these levels were enough for OpenSSH to generate distinct primes.

5. COUNTER-MEASURES

5.1 Impact of non-uniform distribution when generating prime factors

Since we discovered that the prime factors are not uniformly generated in OpenSSL, we tried to evaluate the impact that non-uniform generation has in regard to the results that we found in section 3 and see if it has any impact on security, i.e allowing to find a collision a lot faster.

To do so, we successively generated one key until there was a collision with those previously generated. We measured at which generation in average the collisions occurred by repeating the experiment a hundred time for each size of key. We did this experiment on keys of size 32, 40, 48 and 56 bits, we did not do it for key of size 64 as we didn't have enough time to complete the measures.

We did the previous experiment with two different versions of OpenSSL, one which generates prime factors uniformly and the other which is the basic distribution of OpenSSL, and found the results shown in Table 3 (since we did an average of all the measures, we rounded this number up).

As we can see, from this set of measures, collisions tend to happen four times faster with a non uniform distribution than with a uniform one and this seems to increase slowly with key size. Thus we can conclude that, for keys of size 1024, 2048 bits and more, this does not impact the security of RSA. If we take our extrapolation from part 4.1 for keys of 2048 bits, we would need to generate approximately 10^{150} keys, which is greater than 2^{450}, to have a 0.000001% chance of finding a collision. Using a non-uniform distribution means that we would reduce this number to 2^{448} which is still computationally infeasible. Thus we can say that using a non-uniform distribution, in order to have a quicker generation process, is perfectly safe as long as the key size is sufficient.

It is important to note, however, that this supposes a perfect random key generation process and that failing to fulfill this condition, collisions might be found a lot faster, as shown by the article "Mining your Ps and Qs"[10].

5.2 Evaluation of entropy at key-generation

During our research, we found that the software we studied do not present a way of failing gracefully when not enough entropy is present in the system. We think that such softwares should have a way of evaluating the entropy used to generate a key and fail gracefully in the event that the software estimates the entropy is insufficient. We believe that the security gains largely outweigh the disagreement potentially caused by this behavior.

We also advised that softwares, in the example of Unix distributions, use **/dev/random** to seed their PRNG and not **/dev/urandom**. Once again, even though **/dev/random** can hang unpredictably, it is better to use the best source of randomness available when generating keys. OS developers should also provide an interface so that **/dev/random** doesn't hang endlessly.

We implemented a proof of concept of such a counter-measure by modifying the ssh daemon. We forced the daemon to first read the available entropy the system and then, depending on the read value, either generate the key or not.

5.3 Generating additional entropy

While on a system with keyboard and mouse inputs, we might not need to add additional entropy as there are good sources for random input. However this is slightly different on embedded and headless systems.

The paper at the origin of our work highlighted that a lot of vulnerable keys were found in such devices. If you experience worrying levels of entropy on a system, it might be interesting to generate additional entropy on top of what the system is already generating. One of the popular way of doing that is to use an entropy gathering daemon like Haveged[7]. Such a daemon will use unpredictable source of information to feed entropy to Linux' `/dev/random` such as the behavior from the processor caches for Haveged.

Another daemon of the same type is the Entropy Gathering Daemon (EGD) [8] for which OpenSSL and OpenSSH provide interfaces.

5.4 Considerations on regular, safe and strong primes

In this section, we will briefly discuss the impact of the type of prime numbers used for the key generation process of RSA.

If we solely focus on the gcd attack, one can argue that choosing safe or strong prime numbers would improve the efficiency of the attack (since there are fewer values possible, collisions would happen faster). However, as seen in *section 5.1*, if the length of the key is sufficient (as recommended by the competent authorities) and the key generation process is perfectly random, the gcd attack is not a serious threat.

Thus we advise to follow recommendations for prime number generations, whether it is to generate safe or strong primes which offers better security against more threatening attacks.

6. A FEW WORDS ON DSA

Given the short time frame that we had to treat this subject, we chose to focus mainly on RSA, as it was also the focal point of the article which inspired our work. Moreover, a lot of the observations that we made on RSA are directly applicable to DSA.

We highlighted, in the previous sections, the importance of good entropy when generating keys and situations where this value presents a risk of not being sufficient. Such situations could enable the attacks described in the State of the Art section about DSA.

The same counter-measures that we described in the previous section apply on DSA, we advise that softwares responsible for key generation use a set of mechanisms to determine the entropy of the system in order to avoid generating weak keys or predictable sequences.

7. CONCLUSIONS

First , we decided to further study an attack on RSA and DSA keys which was elaborated by the authors of "Mining your Ps and Qs"[10]. We established a formula that let us compute the probability, given a certain number of RSA keys of a certain size, to encounter a collision and therefore to be able to factorize at least two of these keys. We performed an experiment which confirmed that our formula can be considered as consistent. Then, we studied entropy and key generation at boot time and recreated a situation in which a "boot time entropy hole" can be witnessed and discussed the risks it could entail. Finally, we mentioned several counter-measures we thought could improve the issues we encountered that seemed critical to us.

8. ACKNOWLEDGMENTS

We thank Mr. Jean-Louis Roch for his dedication and his availability as our tutor.

9. REFERENCES

[1] Gnu multi precision library. http://en.wikipedia.org/wiki/Prime-counting_function.

[2] D. J. Bernstein. How to find smooth parts in integers, 2015. http://en.wikipedia.org/wiki/Prime-counting_function. [Online, accessed May-2015]

[3] Wikipedia. Birthday attack — Wikipedia, the free encyclopedia, 2015. http://en.wikipedia.org/wiki/Birthday_attack. [Online, accessed May-2015]

[4] Wikipedia. Birthday problem — Wikipedia, the free encyclopedia, 2015. http://en.wikipedia.org/wiki/Birthday_problem. [Online, accessed May-2015]

[5] Wikipedia. Miller-rabin primality test — Wikipedia, the free encyclopedia, 2015. http://en.wikipedia.org/wiki/Miller-Rabin_primality_test. [Online, accessed May-2015]

[6] Wikipedia. Prime-counting function — Wikipedia, the free encyclopedia, 2015. http://en.wikipedia.org/wiki/Prime-counting_function. [Online, accessed May-2015]

[7] Haveged - An Entropy Gathering Daemon http://www.issihosts.com/haveged/

[8] Entropy Gathering Daemon EGD - Entropy Gathering Daemon http://egd.sourceforge.net/

[9] GNU MPFR - Multi-Precision floating-point computations http://www.mpfr.org/

[10] Mining your Ps and Qs : Detection of Widespread Weak Keys in Network devices https://factorable.net/weakkeys12.extended.pdf

[11] linux random man page http://man7.org/linux/man-pages/man4/random.4.html

[12] SSH Daemon - RedHat https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Deployment_Guide/s1-ssh-configuration.html

[13] TestU01 http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf