Detection of Widespread Weak Keys

https://factorable.net/

Jeevesh Juneja

DTU

November 20, 2021

Outline

1. Introduction

1.1. General Setup

The Protocols in Place

- We will look at two protocols, used at network scale: SSH and TLS.
- These protocols make use of two crypto-primitives: DSA and RSA.
- Both crypto-primitives involve generation and exchange of keys.

The Problem

- When systems are designed, we usually have the model of "ALice talks to Bob and Oscar tries to listen in".
- The current paper explore the question: When lots of Alices talks to lots of Bobs, **possibly using same device/model of device** can Oscar break any one of the secure communications, between any pair?
- Meaning, can we infer the keys of any pair of communications by observing lots of conversations between lots of computers?

1.2. Security of Crypto-Primitives

RSA

- Each device has a different RSA moduli $n = pq$ and $e \in \{2, \cdots, p-2\}$ in its public key.
- The private key consists of $d = e^{-1} \bmod \phi(n)$, where $\phi(n) = (p-1)(q-1)$.
- If an attacker wants to guess the private key, d, he must first factor n, calculate $\phi(n)$, then find e^{-1}.
- RSA is secure since it is difficult to factor n.

DSA

- We will essentially talk about Elgamal Digital Signature, which is a simplified form of DSA, as DSA.
- DSA constitutes of a parameters (p, g), where p is a prime, g is the generator of group $\mathbb{Z}_p^*$. These parameters are known to all. Also, it uses a hash function $H(.)$, also known to all.
- The public key is $y = g^x \bmod p$, where $x \in \{2, \cdots, p-2\}$ is the private key.
- To sign a message m, we generate an ephemeral key $k \in \mathbb{Z}_{p-1}^*$, and append to the message a tuple: (r, s), where $r = g^k \bmod p$ and $s = (H(m) - xr)k^{-1} \bmod p - 1$.
- The security of DSA depends on the hardness of solving the Discrete Log Problem, that is identifying x given y.

2. Attacks

2.1. Core Attacks

Factorable keys attack on RSA

- Let's suppose there are lots of devices communicating with each other.
- If we collect many RSA moduli n, and we happen to find a pair of moduli $n_1 = p_1 q_1$ and $n_2 = p_2 q_2$ such that they share one of their prime factors(say$p_1 = p_2$), then **we have lost all security**.
- We calculate GCD(n_1, n_2), which can be done efficiently using Euclid's Algorithm. We get $p_1 =$GCD(n_1, n_2), and we can find $q_1 = \frac{n_1}{p_1}$ and $q_2 = \frac{n_2}{p_1}$.
- As soon as Oscar has factored the keys, he can calculate $d_1 = e_1^{-1} \mod \phi(n_1)$ and $d_2 = e_2^{-1} \mod \phi(n_2)$, and hence get the private keys for both the conversations.

Same Ephemeral Key Attack for DSA-I

- If by chance, any two conversations, happen to have the same long term key x and sign a message with the same ephemeral key k too, again we have lost all security and **the key x has been leaked**.
- The conversations need not happen at same time.
- If we observe the same r value for two conversations, we can conclude they use the same ephemeral key k.
- Now we have:

$$s_1 = (H(m_1) - xr)k^{-1} \bmod p - 1$$
$$s_2 = (H(m_2) - xr)k^{-1} \bmod p - 1$$

Same Ephemeral Key Attack for DSA-II

- The only unknowns are k and x. Subtracting the two equations, we get:

$$(s_1 - s_2)(H(m_1) - H(m_2))^{-1} = k^{-1} \bmod p - 1$$
$$(s_1 - s_2)^{-1}(H(m_1) - H(m_2)) = k \bmod p - 1$$

- We can easily get x now and sign anything we like.
- A single signature collision between any pair of hosts sharing the same long term key, **at any point in the runtime**, reveals the private(long term) key **for every host using that long term key**.

2.2. Running Attacks at Scale

Finding All Pairs GCD-I

- If we collect RSA moduli for N hosts, instead of running Euclid's algorithm N^2 times, we use an algorithm that allows us to find common factors in $\sim N$ runs.
- We compute product P in $log(N)$ time. The computation below is equivalent to removing N_i from P, i.e., computing P/N_i then taking mod N_i.

Algorithm 1 Quasilinear GCD finding

Input: $N_1, \ldots, N_m$ RSA moduli

1: Compute $P = \prod N_i$ using a product tree.

2: Compute $z_i = (P \bmod N_i^2)$ for all i using a remainder tree.

Output: $\gcd(N_i, z_i/N_i)$ for all i.

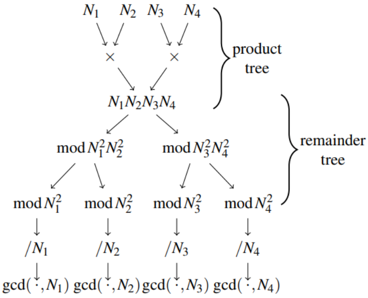


Figure 1: **Computing all-pairs GCDs efficiently** — We computed the GCD of every pair of RSA moduli in our dataset using an algorithm due to Bernstein [6].

Finding All Pairs GCD-II

- We want to find common factors between N_i and all other moduli.
- This is equivalent to finding GCD of N_i and $\prod_{j \neq i} N_j$.
- We observe that $\text{GCD}(N_i, \prod_{j \neq i} N_j) = \text{GCD}(N_i, z_i / N_i)$. Hence we calculate the latter, for all i, as it involves smaller numbers.
- Also, All loops are over i, hence we have a linear time algorithm.
- It is quasi-linear because of the fact that computing $PmodN_i^2$ isn't exactly an $\mathcal{O}(1)$ operation.

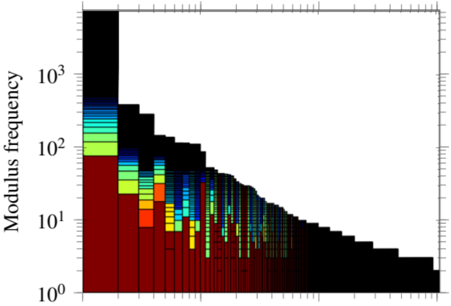
3. Results

Repeated Keys

The paper finds that keys repeat due to the following reasons:

- A server using same key to encrypt messages at its various ports. This is safe, because the server will discard the key soon, and pick a random key again. Also, as both factors are repeated, attacker can't factor using GCD.
- Another reason is many devices have default keys, that are set by manufacturer etc.
- Keys may also repeat due to low-entropy in for e.g., ephemeral key selection in DSA.

Evidence of Low Entropy



(a) Primes generated by Juniper network security devices

On X-axis are various primes p, different colors in a vertical bar correspond to different primes, q. The height of a rectangle correspond to the frequency of repetition of a q value for a given p value. **The long-tailed distribution indicates, rather than flat, indicates a lack of entropy**.

3.1. Linux-Embedded Devices

Why low entropy?

- Most devices are Linux-based. Even embedded ones.
- Entropy is collected from random stochastic events, like mouse movements, kernel interrupt times, keyboard strokes etc.
- This entropy acts as seeds for random number generators.
- Firstly, embedded devices, or devices like routers certainly like many of these sources of entropy. Headless devices are similar too.
- Secondly, there is another issue with Linux Kernel.

Linux RNG

- Entropy is extracted by reading from */dev/random* or */dev/urandom*.
- It is mixed into the **Blocking** or **non-Blocking** pool.
- If the input pool doesn't contain enough entropy to generate the next RNG, the read from Blocking pool block, while, read from non-blocking pool continues.

Experiment: Why low entropy?

- To simulate the conditions for a headless/embedded device, the authors try a new Linux install on their device, and don't allow */dev/(u)random* to take entropy from devices/events usually unavailable in headless/embedded machines.
- They boot their device again and again, each time generating the ssh keys, at the exact time during boot-up when they are generated.
- It is found that the output was entirely predictable and repeatable.
- But why so, when we didn't disable *all* entropy sources?

Boot time Entropy Hole

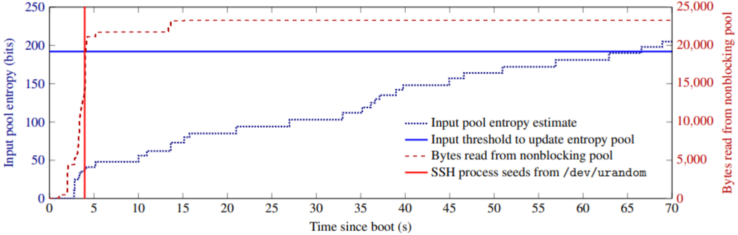


Figure 5: **Linux urandom boot-time entropy hole** — We instrumented an Ubuntu Server 10.04 system to record its estimate of the entropy contained in the Input entropy pool during a typical boot. Linux does not mix Input pool entropy into the Nonblocking pool that supplies `/dev/urandom` until the Input pool exceeds a threshold of 192 bits (blue horizontal line), which occurs here at 66 seconds post-boot. For comparison, we show the cumulative number of bytes generated from the Nonblocking entropy pool; the vertical red line marks the time when OpenSSH seeds its internal PRNG by reading from `urandom`, *well before* this facility is ready for secure use.

- Ubuntu tries to restore entropy from previous boot, but that too, happens slightly *after* the point when **sshd** first reads from */dev/urandom*.

3.2. OpenSSL Application Implementation

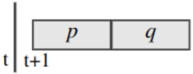
OpenSSL RSA Key Generation-I

- Relies on internal entropy pool seeded on first use with 32 bytes from */dev/urandom*, the process ID, user ID, and current time in seconds.
- Time is added to entropy pool, each time entropy is extracted from the pool.
- RSA key generation generates big random numbers again and again.
- As many keys with one prime common were observed, it is likely that multiple systems start with */dev/urandom* and time in same state, and as we generate more random numbers, the states diverge.

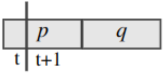
OpenSSL RSA Key Generation-I

Following are the three cases, of clock ticks occurring at various points during key generation of RSA:

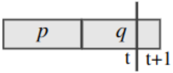
If the second never changes while computing p and q, every execution will generate identical keys.



If the clock ticks while generating p, both p and q diverge, yielding distinct keys with no shared factors.



If instead the clock advances to the next second during the generation of the second prime q, then two executions will generate identical primes p but can generate distinct primes q based on exactly when the second changes.



Future Work

- Extend the study to other protocols/crypto-primitives to Diffie-Hellman key exchange, Elliptic Curve Cryptography.
- Similar vulnerabilities can be searched for in other devices like mobile phones, smart cards etc.
- Try to design protocols/primitives that detect and fail gracefully on weak factorable keys.

Thank you! Questions?