Fermat Factorization in the Wild

Hanno Böck

January 8, 2023

Abstract

We are applying Fermat's factorization algorithm to sets of public RSA keys. Fermat's factorization allows efficiently calculating the prime factors of a composite number if the difference between the two primes is small. Knowledge of the prime factors of an RSA public key allows efficiently calculating the private key. A flawed RSA key generation function that produces close primes can therefore be attacked with Fermat's factorization.

We discovered a small number of vulnerable devices that generate such flawed RSA keys in the wild. These affect devices from two printer vendors - Canon and Fuji Xerox. Both use an underlying cryptographic module by Rambus.

1 Background

1.1 RSA and factoring

We will not give a full description of the RSA algorithm, but we will describe what is necessary to understand the attack.

RSA public keys consist of two values, usually called N and e. N is the modulus and is the product of two large primes, usually called p and q. If someone knows p, q and e then calculating all other values of the private key is trivial.

Therefore, the security of RSA relies on the difficulty of factoring N. It is generally believed that if N is large (2048 bit is the most commonly used value today) and p and q have been independently and randomly generated, then factoring N is not practically feasible with today's technology.

1.2 Fermat's factorization method

Fermat's factorization method is a factorization algorithm that works efficiently on composite numbers composed of two primes where the difference between the primes is small. It has originally been described by Pierre de Fermat in 1643 in a letter [1].

Let's assume a composite number that is composed of two large primes. Given that all primes larger than 2 are odd numbers, there always exists an integer number in the middle of the two primes. We will call that number a. We will call the distance between a and the primes b (the distance is also an integer and identical for both primes, as we have defined a as the middle).

It follows that the smaller prime is a-b and the larger prime is a+b. Therefore, our composite number, which we will call N, can be calculated as:

$$N = (a + b)(a - b)$$

We can rearrange this formula to:

$$N = a^2 - b^2$$
$$a^2 = N + b^2$$
$$b^2 = a^2 - N$$

If the primes are close then we can assume that a is close to the square root of N. We also know that a is larger or equal to the square root of N (it is equal in the special case that the two primes are identical, however this would be an invalid/unusable RSA key).

The factorization works as follows: We will make guesses for a, starting with the integer ceil of the square root of N, and increasing the guess by one in each step. We can check each guess by calculating

$$b^2$$

with the formula above. If the result is a square, we know that we guessed a correct.

This algorithm will always succeed eventually, but it will only be efficient if both primes are close. For this paper, we ran the algorithm for 100 rounds. This is an arbitrary value, larger values lead to more successful factorizations. However, in practice we learned that most vulnerable keys we found in the wild could be factored with one or two rounds, thus we found 100 rounds to be a very conservative choice. The runtime of the algorithm for this value is negligible on a modern PC.

1.3 Breaking RSA with Fermat's factorization method

We learned that the security of RSA relies on the fact that N, a value that is part of the public key, is composed of two large primes, and it is infeasible to calculate these primes for an attacker. We also learned that Fermat's factorization method allows efficiently factoring a composite number composed of two close primes.

[1] A transcript of the letter can be found in the book Oeuvres de Fermat, page 256, an online copy is available at `https://archive.org/details/oeuvresdefermat02ferm/`

The likelihood that two independently, randomly generated large primes are close is negligible. However, it is possible that mistakes in implementing the RSA key generation may create keys with close primes.

A possible scenario is the following: An RSA key generation function creates a large random number. It then searches for the next prime after that random number and uses this as p. It then searches for the next prime after p and uses that as q.

Thus, we end up with two neighboring primes. With common RSA key sizes, neighboring primes will have a difference in the range of thousands, which is easily factorable by Fermat's factorization method.

Other scenarios are also imaginable, e.g. an RSA key generation function could search for the next and previous prime of a random number or it could be based on a flawed random number generator that will output identical numbers for all the higher bits and only create differences in the lower bits.

We would like to stress that Fermat's factorization method is no risk for correct RSA implementations, however a flawed RSA key generation could be vulnerable.

2 Methods

Listing 1: Fermat Factorization in Python

```
def fermat(n):
    tries = 100

    a = gmpy2.isqrt(n)

    c = 0
    while not gmpy2.is_square(a**2 - n):
        a += 1
        c += 1
        if c > tries:
            return False
    bsq = a**2 - n
    b = gmpy2.isqrt(bsq)
    p = a + b
    q = a - b
    return [p, q]
```

We implemented Fermat's factorization method in Python with the gmpy2 library. This is possible in very few lines of code and runs fast enough that we can easily test large numbers of keys. A Python code example is in Listing 1. We implemented a check as part of the badkeys tool and webpage [2] that we

[2] https://badkeys.info/

published under an open source license. The badkeys webpage can calculate private keys for a given vulnerable public key.

We extracted RSA moduli from multiple sets of public keys that were either publicly available or accessible to us. We then applied our attack algorithm and looked for vulnerable keys.

3 Findings

We applied our factorization method to various sets of public RSA keys that we could access.

In a dataset of TLS keys provided by Rapid 7 downloaded in September 2021 we found 97 self-signed certificates with vulnerable public keys. We repeated this with a later data set by Rapid 7 downloaded in December 2021 and found 137 self-signed certificates with vulnerable public keys. Due to overlap overall we found 155 vulnerable certificates in the Rapid 7 datasets. With three exceptions, these certificates were found on the default HTTPS port (443). The three other certificates were on ports 4443, 10443 and 11443.

With three exceptions, all of these certificates had a subject with a common name of the form "FF-1C7D22XXXXXX" or "FX-1C7D22XXXXXX" (with hex digits instead of XX). It is plausible to assume that all certificates with these common names were generated by the same device type.

By using the Censys search engine, we manually checked the IPs belonging to some of these certificates and accessed their web interface. The web interfaces indicated that they belonged to printers of the Fuji Xerox ApeosPort series. (The company has since been renamed to Fujifilm.)

We reported the issue to Fujifilm's security team. We learned that a third-party cryptographic module developed by Rambus was responsible for this vulnerability. Rambus informed us that the vulnerable module is the SafeZone Basic Crypto Module, non-FIPS certified version.[1] Fujifilm has provided a firmware update for the affected printers. Their public advisory contains a full list of affected devices [2]. Rambus has fixed the vulnerability in their library and provided their customers with an update. Based on our communication with Rambus we learned that other vendors were using that library, but Rambus did not share the names of the affected vendors.

Furthermore, we applied our attack on RSA keys from certificates logged in the Certificate Transparency system. Certificate Transparency is a mechanism where certificate issuers need to submit publicly trusted certificates to logs that are public and can be audited by anyone.

Scraping the Certificate Transparency logs is challenging due to their size. Sectigo operates a certificate search engine named crt.sh that offers a public API to access their database. Rob Stradling from Sectigo allowed us to use the database access to extract all RSA moduli from crt.sh. Overall, crt.sh has logged over 6 billion certificates (however not all of them use the RSA algorithm and many share public keys, particularly as the logs often contain both the pre-certificate and the final certificate).

We found 35 affected certificates in this dataset. These certificates contained 17 unique vulnerable keys. Some keys were used on multiple certificates, and for all recent certificates it is common that a pre-certificate gets logged which contains the same public key as the final certificate. All affected certificates were created in 2020 or 2021.

We tried to reach out to the owners of these certificates. In one case, we learned that the certificates were manually created for testing purposes. In two cases, we learned that these certificates belonged to printers from the vendor Canon.

We reported two certificates that were still valid when we found them to the certificate authority that issued them and asked for revocation. It is common practice in the Web PKI ecosystem that certificate authorities revoke certificates within 24 hours if the keys are compromised. In both cases, the certificate authorities revoked the certificates quickly.

The Canon printers, as we learned, offer a feature to generate a certificate signing request (CSR) that includes a vulnerable public key. Certificate signing requests are a common mechanism to deliver a public key to a certificate authority.

We informed Canon about our findings. Canon has informed us that they are preparing a firmware update that will fix this vulnerability.[3]

We also shared our preliminary findings with certificate authorities. Some certificate authorities implemented checks in their issuance process that prevents vulnerable certificates from being issued.[3] The zlint tool that is widely used by certificate authorities to detect malformed certificates has implemented a check in version 3.4.0.[4]

3.1 Negative findings

We applied our factorization method to various other public key collections. That included a collection of SSH host keys from a scan in 2014,[4] multiple collections of GitHub SSH public keys[5], a collection of TLS certificates from the Censys project from 2017, the EEF SSL Observatory collection of TLS certificates from 2010[6] and a dump from an SKS PGP key server.

In the PGP key dataset, we found three vulnerable public keys and one defect key. Two keys had user IDs implying testing ("user_123@test.com", "Test"). One key contained a user id "UserID 3" and an XSS payload as the second user id. The defect key contained no user id and the modulus was a square number (meaning both primes were identical). Such a key is unusable, and it was not possible to import it into GnuPG.

Thus, none of the PGP keys found looked like a "proper" key, the user ids indicate that these keys were created for testing purposes. We therefore believe that these keys were not created by vulnerable software implementations. They

[3]See e.g. Let's Encrypt's check code at `https://github.com/letsencrypt/boulder/pull/5853`

[4]zlint 3.4.0 release notes `https://github.com/zmap/zlint/releases/tag/v3.4.0` and pull request `https://github.com/zmap/zlint/pull/674`

may have been manually crafted by people aware of the Fermat factorization method to deliberately create vulnerable keys.

In all other key sets, we did not find any keys vulnerable to our attack.

We can draw a few conclusions from our findings. All vulnerable TLS certificates we found were of relatively recent origin, and we did not find vulnerable keys in older TLS scan datasets. We did not find any vulnerable SSH keys (neither host nor user keys) and the vulnerable PGP keys we found looked like they were created for testing purposes.

We therefore conclude that vulnerable implementations in the SSH and PGP ecosystems are either very rare or do not exist. We also conclude that all vulnerable TLS implementations are either rarely used or have been created within the past couple of years.

3.2 Checking whether found primes are neighbors

A plausible scenario in which an RSA key generation function would create vulnerable keys is an algorithm that will create neighboring primes p and q. However, other vulnerable implementations are possible, for example ones that fix all the upper bits of a prime up to a certain point and only randomize the lower bits.

We found that all vulnerable keys we found in TLS certificates had neighboring primes, meaning that there was no other prime number between p and q.

Two of the three PGP keys had neighboring primes, one ("user_123@test.com") had two primes between p and q.

This confirms that using neighboring primes is the most likely way an RSA implementation might be vulnerable to Fermat's factorization.

4 Related Work

In 1996 the Austrian terrorist Franz Fuchs sent an encrypted message with a deliberately breakable PGP key to media outlets. According to media reports, the key used primes of similar size, allowing to break the key with Fermat factorization.[7] This likely was not due to a vulnerable implementation, but due to a key deliberately created to be breakable.

We have not found any previous reports of RSA key generation implementations vulnerable to Fermat factorization in the scientific literature.

Flawed RSA key generation algorithms have been found multiple times before. In 2008 Luciano Bello discovered that Debian's OpenSSL package used a flawed random number generator,[8] which limited the number of possible keys.

In 2012 two independent teams discovered that TLS and SSH keys often had shared prime factors,[9][10] allowing to efficiently factor them with the greatest common denominator algorithm. The analysis by Heninger et al indicates that most of these keys were created due to early boot time entropy issues on embedded devices.

In 2017 Nemec, Sys, Svenda, Klinec and Matyas [11] discovered a vulnerability in widely used Infineon hardware chips that created keys with a specific structure that allowed applying a modified version of Coopersmith's attack. The keys were generated with an "optimized" RSA key generation routine that was used in a wrong way. They named this vulnerability "Return of Coopersmith's Attack" or ROCA.

In 2021, it was reported that the keypair JavaScript library had a flaw in the random number generator, making certain keys more likely to appear.[12] It affected SSH keys created by the GitKraken software, which is often used to access code hosting platforms like GitHub.

Most of the previous RSA key generation issues were directly related to the random number generator, which is not directly related to our finding. The major exception is the ROCA vulnerability. The similarity between our finding and ROCA is that in both cases, it looks like implementors tried to be "clever" and optimize the RSA key generation.

Several more modern factoring algorithms like the quadratic sieve or the number field sieve use the ideas from Fermat as their basis. The number field sieve has been used to factor keys with short key sizes.

5 Summary and Conclusion

Fermat's factorization method allows efficiently calculating private RSA keys from public keys if the primes are close. We were able to show that there are vulnerable implementations used in the wild that create such keys for TLS certificates. However, we also learned that these vulnerabilities are not widespread, as we only found a small number of affected public keys and devices.

We did not find evidence of vulnerable devices in SSH keys, and only found a very small number of PGP keys that we believe were created for testing purposes.

We implemented a check for Fermat's factorization method in Python that we shared as part of the badkeys tool[5] that we provide under the MIT open source license. Furthermore, we also provide badkeys as a web service for checking keys for a variety of vulnerabilities.

Checking for Fermat's factorization method is computationally very cheap. We therefore recommend that entities that process public keys that were generated outside of their control to implement checks for this vulnerability. Typical entities that might want to implement such checks are certificate authorities or services that implement user authentication with public keys and support RSA (e.g. code hosting services using Git over SSH). Security audits of cryptographic systems and software implementing RSA should consider this vulnerability.

[5]`https://github.com/badkeys/badkeys/`

6 Misc

We have created a web page with information about this vulnerability at `https://fermatattack.secvuln.info/`.

This work was funded in part by Industriens Fond through the CIDI project (Cybersecure IOT in Danish Industry) and in part by the Center for Information Security and Trust (CISAT) at the IT University of Copenhagen, Denmark.

References

[1] Rambus. Rambus Security Vulnerability Disclosure: SafeZone Basic Crypto Module, non-FIPS certified version. `https://safezoneswupdate.com/`, 2022.

[2] Fujifilm. Notification about the vulnerability for RSA key in our multi-function printers and single-function printers. `https://www.fujifilm.com/fbglobal/eng/company/news/notice/2022/0302_rsakey_announce.html`, 2022.

[3] Canon. Notice of potential vulnerability in RSA key generation. `https://canoncanada.custhelp.com/app/answers/answer_view/a_id/1039057/~/notice-of-potential-vulnerability-in-rsa-key-generation-`, 2022.

[4] Oliver Gasser, Ralph Holz, and Georg Carle. A deeper understanding of SSH: Results from Internet-wide scans. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–9, 2014.

[5] Ben Cox. Auditing GitHub users' SSH key quality. `https://blog.benjojo.co.uk/post/auditing-github-users-keys`, 2015.

[6] Electronic Frontier Foundation. The EFF SSL Observatory. `https://www.eff.org/de/observatory`, 2010.

[7] Markus Sulzbacher. Briefbombenterror: Als das Bundesheer der NSA zuvorkam. *Der Standard*, 2018. `https://www.derstandard.at/story/2000080835367/briefbombenterror-als-das-bundesheer-der-nsa-zuvorkam`.

[8] Debian Security Advisory. DSA-1571-1 openssl – predictable random number generator. `https://www.debian.org/security/2008/dsa-1571`, 2008.

[9] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220, Bellevue, WA, August 2012. USENIX Association.

[10] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Public keys. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 626–642, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[11] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In *24th ACM Conference on Computer and Communications Security (CCS'2017)*, pages 1631–1648. ACM, 2017.

[12] Github Security Lab. GHSL-2021-1012: Poor random number generation in keypair - CVE-2021-41117. `https://securitylab.github.com/advisories/GHSL-2021-1012-keypair/`, 2011.