Factoring RSA Keys With TLS Perfect Forward Secrecy

Sep 10, 2015 • David Wong

Florian Weimer from the Red Hat Product Security team has just released a technical report entitled "Factoring RSA Keys With TLS Perfect Forward Secrecy"

Wait what happened?

A team of researchers ran an attack for nine months, and from 4.8 billion of ephemeral handshakes with different TLS servers **they recovered hundreds of private keys**.

The theory of the attack is actually pretty old, Lenstra's famous memo on the CRT optimization was written in 1996. Basicaly, when using the CRT optimization to compute a RSA signature, if a fault happens, a simple computation will allow the private key to be recovered. This kind of attacks are usually thought of and fought in the realm of *smartcards* and other *embedded devices*, where faults can be induced with lasers and other magical weapons.

The research is novel in a way because they made use of **Accidental Fault Attack**, which is one of the rare kind of remote side-channel attacks.

This is interesting, the oldest passive form of *Accidental Fault Attack* I can think of is **Bit Squatting** that might go back to 2011 at that defcon talk.

But first, what is vulnerable?

Any library that uses the CRT optimization for RSA might be vulnerable. A cheap countermeasure would be to verify the signature after computing it, which is what most libraries do. The paper has a nice list of who is doing that.

Implementation	Verification
cryptlib 3.4.2	disabled by default
GnuPG 1.4.1.8	yes
GNUTLS	see libgcrypt and Nettle
Go 1.4.1	no

Implementation	Verification
libgcrypt 1.6.2	no
Nettle 3.0.0	no
NSS	yes
ocaml-nocrypto 0.5.1	no
OpenJDK 8	yes
OpenSSL 1.0.1l	yes
OpenSwan 2.6.44	no
PolarSSL 1.3.9	no

But is it about what library you are using? Your server still has to be defective to produce a fault. The paper also have a nice table displaying what vendors, in their experiments, where most prone to have this vulnerability.

Vendor	Keys	PKI	Rate
Citrix	2	yes	medium
Hillstone Networks	237	no	low
Alteon/Nortel	2	no	high
Viprinet	1	no	always
QNO	3	no	medium
ZyXEL	26	no	low
BEJY	1	yes	low
Fortinet	2	no	very low

If you’re using one of these you might want to check with your vendor if a firmware update or other solutions were talked about after the discovery of this attack. You might also want to revoke your keys.

Since the tests were done on a broad scale and not on particular machines, it is obvious that **more are vulnerable to this attack**. Also only instances connected to internet that offered TLS on port

443 were tested. The vulnerability could potentially exist in any stack using this CRT optimization with RSA.

The first thing you should do is assess where in your stack the RSA algorithm is used to sign. Does it use CRT? If so, does it verify the signature?

What can cause your server to produce such erroneous signatures

They list 5 reasons in the paper:

- old or vulnerable libraries that have broken operations on integer. For example CVE-2014-3570 was an issue that caused the square operations of OpenSSL to not work properly for some inputs
- race conditions, when applications are multithreaded
- arithmetic unit of the CPU is broken by design or by fatigue
- corruption of the private key
- errors in the CPU cache, other caches or the main memory

Note that at the end of the paper, they investigate if a special hardware might be the cause and end up with the conclusion that several devices leaking the private keys were using Cavium hardware, and in some cases their “custom” version of OpenSSL.

I’m curious. How does that work?

RSA-CRT

Remember, RSA signature is basically $y = x^d \pmod n$ with x the message, d the private key and n the public modulus. Also you might want to use a padding system but we won’t cover that here. And then you can verify a signature by doing $y^e \pmod n$ and verify if it is equal to x (with e the public exponent).

CRT is short for Chinese Remainder Theorem (I should have said that earlier). It’s an optimization that allows to compute the signatures in $\mathbb{Z}_p$ and $\mathbb{Z}_q$ and then combine it into $\mathbb{Z}_n$ (remember $n = pq$). It’s way faster like that.

So basically what you do is:

$$\begin{cases} y_p = x^d \pmod p \\ y_q = x^d \pmod q \end{cases}$$

and then combine these two values to get the signature:

$$y = y\_pq(q^{-1} \pmod p) + y\_qp(p^{-1} \pmod q) \pmod n$$

And you can verify yourself, this value will be equals to $y_p \pmod{p}$ and $y_q \pmod{q}$.

The vulnerability

Let's say that an error occurs in only one of these two elements. For example, y_p is not correctly computed. We'll call it $\widetilde{y_p}$ instead. It is then is combined with a correct y_q to produce a wrong signature that we'll call $\widetilde{y}$.

So you should have:

$$\{ \widetilde{y} = \widetilde{y}_p \pmod{p} \, \widetilde{y} = y_q \pmod{q}$$

Let's notice that if we raise that to the power e and remove x from it we get:

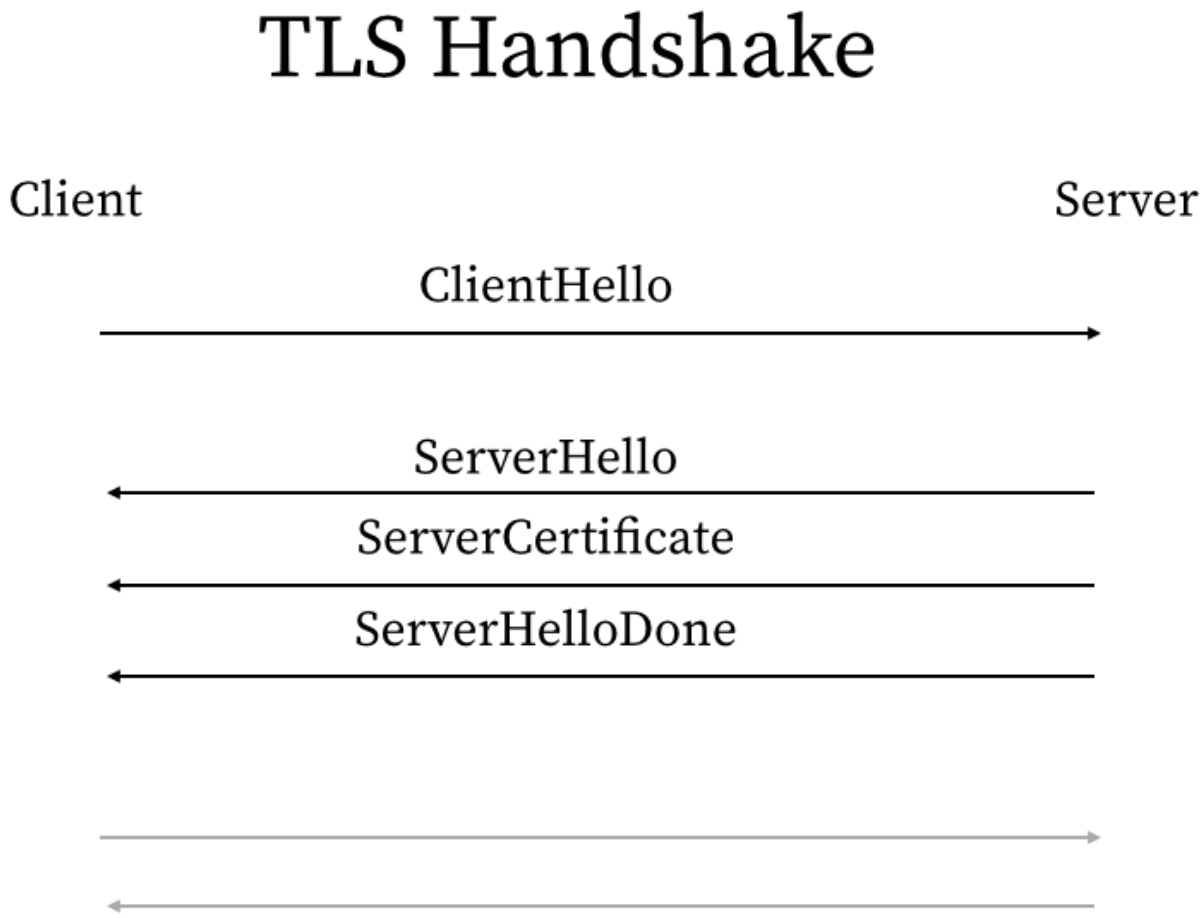
$$\begin{cases} \widetilde{y}^e - x = \widetilde{y}^e\_p - x = a \pmod{p} \\ \widetilde{y}^e - x = y\_q^e - x = 0 \pmod{q} \end{cases}$$

This is it. We now know that $q \mid \widetilde{y}^e - x$ while it also divides n. Whereas p doesn't divide $\widetilde{y}^e - x$ anymore. We just have to compute the Greatest Common Divisor of n and $\widetilde{y}^e - x$ to recover q.

The attack

The attack could potentially work on anything that display a RSA signature. But the paper focuses itself on TLS.

A normal TLS handshake is a two round trip protocol that looks like this:



The client (the first person who speaks) first sends a *helloClient* packet. A thing filled with bytes saying things like “this is a handshake”, “this is TLS version 1.0”, “I can use this algorithm for the handshake”, “I can use this algorithm for encrypting our communications”, etc…

Here’s what it looks like in Wireshark:

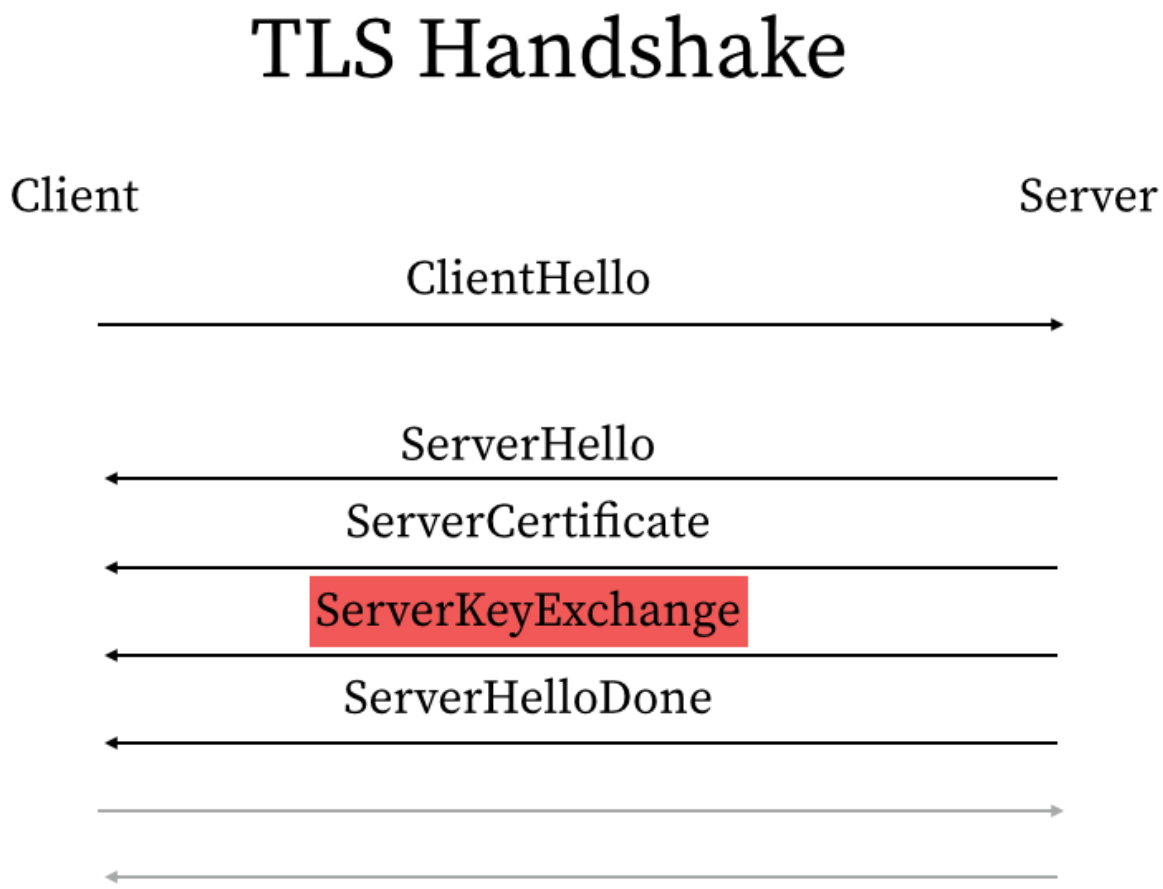
ClientHello

```
▽ Secure Sockets Layer
  ▽ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 302
    ▽ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 298
        Version: TLS 1.2 (0x0303)
      ▷ Random
        Session ID Length: 0
        Cipher Suites Length: 148
      ▽ Cipher Suites (74 suites)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
          Cipher Suite: TLS_DHE_DSS_WITH_AES_256_GCM_SHA384 (0x00a3)
```

The server (the second person who speaks) replies with 3 messages: a similar *ServerHello*, a message with his *certificate* (and that's how we authenticate the server) and a *ServerHelloDone* message only consisting of a few bytes saying "I'm done here!".

A second round trip is then done where the client encrypts a key with the server's public key and they later use it to compute the TLS shared key. We won't cover them.

Another kind of handshake can be performed if both the client and the server accepts ephemeral key exchange algorithms (Diffie-Hellman or Elliptic Curve Diffie-Hellman). This is to provide *Perfect Forward Secrecy*: if the conversations are recorded by a third party, and the private key of the server is later recovered, nothing will be compromised. Instead of using the server's public key to compute the shared key, the server will generate a *ephemeral* public key and use it to perform an *ephemeral handshake*. This key is usually used just for this session or occasionally for a limited number of sessions.



When this occurs, an extra packet called **ServerKeyExchange** is sent. It contains the server's ephemeral public key.

(Interestingly the signature is not computed over the algorithm used for the ephemeral key exchange, that led to a long series of attacks which recently ended with FREAK and Logjam.)

Checking if the signature is correctly performed is how they checked for this potential vulnerability.

I'm a researcher, what's in it for me?

Well what are you waiting for? Go read the paper!

But here are a list of what I found interesting:

- instead of DDoSing one target, they broadcasted their attack.

> We implemented a crawler which performs TLS handshakes and looks for miscomputed RSA signatures. We ran this crawler for several months. The intention behind this configuration is to spread the load as widely as possible. We did not want to target particular servers because that might have been viewed as a denial-of-service attack by individual server operators. We assumed that if a vulnerable implementation is out in the wild and it is somewhat widespread, this experimental setup still ensures the collection of a fair number of handshake samples to show its existence. We believe this approach—

> probing many installations across the Internet, as opposed to stressing a few in a lab—is a novel way to discover side-channel vulnerabilities which has not been attempted before.

- they used public information to choose what to target, like scans.io, tlslandscape and certificate-transparency.
- Some TLS servers need a valid Server Name Indication to complete a handshake, so connecting on port 443 of random IPs should not be very efficient. But they found that it was actually not a problem and most keys found like that were from weird certificates that wouldn't even be trusted by your browser.
- To avoid too many DNS resolutions they bypassed the TTL values and cached everything (they used PowerDNS for that.)
- They guess what devices were used to perform the TLS handshakes from what was written in the x509 certificates in the *subject distinguished name* field or *Common Name* field.
- They used `SSL_set_msg_callback()` (see doc) to avoid modifying OpenSSL.

Cryptography Services

Cryptography Services is a dedicated team of consultants from NCC Group focused on cryptographic security assessments, protocol and design reviews, and tracking impactful developments in the space of academia and industry.