Recover a RSA private key from a TLS Session with Perfect Forward Secrecy

Marco Ortisi

July 25th, 2016

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 ABSTRACT

This whitepaper describes an attack technique against RSA-CRT that whether successfully exploited allows a malicious agent to retrieve the private key from a TLS service supporting Perfect Forward Secrecy cipher suites, just by interacting with it in an active way or passively sniffing the traffic.

Section 2 introduces the history of this attack and the preconditions that must be satisfied for its exploitation.

Section 3 provides a gentle introduction to the RSA cryptographic algorithm and RSA-CRT optimization. Once the basics have been built, Section 4 describes in detail how to exploit the vulnerability. Affected crypto libraries, software solutions and hardware products are covered on Section 5. The proof of concepts developed, including how to set up a test environment to do practice with the exploitation of this vulnerability, is presented in Section 6.

External links and resources are finally listed on Section 7.

2 INTRODUCTION

The RSA-CRT attack has very deep roots in the past but for long time has been believed exploitable only locally. In 1996 Arjen Lenstra[1] demonstrated that the usage of the so-called CRT (Chinese Remainder Theorem) optimization put the RSA implementations at great risk if a fault occurred during the computation of a digital RSA signature. Specifically the risk was the leak of the private key of server.

Around 2000, researchers conjectured that smart cards were sensibly affected by the same problem, but this required the necessity for an attacker to have physically access to the device in order to try to disrupt the math behind RSA and retrieve the private key. In 2001 a pool of researchers discovered what will be known in the IT security history as the "*OpenPGP format attack*"[2]. An attacker could retrieve the server's private key by:

1. getting a local copy of file containing the encrypted private key;
2. tampering with it in order to introduce faulty bits;
3. capturing a single message subsequently signed with the modified encrypted private key;

Of course, once more, this required local access in the system to work. The first researcher that described how the attack could also have remote impacts and implications was *Florian Weimer*[3] from RedHat at the end of 2015. Unfortunately, no proof of concept was released along with his research.

The hereby whitepaper aims to fill up this gap, trying at the same time to explain with an approach more "for the masses" the details and implications of this attack technique.

2.1 RSA-CRT PREREQUISITES

In order to exploit the RSA-CRT vulnerability on a target system, three prerequisites must be met:

- The RSA signature must be calculated using the RSA-CRT optimization;
- The RSA signature must be faulty, namely calculated in a wrong way;
- The RSA signature must be computed on values the attacker knows in clear-text.

Below we analyze all of them one by one.

2.1.1 RSA signature with RSA-CRT

The modular exponentiations required by RSA are computationally heavy. RSA-CRT is an optimization that introduces a less expensive way to do RSA calculations where a private key is involved (decryption and signing operations). It is used by default in every modern crypto library (including openssl, mbedTLS, Java SE, Nettle, libgcrypt, etc…) and due to performance reasons is not advisable to disable such a feature. Because of this, the hereby prerequisite is normally satisfied.

2.1.2 The RSA signature must be faulty

Events causing some faults during computational operations because of CPU overheating, RAM errors, massive exposure of hardware to solar rays, radiation, abnormal voltage, etc… have been well-known and documented in the past. For example, they are the cause that

make possible a bitsquatting attack. The miscalculation of a digital RSA signature is an unpredictable event but this does not mean it will never occur. It is not only is possible but can also happen without the external intervention of a malicious agent.

2.1.3 The RSA signature must be calculated on known values

This condition is normally met if the attacker carefully chooses to negotiate only specific cipher suites and, of course, if the server supports those. When Diffie Hellman is used as a key exchange mechanism and RSA is used as authentication algorithm only, this prerequisite is always satisfied, including when the Elliptic Curve variant is agreed with the TLS connection.

In fact, if cipher suites such as *DHE_RSA_WITH_ANY_ANY_ANY_ANY* or *ECDHE_RSA_WITH_ANY_ANY_ANY_ANY* are agreed, a dynamically generated RSA signature is appended onto a TLS *Server Key Exchange* Message. This signature is applied on three specific, concatenated, pieces of information which are observed during the TLS handshake (for more detail about that process see the section 4.1):

- *Client Random structure* taken from the Client Hello Message;
- *Server Random structure* taken from the Server Hello Message;
- *Server Params structure* taken from the Server Key Exchange Message;

With TLS, a message that needs to be signed is hashed and padded first, so that the result produced in output is a stream of bytes which has the same length of the RSA key. Padding is very important because actually it changes the final shape of a clear-text message. Moreover, for this specific attack technique, it is imperative for the attacker to know the exact clear-text message which is signed. Anyway, the padding scheme used by SSL in every known version nowadays (from SSL 3.0 to TLS 1.2) is a fully deterministic variant of PKCS 1.5. This means that it is predictable.

2.1.3.1 Padding with TLS < 1.2

A RSA digital signature is nothing more than a hash of some plain-text value then encrypted with the private key of server. For example, assuming that the following hash (from now on termed "*payload*") must be signed:

**0D3F8FF87A4D697E73FE86077FD1D10C4ECC59797E759EDD89931B2208B8044CB4A
1B96A**

Below is how the message would be padded before a signature with RSA 2048 was applied and inserted inside a TLS *Server Key Exchange* Message:

0001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00**0D3F8FF87A4D697E73FE86077FD1D
10C4ECC59797E759EDD89931B2208B8044CB4A1B96A**

Basically "0001" (green color) is always prepended to the sequence of padding bytes, instead "00" (blue color) is always appended to the end. In the middle the padding is more or less

composed by a long sequence of "FF" (violet color) depending on the payload length (yellow color), where:

```
number_of_FF = (size_of_RSA_key – green_bytes – blue_byte – payload_len)
```

Finally the payload itself follows.

2.1.3.2 Padding with TLS 1.2

With TLS 1.2 the padding scheme is pretty similar to that one seen in the previous paragraph. There is a small difference anyway. Let us assume the payload to be signed is the following:

3B62EAB7A60E798C9E251FD8399FC3619B1B5B751B042AFE8D7A123DD850D839653
EFBAC11B17C37182FA9532D2C17804F75F7DDBD84D57A4C4E062771F225A3

Below how it would be padded:

```
0001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF003051300D060960
8648016503040203050004403B62EAB7A60E798C9E251FD8399FC3619B1B5B751B042A
FE8D7A123DD850D839653EFBAC11B17C37182FA9532D2C17804F75F7DDBD84D57A4
C4E062771F225A3
```

Basically, the new entry here is the grey part. It represents the DER encoded form of the Object Identifier (OID) for the hashing algorithm used to hash the payload (yellow color). In this specific case it indicates **SHA512**[6]:

```
const unsigned char sha512_der_encoded[] =
"\x30\x51\x30\x0d\x06\x09\x60\x86\x48\x01\x65\x03\x04\x02\x03\x05\x00\x04\x
40";
```

With TLS < 1.2 the indication of the hashing algorithm used is unnecessary because the payload is always the outcome of two hashes concatenated each other and respectively calculated by using the *MD5* and *SHA1* hashing algorithms (see 4.3.1).

3 GENTLE INTRODUCTION TO RSA

Below follows a gentle introduction to RSA. It is about how RSA works and why RSA-CRT poses a problem. This chapter can be skipped if the reader deems to own solid basics about this crypto standard.

3.1 WHAT IS A RSA SIGNATURE

To understand the basis of this whitepaper is very important to have a gentle introduction to the main concepts around RSA and public key encryption. With RSA a *public key* (freely distributable) is used to encrypt a message, instead the *private key* (that as the name suggests must be kept secret) is used to decrypt that message. To oversimplify we can say that there is a case when the private key is instead used to encrypt, namely when an entity wants to sign a message. In that specific case, the *public key* is conversely used to decrypt such a value and prove the authenticity of the signature because of the "mathematical" connection between *private key* and *public key*.

3.2 HOW RSA WORKS

RSA is a very simple standard. The encryption and decryption functions can be merely described in terms of *multiplication*, *exponentiation* and *modulo* operations:

```
encryption =  c = m^e % n
decryption =  m = c^d % n
```

Where:

```
^ = exponentiation operation
% = modulo operation
* = multiplication (when used below)
```

In this context:

- **c** is the *ciphertext* (the encrypted message);
- **m** is the *message* to encrypt;
- **n** and **e** are public information and in fact they compose the public key. While "**e**" is a little exponent (usually a value such as 3 or 65537) that is found inside the server certificate (Figure 4), **n** is instead a big semi prime number which is also found inside the server certificate (Figure 3) and is given by the multiplication of two prime numbers termed "**p**" and "**q**". The security of RSA rotates entirely around the fact that these two prime numbers must not be revealed or discovered.
- **d** is the private key mathematically tied with **n**. It is calculated by feeding the modular inverse function[4] with "**e**" and the result of **p-1** multiplied by **q-1** (see the formula below):

  ```
  d = inverse_mod(e,(p-1) * (q-1))
  ```

Just a couple of sentences above, we have said that the security of RSA rotates around the robustness of "**p**" and "**q**". To say the truth, it is even worse than this. Actually, the discovery of just one of "**p**" and "**q**" is enough to derive a private key, because "**e**" is already a public information. This can be verified very easily. Reasoning on small numbers, let us assume that **n** (the public key into the certificate) has the value 77. If at some point, an attacker discoveres the value of "**p**" is 7, the other prime number "**q**" can be recovered just dividing **n** by **p**:

```
q = 11 -> 77 / 7
```

The same is whether the attacker discovers that **q** is equals to 11. In that case, **p** can be retrieved just by simply calculating:

```
p = 7 -> 77 / 11
```

And naturally **p** multiplied by **q** gives **n** as a result:

```
n = 77 -> 7 x 11
```

Once again, this means that by deriving a prime factor of **n** (whatever of the two) one can determine the other one and recover the private key. Here is where the problem with **RSA-CRT** comes. When the Lenstra attack[1] is successful, it allows to acquire one of the prime numbers ("**p**" or "**q**"). Let us see how.

3.3 RSA-CRT AND LENSTRA ATTACK

RSA-CRT (Chinese Remainder Theorem) has been developed as a performance optimization for RSA operations where the RSA calculation is broken down in two smaller parts. The following values are precomputed:

```
qInv = (1/q) mod p
dP = d * (mod p - 1)
dQ = d * (mod q - 1)
```

These other values shown below are instead calculated dynamically because the message to sign is of course supposed to be always different:

```
s1 = m^dP mod p
s2 = m^dQ mod q
h = (s1 – s2) * qInv mod p
s = s2 + q * h
```

If during the calculation of **s1** or **s2** there is a fault, a wrong signature is generated. An attacker can retrieve a prime factor of **n** from a miscalculated signature when RSA-CRT is used for signing operation by using the greatest common divisor[5] function shown below (see [1] for further details):

```
gcd(y^e – x, n)
```

The information required to feed the "`gcd`" function are all public and visible during the TLS handshake. In addition, those can be also captured with no effort with a passive sniffer. In fact:

- **y** is the faulty/miscalculated signature. This can be taken directly from the TLS *Server Key Exchange* packet (Figure 6);
- **e** is the public exponent found inside the TLS *Certificate* Message (Figure 4);
- **x** is the original value before to be signed. It is padded but PKCS 1.5 padding scheme is deterministic with TLS (see the paragraph 2.1.3);
- **n** is the public key (**p** * **q**) also found inside the *Server Certificate* (Figure 3);

When the greatest common divisor formula is applied, one of the prime factors of **n** leaks out, becoming known to the attacker. At this point he/she can derive very easily the other one:

```
other_prime_factor = n / leaked_prime_factor
```

Finally, the attacker has the control of both **p** and **q** and then the private key is lastly computed:

```
d = inverse_mod(e,(prime_factor_P - 1) * (prime_factor_Q -1 ))
```

3.4 Double Check RSA-CRT

A possible suggested check to prevent that **p** or **q** can leak out is to verify that the digital signature calculated with RSA-CRT is actually not faulty by checking:

```
y^e = x % n
```

This check is still less expensive rather than consider the possibility to switch back to the original RSA implementation without any optimization for the calculation of digital signature:

```
m^d % n
```

This is due to the exponentiation. In fact "**e**" is a very small exponent compared to "**d**" (the private key of server).

4 EXPLOITING RSA-CRT

How the RSA-CRT attack works is very simple. The attacker establishes a TLS handshake with the target by negotiating Perfect Forward Secrecy cipher suites only (see 2.1.3). Each packet sent out or received, the attacker collects a precise piece of information that will allow him/her to move forward with the attack. At a specific point of the handshake, a RSA digital signature is found and it is checked for congruency. If not valid, the recovery of the private key of server is attempted via the Lenstra attack of 1996 (3.3). In contrary case, a new TLS handshake is established. It is said *active approach*.

In addition, instead of actively establish TLS connections, the attacker can also exploit the RSA-CRT vulnerability just by capturing the network traffic (*passive approach*).

4.1 ACTIVE APPROACH

With the active approach, multiple sequential TLS handshakes are established with the target so to negotiate Perfect Forward Secrecy cipher suites, until a failure in the RSA digital signature is detected. At the point, the attacker attempts to calculate the private key of server.

First, the attacker sends a TLS *Client Hello* message. This packet includes a list of supported cipher suites (remember, only PFS cipher suites must be agreed) but more importantly contains a 32 bytes ***Random*** structure (Figure 1) which will be useful later.

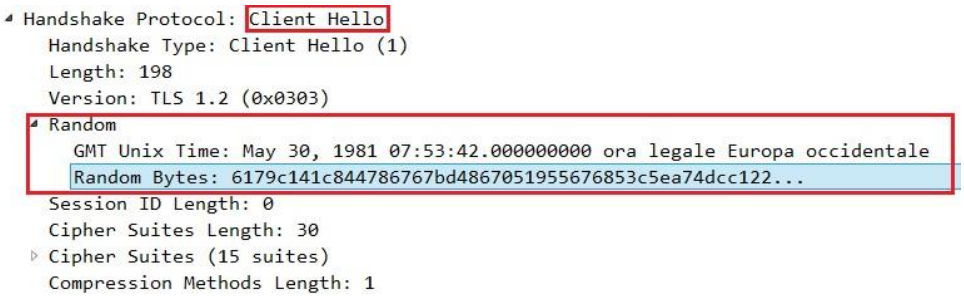

Figure 1 – Random Structure from TLS Client Hello Message

The server answers with a TLS *Server Hello* message. This packet indicates which cipher suite has been chosen by server. Anyway, in particular, the attacker is specifically interested on a 32 bytes ***Random*** structure (Figure 2) which will be useful later.

Figure 2 – Random Structure from TLS Server Hello Message

Afterwards, the server transmits back to the client a TLS *Server Certificate* message. For the purposes of the attacker, the most interesting pieces of information here are two values called

"**n**" (Figure 3) and "**e**" (Figure 4) found inside the digital certificate. They represent the public key of server.

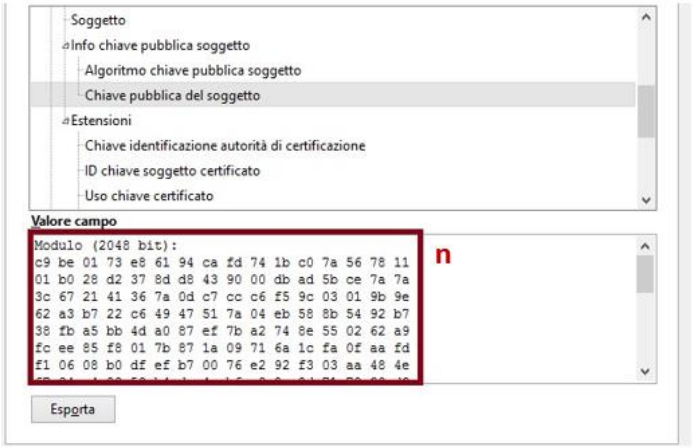

Figure 3 – "n" value (server public key) from TLS Server Certificate Message

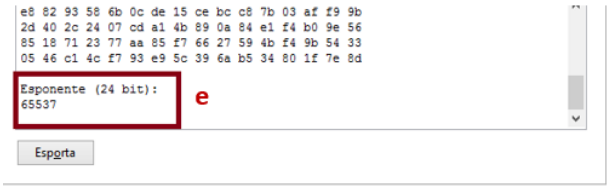

Figure 4 – "e" exponent from TLS Server Certificate Message

Then the server continues the handshake by sending a TLS *Server Key Exchange* Message. From that, the attacker wants to collect the *Server Params* structure (Figure 5).

Figure 5 – Server Params Structure from TLS Server Key Exchange Message

Within the same TLS packet, the server appends a signature that is just applied on the three pieces of information the attacker has harvested earlier:

- *Client Random structure* taken from the Client Hello Message;
- *Server Random structure* taken from the Server Hello Message;
- *Server Params structure* taken from the Server Key Exchange Message;

Figure 6 – RSA Signature from TLS Server Key Exchange Message

The validity of the signature is finally checked. If it is not valid (we say that we are in presence of a *faulty signature*) the attacker can try to retrieve the private key of server through the Lenstra attack of 1996 (3.3).

4.2 Passive Approach

If the approach chosen is passive, it means that the attacker does not directly interact with the target TLS service. Instead, the packets are passively sniffed from the network. The attack is still possible because all the information exchanged that are useful to derive the private key of server are transmitted in clear-text over the network, during the preliminary phase of a TLS handshake, when data encryption and anti-tampering countermeasures are not applied yet.

4.3 How to detect the presence of a faulty signature

The way in which a faulty signature can be detected is really simple but a little laborious and depends on the version of TLS agreed.

Basically, the attacker can't simply generate the digital signature for the three collected structures just as the server does and compare this value with that one extracted from the *Server Key Exchange* message, because he/she does not own the private key of server. However as a client, the attacker knows the plain-text message signed by server and therefore can:

1. Use the public key of server extracted from the TLS *Server Certificate* Message and decrypt the signature from the TLS *Server Key Exchange* Message;
2. Remove the padding from the value got at point 1
3. Compare the value obtained at point 2 with the expected hash of the plain-text message. If there is a mismatch, the signature is faulty.

4.3.1 TLS < 1.2

When the protocol TLS 1.0 or 1.1 is used, the value signed by the server and transmitted as RSA digital signature with the *TLS Key Server Exchange* Message is the concatenation of two hashes (MD5 and SHA1) calculated on *Client Random*, *Server Random* and *Server Params* structs.

After having collected all the TLS messages of interest (from *Client Hello* to *Server Key Exchange*) the attacker calculates offline the expected value for the signature (before to be encrypted) which produces 36 bytes in output (16 bytes for the MD5 hash and 20 bytes for the

SHA1 hash). Then the attacker uses the public key of server extracted from the *TLS Server Certificate* Message to decrypt the signature found in the TLS *Server Key Exchange* message. Afterwards the padding is removed from there (see section 2.1.3.1) to get the raw value originally signed by server. This value is finally compared with that one computed offline. If there is a match, a new TLS handshake is tried (it means the signature is valid). In contrary case the recovery of the private key is attempted (3.3). The entire process is depicted in Figure 7.

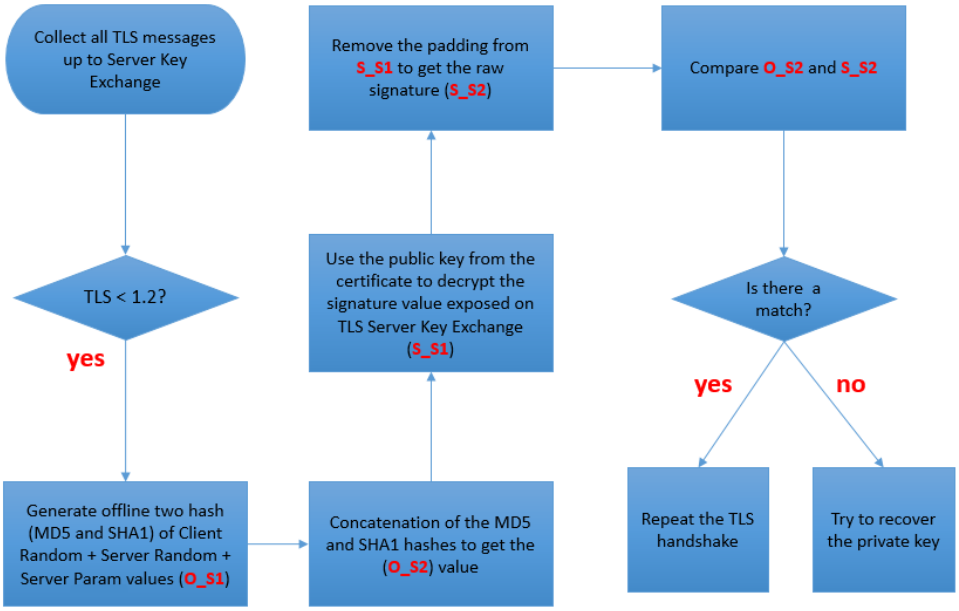

Figure 7 – How to check if a RSA digital signature is faulty (TLS < 1.2)

4.3.2 TLS 1.2

In case the handshake is based on TLS 1.2, the process to determine the presence of a faulty RSA digital signature is quite similar than that one described in the previous paragraph.

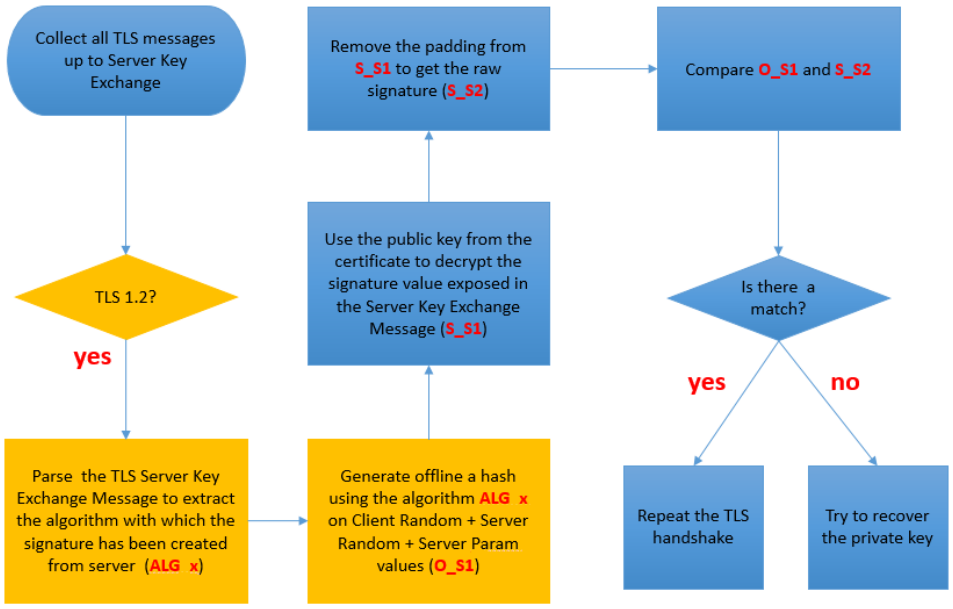

Figure 8 – How to check if a RSA digital signature is faulty (TLS 1.2)

The only difference is that the hashing algorithm the attacker uses to compute offline the hash of the values collected over the network is not static (concatenation of MD5 and SHA1 hashes) but it is specified inside the TLS *Server Key Exchange* Message itself.

So this time the value the attacker have to calculate offline actually must be hashed with the signature algorithm specified inside the TLS *Server Key Exchange* Message. Furthermore he/she must be also aware of the padding scheme implemented (see section 2.1.3.2) which is slightly different compared to that one used for TLS handshake < 1.2. The entire procedure is depicted in Figure 8.

4.4 Other Protocols

TLS is not the only network protocol where Perfect Forward Secrecy cipher suites, RSA and specifically RSA-CRT play an important security role. Another one where is made intensive usage of these crypto standards and algorithms is **IPSEC**, specifically the **IKE** component (Internet Key Exchange) listening on udp ports 500 or 4500.

4.4.1 IPSEC and IKE

Before to establish a protected channel with IPSEC, peers have to authenticate each other and negotiate the so-called "*Security Associations*" (SA). This is the purpose of IKE[7]. For the **Phase 1** (*Authentication* of peers and *Negotiation* of SA) it provides two different modes: *main* and *aggressive*. It has been always believed that only by sniffing the network traffic it is not possible to recover a private key (just as happens for TLS) and an active approach is necessarily requested for doing that.

Among the diverse authentication mechanisms supported both in *main* and *aggressive* mode, IKEv1 supports the "*signature authentication*". It is a kind of authentication based on digital certificates. In *main* mode, what is shown below is the exact sequence of packets exchanged:

IKEv1 Phase 1 Main Mode (Signature Auth)

```
Initiator                              Responder

-----------                            -----------

(1) HDR, SA                      -->

                                 <--   (2) HDR, SA

(3) HDR, KE, Ni                  -->

                                 <--   (4) HDR, KE, Nr

(5) HDR*, IDii, [ CERT, ] SIG_I  -->

                                 <--   (6) HDR*, IDir, [ CERT, ] SIG_R
```

** Indicates payload encryption*

Here are observable two order of problems:

1. The *Initiator* (attacker) is the first sending a Certificate (**CERT**) on packet 5. If the responder (server) does not recognize it, the connection should be discarded. This means the attacker must own a valid certificate.
2. The Signature (**SIG_R**) from server that has to be checked is sent on packet 6, but it is exchanged inside an encrypted channel when it is transmitted. So this means that only a peer that participates actively in the handshake can try to exploit the RSA-CRT vulnerability.

Anyway, these premises do not take in consideration what happens when the server supports the *aggressive* mode.

IKEv1 Phase 1 Aggressive Mode (Signature Auth)

```
Initiator                              Responder

-----------                            -----------

(1) HDR, SA, KE, Ni, IDii       -->

                                <--  (2)HDR, SA, KE, Nr, IDir,

                                                [ CERT, ] SIG_R

(3) HDR, [ CERT, ] SIG_I        -->
```

Basically, the aggressive mode makes the IKEv1 phase 1 more compact. This goes at complete benefit of the attacker. Indeed on packet 2 the server transmits first its Certificate (**CERT**). In addition, with the same packet, the Signature (**SIG_R**) is attached. These exchanges occur in a not encrypted channel. It means that with just two packets, if all the preconditions are met, the attacker can recover the private key of server without being active part in the communication and by simply sniffing the network traffic.

Anyway, it is worth to note that aggressive mode implies not negligible security risk with only minor returns in terms of negotiation speedup. Its usage has not been recommended during these last few years because of security implications especially in combination with other authentication mechanisms, in particular PSK (*Pre-Shared-Key Auth*), where the hash of the password used to connect to the VPN endpoint can be sniffed in plain text over the network.

However, a complete research on this protocol and the impacts the RSA-CRT vulnerability can have on it are out of scope for this whitepaper.

5 AFFECTED PRODUCTS

In order to exploit the RSA-CRT vulnerability an attacker have to find a piece of software linked to a vulnerable crypto library that implements RSA-CRT and that does not verify the correctness of the generated RSA digital signatures. Different crypto libraries, software and hardware products having these features have been found affected in real-life scenarios or declared to be affected by their respective vendors (see Table 1 and Table 2). Florian Weimer of Redhat and his research have given a sensible contribution in that sense.

In some cases it was not possible determine remotely the exact version of software or hardware products. It is worth to note that the list of Table 2 is based on what was found inside the leaked digital certificates and could not be complete or accurate.

crypto library, software or vendor	Version and notes
mbedTLS (formerly PolarSSL)	< 2.1.1, 1.3.13 and 1.2.16: *MBEDTLS_RSA_NO_CRT* can be defined to disable RSA-CRT but this option is off by default)
libgcrypt	< 1.6.3 (equivalent to *CVE-2015-5738*)
Nettle	< 3.1: used by GnuTLS
Java SE	< 5.0u81, 6u91, 7u76, 8u40 (*CVE-2015-0478*)
JRockit	< R28.3.5 (*CVE-2015-0478*)
EMC	RSA BSAFE Micro Edition Suite (MES) 4.0.x and 4.1.x before 4.1.5, RSA BSAFE Crypto-C Micro Edition (CCME) 4.0.x and 4.1.x before 4.1.3, RSA BSAFE Crypto-J before 6.2.1, RSA BSAFE SSL-J before 6.2.1, and RSA BSAFE SSL-C before 2.8.9 (*CVE-2016-0887*)
OpenSSL	<= 0.9.7 and *potentially* between 1.0.2 and 1.0.2d because of *CVE-2015-3193* only on x86_64 architectures + custom versions
Go	< 1.6.2
Cryptlib	up to latest 3.4.3 (*CRYPT_OPTION_MISC_SIDECHANNELPROTECTION* would prevent the attack but it is set to false by default)
wolfSSL (formerly CyaSSL)	< 3.6.8 (*CVE-2015-7744*)
Libtomcrypt	< 2.00
Eldos SecureBlackbox	< 13.0.280 and 14.0.281
MatrixSSL	< 3.8.3
Openswan	up to latest version 2.6.47 vulnerable when not compiled with NSS

Table 1 – RSA-CRT vulnerability: affected crypto libraries, software and vendors

By analyzing the affected solutions, it is observable that they are fundamentally embedded devices of some kind, with a high prevalence for devices providing network connectivity or IT security features (firewalls, routers, SSL accelerators, VPN concentrators, etc…). Another category found as affected is represented by consumer or SOHO devices such as network surveillance cameras and printers.

5.1 THE FIX

A few of crypto libraries allow users to disable RSA-CRT (for example the configuration option *MBEDTLS_RSA_NO_CRT* supported on mbedTLS) but this is not convenient due to performance issues. Most vendors have issued a patch between the end of 2015 and the next months of 2016 to address this problem. See Table 1 and Table 2 to determine whether a vulnerable products or solutions has been used inside your company or office, and look for a patch from the vendor's website.

Hardware	Version and notes (when known)
FORTINET	Series 300 / FortiGate < 5.0.13 / 5.2.6 / 5.4.0
Dell	(SonicWALL< SonicOS 6.1.1.12)
F5	(Traffix SDC)
ZTE ZXSEC	Firewall (models US2640B, US2630B, US2620B)
LANCOM	wireless devices (version 8.84) <- apparently silently patched in 2014
D-Link-DCS-933L	Surveillance camera
HILLSTONE NETWORKS	(SG-6000 Firewall)
CITRIX	
ZYXEL	
NORTEL	
QNO	
Viprinet	

Table 2 – RSA-CRT vulnerability: Hardware vendors / products affected

6 TOOLS

In order to demonstrate the exploitability of the RSA-CRT flaw, two different proof-of-concepts have been developed. They are described in the subsequent paragraphs.

6.1 HIGH VOLTAGE

High Voltage implements the *active approach* method (see 4.1) to exploit the RSA-CRT vulnerability. It is a C-language application that requires as only dependency the *openssl-devel* package on CentOS/Redhat (*libssl-dev* on Ubuntu, Kali Linux and Debian).

```
Usage: hv -h <host_name> -p <port> [-t] [-v] [-f] [-c num]
-h: host or ip address to connect to
-p: tcp port to connect to (1-65535)
-t: use TLS_v1.2 (default: no)
-v: enable verbose mode (default: no)
-c: number of childs (default: 1)
-f: do not stop when a signature fault is detected (default: no)
```

The PoC requires two mandatory options from command line: the hostname (`-h`) and the TCP port to attack (`-p`). By default the connection is established with the TLSv1.0 protocol. The option `-t` overrides this setting by establishing connections with the TLSv1.2 protocol. The verbose mode is disabled by default but can be enabled with the option `-v`.

By default only one process establishing TLS connections is forked. The `-c` option allows to specify a higher number of forked childs. Every child will start to establish TLS connections independently by the others.

Always by default, once a faulty digital signature is detected and the recovery of the server private key is attempted, the program stops. The `-f` option prevents this behavior and lets the program run forever until it is interrupted by the user.

Example:

```
$ hv -h 192.168.1.1 -p 443 -t -c 5
```

When a TLS session is successfully exploited, a dedicated folder with a random name inside the "`results`" directory is created containing the private/public key and certificate of the server. The file "`host.txt`" contains the server's IP address. If instead a file called "`incomplete.txt`" is found, it is the signal that the public key calculated is not corresponding to the private key recovered. When this happens it means that an error has occurred during the RSA-CRT attack, for example because both of **s1** and **s2** (see 3.3) were wrong.

6.2 PICIOLLA

Piciolla implements the *passive approach* method (see 4.2) to exploit the RSA-CRT vulnerability. It consists of two components: a bash script and a C-language application. The only dependencies are the *openssl-devel* package on CentOS/Redhat (*libssl-dev* on Ubuntu, Kali Linux and Debian) and the *tcpflow* package. To compile the C file run:

```
$ gcc piciolla.c -o piciolla -lcrypto -lssl
```

Piciolla works on network dump files. Firstly *tcpflow* has to be launched uncompress the single TCP streams from one or more PCAP files:

```
$ mkdir streams
$ cp ../pcaps/bigfile.pcap ./streams
$ cd streams
$ tcpflow -r bigfile.pcap
$ ls -al
192.168.002.144.59105-172.016.032.001.443
172.016.032.001.443-192.168.002.144.59105
192.168.002.144.31337-010.010.172.250.995
010.010.172.250.995-192.168.002.144.31337
[...]
```

Afterwards "`piciolla.sh`" is executed from the command prompt, providing as an input the directory containing the extracted TCP streams.

```
$ ./piciolla.sh ./streams
```

Then each file is analyzed and whether it appears to be a TLS data stream, the C application "`piciolla`" is invoked via the bash script.

At the end of the analysis, one or more of the following directories could be created:

- `not_tls`: here are copied all the TCP stream files that are not TLS sessions.
- `incomplete`: here are copied all the incomplete TCP stream files (for example when only the client payload is detected but there is no server payload or vice-versa).
- `tls_not_affected`: here are copied all the TCP stream files that are TLS sessions for which nothing has been found.
- `results`: if this directory exists it means one or more private keys have been recovered. For each session successfully exploited a dedicated folder with a random name is created containing the private/public key and certificate of the server, in addition to the corresponding TCP stream files that allow to identify the vulnerable entity and the payloads exchanged.

6.3 Prepare a Test Environment

As one of the three preconditions (see 2.1) needed for the exploitation of the RSA-CRT vulnerability is unpredictable (it requires the generation of a faulty signature, namely an event that cannot be controlled by the attacker) the faster way to set up a test environment is actually to patch the openssl crypto library in order to deliberately inject RSA digital signatures calculated in a wrong way.

The author of this whitepaper has adopted the procedure reported below in order to create a working test environment to use as testbed for *High Voltage* and *Piciolla*. On the machine that will play the role of vulnerable target:

1. Install the *openssl* and *openssl-libs* packages.

2. Download the source codes of openssl from https://www.openssl.org/. The author used the version *1.0.1l* but any other recent version should work;

3. Apply the Florian Weimer's openssl patch[3] specified on page 13 "*Figure 1: Patch to inject a RSA-CRT-related fault into OpenSSL for testing purposes*";

4. Compile the openssl source codes being sure to disable the cryptographic hardware support. Run from command line `./config --openssldir=/usr/local/ssl4 -no-hw –no-engine shared; make; make install`

Now the target machine has two installations of *openssl*. The first one from point 1 which is good and perfectly working. The other one inside the folder "`/usr/local/ssl4`" that instead generates faulty RSA signatures. With the good version of openssl let us create a new private key and digital certificate:

```
$ openssl req –x509 –newkey rsa:4096 –keyout newone.key –out newone.pem
-days 365 -nodes
```

Instead with the vulnerable version of *openssl* let us run a TLS service by referencing the private key and certificate just created with the previous step:

```
$ cd /usr/local/ssl4/bin

$ ./openssl s_server –www –cert /path/to/newone.pem –key
/path/to/newone.key –accept 443

$ netstat –tupan | grep 443
tcp    0    0 0.0.0.0:443     0.0.0.0:*     LISTEN     6543/./openssl
```

From this moment on, *High Voltage* or *piciolla* can be used to recover the private key of the vulnerable server from another network host, for example by executing:

```
$ ./hv –h server_IP –p 443
```

7 RESOURCES

[**1**] *Memo on RSA signature generation in the presence of faults* – Arjen Lenstra (https://infoscience.epfl.ch/record/164524/files/nscan20.PDF)

[**2**] *Attack on Private Signature Keys of the OpenPGP format, PGP TM programs and other applications compatible with OpenPG* – Vlastimil Klíma and Tomáš Rosa (http://eprint.iacr.org/2002/076.pdf)

[**3**] *Factoring RSA Keys With TLS Perfect Forward Secrecy* – Florian Weimer (https://people.redhat.com/~fweimer/rsa-crt-leaks.pdf)

[**4**] *Modular Multiplicative Inverse* – Wikipedia (https://en.wikipedia.org/wiki/Modular_multiplicative_inverse)

[**5**] Greatest Common Divisor – Wikipedia (https://en.wikipedia.org/wiki/Greatest_common_divisor)

[**6**] *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1* – J. Jonsson, B. Kaliski (http://www.ietf.org/rfc/rfc3447.txt)

[**7**] *The Internet Key Exchange* – D. Harkins, D. Carrel (https://tools.ietf.org/html/rfc2409)