Factoring RSA Keys With TLS Perfect Forward Secrecy

Florian Weimer
Red Hat Product Security

September 2015

Abstract

This report describes the successful factorization of RSA moduli, by connecting to faulty TLS servers which enable forward secrecy and which use an insufficiently hardened RSA-CRT implementation. The history of this particular RSA-CRT implementation defect is discussed, and the current state of countermeasures is reviewed. Some familiarity with the RSA cryptosystem and the Transport Layer Security protocol suite is assumed.

1 The RSA-CRT optimization

The RSA cryptosystem works in a ring $\mathbf{Z}/n\mathbf{Z}$. Knowledge of the private key $pq = n$ allows one to construct a ring isomorphism between $\mathbf{Z}/n\mathbf{Z}$ and $\mathbf{Z}/p\mathbf{Z} \oplus \mathbf{Z}/q\mathbf{Z}$.

In practice, both p and q are about the size of $\sqrt{n}$. The cost of RSA operations grows quite a bit faster than $O\big((\log n)^2\big)$, for typical implementations. Therefore, doing the computations in $\mathbf{Z}/p\mathbf{Z}$ and $\mathbf{Z}/q\mathbf{Z}$ separately is more efficient, even after taking into account the cost of separating the input and combining the result. (Not just are the moduli much shorter, but the decryption exponent d can be reduced as well, and many of the values needed can be pre-computed during key generation.) This optimization is usually called the Chinese Remainder Theorem (CRT) optimization.

1.1 Arjen Lenstra's CRT attack

A widely-cited memo by Arjen Lenstra [1] points out a problem with this optimization in the sense that it can lead to an additional side-channel attack. We summarize Lenstra's observation below, using slightly different language.

With the CRT optimization, the bulk of the computation happens in the ring $\mathbf{Z}/p\mathbf{Z} \oplus \mathbf{Z}/q\mathbf{Z}$. If a fault happens during the computation of a RSA signature in exactly one of the components, say $\mathbf{Z}/p\mathbf{Z}$, and the attacker gets hold of that faulty signature y, and the attacker knows the original value x which is being signed, then

$$1 < \gcd(y^e - x, n) < n \tag{1}$$

over the integers, where e is the public RSA exponent and a suitable integer representative for $y^e \pmod n$ has been chosen. This means that the greatest common divisor reveals a prime factor of n.

Here is why this works, assuming such a fault has occurred: The ring isomorphism $\varphi : \mathbf{Z}/n\mathbf{Z} \to \mathbf{Z}/p\mathbf{Z} \oplus \mathbf{Z}/q\mathbf{Z}$ behind the CRT optimization can be realized as the reduction modulo p and q in the respective components. Assume $\varphi(y) = (y_p, y_q)$ for a signature y which is faulty in the component y_p, then $\varphi(y^e) = (y_p^e, y_q^e)$: the fault is still restricted to the first component. Writing the original value-to-be-signed x in the form $\varphi(x) = (x_p^e, x_q^e)$, we see that

$$\varphi(y^e - x) = (a, 0)$$

for some $a \in \mathbf{Z}/p\mathbf{Z}$. But this means that any representative of $y^e - x$ modulo n is a multiple of q. If the representative is less than n (which is always possible to arrange), the greatest common divisor of $y^e - x$ with n is q, revealing the factorization.

1.2 Faults enabling Lenstra's attack

The nature of the fault which affects the CRT optimization can take various forms:

1. The arbitrary precision integer library used by the RSA implementation could have a defect

that causes it to produce incorrect results (see CVE-2014-3570 in [2], although the practical impact of this particular defect is minimal, as explained in the security advisory).

2. There could be a race condition involving data access from multiple threads in an unsynchronized fashion.

3. The arithmetic unit in the CPU could produce incorrect results, either in a deterministic fashion for some inputs (as described in [3]), or randomly due to some environmental conditions (perhaps after overheating, as a result of its extensive use during an RSA operation).

4. Critical parts of the private key could have been corrupted after all the integrity checks (if any) have been performed (see [4]), causing all future signatures to reveal the private key.

5. CPU caches, other caches, or main memory could be defective and introduce bit errors into intermediate results.

If the target is a smartcard (or other device which is supposed to prevent even its owner from accessing the private key material), other forms of faults may be relevant, typically deliberately introduced by someone who has physical access to the device. We are eventually interested in faults in TLS implementations where we do not have physical access, which is why we did not consider these additional faults further.

1.3 Countermeasures

The obvious countermeasure is to verify the correctness of the signature y by checking $y^e \equiv x \pmod{n}$, using arithmetic in $\mathbf{Z}/n\mathbf{Z}$ (without the CRT optimization). This is already alluded to in [1]. This verification step is relatively cheap compared to the signing operation itself because the public encryption exponent is typically $e = 65537$, which is much shorter than the private exponent (even with the CRT optimization in place). The difference is even more pronounced with the other popular choice, $e = 3$,

In response to Klima and Rosa's attack on the secret key storage in some implementations of OpenPGP [4], several RSA-CRT implementations (notably GnuPG and OpenSSL) introduced hardening against Lenstra's RSA-CRT attack (because it was used in the RSA part of the Klima/Rosa attack). It has been conjectured that this kind of hardening is no longer relevant (see [5] for a recent example), at least not in the context of non-smartcard implementations, so we examined the state of current implementations.

Implementation	Verification
cryptlib 3.4.2	disabled by default
GnuPG 1.4.18	yes
GNUTLS	see libgcrypt and Nettle
Go 1.4.1	no
libgcrypt 1.6.2	no
Nettle 3.0.0	no
NSS	yes
ocaml-nocrypto 0.5.1	no
OpenJDK 8	yes (see text)
OpenSSL 1.0.1l	yes (see section 1.3.1)
OpenSwan 2.6.44	no
PolarSSL 1.3.9	no

Table 1: RSA-CRT and signature verification

Table 1 shows the result of a source code review of the major free-software RSA implementations. The versions reviewed were the most recent stable versions available at the time of review. All of them implemented the RSA-CRT optimization. The table reflects the default upstream implementation only and does not cover alternative RSA implementations which may have been installed using a plug-in mechanism (which is an optional feature offered by several of the cryptographic libraries surveyed).

Cryptlib supports checking after CRT-based RSA signing operations, but the configuration option which controls this functionality is disabled by default. The cryptlib author reports in [7] that "enabling this option" for general protection against side-channel attacks "will slow down all private-key operations by up to 10%". The post-signing check is deliberately included among the general protection mechanisms, so it is disabled by default as well.

NSS simply reports an error to the caller if a signature fails to verify immediately after its computation. We think this is the best way to deal with this situation.

OpenJDK was fixed prior to publication, in re-

sponse to this research, under the vulnerability identifier CVE-2015-0478 [6].

1.3.1 Countermeasures in OpenSSL

The OpenSSL case is slightly complicated because this library performs a verification and re-computes the signature without the CRT optimization if a fault has been detected. We believe that this poses no immediate danger—Lenstra's attack on RSA-CRT certainly does not work even if a second fault occurs because of the unoptimized second computation. Depending on the nature of the faults, this could conceivably still reveal the key over time, but several thousand faulty signatures are needed for current key sizes [1, 8], each one preceded by an RSA-CRT fault. As shown in [8], it is possible to construct a faulty CPU which indeed leaks keys in this way, which is why there are some lingering concerns about the reliability of the OpenSSL hardening.

There may also be OpenSSL variants in circulation which omit RSA-CRT verification altogether, either because they have been patched, or a completely different RSA implementation has been swapped in, using the OpenSSL plug-in mechanism. The comments in the preceding paragraph apply only to current (1.0.1l) upstream sources and the `rsa_eay.c` implementation.

1.4 RSA blinding

Since Lenstra has described his attack, RSA implementations have started to follow additional recommendations, particularly as a form of hardening against timing attacks. One common hardening measure is RSA blinding [9], which multiplies the value-to-be-signed with a random number before computing the signature and compensates for this additional factor afterwards (details vary). But no matter how RSA blinding is actually implemented, it will not propagate a computation error which affects only one CRT component to the other component (again, this is a consequence of the ring isomorphism φ), and RSA blinding does not stop Lenstra's attack.

2 RSA use in TLS

Traditionally, SSL 3.0 and all TLS versions up to 1.2 ([10], the current version) require that the server performs an RSA decryption of a client-supplied challenge as part of the handshake, thus authenticating the server to the client. But this approach does not work if the server uses an X.509 certificate which does not embed an encryption-capable public key. (Historically, such keys were DSA keys.) To use such keys, SSL 3.0 provides a different authentication mechanism: the server signs a message containing, among other things, a client-supplied random string (to ensure freshness) and parameters for establishing a session key using an instance of the Diffie-Hellman key agreement scheme.

It was then noticed that it is possible to pretend that RSA keys do not support encryption, and use their signing capabilities only, just as one would do with a DSA key. This protocol tweak has the property that as long as there is no practical offline attack on the Diffie-Hellman variant being used, it is not possible to recover the plaintext of TLS sessions even if the server private key is leaked after the fact, which is why this protocol variant goes by the somewhat pompous name *Perfect Forward Secrecy*, or just forward secrecy.

This protocol change opens up RSA keys to Lenstra's attack, assuming that the underlying RSA-CRT implementation lacks appropriate checking. (The Diffie-Hellman variant used for session key negotiation does not matter here.)

Recall that Lenstra's attack needs knowledge of the value x which has been signed, in addition to the corrupted signature y. This means that a deterministic padding scheme is required for the attack to work. Curiously, SSL 3.0 and all versions of TLS up to 1.2 use a variant of the fully deterministic PKCS #1 version 1.5 RSA signature padding [11]. Versions up to TLS 1.1 concatenate the MD5 and SHA-1 hash of the message and do not include the DER-encoded OID of the digest algorithm. TLS 1.2 added digest algorithm negotiation, switched to a standard digest algorithm (no more concatenation of two digests), and includes the OID in the padding.

But the key point remains: These schemes are all deterministic. There have been discussions to use randomized padding schemes for RSA signatures in

TLS (predominantly RSASSA-PSS [12]), and the new negotiation capabilities in TLS 1.2 would have provided a backwards-compatible way to introduce them. It is surprising that this has not happened.

The actual key recovery attack can be described in very simple terms: establish a TLS session, negotiate forward secrecy, and watch out for a miscomputed signature. If a signature mismatch is detected, attempt Lenstra's attack to recover the RSA private key. Figure 1 shows the patch we used to experimentally validate that the approach works. The patch introduces the fault (the time dependency will be used in section 4) and removes OpenSSL's countermeasure against faults during the RSA-CRT computation.

3 The experiment

As explained above, implementers no longer consider hardening RSA-CRT signing operations against faults a necessity. The increased deployment of TLS with forward secrecy provides a way to test if this decision is reasonable: 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.

3.1 Setup

The crawler ran in two instances:

- One instance used the OpenSSL 1.0.1k-3 Debian package, recompiled to add back SSL 3.0 support (to reduce the amount of spurious handshake failures). It ran on a Debian 8 operating system (amd64 architecture).
- Another instance ran on Red Hat Enterprise Linux 7.1, using the OpenSSL packages with versions `1.0.1e-42.el7_1.6` and later (which reject weak Diffie-Hellman primes).

An OpenSSL function, `SSL_set_msg_callback`, is used to install a callback function which OpenSSL invokes after processing handshake messages. This enables the crawler to run with an unmodified OpenSSL library (and will make it possible to rerun the experiment in the future, potentially with a different OpenSSL version). The callback function is invoked even in the presence of handshake failures.

We chose to include cipher suites without forward secrecy in the client hello message, to make it somewhat less obvious to server operators what we were after in this experiment. The crawler advertised TLS 1.2 support, but would retry connections with TLS 1.0 and SSL 3.0 in response to certain handshake failures.

Three different forms of target selection were used:

1. *Domain-based target selection.* The domain list consisted of a manually curated list of critical domains, plus Hubert Kario's domain list used in his TLS scans [13]. We also included domain names from a subset of the certificates documented by the Rapid7 Sonar SSL scans [14]. Later on, we added host names extracted from certificate transparency logs [15], too.

   The total set of targets had 25 million host names (and IP addresses), and we saw handshakes with forward secrecy for 8 million of these targets. We conducted about 1.7 billion TLS handshakes in total, of which 1.4 billion involved forward secrecy.

2. *IP-based target selection.* We used IP addresses extracted from the same Rapid7 Sonar SSL scans. We also experimented with the University of Michigan ZMap data [16], but the published data was not current at the time we noticed its availability.[1]

[1] Originally, the published ZMap/zgrab data did not include the client hello and the complete `ServerKeyExchange` message, so the TLS traces do not contain sufficient information to attempt key recovery. The ZMap team has since switched to publishing more complete data.

The consolidated target set contained 32 million IP addresses, of which 10 million responded with a forward secrecy handshake. Here, the number of TLS handshakes was around 3.8 billion, of which 3.4 billion had forward secrecy enabled.

3. *Certificate-based target selection.* We selected targets based on X.509 certificate properties, mainly vendor strings. There were 26,000 targets in that category, with about 40 million forward secrecy handshakes combined, yielding a high number of such handshakes per target.

In both cases, we switched, after a few probing runs over the entire target set, to the subset of targets which performed a TLS handshake which involved a `ServerKeyExchange` message (because that is the only way a key leak could occur).

We assumed that the first option would yield better results: Port scanning, without domain names, should no longer be fully effective because some TLS servers require a valid Server Name Indication before they complete the TLS handshake. But this turned out not to be the case, and the second approach was far more effective. As the results below show, most of the affected devices were not in the browser PKI. They do not even contain a matching host name in their X.509 certificates, and do not show up in the certificate transparency logs.

In addition, for the domain-based approach, DNS resolution turned out to be a major bottleneck. We used the PowerDNS recursor on the same machine, configured to cache records aggressively (mostly by increasing the cache size and ignoring server-provided TTL values). However, this caching was only effective while we were using a smaller set of host names, and we eventually had to cache name resolution results in an SQLite database, completely bypassing TTL values.

3.2 Key recovery implementation

If OpenSSL reports any handshake failure, the RSA signature is extracted from the `ServerKeyExchange` handshake message using a custom-written partial TLS parser. A key recovery based on equation (1) is attempted: y is the extracted signature, e and n are extracted from the X.509 certificate sent by the server, and the expected signature value x is formed by applying PKCS#1 version 1.5 padding to the appropriate digest value, as required by the TLS protocol.

The dominating operation in equation (1) is the public RSA operation, y^e. Therefore, the key recovery computation is slightly more costly than verifying an RSA signature.

Key recovery can fail for various reasons:

1. If the signature is not actually faulty, then $y^e \equiv x \pmod{n}$, and the handshake failed for other reasons.

2. If the signature has faults in both CRT components, the greatest common divisor $\gcd(y^e - x, n)$ will be 1, and not reveal a factorization.

   (a) The public key in the X.509 certificate and the private RSA key used to create the signature do not match: The private RSA key is internally consistent, and no fault occurs during signature computation. The signature is just for the wrong public key, and cannot be verified against the public key in the X.509 certificate.

   (b) Multiple faults occur during a RSA-CRT computation, affecting all CRT components and not just one.

   (c) A fault occurred, but the RSA-CRT optimization was not used to compute the signature as it is disclosed to the client. (This case includes the OpenSSL fallback scenario, discussed in section 1.3.1 and [8].)

   (d) The `ServerKeyExchange` message was corrupted after the signature was computed (perhaps during transmission over the Internet).

   (e) The signature was computed correctly, using the expected key, but over the wrong digest. This can happen if the digest was miscomputed on the server side, or earlier handshake messages were corrupted so that the client computes the wrong expected digest.

   Except in the last case, y^e will not have valid PKCS#1 version 1.5 padding.

We discuss key recovery failures in more detail in section 3.4.

3.3 Results

Table 2 summarizes the results of the crawler. "Vendor" refers to the putative device maker; see below for a description how devices were attributed to specific vendors.

The counts in the "Keys" column is based on the number of distinct RSA moduli (and not full X.509 certificates, or device or IP address counts).

The "PKI" column indicates whether the public key was signed by a certificate authority which is part of the browser PKI; "expired" means the certificate was expired at time of factorization.

The "Rate" column indicates how often the key leaked; most key leaks were seen multiple times, but the rate varied from one-time leaks ("very low"), to the occasional odd handshake ("low") to the majority of handshakes ("high"; at this point, faults *prevent* leaks by happening in both CRT components at the same time), or even all handshakes ("always"). The location in the "Geo" column was determined based on WHOIS data.

3.3.1 Discussion of individual key leaks

The two keys leaked first were used by the same organization, on devices with neighboring IP addresses. An investigation with the help of the certificate authority and the system administrator showed that the keys were stored in the same hardware accelerator card of a TLS-terminating load balancer, a Citrix Netscaler device. The load balancer was quite old and was already scheduled for replacement. This replacement happened after we contacted the system administrator, and the certificates were reissued.

The next key leak we observed was from a device made by Hillstone Networks. The first leaked key of this kind was picked up by accident during our domain-based crawler runs. The certificate was self-signed, and the subject distinguished name in the X.509 certificate contained the strings "`Hillstone Networks`" and "`SG-6000`". Further probing based on this vendor name identified more than 200 additional public keys whose private key leaked. Devices with the "`DCFW-1800`" string in their X.509 certificates are from them same manufacturer and are included in these counts. Hillstone Networks as since release a firmware update [17] to correct this issue.

Vendor	Keys	Geo	PKI	Rate
Citrix	2	DE	yes	medium
Hillstone Networks	231	CN	no	low
	1	CZ	no	low
	1	PL	no	low
	1	TH	no	low
	1	US	no	low
Alteon/ Nortel	1	US	expired	high
	1	US	no	high
Viprinet	1	NL	no	always
QNO	2	CN	no	medium
	1	TW	no	medium
ZyXEL	4	AT	no	low
	1	CH	no	low
	1	DE	no	low
	2	DK	no	low
	1	FR	no	low
	7	IE	no	low
	1	IT	no	low
	2	NL	no	low
	1	SE	no	low
	5	UK	no	low
	1	US	no	low
BEJY	1	DE	yes	low (?)
Fortinet	1	US	no	very low
	1	IN	no	very low

Table 2: Crawler results

The key leaks attributed to an Alteon/Nortel device occured from two IP addresses on the same subnet, and because of that proximity and the similarity in behavior, we assumed it was a single device. Only one of the certificates embedded the string "`Alteon/Nortel Generated Certificate`", on which the attribution is based. For both IP addresses failure rate was so high that the TLS implementation was unable to complete the majority of handshakes successfully. The validity periods of the the X.509 certificates suggest that they were generated many years ago. If the key is still loaded into the same device on which it was generated, these two keys were leaked by an old piece of equipment, like those from the Citrix device.

In the case of the Viprinet device, the key was apparently corrupted in memory or in persistent storage because the RSA signature in *every* `ServerKeyExchange` message generated by the device exposes the RSA private key. (This is similar to what happens in case of the Klima/Rosa attack [4].) See section 3.4.3 for another defect related to this vendor.

The devices made by QNO Technology were first identified using the second probing approach (based on IP addresses), followed by further tests targeting a subset of addresses based on X.509 certificate properties (on which the attribution is based).

The ZyXEL devices we observed were discovered in the same way. The Common Name attribute in their X.509 certificates started with "`zw110`" or "`nxc2500`". ZyXEL confirmed [18] that they had already addressed this issue in a firmware update in March 2013, but as a functionality defect and not as a security vulnerability.

BEJY is a proprietary, custom Java implementation of RSA, TLS, and HTTP/HTTPS, which is not based on the usual Java cryptographic frameworks (JCE and JSSE). The server operator fixed the code rather quickly, and we only saw a single key leak. This particular code base does not appear to be widely used, although JAR files with the server code can be downloaded for free.

The Fortinet devices identified themselves with a Common Name starting with "`FG300B`". We observed just two key leaks from two different devices, in over 90 million TLS handshakes with forward secrecy, across many different devices with similar certificates. Clearly, the TLS implementation lacked RSA-CRT hardening, but beyond that, it was unclear what was happening.

3.3.2 A note on interpreting the results

Various versions of the crawler ran over a time period of nine months. As a result, lots of handshakes were analyzed, but at 200 handshakes per second on average, the number of handshakes is still just a negligible fraction of all TLS handshakes on the Internet during that time period. Very close to publication of this report, we picked up one case with an extremely low key leak rate (Fortinet) by sheer luck, and other such cases may exist as well.

In all likelihood, the distribution between vendors and countries shown in table 2 does not accurately reflect the actual distribution of devices which leak RSA keys in this way. In order for the crawler to pick up the device at all, a few requirements need to be met:

1. The device must implement TLS.
2. It must offer a TLS service on port 443/TCP.
3. This TLS service must be accessible to the general Internet.

The third item is not just a property of the device, but also depends on how the device is deployed in the field. An Internet service provider that installs customer premises equipment with a world-accessible HTTPS management interface on port 443/TCP can easily shift these numbers, and some of the observations in table 2 may be the result of that.

3.3.3 A common root cause?

Citrix [19], Hillstone Networks [20], and ZyXEL [18] confirmed that they use Cavium as a hardware supplier. Documentation [21] from Radware (the company that acquired the Alteon/Nortel product lines) suggests that they use this that they use this supplier as well, but we could not confirm that the referenced documentation actually applied to the device we observed.

In case of Hillstone Networks and ZyXEL, Cavium supplied custom versions of OpenSSL, which are not available to the general public in source

code form. Both vendors disabled the hardware acceleration in order to prevent the key leak and to re-enable the RSA-CRT hardening present in the OpenSSL upstream version.

Cavium kindly provided us with the following statement [22]:

> Cavium has issued a patch and notified all customers (CVE-2015-5738) that are using older SDK 2.x Cavium Cryptographic Software under Linux on older OCTEON II CN6xxx Hardware with details of the vulnerability. OCTEON II running Simple Exec (SE-S) applications and OCTEON III CN7xxx processors are not affected.

The relationship of QNO or Fortinet with this hardware supplier is unknown. Viprinet and BEJY do not use this supplier.

3.4 Other faulty signatures

In addition to faulty signatures which resulted in recovery of the private key, we saw faulty signatures were key recovery failed. As explained in section 3.2, there are several potential causes. We attempted to attribute these failures to specific causes.

3.4.1 Public/private key mismatch

In this scenario, the server public key, as contained in its X.509 certificate sent during the TLS handshake, does not match the private key which is used to compute the signature, but apart from that, the signature is computed correctly.

We used the following approach to recover the public key n from two faulty signatures (x, y), (x', y'), where x, x' are the expected padded values over which the signature is computed, and y, y' are the server-provided signatures. Assuming the RSA signatures have been computed correctly using the same private key, $n \mid y^e - x$ and $n \mid y'^e - x'$, so that

$$n \mid \gcd(y^e - x, y'^e - x'), \tag{2}$$

computed over $\mathbf{Z}$. (If our assumption about correct computation was incorrect, the greatest common divisor will likely be 1 or a small integer, see below.)

It is still necessary to guess the public encryption exponent e. We tried the values 3, 17, and 65537. The latter makes the computation of the greatest common divisor in (2) somewhat cumbersome, but it is well within reach of modern systems.

$y^e - x$ and $y'^e - x'$ should be (pseudo-)random multiples of n (otherwise, this might point to a flaw in the RSA cryptosystem), which means that the respective cofactors of n are relatively prime with probability $6\pi^{-2}$. (This result goes back to Dirichlet, see exercise 4.5.2(10) in [23].) Consequently, there is a good chance that (2) is, in fact, an equality. The greatest common divisor with additional pairs can be computed to make sure that the inferred value of n is indeed the right one.

We did not fully automate this process because our main goal here was to weed out known causes of miscomputed signatures. The handshake failure due to this misconfiguration is harmless because it does not leak the private key, and so it does not warrant further consideration as part of this report.

3.4.2 Accidental factors of a corrupted public key

We observed one rather peculiar factorization of a RSA modulus, involving factor 23. What happened was that the public key in the X.509 certificate was corrupted in some (there was a bit flip, according to the server operator), and equation (1) accidentally revealed the factor 23. The corrupted modulus had other small factors, too, and a large composite factor with an unknown factorization.

Even if a complete factorization of the corrupted modulus had leaked, this would not have been useful for attacking the correct modulus. The server had RSA-CRT hardening, so a key leak was not possible, but the operator quickly retired the defective server and replaced the X.509 certificate as a precaution.

3.4.3 No-op RSA

Several deployed VPN devices whose certificates suggest they were manufactured by Viprinet consistently generate `ServerKeyExchange` messages where the RSA signature is unencrypted. That is, $x = y$ and $e = d = 1$ for the computation, although the X.509 certificate says that the public exponent is 65537. This defect leads to an interoperability

failure with properly behaving clients, but it does not leak the server private key.

If this is not the result of memory implementation, but of a defect in the TLS implementation which uses this no-op RSA variant, and this TLS implementation is used on both ends, TLS will not provide any cryptographic protection.

3.4.4 Zero RSA

Some servers occasionally or consistently produce `ServerKeyExchange` messages which contain RSA signatures which are zero. Encoding of the number zero varied. In some cases, zero or one bytes were transmitted. Sometimes the length of the signature matched the size of the RSA modulus. The latter suggests that the server implementation may have omitted the copy of the computed signature. This could happen if RSA-CRT hardening detects a faulty signature, does not write the result to a caller-provided buffer, returns an error. If the caller is defective and ignores RSA signature computation failures, it may then proceed to send out the unwritten buffer.

4 Browser behavior

This section presents a curious observation which is only peripherally related to the main results.

The timing dependency in our corrupted OpenSSL version (see figure 1) gives clients who retry on handshake error a second chance. Most web browsers perform such second tries to increase compatibility, reducing security at the same time ("insecure protocol downgrade"). To see what this means in the presence of corrupted forward secrecy signatures, we checked several web browsers against the example web server in OpenSSL ("`openssl s_server -www`").

The results are shown in table 3. Most of the tested browsers are binaries provided by Debian. The Firefox binaries are official builds from Mozilla. Neither the Iceweasel nor the Firefox binaries use the system NSS library. The test system was a Debian 7.8 amd64 installation, except for the Internet Explorer test, where a German version of Windows 8.1 (updated to the January 2015 version) was used.

Most of the tested browsers hide server problems in the TLS signing operation. Between version 31.0 and 32.0, Firefox switched from an unconditional retry on this particular error to reporting a failure (with a "Try Again" button, though). However, the error is only reported on subsequent connection attempts during the same browser session. Even in version 32.0 (and later versions up to 37.0), Firefox silently retries a second time if the error happens during the initial connection attempt. (We believe this behavior is not related to TLS session resumption, but caused by a cache introduced to reduce protocol downgrades due to network glitches.) Beginning with version 38.0, Firefox always reports handshake failures due to faulty `ServerKeyExchange` signatures (with a "Try Again" button).

Browser	Behavior
Chromium `37.0.2062.120-1~deb7u1`	silent retry
Iceweasel `31.4.0esr-1~deb7u1`	silent retry
Internet Explorer `11.0.9600.17498/11.0.15`	limited retry
Firefox `31.0`	silent retry
Firefox `32.0`	limited retry
⋮	⋮
Firefox `37.0`	limited retry
Firefox `38.0`	failure
w3m `0.5.3-8`	failure

Table 3: Browser behavior in response to server signature failures

The tested version of Internet Explorer shows the reverse retry behavior: Initial connections with a faulty signature fail, but if an unsuccessful session resumption attempt leads to a full handshake with a faulty signature, Internet Explorer silently retries the connection, covering up the signing error.

We need to stress here that all these behavioral variants in browsers are not themselves vulnerabilities. But these silent retries help to hide the fact that the server signing operation is vulnerable to Lenstra's attack.

5 Impact on other protocols

The RSA cryptosystem is used in other protocols besides TLS, and deterministic padding schemes such as PKCS #1 version 1.5 are widely used. How-

ever, some protocols make key leaks from miscomputed RSA signatures invisible to passive observers.

5.1 IPsec

The majority of the leaked keys affected devices whose X.509 certificates suggested they are used as VPN gateways. Some operating modes for IPsec-based VPN gateways use RSA, and until recently [24], there was not a standard option in the Internet Key Exchange (IKE) to avoid deterministic signatures when RSA keys are used. Therefore, it is conceivable that IPsec VPN gateways with faulty RSA implementations leak private key material not just over their web front ends, but also through IKE (that is, as part of regular IPsec operation). Due to the early use of a Diffie-Hellman key exchange in IKE protocol, such key leaks may not be visible to passive observers, only to those actively participating in the handshake.

Most VPN devices will not expose the administrative front end to the public Internet, which means that the results presented in section 3.3 likely underestimate the number of affected VPN devices and the range of vendors. Further research, using a different crawler implementation, might be able to settle this question.

5.2 SSH

Like IPsec, SSH performs a Diffie-Hellman key exchange early in the protocol, so active participation in the handshake is required to observe key leaks.

5.3 DNSSEC

Due to its use of offline signatures, DNSSEC is less exposed than other RSA-using protocols. Furthermore, it is recommended practice to check zone files for consistency and cryptographic correctness prior to publication. A key-leaking RSA signature would be noticed at this step, and prevent publication of the zone.

6 Conclusion

This report shows that it is still possible to use Lenstra's attack to recover RSA private keys, almost two decades after the attack has been described first, and that fault-based side-channel attacks can be relevant even in scenarios where the attacker does not have physical access to the device.

Several factors had to contribute to make this attack possible, namely:

1. Many RSA implementations still do not verify signatures that have been computed with the CRT optimization.

2. TLS with forward secrecy uses RSA signatures with deterministic padding, which are, in principle, vulnerable to Lenstra's attack.

3. A campaign is under way to deploy forward secrecy in TLS more widely, in effect making more servers potentially vulnerable to this attack.

4. Browsers cover up faulty server-computed RSA signatures, hiding even fairly high error rates.

The net effect is that a passive observer with visibility into global Internet traffic is likely able to recover quite a few RSA keys in a completely non-attributable fashion.

Initially, we suspected that these puzzle pieces fit together in a rather strange manner. But discussions with TLS library implementers showed they assumed that they had good reasons to avoid adding the check to RSA-CRT-based signing. The campaign for Perfect Forward Secrecy is likely based on its compelling name, and not on a careful analysis of the security trade-offs involved. The browser fallback behavior appears just to be an unrelated accident.

In the short term, implementing a checked RSA-CRT signing operation, like NSS or even OpenSSL already do (despite some theoretical concerns about the effectiveness in the case of OpenSSL), seems a very reasonable hardening measure. Longer term, TLS should perhaps switch to a non-deterministic signature scheme like RSASSA-PSS. However, none of these measures will help those operators who have been using unchecked RSA-CRT implementations for years, and are now wondering if their RSA private keys have already leaked.

Acknowledgments

Nikos Mavrogiannopoulos kindly answered questions about the RSA implementations used by GNUTLS. Xuelei Fan unearthed an online copy of Lenstra's 1996 memo, and Arjen Lenstra confirmed that is genuine. Werner Koch helped to link the introduction of RSA-CRT checking to the attack by Klima and Rosa. Hubert Kario provided helpful comments on a draft of this report. Peter Gutmann answered questions regarding cryptlib. Paul Wouters clarified some IPsec aspects.

References

[1] Arjen Lenstra, *Memo on RSA signature generation in the presence of faults*. `https://infoscience.epfl.ch/record/164524/files/nscan20.PDF` (retrieved 2015-01-22)

[2] OpenSSL Project, *OpenSSL Security Advisory 08 Jan 2015*. January 2015. `http://openssl.org/news/secadv_20150108.txt` (retrieved 2015-01-21)

[3] Eli Biham, Yaniv Carmeli, Adi Shamir, *Bug Attacks*. CRYPTO 2008. `http://iacr.org/archive/crypto2008/51570222/51570222.pdf` (retrieved 2015-06-15)

[4] Vlastimil Klíma and Tomáš Rosa, *Attack on Private Signature Keys of the OpenPGP format, PGP programs and other applications compatible with OpenPGP*. March 2001. `http://eprint.iacr.org/2002/076.pdf` (retrieved 2015-01-24)

[5] Peter Gutmann, *Vulnerability of RSA vs. DLP to single-bit faults*. Mailing list posting, November 2014. `http://www.metzdowd.com/pipermail/cryptography/2014-November/023464.html` (retrieved 2015-01-22)

[6] Oracle Corporation, *Oracle Critical Patch Update Advisory—April 2015*. `http://www.oracle.com/technetwork/topics/security/cpuapr2015-2365600.html#AppendixJAVA` (retrieved 2015-06-11)

[7] Peter Gutmann, *cryptlib Security Toolkit Version 3.4.2*. `ftp://ftp.franken.de/pub/crypt/cryptlib/manual.pdf` (retrieved 2015-01-21)

[8] Andrea Pellegrini, Valeria Bertacco, Todd Austin, *Fault-Based Attack of RSA Authentication*. Design Automation and Test in Europe (DATE), March 2010. `http://web.eecs.umich.edu/~valeria/research/publications/DATE10RSA.pdf` (retrieved 2015-01-21)

[9] Paul C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. CRYPTO, 1996. `http://www.cryptography.com/public/pdf/TimingAttacks.pdf` (retrieved 2015-01-21)

[10] Tim Dierks, Eric Rescorla (eds.), *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246, August 2008. `http://tools.ietf.org/html/rfc5246` (retrieved 2015-01-22)

[11] Burt Kaliski, *PKCS #1: RSA Encryption Version 1.5*. RFC 2313, March 1998. `http://tools.ietf.org/html/rfc2313` (retrieved 2015-01-22)

[12] Jakob Jonsson, Burt Kaliski, *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. RFC 3447, February 2003. `http://tools.ietf.org/html/rfc3447` (retrieved 2015-01-22)

[13] Hubert Kario, *TLS landscape*. January 2015. `https://securityblog.redhat.com/2014/09/10/tls-landscape/` (retrieved 2015-01-24)

[14] Rapid 7, *Project Sonar SSL Certificates*. January 2015. `https://scans.io/study/sonar.ssl` (retrieved 2015-01-22)

[15] *Certificate Transparency—Known Logs*. `http://www.certificate-transparency.org/known-logs` (retrieved 2015-02-03)

[16] University of Michigan, *Full IPv4 HTTPS Handshakes*. `https://scans.io/series/443-https-tls-full_ipv4` (retrieved 2015-06-08)

[17] Hillstone Networks, 山石网科关于第三方某型号硬件器件设计缺陷可能导致信息泄露问题的声明. July 2015. `http://www.hillstone.com.cn/services/2015/0728/20150728.html` (retrieved 2015-08-26)

[18] ZyXEL Communications Corp., personal communications, June 2015.

[19] Citrix Systems, Inc., personal communications, April 2015.

[20] Hillstone Networks, personal communications, June 2015.

[21] Radware, *BP15958: Alteon Support for FIPS 140-2*. `http://kb.radware.com/Questions/Alteon/Public/Alteon-Support-for-FIPS-140-2` (retrieved 2015-08-25)

[22] Cavium, Inc., personal communcations, August 2015.

[23] Donald E. Knuth, *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Second edition. Addison-Wesley, 1981.

[24] Tero Kivinen, Joel Snyder, *Signature Authentication in the Internet Key Exchange Version 2 (IKEv2)*. RFC 7427, January 2015.

```
diff --git a/crypto/rsa/rsa_eay.c b/crypto/rsa/rsa_eay.c
index 3e08fe7..87c1f09 100644
--- a/crypto/rsa/rsa_eay.c
+++ b/crypto/rsa/rsa_eay.c
@@ -844,6 +844,17 @@ static int RSA_eay_mod_exp(BIGNUM *r0, const BIGNUM *I, RSA *rsa, BN_CTX *ctx)
        if (!rsa->meth->bn_mod_exp(m1,r1,dmq1,rsa->q,ctx,
                rsa->_method_mod_q)) goto err;

+       // Occasionally inject a fault.
+       {
+         static time_t last;
+         time_t current = time(NULL);
+         if (current - last > 5) {
+           last = current;
+           BN_add_word(m1, 0x80);
+           printf("Fault injected.\n");
+         }
+       }
+
        /* compute I mod p */
        if (!(rsa->flags & RSA_FLAG_NO_CONSTTIME))
                {
@@ -897,7 +908,7 @@ static int RSA_eay_mod_exp(BIGNUM *r0, const BIGNUM *I, RSA *rsa, BN_CTX *ctx)
        if (!BN_mul(r1,r0,rsa->q,ctx)) goto err;
        if (!BN_add(r0,r1,m1)) goto err;

-       if (rsa->e && rsa->n)
+       if (rsa->e && rsa->n && 0)
                {
                if (!rsa->meth->bn_mod_exp(vrfy,r0,rsa->e,rsa->n,ctx,rsa->_method_mod_n)) goto err;
                /* If 'I' was greater than (or equal to) rsa->n, the operation
```

Figure 1: Patch to inject a RSA-CRT-related fault into OpenSSL for testing purposes