Variants of Bleichenbacher's Low-Exponent Attack on PKCS#1 RSA Signatures

Ulrich Kühn*
Sirrix AG security technologies, Bochum
ukuehn@acm.org

Andrei Pyshkin, Erik Tews, Ralf-Philipp Weinmann
Department of Computer Science, Universität Darmstadt
{pyshkin,e_tews,weinmann}@cdc.informatik.tu-darmstadt.de

Abstract: We give three variants and improvements of Bleichenbacher's low-exponent attack from CRYPTO 2006 on PKCS#1 v1.5 RSA signatures. For each of these three variants the *fake* signature representatives are accepted as valid by a flawed implementation. Our attacks work against much shorter keys as Bleichenbacher's original attack, i.e. even for usual 1024 bit RSA keys.

The first two variants can be used to break a certificate chain for vulnerable implementations, if the CA uses a public exponent of 3. Such CA certificates are indeed deployed in many browsers like Mozilla, Opera and Konqueror. The third attack works against the Netscape Security Services only, and requires the public exponent 3 to be present in a site's certificate, not the CA certificate.

Using any of these attack vectors, an active adversary can mount a full man-in-the-middle attack on any SSL connection initiated by a vulnerable client.

1 Introduction

RSA is the most widely used public key crypto-system, used both for digitally signing and encrypting messages. Its security relies on the problem of extracting d-th roots modulo a composite number N, which is believed to be hard if N is composed of large primes.

In order to boost the efficiency of public-key operations, the public exponent e is usually chosen to be of low Hamming weight: common public exponents are $e \in \{3, 17, 65537\}$. Although well-known attacks exist on certain padding formats when a small public exponent is used for RSA *encryption* [Bon99], small public exponents such as 3 are sometimes even recommended for RSA *signatures* for the sake of efficiency [Eas01] – provided that the key is used for signatures only.

Recently Bleichenbacher exposed (see [Fin06a]) a critical problem arising from a common flaw in implementations of the RSA signature verification as specified in PKCS #1 v1.5. This flaw can lead to faked signatures being accepted as valid, hence rendering the scheme insecure. This attack requires a large modulus (more than 3000 bits) compared to the

*Some of this author's contribution was carried out while at Deutsche Telekom Laboratories.

exponent size. As a consequence, the choice of small exponents for RSA signatures is considered problematic as well.

CONTRIBUTION. In this paper we first show how Bleichenbacher's low-exponent attack on PKCS#1 RSA signatures can be modified to work for key sizes used commonly by certificate authorities, namely 1024 to 2048 bits and $e = 3$. It works against older versions of Firefox, Opera, and Konqueror. The second attack highlights how to exploit a similar but different implementation flaw in signature verification, which is present in older versions of OpenSSL for example. Our attack algorithms is general enough to work for a large number of different parameter sets. A third attack is presented that exploits a flaw in older versions of the Netscape Security Services (NSS). All three attacks produce *fake signatures*, i.e. message representatives that do not conform to the standard but are accepted as valid by flawed implementations nevertheless.

The impact of the first two attacks is that they break the certificate chain authenticating a site's certificate and allows an adversary to set up its own false certificates which are accepted as valid. The third attack breaks the authentication included in the SSL / TLS protocol. Thus all of these attacks allow to mount man-in-the-middle impersonations to intercept and decrypt arbitrary SSL/TLS-encrypted sessions, including HTTPS, POP3S, or IMAPs, without the user being warned.

Security advisories about the underlying implementation flaws have been published soon after their discovery. Thus, affected software should be fixed by now.

STRUCTURE. The structure of the paper is as follows: We briefly describe the PKCS#1 v1.5 signature format in Section 2 and Bleichenbacher's attack in Section 3. In Section 4 we describe our variant that works for RSA moduli of commonly used bit lengths, in Section 5 we describe a variant that exploits an unchecked algorithm parameters field, and in Section 6 we show an attack method against further implementation flaw in the Netscape Security Services. In Section 7 we discuss issues with the underlying PKCS#1 standard. Finally, in Section 8 we highlight how the affected implementations were fixed.

2 PKCS#1 v1.5 Signature Format

First, we briefly describe the PKCS#1 v1.5 format for RSA signatures [RSA93]. We start with some notation. Let N be the RSA modulus, e the public and d the private exponent. Denote the bitlength of N by $|N|$, i.e. the smallest number of bits to represent N, and its length in octets by $n = \lceil |N|/8 \rceil$. Denote the concatenation of bit or octet strings by $||$. Call the input $m \in \mathbb{Z}_N^*$ to the RSA transform the *message representative* or *signature input*, using a representative $0 < m < N$.

Definition 1. Let $B = B_{l-1}|| \ldots ||B_0$ be a string of l octets. Then define

$$m_B = \mathrm{rep}(B) = \sum_{i=0}^{l-1} B_i 2^{8i}.$$

Let $\mathrm{i2o}_l(m)$ denote the l-octet string of a message represenative $m < 2^{8l}$, such that

$\mathrm{i2o}_l(\mathrm{rep}(B)) = B$ for all octet strings of length l.

In the PKCS#1 v1.5 signature format [RSA93] the message representative M for a message m is formed by converting the octet string

$$\mathtt{00}_x||\mathtt{01}_x||\mathtt{FF}_x||\ldots||\mathtt{FF}_x||\mathtt{00}_x||a||H(m)$$

to a positive integer, where the string a consists of the ASN.1 encoded object identifier of the hash function H used for computing $H(m)$ and possible algorithm parameters. The padding string of $\mathtt{FF}_x$ octets is made as long as it takes to have the whole signature input be representable by a string of exactly n octets. However, a minimum size of eight $\mathtt{FF}_x$ octets is required.

When creating a signature, the message representative M formed as described is passed to the RSA private key transform $M^d \bmod N$. When verifying the signature S, the RSA public key transform $S^e \bmod N$ is applied, and the resulting message representative M' is checked for compliance with the standard, as well as that the included hash code is consistent with the hash code of the (allegedly) signed message.

It is exactly the compliance check of the message representative that seems difficult to get right, as we will show during the rest of this paper using actual implementation flaws, and how to exploit them. In fact, all attacks given in this paper produce message representatives that are *not* compliant to the PKCS#1 v1.5 standard, therefore we denote them by *fake signatures*. Nevertheless, flawed implementations will accept them as valid signatures.

3 Bleichenbacher's Low-Exponent Attack

During the rump session of the Crypto 2006 conference Daniel Bleichenbacher presented an attack on implementations of the RSA PKCS#1 v1.5 signatures, as summarised subsequently by Finney [Fin06a]. This attack requires the public exponent to be very small compared to the modulus, e.g. $e = 3$. Bleichenbacher made use of the fact that some implementations don't verify that there is no data after the hash. Thus, these implementations accept message representatives with a format

$$\mathtt{00}_x||\mathtt{01}_x||\mathtt{FF}_x||\ldots||\mathtt{FF}_x||\mathtt{00}_x||a||H(m)||g$$

as valid, where g denotes arbitrary data.

Bleichenbacher showed that this can be exploited to forge RSA signatures, and demonstrated this for an RSA key with a 3072 bit modulus and $e = 3$. With these parameters, it is very easy to arrange this data to be a perfect cube smaller than 2^{3072}. The cube root of this number computed over $\mathbb{R}$ is accepted as a valid RSA signature by broken implementations.

4 Variants for Smaller RSA Moduli

While Bleichenbacher's example outlined a serious flaw in some RSA implementations, usually no CA uses an RSA key size of 3072 bits. However, CAs with public exponent 3 and key size of 1024 bits exist and are furthermore included in all major web browsers, while CAs with exponent 3 and a key size of 2048 bits are less common. Here we show how to modify Bleichenbacher's attack to work with moduli as short as 1024 bits.

4.1 Signature Input

The PKCS#1 standard requires at least 8 octets in the $\mathtt{FF}_x$ part of the padding. However, some implementations do not check the number of padding octets at all, some require indeed 8 octets of padding, some just require 7 octets of padding. To be able to exploit every implementation that does not check trailing data after the hash value, we will use 8 octets of $\mathtt{FF}_x$ values. Thus, breaking down the lengths of the individual parts of the padding, we get for the most significant fixed part of the signature input the octet string

$$A = \mathtt{00}_x\mathtt{01}_x||\mathtt{FF}_x \ldots \mathtt{FF}_x||\mathtt{00}_x||a||H(m) \tag{1}$$

where the parts have sizes of 16, 64, 8, 120, and 160 bits (assuming SHA-1 as H and a being the ASN.1 encoded prefix), yielding a total 46 octets. For adaptions to other hash value lengths see the analysis below.

4.2 Attack Algorithm

For the attack to work on rather short key sizes we need to be able to vary the hash value $H(m)$ of the message. To do so for a certificate obeying the X.509 standard, we vary freely choosable fields such as the serial number of the certificate, time of issue, or time of validity.

Given an X.509 certificate, the attack algorithm is as follows:

1. Add one second to the *notAfter* value.
2. Hash it, and construct an octet string A with length 46 octets as decribed (see (1)). Set $a' \leftarrow \mathrm{rep}(A)$ and construct a message representative with minimal padding and trailing $\mathtt{FF}_x$ octets:
$$s' \leftarrow a \cdot 2^{8\cdot 82} + \sum_{i=0}^{81} (\mathtt{FF}_x)\, 2^{8i}, \tag{2}$$
3. Compute $t \leftarrow \left\lfloor \sqrt[3]{s'} \right\rfloor^3$. Check if $\lfloor t/2^{1024-46\cdot 8} \rfloor = \lfloor s'/2^{1024-46\cdot 8} \rfloor$, i.e. if the 46 most significant octets are equal in t and s'. If so, $s \leftarrow \sqrt[3]{t}$ is now the fake signature. If not, go to step 1.

Analysis. For a 1024-bit modulus the complete length of the signature input is 128 octet. With 46 octets fixed, we have 82 octets (=656 bits) to fill with arbitrary data.

As the message representative starts with $\texttt{00}_x\ \texttt{01}_x\ \texttt{FF}_x$, it is a number $< 2^{1009}$. Thus the distance between two perfect cubes of this size is smaller than 2^{673}. This is 17 bits more than the number of bits we can choose arbitrarily. Consequently the probability of choosing a number with a distance smaller than 2^{656} to the next perfect cube is about 2^{-17}.

More general, let c be the length (in bits) of the hash code of the employed hash function and d the length of the ASN.1 data not including the hash code (in bits). Then there are $n_f = 8(2 + 8 + 1) + d + c = 15 + 73 + d + c$ fixed bits in our approach (including the 15 leading zero bits). Thus, for 1024 bit moduli, any $d + c < \lfloor 1009/3 \rfloor - 73 = 263$ should yield directly a perfect cube without much problems. For $d + c \geq 263$ we expect that the probability of success of a single guess in step 3 is about $p \approx 2^{-(d+c-263)}$. Thus, the expected work factor for the attack algorithm is about $1/p$ trials.

For example, for MD5 with $d = 144$ and $c = 128$ this is about $p \approx 2^{-9}$ with 2^9 expected iterations, for SHA-1 we have $d = 15 \cdot 8 = 120$ and $c = 160$, thus $p \approx 2^{-17}$ with 2^{17} expected iterations. However, for SHA-256, $d = 152$ and $c = 256$ yield an extremely small $p \approx 2^{-145}$.

Experimental Results. We implemented this attack in Java. Executing it took 2-3 minutes on average on an IBM T41p Thinkpad using an Intel Pentium M processor with 1700MHz running SUN JDK 1.5.0_06 under Linux 2.6.

Impact. An X.509 certificate containing such a fake signature for a CA public key with exponent $e = 3$ is accepted as valid by flawed implementations. This allows to "issue" false certificates in the name of the CA. Trusted CA certificates with $e = 3$ are included in affected versions of Mozilla, Firefox, Thunderbird, Opera, Konqueror, and possibly other browsers, too. Thus, this breaks the certificate chain, allowing an adversary to intercept and decrypt any SSL/TLS-protected traffic from, e.g., the HTTPS, POP3s, IMAPs protocols.

5 Exploiting the Algorithm Parameters Field

Another implementation flaw in PKCS#1 signature verification is not to check the algorithm parameter field for the hash function. This has first beed described by Oiwa, Kobara, and Watanabe in [OKW06]. However, they give only an attack on 1024 bit RSA with MD5. Below we present a practical algorithm for computing fake signatures that are accepted by such flawed implementations. While we use SHA-1 as an example, other hash functions could be used also. Further, we point along with limiting factors for the algorithm.

5.1 Signature Input

The ASN.1 encoded data in the signature input indicates the hash algorithm used during hash calculation. Technically, this consists of an object identifier of the actual hash functions and additional input parameters like an intialization vector. For all hash functions in use today a `NULL` value or a zero-length field must be used as algorithm parameters.

While the attack is general in nature, we use $H = \text{SHA-1}()$ as a concrete example. The message representative can be broken down into $r = a||z||b||h$ where a is fixed data, determined by the padding structure and a prefix of the ASN.1 encoded data, z is arbitrary data placed in a maximal-size algorithm parameters field representing the adversary's playground, b the suffix of the ASN.1 encoded data, and $h = H(m)$. In more detail, using the SHA-1 OID we have

$$a = 00_x01_x||\texttt{FF}_x \ldots \texttt{FF}_x||30_x??_x30_x??_x||06_x05_x2\text{B}_x0\text{E}_x03_x02_x1\text{A}_x||04_x??_x \tag{3}$$

where the $??_x$ indicate octet values that depend on the exact length of the resulting ASN.1 structure. Further, we have $b = 04_x14_x$ to indicate the length of the hash code. Other hash functions yield very similar octet strings, changes occur after the 06_x octet and within b.

5.2 Attack Algorithm

The algorithm to compute a perfect cube showing the format given above is split into two parts: (i) one part that computes a most significant part of the fake signature which only dependens on the octet-length of the modulus, but not on any specific message, and (ii) a part that computes the least significant part of the fake signature, which does depend on the message to be signed and, if the modulus is rather short, on the most significant part from the first part of the algorithm. Note that b is fixed for any fixed hash code length. Further, we use $d = |b| + |h|$. So for SHA-1 we have $d = 176$.

1. Compute the most significant part $u2^{d'}$ such that $(u2^{d'})^3$ has the prefix a at the correct position with d' to be determined in the process:
   (a) Let $c \geq 0$ be some value $c < N$ (this c will be determined by binary search). Set $a' \leftarrow \text{rep}(a) \cdot 2^{|z|+|b|+|h|} + c$, i.e. a' is the prefix of the fake message representative up the the point where the algorithm parameters start, with c compensating the truncation error in the next steps.
   (b) Set $s \leftarrow \lfloor \sqrt[3]{a'} \rfloor$, and compute $\hat{a} \leftarrow (s)^3$.
   (c) If $\lfloor \hat{a}/2^{|z|+d} \rfloor \neq \text{rep}(a)$ restart at step 1a, updating c by binary search such that $\text{rep}(a)$ is approximated from below. If equality cannot be reached, abort.
   (d) Find the largest $d' \leq d = |b| + |h|$ such that $v \leftarrow s - (s \bmod 2^{d'})$ and $\lfloor v^3/2^{|z|+d} \rfloor = \text{rep}(a)$, i.e. the maximal number of low bits we can cut out to still have the correct prefix after cubing.

(e) Record the most significant part u of the fake signature as

$$u \leftarrow (\hat{a} - (\hat{a} \bmod 2^{d'}))/2^{d'}. \tag{4}$$

2. Obtain the data m to be signed, preferrably with some variable parts. m could be a suitable X.509 certificate, the variable parts being, e.g., a serial number. Use this to compute the least significant part of the fake signature as follows:

   (a) Compute the hash code $h \leftarrow H(m)$. If the least significant bit of h is not 1, alter some variable part in m and repeat.

   (b) Set $w \leftarrow \mathrm{rep}(b||h)$, $d \leftarrow |w|$ and compute $x < 2^d$ such that $x^3 \equiv w \bmod 2^d$ by inverting 3 modulo 2^{d-1}.

   (c) Check if

   $$u \bmod 2^{d-d'} = \lfloor x/2^{d'} \rfloor. \tag{5}$$

   If not, alter some variable part in m and repeat from step 2a.

3. Now compute a candidate fake signature

$$\tilde{s} = \lfloor u/2^{d-d'} \rfloor \cdot 2^d + x \tag{6}$$

for the current instance of the data m.

Analysis. To show that the algorithm yields indeed a fake signature we have to show that, first, the verification procedure sees the correct ASN.1 part for the hash value, and second, it sees the correct prefix with an arbitrary algorithm parameters field.

For the first part of the analysis, we rewrite (6) using $u' \leftarrow \lfloor u/2^{d-d'} \rfloor$, i.e. the non-overlapping part of u (for $d' = d$ we have $u' = u$). Then $\tilde{s} = u'2^d + x$. When verifying, we find $\tilde{s}^3 = (u'2^d + x)^3 = (2^d)(u'^3(2^d)^2 + 3u'^2(2^d)x + 3u'x^2) + x^3$. Thus, the least significant d bits of $\tilde{s}^3$ are only influenced by x^3. As $x^3 = w \bmod 2^d$, the bits in the b and h parts are correct. The reason we require h to have the least significant bit set is that the computation of x works always in this case.

For the second part of the analysis, we partition $x = x''2^{d'} + x'$ with $x'' \leftarrow \lfloor x/2^{d'} \rfloor$ and $x' \leftarrow x \bmod 2^{d'}$. From (5) we have $u \bmod 2^{d-d'} = x''$ and $\tilde{s} = u2^{d'} + x'$. Thus we have

$$\tilde{s}^3 = \underbrace{(u2^{d'})^3}_{\gamma} + \underbrace{(3(u2^{d'})^2x' + 3(u2^{d'})x'^2 + x'^3)}_{\delta}.$$

We have to prove that the prefix a' of γ, resulting from the u part in $\tilde{s}$, cannot be disturbed by adding δ. We observe that

$$A := (u2^{d'})^3 \leq \mathrm{rep}(a)2^{d+|z|} \leq \tilde{s}^3 \leq (u2^{d'} + 2^{d'})^3 =: B, \tag{7}$$

and thus $B - A = 3u^2 2^{2d'} 2^{d'} + 3u2^{d'} 2^{2d'} + 2^{3d'} = 2^{3d'}(3u^2 + 3u + 1)$. Introducing a formal bound $t \geq 2$ such that $u < 2^t$, we obtain $3u^2 + 3u + 1 < 4 \cdot 2^{2t} = 2^{2t+2}$, and $B - A < 2^{3d'+2t+2}$. As a consequence, we find $|\tilde{s}^3 - \mathrm{rep}(a)2^{d+|z|}| \leq B - A < 2^{3d'+2t+2}$.

In order to have a fake signature where the message representative has the correct prefix (see step 1), we get the necessary condition $3d' + 2t + 2 < |z| + d$. Further, from the modulus size of n octets, we also have $8n = |a| + |z| + |b| + |y| = |a| + |z| + d$. Thus, we have $8n - |a| - 3d' - 2 > 2t$. In order to obtain a correct prefix $\mathrm{rep}(a)$ from the u part after cubing $\tilde{s}$ we need, applying entropy arguments, at least $t \geq |a|$ bits to code for u. Thus we obtain the necessary condition

$$8n - 2 > 3(|a| + d'). \tag{8}$$

Note that the $d - d'$ overlapping bits of u and x must coincide, so that we get a partial brute-force condition for $d - d' > 0$.

Experimental Results. When determining $|a|$ there is a technical complication from the ASN.1 notation for the length fields (values indicated with $??_x$ in (3)), which stems from the variable-size encoding of length fields. As a consequence $|a|$ depends on n, and for usual key sizes we have $192 \leq |a| \leq 240$. Concretely, for $n \leq 140$, i.e. $|N| < 1120$, only a single octet is needed for the length encoding of the first sequence (and also the other length encodings). Therefore $|a| = 192$ in these cases. For $177 \leq n \leq 269$, i.e. $1416 \leq |N| < 2152$, all the length fields are encoded by 2 octets, so $|a| = 216$.

Our analysis above indicates that for $|N| < 1120$ we may have $d' < d$, so that possibly a large number of steps in the second part of the algorithm is necessary. We obtain $d = d'$ for $|N| \geq 1120$. To verify our analysis in practicewe implemented the algorithm. For varying modulus sizes (in bits) we obtained the following values for $d - d'$ which indicates number of bits to brute-force:

n	768	1024	1056	1088	1120
$d - d'$	101	21	9	0	0

This shows that for key length ≥ 1024 bit the attack is feasible. Producing fake signatures for a 1024 bit RSA key in a couple of minutes on a Pentium M 1.5 GHz machine. For larger key sizes the only constraint is the least significant bit of the hash code being 1. Verification using a vulnerable version of `OpenSSL` succeeded as expected.

Impact. Similar to the attack in the last section this attack allows to produce fake signatures for CA public keys. Thus, a flawed implementation will accept a false certificate containing such a fake signature. This completely breaks the certificate chain.

6 Attack Variant against the Netscape Security Services

Here we present an attack variant that works against yet another, but similar implementation flaw as the ones before. This flaw is present in a part of the Netscape Security Services NSS up to and including version 3.10.2 that deals with PKCS#1 signatures as they are used inside the SSL/TLS protocol. This version of the NSS was used up to version 1.5.0.7 of the

Firefox browser. Below we show an algorithm that can break the SSL/TLS authentication when the server uses the exponent 3 in its certificate.

6.1 Signature Input

The file `security/nss/lib/softoken/rsawrapr.c` of NSS contains two functions which decrypt RSA signature and check for correct PKCS#1-padding. One is used for verifying PKCS#1 signature, while the other, `RSA_CheckSign()`, is used for verifying the signature in an `ServerKeyExchange` message of SSL or TLS.

This latter function checks for the leading $\texttt{00}_x\texttt{01}_x$ octets, and due to an implementation flaw, accepts any number of $\texttt{FF}_x$ octets, including zero[1] . Further, the comparison of the expected and the given hash value are done by taking the appropriate number of octets from the end of the octet string, e.g. the least significant octets in the message representative.

In fact, the function `RSA_CheckSign()` accepts, instead of a fully standard-conforming message representative, an octet string

$$B = \texttt{00}_x||\texttt{01}_x||\texttt{00}_x||g||h \tag{9}$$

as valid, provided that (i) h has the expected value and length, and (ii) the length of g is such that B has the correct length, i.e. g consists of $n - 3 - \lceil |h|/8 \rceil$ octets.

6.2 Attack Algorithm

The key idea of the attack is, like in the attacks presented here, to turn the computation of the 3rd root modulo N into the computation of a 3rd root over the integers. That is, we will produce a value whose bit representation, when run through the RSA verification transformation, is described by equation (9). Note that larger exponents can also be attacked if correspondingly larger keys are used. We construct an octet string

$$S = Y||\texttt{00}_x \ldots \texttt{00}_x||X \tag{10}$$

with $Y \neq 0$ such that $v = \mathrm{rep}(S)^3 < N$ and $\mathrm{i2o}_n(v)$ is formatted according to (9). We compute a prefix

$$Z = \texttt{00}_x||\texttt{01}_x||\texttt{00}_x \tag{11}$$

such that for $y' \leftarrow (\mathrm{rep}(Y) \cdot 2^a)^3$ the octet string $Y' \leftarrow \mathrm{i2o}_n(y')$ has the 3-octet prefix Z. Let $b = n - 3$, the number of octets in B (see (9)) following the prefix Z (e.g. $b = 125$ for 1024 bit keys). Let k denote the bitlength of the hash code h expected in the fake signature. Note that $\phi(2^k) = 2^{k-1}$ is relative prime to the exponent $e = 3$, so computation of e-th roots $\bmod 2^k$ succeeds whenever h is odd. Then the algorithm is as follows:

[1] This problem was noted independently by Finney [Fin06b].

1. Let $0 < x_h < N$ be such that

$$x_h^3 \equiv h \bmod 2^k, \tag{12}$$

and let $X_h = \mathrm{i2o}_{\lceil k/8 \rceil}(x_h)$ be its octet representation.

2. Distinguish three cases:

$b \equiv 2 \pmod 3$**:** Set $y' \leftarrow 2^{16} \cdot 2^{8(b-5)/3}$.
$b \equiv 1 \pmod 3$**:** Set $y' \leftarrow \texttt{2857}_x \cdot 2^{8(b-4)/3}$.
$b \equiv 0 \pmod 3$**:** Set $y' = \texttt{065A}_x \cdot 2^{8(b-3)/3}$.

Now set $s = y' + x_h$, the fake signature for h.

Analysis. To show the correctness of the algorithm, i.e. that s is a fake signature, we note that $s^3 \equiv x_h^3 \equiv h \pmod{2^k}$, thus the expected hash value will be found in the message representative. Further, distinguishing the three cases in the second step, we see from a dirct computation that $y'^3 = 2^{48} \cdot 2^{8(b-5)}$ for $b \equiv 2 \pmod 3$, $y'^3 = \texttt{0001006D260447}_x \cdot 2^{8(b-4)}$ for $b \equiv 1 \pmod 3$, and $y'^3 = \texttt{0001003CA7A8}_x \cdot 2^{8(b-3)}$ for $b \equiv 0 \pmod 3$. We note that for $x_h < 2^{8(b-5)/3}$, $x_h < 2^{8(b-4)/3}$, and $x_h < 2^{8(b-3)/3}$, respectively, these prefixes will stay intact even $s^3 = (y' + x_h)^3$.

Example. For a usual key length of a 1024 bit modulus we get bounds for x_h of $|x_h| < 320$, i.e. we can fake signatures that include hash codes of at most this size. As the function `RSA_CheckSign()` is used for verifying a certain signature during the SSL/TLS handshake with the hash code being the concatenation of an SHA-1 and MD5 hash code, we have $|x_h| = 160 + 128 = 288$. Therefore the attack succeeds for 1024 bit RSA keys. The attack can be done on the fly: The most time consuming part is the inversion in equation (12).

Impact. During SSL/TLS handshake the server has the option to issue a new key by sending a `ServerKeyExchange` message to the client[2] . This message is signed using the concatenation of MD5 and SHA-1, and is checked by the `RSA_CheckSign` function attacked above. As a consequence, an adversary can send a new key, along with a fake signature on the `ServerKeyExchange` message to the client, and is thus able to act as a man in the middle. This breaks the SSL/TLS authentication.

7 Issues in the PKCS#1 Standard Documents

The PKCS#1 standard (see [JK03, RSA93]) specifies and documents well-known and widespread way to format the message representative for an RSA signature. Here we

[2]This mechanism was introduced to support short keys for encryption due to export restrictions while keeping the strength of the authentication. It is kept in TLS to support authentication-only server certificates.

discuss the issues in the PKCS#1 standard regarding signature verification that are indeed at the heart of the problems described in this paper.

Comparison-based verification. The signature verification procedure suggested in the current version 2.1 of the standard is to build up the message representative as it would be expected, applying the RSA verification transformation to the signature and then doing a bit-by-bit comparison of the received and the expected message representative. This method is based on the assumption that the formatting of the message representative is unique.[3] However, this is not really the case. The algorithm parameter field is *optional* in the sequence comprising the algorithm identifier. As current hash functions only have a `NULL` parameter here, this field can be either present or not. Thus, verifying using the comparison-based method needs to take this into account to not falsely reject valid signatures.

Parsing-based verification. Further, the PKCS#1 v2.1 standard includes a footnote mentioning an alternative method of verifying a v1.5 signature. This alternative method parses the ASN.1 data and checks that everything is as expected and required:

1. Checking the padding
2. Parsing the ASN.1 encoded data while checking every octet for wrong data and making sure that no additional data is present.
3. Checking for the correct OID of the hash algorithm
4. Checking the Algorithm Parameters field
5. Comparing the hash code data.

Each of these steps must be carefully implemented in order to avoid vulnerabilities of the resulting code. In fact, despite being only briefly mentioned in the standard, this method is used by the major implementations of the v1.5 signature verification procedure.

While we view the comparision-based method as much better-suited[4] for correct implementation, it might falsly reject some valid signatures. However, we believe the real-world impact of this change is very small. Nevertheless there might be some areas of application where it indeed has an impact, e.g. when embedded software or hardware cannot be upgraded.

[3] The most important changes from version 1.5 [RSA93] to the current version 2.1 [JK03] is the use of the Distinguished Encoding Rules DER instead of the Basic Encoding Rule BER as the ASN.1 encoding scheme, possibly in order to remove some ambiguity. In fact, BER allows length encodings for ASN.1 items in many equivalent forms, while DER gives indeed a unique encoding of the data. Thus, a verifier requiring DER might reject previously valid signatures.

[4] We note that the comparision-based verification method for version 1.5 signature padding would not work when keyed universal one-way hash functions (see [BR97]) are employed. Here the algorithm parameters would contain the key, which has to be extracted *before* verifying the message representative.

8 Fixing Affected Implementations

Here we describe how some of the major affected implementations were fixed regarding the attacks described in this paper.

GnuTLS. This SSL library [Gnu] suffered from the problem of not properly checking the algorithm parameters field (see Section 5). This problem is fixed in version 1.4.4. However, a source code inspection[5] revealed that the minimum size of 8 octets of $\texttt{FF}_x$ in the padding string is not enforced.[6] The attacks using trailing data after the ASN.1 data (see Sections 3 and 4) are avoided by using an ASN.1 parser that returns an error on additional data after an ASN.1 structure.

OpenSSL. The patch for the affected OpenSSL version [Ope] fixed two problems: not checking for trailing data (see Sections 3 and 4), and not checking the algorithm parameters field (see Section 5).[7]

Netscape Security Services (NSS). The problem described in Section 4 has been fixed in the NSS prior to version 3.11.3 (contained in Firefox and Thunderbird prior to version 1.5.0.7) by switching to another ASN.1 parser[8]. The implementation error described in Section 6 is fixed since NSS version 3.11.3. While this problem was fixed for Firefox 2.0 [Moz], it went unnoticed that this fixed a critical problem still present in the older 1.5 branch of Firefox and Thunderbird (fixed since 1.5.0.8).

Bouncy Castle Java Cryptographic Service Provider. The cryptographic service provider for Java [Bou] was affected up to version 1.33. In later versions this problem is fixed in a rather radical way. The code was switched from a parsing-based method to the comparison-based method (see Section 7). Further, the verification code contains a hard-coded resolution of the ambiguity from the algorithm parameters field being optional.

9 Conclusion

We have shown how Bleichenbacher's attack on PKCS#1 signature verification can be extented to much shorter key sizes. Further, we have shown how similar implementation flaws in signature verification can be efficiently exploited to create fake signatures. All implementation flaws discussed here stem from the problem of not checking some parts of the message representative. This highlights (again) that it is essential to check carefully all fields in the message representative for compliance with the specification.

[5] File `lib/gnutls_pk.c`, function `_gnutls_pkcs1_rsa_decrypt()`

[6] Interestingly the original security advisory in [OKW06] detailing the algorithm parameter attack gives an example where the padding uses only two octets of $\texttt{FF}_x$ in the padding string, and consequently is not conforming to the standard, which requires at least eight octets.

[7] The function `RSA_verify()` the file `crypto/rsa/rsa_sign.c` is fixed by checking for trailing data after the ASN.1-encoded data and for an ASN.1-object of type `NULL` as the algorithm parameters. This latter check is hard-coded, blocking future use (without changing the code) of hash functions that do have parameters.

[8] However, the security-critical behavior of both parsers, allowing trailing data for backward compatibility vs. disallowing it, is documented only by comments deeply embedded into the source code. Thus, a security critical feature of the code is used without being well-documented.

References

[Bon99] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[Bou] http://www.bouncycastle.org.

[BR97] Mihir Bellare and Phillip Rogaway. Collision-Resistant Hashing: Towards Making UOWHFs Practical. In Burton S. Kaliski, Jr., editor, *Advances in Cryptology – CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 470–484. Springer-Verlag, 1997.

[Eas01] Donald E. Eastlake. RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS). Internet Request for Comment RFC 3110, Internet Engineering Task Force, May 2001. http://www.ietf.org/rfc/rfc3110.txt.

[Fin06a] Hal Finney. Bleichenbacher's RSA signature forgery based on implementation error. Post to the IETF OpenPGP mailing list, August 2006. http://www.imc.org/ietf-openpgp/mail-archive/msg14307.html.

[Fin06b] Hal Finney. Re: Why the exponent 3 error happened. Post to cryptography mailing list, September 2006. http://www.mail-archive.com/cryptography@metzdowd.com/msg06693.html.

[Gnu] http://www.gnu.org/software/gnutls/.

[JK03] John Jonsson and Burt Kaliski. PKCS #1: RSA Cryptography Specifications Version 2.1. RFC 3447, February 2003.

[Moz] http://www.mozilla.org.

[OKW06] Yutaka Oiwa, Kazukuni Kobara, and Hajime Watanabe. GNUTLS-SA-2006-4 vulnerability report. Post to the gnutls-dev mailing list, September 2006. http://lists.gnupg.org/pipermail/gnutls-dev/2006-September/001240.html.

[Ope] http://www.openssl.org.

[RSA93] RSA Laboratories. PKCS #1: RSA Encryption Standard, Version 1.5, November 1993. ftp://ftp.rsasecurity.com/pub/pkcs/ascii/pkcs-1.asc.