A Decade After Bleichenbacher '06, RSA Signature Forgery Still Works

Sze Yiu Chau  Purdue University

schau@purdue.edu
Who am I?

• Born and raised in Hong Kong

• PhD in CS from Purdue

• Joining CUHK IE as AP in 2020

• Interests: (in)secure design and implementations of protocols
A little brain teaser

What is common among these protocols?

- They all have RFCs
- They're all security-critical
- They all can benefit from PKCS#1 v1.5 RSA signatures
Textbook RSA signature

- Signing message $m$:
  
  $m \\ H(m) \\ H(m)^d \mod n \\ S$

  where $d =$ private exponent
  
  $n =$ modulus

- Given $(S, m, e, n)$, verifying $S$ is a valid signature of $m$

  $S \\ S^e \mod n \\ ? \\ H(m)$

  where $e =$ public exponent
Beyond textbook RSA

• Reality is more complex than that

1. Which H() to use?
   • SHA-1, SHA-2 family, SHA-3 family …

2. n is usually much longer than H(m)
   • $|n| \geq 2048$-bit
   • $|SHA-1| = 160$-bit, $|SHA-256| = 256$-bit
   • Need meta-data and padding
Beyond textbook RSA

• The PKCS#1 family of standards
• Both encryption and signature schemes
  • version 2+ adapted schemes from Bellare et al.
• For signatures, PKCS#1 v1.5 most widely used
  • e.g. certificates of Google, Wikipedia
PKCS#1 v1.5 Signature Scheme

- Signing:

\[
\begin{align*}
\text{m} \\
\text{H(m)} \\
\text{H(m)}^d \\
\text{H(m)}^d \mod n \\
k \\
k^d \\
k^d \mod n
\end{align*}
\]

For signature, BT (Block Type) = 0x01

PB (Padding Bytes) = 0xFF 0xFF \ldots 0xFF

- At least 8-byte long
- Pad k to the size of n

AS is a DER-encoded ASN.1 structure, containing:
- Meta-data describing H()
- The actual H(m)
PKCS#1 v1.5 Signature Scheme

• Encoded AS looks like this:

```
30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14 2A AE 6C 35 C9 4F CF B4 15 DB E9 5F 40 8B 9C E9 1E E8 46 ED
```

• $H() = \text{SHA-1}()$, $m$ = “hello world”
• altogether 35 bytes
• DER encoded object is a tree of $<T,L,V>$ triplets
PKCS#1 v1.5 Signature Scheme

- Encoded AS looks like this:

```
30 21
30 09 06 05 2B 0E 03 02 1A 05 00 04 14 2A AE 6C
35 C9 4F CF B4 15 DB E9 5F 40 8B 9C E9 1E E8 46 ED
```

- $H() = \text{SHA-1}()$, $m = \text{“hello world”}$
- altogether 35 bytes
- DER encoded object is a tree of $<T,L,V>$ triplets
PKCS#1 v1.5 Signature Scheme

- Encoded AS looks like this:

```
T 30 21 T 30 09 V 06 05 2B 0E 03 02 1A 05 00
  04 14 2A AE 6C 35 C9 4F CF B4 15
  DB E9 5F 40 8B 9C E9 1E E8 46 ED
```

- $H() = \text{SHA-1()}$, $m = \text{“hello world”}$
- altogether 35 bytes
- DER encoded object is a tree of $<T,L,V>$ triplets
PKCS#1 v1.5 Signature Scheme

- Encoded AS looks like this:

```
30 21 09 06 05 2B 0E 03 02 1A 05 00 04 14 2A AE 6C 35 C9 4F CF B4 15 DB E9 5F 40 8B 9C E9 1E E8 46 ED
```

- \( H() = \text{SHA-1}() \), \( m = \) “hello world”
- altogether 35 bytes
- DER encoded object is a tree of \(<T,L,V>\) triplets
PKCS#1 v1.5 Signature Scheme

- Encoded AS looks like this:

  ![Diagram of DER encoded object]

  - $H() = \text{SHA-1}(\text{m})$, $m = \text{“hello world”}$
  - altogether 35 bytes
  - DER encoded object is a tree of $<T,L,V>$ triplets
PKCS#1 v1.5 Signature Scheme

- Encoded AS looks like this:

```
H() = SHA-1(), m = “hello world”
altogether 35 bytes
DER encoded object is a tree of <T,L,V> triplets
```
PKCS#1 v1.5 Signature Scheme

- Given \((S, m, e, n)\), verifier computes \(H(m)\) and \(r = S^e \mod n\)

\[
m
\]
\[
H(m)
\]
\[
? \quad \Rightarrow \quad \text{Check if } r \text{ is well-formed?}
\]
\[
r = S^e \mod n
\]
\[
k^d
\]
\[
\Rightarrow \quad \text{All kinds of leniencies ...}
\]
\[
k^d \mod \beta
\]
RSA and Factorization

Given \((e,n)\) can we find \(d\)?

- \(ed = 1 \pmod{\phi(n)}\)
  - \(d\) is the multiplicative inverse of \(e\) mod \(\phi(n)\)
  - if we know \(\phi(n) = (p-1)(q-1)\) then easy to find \(d\) (use Extended Euclidean Algorithm)
- if we know \(n = pq\) then easy to find \(\phi(n)\)

**RSA-640**

RSA-640 has 640 bits (193 decimal digits). A cash prize of US$20,000 was offered by RSA Security for a successful factorization. On November 2, 2005, F. Bahr, M. Boehm, J. Franke, and T. Kleinjung of the German Federal Office for Information Security announced that they had factorized the number using GNFS as follows:

\[
\begin{align*}
\text{RSA-640} &= 310741824049804372313597500585885679390373460228427275457 \\
&
28161948232366405188195456346829671723286782437916272 \\
&
03983341547107310956191954832908733772426278352374230645 \\
&
4814697130692477053434669
\end{align*}
\]

The computation took five months on 80 2.2 GHz AMD Opteron CPUs. The slightly larger RSA-200 was factored in May 2005 by the same team.

**RSA-200**

RSA-200 has 200 decimal digits (663 bits), and factors into the two 100-digit primes given below. On May 9, 2005, F. Bahr, M. Boehm, J. Franke, and T. Kleinjung announced that they had factorized the number using GNFS as follows:

\[
\begin{align*}
\text{RSA-200} &= 279978393112113278788294673877226016121879446786955428537576506992932612840108 \\
&
7609345671952953308056618223519189513658786373809544820665767750985800557613 \\
&
57998973459591441786831379462945187237896221823983
\end{align*}
\]

The CPU time spent on finding these factors by a collection of parallel computers amounted — very approximately — to the equivalent of 75 years work for a single 2.2 GHz Opteron-based computer. Note that while this approximation serves to suggest the scale of the effort, it leaves out many complicating factors; the announcement states it more precisely.
Bleichenbacher’s low exponent attack

- Yet another crypto attack attributed to D. Bleichenbacher
- CRYPTO 2006 rump session
- Some implementations accept malformed \( r' \)

\[
\begin{array}{cccccc}
0x00 & \text{BT} & \text{PB} & 0x00 & \text{AS} & \text{Garbage}
\end{array}
\]

- Existential forgery possible when \( e \) is small
- Generate signatures for some \( m \) without \( d \)
Bleichenbacher’s low exponent attack

- A contributing factor to the push for bigger $e$ (e.g. 65537)
- Smaller $e$ more efficient for signature verifier
- $e = 3$ prescribed in some protocols e.g. DNSSEC [RFC3110, Sect. 4]

4. Performance Considerations

General signature generation speeds are roughly the same for RSA and DSA [RFC2536]. With sufficient pre-computation, signature generation with DSA is faster than RSA. Key generation is also faster for DSA. However, signature verification is an order of magnitude slower with DSA when the RSA public exponent is chosen to be small as is recommended for KEY RRs used in domain name system (DNS) data authentication.

A public exponent of 3 minimizes the effort needed to verify a signature. Use of 3 as the public exponent is weak for confidentiality since, if the same data can be collected encrypted under three different keys with an exponent of 3 then, using the Chinese Remainder Theorem [NETSEC], the original plain text can be easily recovered. If a key is known to be used only for authentication, as is the case with DNSSEC, then an exponent of 3 is acceptable. However other applications in the future may wish to leverage DNS distributed keys for applications that do require confidentiality. For keys which might have such other uses, a more conservative choice would be 65537 (F4, the fourth fermat number).
Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard
PKCS #1

Efficient Ciphertext-Only Attacks on Cryptographic Hardware

Return Of Bleichenbacher’s Oracle Threat (ROOT)

The 9 Lives of Bleichenbacher’s CAT: New Cache ATtacls on Implementations
A little brain teaser

What is common among these protocols?

- They all have RFCs
- They're all security-critical
- They all can benefit from PKCS#1 v1.5 RSA signatures
Why was the attack possible?

• Problem: verifier accept malformed input w/ GARBAGE unchecked
  • Can be in many different locations, not only at the end

• Longer modulus makes forgery easier
  • More GARBAGE bits to use
  • Can handle longer hashes / slightly larger e
To find these attacks

• Want to see how input bytes are being checked

• If enough unchecked GARBAGE then
Automatically generate concolic test cases

- Observation: size of components exhibit linear relations
  - e.g. \( \sum \text{length}(\text{components}) = |n| \); ASN.1 DER
- Programmatically capture such linear constraints
- Ask Symbolic Execution to find satisfiable solutions

- Based on that, automatically pack symbolic/concrete components into test buffers
Testing with Symbolic Execution

- Symbolic Execution with concolic test cases
- Very useful abstraction
  - What and how things are being checked in code?
- Formulas can help cross-validate implementations
Finding root causes

• Locate the piece of code that imposes wrong constraints
• Can we go from formula abstraction back to code?
• Constraint Provenance Tracking
  • Keep a mapping of \langle\text{clause, source-level origin}\rangle
  • Function filtering, e.g. `memcmp()`
  • Tiny space & time overhead
## Implementations Tested

<table>
<thead>
<tr>
<th>Name - Version</th>
<th>Overly lenient</th>
<th>Practical exploit under small e</th>
</tr>
</thead>
<tbody>
<tr>
<td>axTLS - 2.1.3</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>BearSSL - 0.4</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>BoringSSL – 3112</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Dropbear SSH – 2017.75</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>GnuTLS – 3.5.12</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>LibreSSL – 2.5.4</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>libtomcrypt – 1.16</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>MatrixSSL – 3.9.1 (Certificate)</td>
<td>YES</td>
<td>No</td>
</tr>
<tr>
<td>MatrixSSL – 3.9.1 (CRL)</td>
<td>YES</td>
<td>No</td>
</tr>
<tr>
<td>mbedTLS – 2.4.2</td>
<td>YES</td>
<td>No</td>
</tr>
<tr>
<td>OpenSSH – 7.7</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>OpenSSL – 1.0.2l</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Openswan – 2.6.50 *</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>PuTTY – 0.7</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>strongSwan – 5.6.3 *</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>wolfSSL – 3.11.0</td>
<td>No</td>
<td>-</td>
</tr>
</tbody>
</table>

Discussion of signature forgery assumes $e = 3$ and SHA-1, attacks also applicable to newer hash algorithms.

* configured to use their own internal implementations of PKCS#1 v1.5
Leniency in Openswan 2.6.50

- Ignoring padding bytes [CVE-2018-15836]
- Simple oversight, severe implications
  - Exploitable for signature forgery
- Use this r'

\[
\text{Want: } (a + b)^3 = a^3 + 3a^2b + 3b^2a + b^3, \text{ s.t.}
\]
  - MSBs of \( a^3 \) give what is before GARBAGE
  - LSBs of \( b^3 \) give what is after GARBAGE
  - \( (\text{LSBs of } a^3) + 3a^2b + 3b^2a + (\text{MSBs of } b^3) \) stay in GARBAGE
  - fake signature \( S' = (a+b) \)

```c
/* check signature contents */
/* verify padding (not including any DER digest info) */
padlen = sig_len - 3 - hash_len;
... ...

/* skip padding */
if(s[0] != 0x00 || s[1] != 0x01 || s[padlen+2] != 0x00)
  return "3""SIG padding does not
s += padlen + 3;
```
New unit test in Openswan

wo#7449 . test case for Bleichenbacher-style signature forgery

Special thanks to Sze Yiu Chau of Purdue University (schau@purdue.edu) who reported the issue, and made major contributions towards defining this test case.

master (#330)  v2.6.51.2  ...  v2.6.50.1

bartman committed on Aug 20

1 parent 9ea8c2

Showing 6 changed files with 218 additions and 0 deletions.

1 tests/unit/llbopenswan/Makefile

@@ -23,6 +23,7 @@ clean check:

  23  23
  24  24
  25  25
  26 +  @$(MAKE) -C lo04-verifypubkeys $@
  27
  28  @$(MAKE) -C lo05-datatot $@
  29  @$(MAKE) -C lo06-verifybadsigs $@
  30
  31 +  @$(MAKE) -C lo07-bleichenbacher-attack $@
Leniency in strongSwan 5.6.3

1. Not checking `AlgorithmParameter` [CVE-2018-16152]
   - classical flaw found in GnuTLS, Firefox years ago
   - Exploitable for signature forgery
     - hide GARBAGE in `AlgorithmParameter`
     - follow the Openswan attack algorithm
       - adjust what $a^3$ and $b^3$ represent, fake signature $S' = (a+b)$
2. Accept trailing bytes after Algorithm OID [CVE-2018-16151]
   • interestingly, **Algorithm OID** is not matched exactly
   • a variant of longest prefix match

```
/* [AlgorithmIdentifier] */
30 09
  06 05 2B 0E 03 02 1A
  05 00

/* [AlgorithmIdentifier] */
30 0C
  06 08 2B 0E 03 02 1A
  AB CD EF
  05 00
```

both would be recognized as OID of SHA-1

• knowing this, one can hide **GARBAGE** there
  • follow the Openswan attack algorithm
  • adjust what $a^3$ and $b^3$ represent, **fake signature** $S' = (a+b)$
3. Accepting less than 8 bytes of padding
   • Can be used to make the other attacks easier
     • Use no padding, gain more bytes for GARBAGE
strongSwan Security Update

**Version 5.3.5-1ubuntu3.7:**

* SECURITY UPDATE: Insufficient input validation in gmp plugin
  - debian/patches/strongswan-5.3.1-5.6.0_gmp:pkcs1-verify.patch: don't parse PKCS1 v1.5 RSA signatures to verify them in
    src/libstrongswan/plugins/gmp/gmp_rsa_private_key.c,
    src/libstrongswan/plugins/gmp/gmp_rsa_public_key.c.
  - CVE-2018-16151
  - CVE-2018-16152
Some key generation programs still forces e=3

- e.g., ipsec_rsasigkey on Ubuntu

**NAME**

ipsec_rsasigkey - generate RSA signature key

**SYNOPSIS**

```
ipsec rsasigkey [--verbose] [--seeddev device] [--seed numbits] [--nssdir nssdir]
                   [--password nsspassword] [--hostname hostname] [nbits]
```

**DESCRIPTION**

`rsasigkey` generates an RSA public/private key pair, suitable for digital signatures, of (exactly) `nbits` bits (that is, two primes each of exactly `nbits/2` bits, and related numbers) and emits it on standard output as ASCII (mostly hex) data. `nbits` must be a multiple of 16.

The public exponent is forced to the value 3, which has important speed advantages for signature checking. Beware that the resulting keys have known weaknesses as encryption keys and should not be used for that purpose.
Leniency in axTLS 2.1.3

1. Accepting trailing GARBAGE [CVE-2018-16150]
   • original Bleichenbacher '06 forgery also works
Leniency in axTLS 2.1.3

2. Ignoring prefix bytes

```c
i = 10;
/* start at the first possible non-padded byte */
while (block[i++] && i < sig_len);
size = sig_len - i;
/* get only the bit we want */
if (size > 0) {... ...}
```

- First 10 bytes are not checked at all
Leniency in axTLS 2.1.3

2. Ignoring prefix bytes
   • First 10 bytes directly skipped
   • Make forgery easier, use this r' (first 90 bits are all zeros)
     
     ```
     /** all numbers below are hexadecimals **/
     00 00 00 00 00 00 00 00 00 00 00 30 21 ... ... 04 16 SHA-1(m') GARBAGE
     ```
   • Reduce the distance between two consecutive cubes
     • Easier to find S’

   • roughly 19 bits < b^3, so ~ 2^19 trials to find S'
Leniency in axTLS 2.1.3

3. Ignoring AS.AlgorithmIdentifier [CVE-2018-16253]

```c
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 21
/* [AlgorithmIdentifier] */
30 09
  06 05 2B 0E 03 02 1A
  05 00
/* [Digest] */
04 14
  /* H(m), H()=SHA-1(), m = "hello world" */
  2A AE 6C 35 C9 4F CF B4 15 DB
  E9 5F 40 8B 9C E9 1E E8 46 ED
```

- Probably because certificates have an explicit signature algorithm field, which gives $H()$
3. Ignoring AS.AlgorithmIdentifier [CVE-2018-16253]
   • Just because H() is known from outside
   • Doesn’t mean it can be skipped

   • Use this r’:
     ```
     /** all numbers below are hexadecimals **/
     00 01 FF FF FF FF FF FF FF FF FF FF 00
     30 5D 30 5B GARBAGE 04 16 SHA-1(m’)
     ```
   • hide GARBAGE in AlgorithmIdentifier
   • follow the Openswan attack algorithm
     • adjust what a³ and b³ represent, fake signature S’ = (a+b)
4. Trusting the declared ASN.1 DER lengths w/o sanity checks [CVE-2018-16149]

/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w
/* [AlgorithmIdentifier] */
30 x
06 u 2B 0E 03 02 1A
05 y
/* [Digest] */
04 z

/** H(m), H()=SHA-1(), m = "hello world" */
2A AE 6C 35 C9 4F CF B4 15 DB
E9 5F 40 8B 9C E9 1E E8 46 ED

• DoS PoC: making z exceptionally large crashed the verifier
Leniency in axTLS 2.1.3

4. Trusting the declared ASN.1 DER lengths w/o sanity checks [CVE-2018-16149]
   • DoS PoC: making z exceptionally large crashed the verifier
   • Particularly damaging
   • axTLS does certificate chain validation bottom-up
   • Even if no small $e$ in the wild
     • any MITM can inject a fake certificate with $e = 3$
     • crash verifier during chain traversal
patching axTLS

axTLS Embedded SSL
Brought to you by: cameronrich

Download Latest Version
axTLS-2.1.5.tar.gz (1.3MB)

Files
- 2.1.5 (2019-03-15, 15 downloads/week)
- 2.1.4 (2017-08-31, 10 downloads/week)
- 2.1.3 (2017-02-16, 9 downloads/week)
- 2.1.2 (2016-12-30, 0 downloads/week)
Leniency in libtomcrypt 1.16

1. Accepting trailing GARBAGE
   • original Bleichenbacher '06 forgery also works

2. Accepting less than 8 bytes of padding
   • Use no padding, gain more bytes for GARBAGE
     • Make signature forgery easier

• Flaws independently found by other researchers, fixed in v1.18
Leniency in MatrixSSL 3.9.1 (CRL)

1. Mishandling Algorithm OID

```c
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w
    /* [AlgorithmIdentifier] */
    30 x
        06 u 2B 0E 03 02 1A
        05 y
    /* [Digest] */
    04 z
        /* H(m), H()=SHA-1(), m = "hello world" */
        2A AE 6C 35 C9 4F CF B4 15 DB
        E9 5F 40 8B 9C E9 1E E8 46 ED
```

- Some bytes in the middle of `AS` can take any values
  - Depends on choice of $H()$, SHA-1: 5 bytes, SHA-256: 9 bytes
- Doesn’t seem to be numerous enough for practical attacks
Other leniencies

- Lax checks on ASN.1 DER lengths in MatrixSSL (CRL)

```cpp
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 \(w\)
/* [AlgorithmIdentifier] */
30 \(x\)
06 u 2B 0E 03 02 1A
05 \(y\)
/* [Digest] */
04 \(z\)
/* H(m), H()=SHA-1(), m = "hello world" */
2A AE 6C 35 C9 4F CF B4 15 DB
E9 5F 40 8B 9C E9 1E E8 46 ED
```

- Some bits in the middle of AS can take any values
- Doesn’t seem to be numerous enough for practical attacks
- Variants of this leniency also found in mbedTLS, libtomcrypt, MatrixSSL (Certificate)
MatrixSSL 4.x changelog

Changes between 4.0.0 and 4.0.1 [November 2018]

This version improves the security of RSA PKCS #1.5 signature verification and adds better support for run-time security configuration.

- Crypto:
  - Changed from a parsing-based to a comparison-based approach in DigestInfo validation when verifying RSA PKCS #1.5 signatures. There are no known practical attacks against the old code, but the comparison-based approach is theoretically more sound. Thanks to Sze Yiu Chau from Purdue University for pointing this out.
  - (MatrixSSL FIPS Edition only) Fix DH key exchange when using DH parameter files containing optional privateValueLength argument.
  - psX509AuthenticateCert now uses the common psVerifySig API for signature verification. Previously, CRLs and certificates used different code paths for signature verification.
Summary

• RSA signature verification should be robust regardless of the choice of $e$
  • Flawed verification can break authentication in different scenarios

• To analyze this, we extend symbolic execution with
  • Automatic generation of concolic test cases
  • Constraint Provenance Tracking

• Found new variants of Bleichenbacher '06 attacks after more than a decade, 6 new CVEs
  • And some other unwarranted leniencies
Lessons Learned

• Corner-cutting is not cool
• Parsing is hard
• Learn from previous mistakes