



ekoparty 10



BERserk: New RSA Signature Forgery Attack

Andrew Furtak, Yuriy Bulygin, Oleksandr Bazhaniuk, John Loucaides,
Alex Matrosov, Mikhail Gorobets



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
```

Fix (Well.. one of)

```
(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
```

Make sure there's
no extra data left
after the
DigestInfo | Hash

Wait! Parsing DigestInfo!?

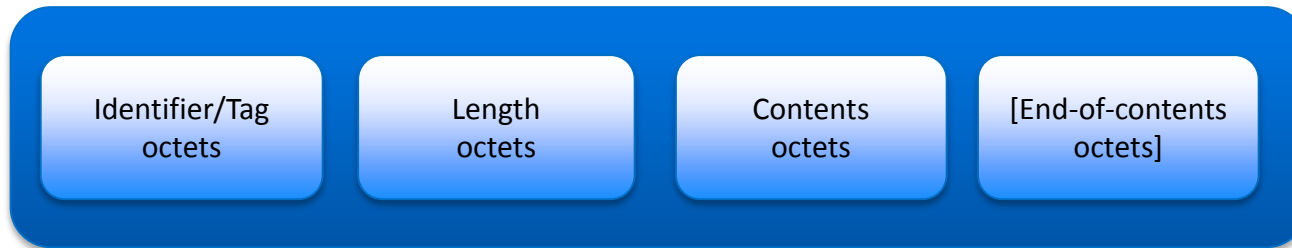
```
RSA_BER_Parse_DigestInfo( T, T_size )
```

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

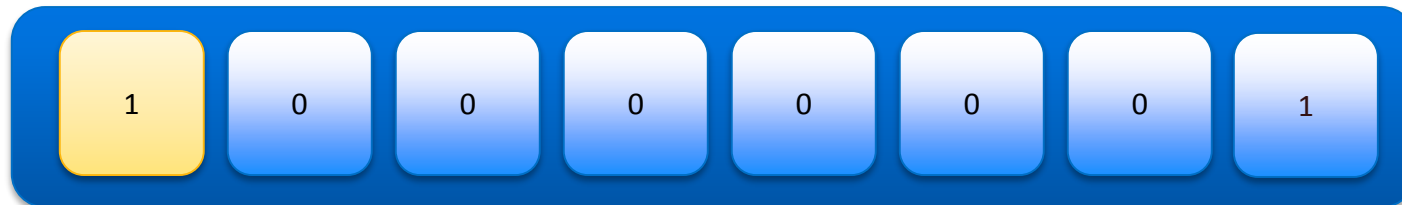
```
30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20
```

BER/DER Encoding of ASN.1 Lengths

BER Structure



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"

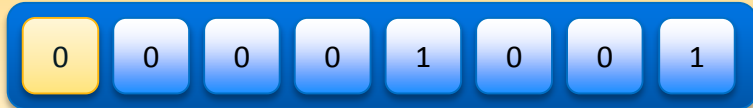
BER vs DER

- 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



Valid BER, Valid DER



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 XXXXXXXXXXXXXXXXXXXXXXXXXX

Tag Length
30 (SEQUENCE) 31

Tag Length
30 (SEQUENCE) 0d

Tag Length
06 (OID) 09

Tag Length
06 (OID) 09

OID

60 86 48 01 65 03 04 02 01

Tag Length
05 (NULL) 00

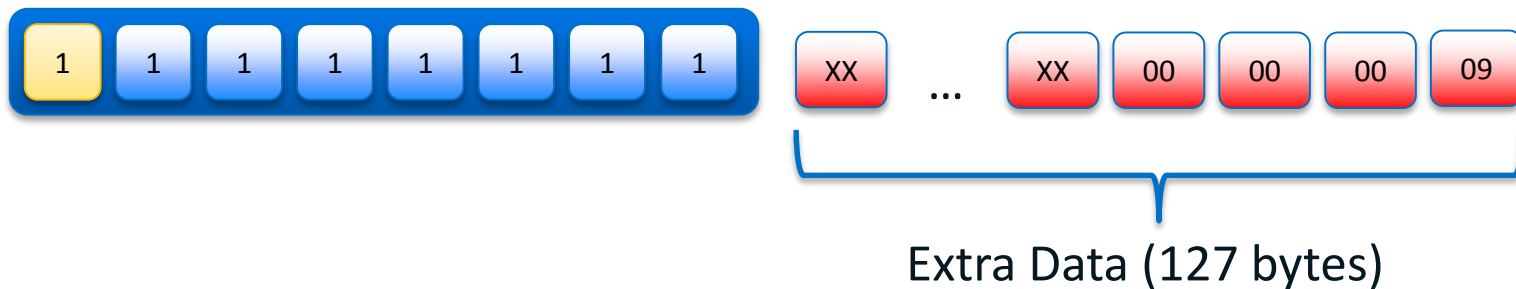
Tag Length
04 (OCTET STRING) 20

Tag Length
04 (OCTET STRING) 20

octet string (SHA-256 hash)
XXXXXXXXXXXXXXXXXXXXXXX

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 ..  
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
Tag          Length (long form)  
30 (SEQUENCE) 32
```

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

```
Tag          Length  
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)  
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Adding Extra Data in DigestInfo

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

“ .. “ Describes extra “Garbage” data

```

00 01 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
  
```

Correct PKCS#1 v1.5 with DER DigestInfo

```

FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20
48 AC E9 B7 BC 30 CB 37 33 84 19 F7 71 6D 4E 9F
50 AA 0A D2 A4 25 BC F3 8C 2A 11 66 9F 85 CF D5
  
```

```

00 01 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 .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. 04
FF .. .. .. .. .. .. .. .. .. .. .. .. .. ..
  
```

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

```

.. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
48 AC E9 B7 BC 30 CB 37 33 84 19 F7 71 6D 4E 9F
50 AA 0A D2 A4 25 BC F3 8C 2A 11 66 9F 85 CF D5
  
```

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 (Cont'd)

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

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

- Finding fixed octets in the middle part:
 1. If fixed octets are adjacent to Prefix or Suffix parts, \mathbf{m} can be solved as part of calculating Prefix or Suffix
 2. If number of fixed octets is small, \mathbf{m} can be found by exhaustive search (by incrementing bytes of \mathbf{m} above 1)
 3. \mathbf{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	32	cb	fd	4a	7a
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	2c	7a
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')

```
00000000 00 01 ff ff ff ff ff ff ff 00 30 31 30 0d 06
00000010 09 60 86 48 01 65 03 04 02 01 05 c3 68 9b 67 e3
00000020 6c 25 a4 a2 f3 23 2d 63 cf af 1a 19 f0 d4 a8 78
00000030 b9 bf b9 12 fd 1c 00 95 74 f1 1f ed 69 23 14 46
00000040 f6 c2 aa b2 21 54 7e ce 2f 18 8d 5f 45 bb d6 cd
00000050 ad 25 06 50 98 68 11 b9 2a a1 0b b8 ca 7d 59 04
00000060 ff 4f da 00 db 7f 2a c3 39 a0 ff ca ba ca 6f d8
00000070 a2 19 3f 6b 42 07 9d 11 58 fc 59 7d 51 d7 08 98
00000080 42 5a f8 92 16 ee 07 8b 5b 9a 6d c5 f8 00 80 d5
00000090 b1 a3 47 56 b2 dd c6 d6 5c 13 98 4d bf 03 ad b0
000000a0 32 f8 8b 4d 5b 40 b7 ef 8f fc 4d 6b e3 e1 bb 1f
000000b0 58 a8 a3 41 55 22 00 84 4c b0 eb 26 9f 64 a6 28
000000c0 4b a9 a5 62 a5 6a ae ef 00 f3 a9 d2 3d 6b f8 83
000000d0 2b e1 86 55 22 55 16 f4 3d 88 7c 74 b5 df a4 b2
000000e0 48 ac e9 b7 bc 30 cb 37 33 84 19 f7 71 6d 4e 9f
000000f0 50 aa 0a d2 a4 25 bc f3 8c 2a 11 66 9f 85 cf d5
```

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

```
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;  
}
```

Verifies no extra
data left after
decoding ASN.1
sequence

How Many Bytes Can BER Length Have?

```
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 unsigned integer

len_count can be up to 127

What about other 123 bytes?

Malformed ASN.1 DigestInfo (SHA-1)

```
30 db .. garbage .. 00 00 00 a0 30 ff .. garbage .. 00 00 00 09 06 05 2b 0e 03 02 1a
05 00 04 14 XXXXXXXXXXXXX
```

```
Tag          Length (long form)
30 (SEQUENCE) db (80|5b) .. garbage .. 00 00 00 a0
```

```
Tag          Length (long form)
30 (SEQUENCE) ff (80|7f) .. garbage .. 00 00 00 09
```

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

```
OID
2b 0e 03 02 1a
```

```
Tag          Length
05 (NULL)    00
```

```
Tag          Length
04 (OCTET STRING) 14
```

```
octet string (the SHA1 hash)
XXXXXXXXXXXX
```


Conclusions / Recommendations

- “BERSerk” is not one bug but rather a class of bugs when parsing BER lengths. Each implementation is different
- Parsing DigestInfo as ASN.1 is a bad idea. There are just a few short DigestInfo sequences that an implementation needs to compare with
- Detecting forged PKCS#1 v1.5 signatures:
 - Does the signature have a bunch of zeros? → Suspicious!
 - More precise, does $(\text{signature})^3 < \text{RSA modulus}$ for any certificate in the chain? → Forged certificate!

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](#)

Thank You!

Many thanks to Francisca Moreno and Charles McFarland for help with the presentation!