Or why won't BrainpoolP384r1 work with NSS?

MiWCryptoCurrency@gmail.com   January 31st 2015

Summary

After three weeks of studying the freebl softtoken component (specifically the ECL libraries),
I have found a long standing bug in the ec_GFp_sub_6 function that causes it to return incorrect values while testing the brainpoolP384r1 curve.
This bug was introduced in 3.11.1 as optimised prime Galois Field arithmetic functions.

Additionally, this off-by-one implementation of Algorithm 2 from (Brown, Hankerson, J.Lopez, & Menezes, 2000) also exists in the ec_GFp_sub_5 function.
In the case of prime256v1 (Suite-B P256), there are additional routines that use curve optimised functions instead of the ec_GFp_sub_5.
This would prevent this from introducing arithmetic errors under P256.

The Suite-B curve P384 does call ec_GFp_sub_6 for its field arithmetic and but is not directly impacted by this bug.
Practically, the ec_GFp_sub_6 function fails to add the most significant 64 bit word from the prime to the result in the event of sub_borrow carry.

In practice this does not trigger for P384 because p is:

```
p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFF0000000000000000FFFFFFFF
x is a 64 bit integer
x + FFFFFFFFFFFFFFFF = x
```

The result is NSS will fail to verify a correct signed digest on the ServerKeyExchange Record and fail the handshake, or fail the public key validity check when generating an ECDHE key
In firefox this will manifest as SSL error SEC_ERROR_BAD_SIGNATURE in the case of ECDSA and SEC_ERROR_BAD_KEY in the case of ECDHE.

This bug will occur in a 384 bit curve when the most significant 64 bit word of prime is not `FFFFFFFFFFFFFFFF` like in brainpool curves.

Motivation

I have been porting the brainpool set of curves standardized in TLS to NSS, to offer diversity in choice of curves. Presently, most cryptographic products only support the NIST/NSA Suite-B curves – secp256r1, secp384r1 and secp521r1. Legal/Political issues surrounded the removal of the 22 alternative NIST and SECg curves. Newer 'safer' curves are set to be introduced in the future, such as Curve25519, will require additional rework to NSS as this does not support ECDSA or standard public key validation routine as below.

The brainpool curves have been assigned named_curve id's 26, 27 and 28 in www.iana.org/assignments/tls-parameters/tls-parameters.xhtml and can be used in a standard TLS implementation.
OpenSSL 1.0.2 released January 2015 includes these curves.

Adding the new curves was effectively plumbing; the codebase allowed addition of short Weierstraß form curves with relative ease.
Other example patches on Mozilla Bugzilla provided hints on where switch cases or definitions needed to be added.

After firefox was compiled with the additional curve parameters, it was able to handle certificates with public keys defined over the 25 TLS standard curves, and generate keys used for ECDHE on these curves.

Issue

Extending the ClientHello to send 28 curves worked with the brainpool curves at 256 & 512 bits; but would always fail with a signature error in the client.

The first instance of failure would occur when NSS was trying to verify the signature of the ServerKeyExchange record.

The call stack would look as follows:

```
freebl3.dll!ec_GFp_validate_point
freebl3.dll!ECPoint_validate
freebl3.dll!EC_ValidatePublicKey
softokn3.dll!EC_ValidatePublicKey
softokn3.dll!sftk_handlePublicKeyObject
softokn3.dll!sftk_handleKeyObject
softokn3.dll!sftk_handleObject
softokn3.dll!NSC_CreateObject
nss3.dll!PK11_CreateNewObject
nss3.dll!PK11_ImportPublicKey
nss3.dll!PK11_Verify
nss3.dll!vfy_VerifyDigest
nss3.dll!VFY_VerifyDigestDirect
nss3.dll!ssl3_VerifySignedHashes
nss3.dll!ssl3_HandleECDHServerKeyExchange
nss3.dll!ssl3_HandleServerKeyExchange
```

The signature verify routine would import the public key from the certificate, and validate if it is a correctly formed and useable for checking the record signed hash.

```
nss/lib/freebl/ecl/ecp_aff.c
ec_GFp_validate_point(const mp_int *px, const mp_int *py, const ECGroup *group)
```

This function performs 4 checks with point (px,py) defined on the group:

- `Verify that publicValue is not the point at infinity`

(px,py) != point at infinity (which is 0,0 in NSS notation).
checks that px != 0, py !=0

- `Verify that the coordinates of publicValue are elements the field.`

px is not negative, px <= prime, py is not negative, py <= prime

- `Verify that publicValue is on the curve.`

In the equation of the curve, given in short weistrass form, calculate value of both sides.
Check that LHS – RHS = 0

- `Verify that the order of the curve times the publicValue is the point at infinity.`

Calculate and verify point multiplication: order * (px, py) = 0

If conditions 1, 2, and 3 are met, 4 is implied. That is to say if point parameters (x,y) are on the curve, and are not the point at infinity, the order of the curve point multiplied by the point must be the point at infinity.

If this test fails there must be something very badly defined with the curve parameters.

These are the same 4 tests are specified in (Antipa, Brown, Menezes, Struik, & Vanstone, 2003)

Now, this last test would always fail on brainpoolP384r1.
Parameters for this were compared again to the RFC and openssl implementations, to find they were correctly specified.
This last step in the verify routine executes

```
MP_CHECKOK( ECPoint_mul(group, &group->order, px, py, &pxt, &pyt) )
```

which should return 0 and set pxt and pyt to the scalar multiplication of curve order * point(px, py).
For all 27 other curves excepting brainpoolP384r1, this would leave pxt=0, and pyt=0, as expected, verifying the public key supplied on the certificate was valid.

My test parameters are as follows:

OpenSSL 1.0.2 generated the follow certificate(s) containing a brainpoolP384r1 EC public key and secp256k1 root certificate.

```
Root CA:
-----BEGIN CERTIFICATE-----
MIICwDCCAmWgAwIBAgIJAMbMIMTFqrAuMAoGCCqGSM49BAMCMIG2MQswCQYDVQQG
EwJVUzETMBEGA1UECAwKQ2FsaWZvcm5pYTEWMBQGA1UEBwwNTW91bnRhaW4gVmll
dzENMAsGA1UECgwEVEVTVDEfMB0GA1UECgwWU2VsbGluZyByYW5kb20gbnVtYmVy
czEQMA4GA1UECwwHVGVzdGluZzE4MDYGA1UEAwwvVW51c3VhbCBDZXJ0aWZpY2F0
ZSBUZXN0IFJvb3QgQ0EgKERvIE5vdCBUcnVzdCkwHhcNMTUwMTE3MTAxMjI3WhcN
MTUwMjE2MTAxMjI3WjCBtjELMAkGA1UEBhMCVVMxEzARBgNVBAgMCkNhbGlmb3Ju
aWExFjAUBgNVBAcMDU1vdW50YWluIFZpZXcxDTALBgNVBAoMBFRFU1QxHzAdBgNV
BAoMFlNlbGxpbmcgcmFuZG9tIG51bWJlcnMxEDAOBgNVBAsMB1Rlc3RpbmcxODA2
BgNVBAMML1VudXN1YWwgQ2VydGlmaWNhdGUgVGVzdCBSb290IENBIChEbyBOb3Qg
VHJ1c3QpMFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEkOTRWhg0iMCYyYxCMY0opOdK
D/5HFt3Rrl/IoNiwajVfHJGUFwiVk4FCsOYZXUaudjX7pVw6A1QAw4nxY6nXZ6Nd
MFswHQYDVR0OBBYEFOCOIJy1Tw7+LRYbrzkir/+bBvfVMB8GA1UdIwQYMBaAFOCO
IJy1Tw7+LRYbrzkir/+bBvfVMAwGA1UdEwQFMAMBAf8wCwYDVR0PBAQDAgEGMAoG
CCqGSM49BAMCA0kAMEYCIQCBZrGyDN5gqRje3QNADi8uO85Zdeg1KNl0JMjKVgWH
dQIhAM3iRBYcxOYgMsBD5lSQYoKL3afGWf/fIBk2Ku4P541C
-----END CERTIFICATE-----

Server Certificate:
-----BEGIN CERTIFICATE-----
MIICVjCCAf0CAUUwCgYIKoZIzj0EAwIwgbYxCzAJBgNVBAYTAlVTMRMwEQYDVQQI
DApDYWxpZm9ybmlhMRYwFAYDVQQHDA1Nb3VudGFpbiBWaWV3MQ0wCwYDVQQKDARU
RVNUMR8wHQYDVQQKDBZTZWxsaW5nIHJhbmRvbSBudW1iZXJzMRAwDgYDVQQLDAdU
ZXN0aW5nMTgwNgYDVQQDDC9VbnVzdWFsIENlcnRpZmljYXRlIFRlc3QgUm9vdCBD
QSAoRG8gTm90IFRydXN0KTAeFw0xNTAxMDkwODA3MDJaFw0xNjAxMDkwODA3MDJa
MIGWMQswCQYDVQQGEwJVUzETMBEGA1UECAwKQ2FsaWZvcm5pYTEWMBQGA1UEBwwN
TW91bnRhaW4gVmlldzENMAsGA1UECgwEVEVTVDEfMB0GA1UECgwWU2VsbGluZyBy
YW5kb20gbnVtYmVyczEQMA4GA1UECwwHVGVzdGluZzEYMBYGA1UEAwwPZGViaWFu
LWRldi5ob21lMHowFAYHKoZIzj0CAQYJKyQDAwIIAQELA2IABBJKzf4E0DKN4Ym2
Ul8AdyZFjKDQBUhjJcvcayxPgO6A1Um3b5maAbkNAC4u1hSioBXff4oNbkPK2URp
JduccEiHw7pufAVyLRhQnAjmWo4TwPym/HCGNc0IWzlSNrHTWjAKBggqhkjOPQQD
AgNHADBEAiATZc59P4nxNFs72ByU3JhFba+hnyApTOgHXVL5qVtpuwIgSVljCCeG
p+CodlTLsNbzJLWM84HypHA7rUcjGJVeuM4=
-----END CERTIFICATE-----


Server Key:
-----BEGIN EC PRIVATE KEY-----
MIGoAgEBBDBNq9v+5u26ObxiAeJHAMordhdD0H7zL41x65vpzNsdqh1G6rGGRBVR
OjQTmxyeZu+gCwYJKyQDAwIIAQELoWQDYgAEEkrN/gTQMo3hibZSXwB3JkWMoNAF
SGMly9xrLE+A7oDVSbdvmZoBuQ0ALi7WFKKgFd9/ig1uQ8rZRGkl25xwSIfDum58
BXItGFCcCOZajhPA/Kb8cIY1zQhbOVI2sdNa
-----END EC PRIVATE KEY-----
```

The server public key corresponds to:
pub:
```
                    04:12:4a:cd:fe:04:d0:32:8d:e1:89:b6:52:5f:00:
                    77:26:45:8c:a0:d0:05:48:63:25:cb:dc:6b:2c:4f:
                    80:ee:80:d5:49:b7:6f:99:9a:01:b9:0d:00:2e:2e:
                    d6:14:a2:a0:15:df:7f:8a:0d:6e:43:ca:d9:44:69:
                    25:db:9c:70:48:87:c3:ba:6e:7c:05:72:2d:18:50:
                    9c:08:e6:5a:8e:13:c0:fc:a6:fc:70:86:35:cd:08:
                    5b:39:52:36:b1:d3:5a
```
On brainpoolP384r1

Or
(Least Significant Digit first – mpi (native NSS arbitrary length integer encoding, copied from memory):
```
px: a0 a2 14 d6 2e 2e 00 0d b9 01 9a 99 6f b7 49 d5 80 ee 80 4f 2c 6b dc cb 25 63 48 05 d0 a0 8c 45 26 77 00 5f 52 b6 89 e1 8d 32 d0 04 fe cd 4a 12
py: 5a d3 b1 36 52 39 5b 08 cd 35 86 70 fc a6 fc c0 13 8e 5a e6 08 9c 50 18 2d 72 05 7c 6e ba c3 87 48 70 9c db 25 69 44 d9 ca 43 6e 0d 8a 7f df 15
```

group = brainpoolP384r1 has:
(LSD first – mpi encoding)
order:65 65 04 e9 02 32 88 3b 10 c3 7f 6b af b6 3a cf a7 25 04 ac 6c 6e 16 1f b3 56 54 ed 09 71 2f 15 df 41 e6 50 7e 6f 5d 0f 28 6d 38 a3 82 1e b9 8c

The call result to ECPoint_mul(group, **group->order**, px, py, &pxt, &pyt) would result in:
(LSD first – mpi encoding):
```
pxt: e0 57 2c 68 65 e1 63 5c 9c 03 72 4a f4 18 2d f5 8b c3 9b 48 b2 ba a8 54 89 82 d7 aa 83 01 84 f0 7f c5 ff 08 fa 04 40 43 27 20 a8 37 2a 16 21 63
pxy: 1b 8a f6 48 6d 6c bd 5c 34 25 1a e7 63 7e 6c c7 fb 2f c5 64 b8 59 dc ef 0a b4 fd 3c 5b 0d 1a 95 4b 85 63 7c ab f4 12 3c 49 eb 07 ef 65 8d 71 11
```

The expected result is pxt: 00, pyt: 00

Thinking the problem might be with the point multiplication function, I changed the particular ECGroup from its default case of ECGroup_consGFp_mont to ECGroup_consGFp.

These group definitions provide slightly different implementations of the field arithmetic, the mont version performs these over Montgomery coordinates.
The generic version performs arithmetic on affine coordinate system.
Mont encoding offer an efficiency speedup as less operations need to occur to calculate the result compared to affine form. #REF

Curiously, it returned a different results depending on the coordinate arithmetic.

The group with affine coordinate arithmetic would result the following result to ECPoint_mul(group, **group->order**, px, py, &pxt, &pyt):
(LSD first – mpi encoding):
```
Pxt: 72 05 74 e8 bd ce cc f0 fb c8 24 25 ce 6c 61 f5 e0 c9 02 27 83 ef 0e b7 12 a1 33 ae c1 80 74 75 cb 63 9d d2 ab e3 0d 9d be f0 cd 7d 99 a7 b5 15
Pyt: 44 a0 2f 24 db 95 1b cc 0b 4e 6a 82 4a df 8b 37 84 f8 99 57 c6 72 14 c1 d8 d6 3d ee 1b 09 8f 9f 0b 85 69 9e 6a 79 e9 34 ce a4 2b 08 10 11 a1 50
```

It is expected that the coordinate system should not return different values for multiply if everything is well defined.

Rather than looking too deeply into the optimized generic point multiplication function 'multiplication on modified jacobian coordinates with in windowed Non Adjacent Form' (we shall revisit this): `ec_GFp_pt_mul_jm_wNAF`, I compiled the 'reference' affine multiplication function back into the source and defined a new group ECGroup_consGFp_aff and substituted the simpler affine multiplication above. #ifdef ECL_DEBUG was also used to create use /* basic double and add method */

In practice the affine multiplication function should not be used as there are more efficient algorithms that do not create so many power and time information side channels.

The `ec_GFp_pt_mul_aff(n, px, py, rx, ry, group)` performs the following (simplified) multiplication algorithm, where n is scalar, (px,py) are point, and (rx,ry)=n*(px,py) and group is collection of arithmetic parameters and functions:

```
copy (px, py) into (qx, qy), n into k
If n=0 then (rx, ry) = (0, 0) and return
If n<0 then (qx, qy) = -(qx,qy), k = -n

set L to number of significant bits in k -1
copy (qx, qy) into (sx, sy)

for I = L to 0:
      (sx, sy) = double( (sx, sy) )
      if (bit value at k[i] = 1) then
            (sx, sy) = (sx, sy) + (qx, qy)

copy (sx, sy) into (rx, ry) and return
```

Tracing execution of this function with the following parameters, we find at the first arithmetic function point double(sx,sy) returns an incorrect result:

(LSD first – mpi encoding):

```
sx:     a0 a2 14 d6 2e 2e 00 0d b9 01 9a 99 6f b7 49 d5 80 ee 80 4f 2c 6b dc cb 25 63 48 05 d0 a0 8c 45 26 77 00 5f 52 b6 89 e1 8d 32 d0 04 fe cd 4a 12
sy:     5a d3 b1 36 52 39 5b 08 cd 35 86 70 fc a6 fc c0 13 8e 5a e6 08 9c 50 18 2d 72 05 7c 6e ba c3 87 48 70 9c db 25 69 44 d9 ca 43 6e 0d 8a 7f df 15
k:      65 65 04 e9 02 32 88 3b 10 c3 7f 6b af b6 3a cf a7 25 04 ac 6c 6e 16 1f b3 56 54 ed 09 71 2f 15 df 41 e6 50 7e 6f 5d 0f 28 6d 38 a3 82 1e b9 8c
2sx:    c1 fa 33 d8 9b cf 85 fb 60 95 cd 42 c7 05 5f 1f 05 4e 9a 1c 44 50 d8 69 e6 aa 54 87 f1 fb 12 65 5d d1 e0 c7 63 08 54 c6 9f 00 d3 05 cb 84 4e f8
2sy:    2f 81 79 01 db 14 37 16 6e f4 f1 19 66 2e f1 30 02 38 3a 76 27 08 57 06 1b c7 5c 41 a9 a2 59 11 2b 04 75 f6 5e d3 13 df cf 67 59 59 68 fa a9 0d
```

Or standard MSD first, comparing with the correct answer from OpenSSL 1.0.2 ['incorrect point double result']:

```
NSS-P_x: 0xF84E84CB05D3009FC6540863C7E0D15D6512FBF18754AAE669D850441C9A4E051F5F05C742CD9560FB85CF9BD833FAC1
OSS_P_x: 0x8507A34DA90B6DC7C6540863C7E0D15D6512FBF18754AAE669D850441C9A4E051F5F05C742CD9560FB85CF9BD833FAC1
         0xoooooooooooooooo................................................................................
NSS-P_y: 0x0DA9FA68595967CFDF13D35EF675042B1159A2A9415CC71B06570827763A380230F12E6619F1F46E163714DB0179812F
OSS-P_y: 0x0DA9FA68595967CFDF13D35EF675042B1159A2A9415CC71B06570827763A380230F12E6619F1F46E163714DB0179812F
         0x................................................................................................
```

The 8 most significant bytes are incorrect, for the x point value only. Something component within the double function appears broken.

The point double function, or more specifically `ec_GFp_pt_dbl_aff(px, py, rx, ry, group)` calls `ec_GFp_pt_add_aff(px, py, px, py, rx, ry, group)`

The `ec_GFp_pt_add_aff(px, py, qx, qy, rx, ry, group)` function performs the (simplified) point addition algorithm, where (rx, ry) = (px, py) + (qx, qy), group contains curve parameter a for `y^2 = x^3+ax+b`

```
if (px, py) = (0,0) then (rx, ry) = (qx, qy) return

if (qx, qy) = (0,0) then (rx, ry) = (px, py) return

if px is not equal to qy, then set lambda = (py – qy ) / (px - qx)

else

      if py != qy OR qy = 0 then (rx, ry) = (0, 0) return

      lambda = (3*qx^2+a) / (2qy)

rx = lambda^2 – px – qx

ry = (px – qx) * lambda – y1

return
```

Breaking this down further, we try to find which field arithmetic operator is causing the incorrect result.

Most of this code is an implementation of (Brown, Hankerson, J.Lopez, & Menezes, 2000)

In our case px=py, so we want to determine if the

`lambda = (3*qx^2+a) / (2qy)` is the expected value. Following execution of each field operator in order, we find:

<u>Field Square function:</u>
(LSD first – mpi encoding):

```
qx:
a0 a2 14 d6 2e 2e 00 0d b9 01 9a 99 6f b7 49 d5 80 ee 80 4f 2c 6b dc cb 25 63 48 05 d0 a0 8c 45 26 77 00 5f 52 b6 89 e1 8d 32 d0 04 fe cd 4a 12
qx^2:
8a bc 6f 5b 77 82 b5 6f d1 2c 9f 3e 96 71 28 b2 f1 e0 6f b4 9a ee 93 cd a4 00 5f d5 af a5 4e 86 c1 39 ae e1 53 3f 72 01 11 99 c4 69 73 42 ff 75
```

OpenSSL answer (MSB first):
0x75FF427369C4991101723F53E1AE39C1864EA5AFD55F00A4CD93EE9AB46FE0F1B22871963E9F2CD16FB582775B6FBC8A

Correct.

<u>Field Multiply function:</u>
(LSD first – mpi encoding):

```
qx^2:
8a bc 6f 5b 77 82 b5 6f d1 2c 9f 3e 96 71 28 b2 f1 e0 6f b4 9a ee 93 cd a4 00 5f d5 af a5 4e 86 c1 39 ae e1 53 3f 72 01 11 99 c4 69 73 42 ff 75
3*qx^2:
f8 5c 3f b0 3f 87 92 40 91 51 a3 9b 6f 06 d2 bc 8d 80 e1 1d 9d 17 58 43 86 54 74 a5 fb 0e 8d 68 86 29 3e 03 ff de 9b e5 e2 f0 dc f6 54 8a 8b 48
```

OpenSSL answer (MSB first):

0x488B8A54F6DCF0E2E59BDEFF033E2986688D0EFBA57454864358179D1DE1808DBCD2066F9BA351914092873FB03F5CF8

Correct.

<u>Field Addition function:</u>
(LSD first – mpi encoding):

```
3*qx^2:
f8 5c 3f b0 3f 87 92 40 91 51 a3 9b 6f 06 d2 bc 8d 80 e1 1d 9d 17 58 43 86 54 74 a5 fb 0e 8d 68 86 29 3e 03 ff de 9b e5 e2 f0 dc f6 54 8a 8b 48
a:
99 3c 6c 46 95 b8 26 db 7b b0 57 f1 fe f3 d7 75 b4 0d f1 d7 b9 71 67 93 74 93 52 35 e5 e9 ff e7 60 0c b0 42 df 8f 0a 40 d1 c0 e8 a2 21 80 33 7c
3*qx^2+a:
cb 98 05 a2 09 4f f4 bd 0a 0c c1 5b 90 e0 a3 9a 79 68 bc 58 73 a3 37 44 59 25 d2 07 80 40 1c 16 48 97 5d 80 8b 77 b0 12 c7 98 30 91 98 ee 95 37
```

OpenSSL answer (MSB first):

0x3795EE98913098C712B0778B805D9748161C408007D225594437A37358BC68799AA3E0905BC10C0ABDF44F09A20598CB

Correct.

<u>Field Division function:</u>
(LSD first – mpi encoding):

```
3*qx^2+a
cb 98 05 a2 09 4f f4 bd 0a 0c c1 5b 90 e0 a3 9a 79 68 bc 58 73 a3 37 44 59 25 d2 07 80 40 1c 16 48 97 5d 80 8b 77 b0 12 c7 98 30 91 98 ee 95 37
2qy:
b4 a6 63 6d a4 72 b6 10 9a 6b 0c e1 f8 4d f9 81 27 1c b5 cc 11 38 a1 30 5a e4 0a f8 dc 74 87 0f 91 e0 38 b7 4b d2 88 b2 95 87 dc 1a 14 ff be 2b
lambda:
92 e2 b4 bb 46 29 32 47 f7 84 41 0b 30 a9 5d 15 55 45 03 64 56 f9 2e 67 8c b8 0e 86 b6 c5 1c bb 0b 23 05 e6 8f 29 73 85 37 45 c9 42 73 52 a9 12
```

OpenSSL answer (MSB first):

0x12A9527342C945378573298FE605230BBB1CC5B6860EB88C672EF95664034555155DA9300B4184F747322946BBB4E292

Correct.

Field Square function:

Calculating rx, we assume lambda^2 is correct from above. NSS returns expected result.

Field Subtraction function (-px):
(LSD first – mpi encoding):

```
Lambda^2:
ae 53 55 53 e6 2b 3f 8e 61 7e e4 e5 7c cd 1e 1d e3 19 e5 3b 83 4c df ee 7d 1a 91 a4 87 cc fc da ca 7d fb 34 8a 05 0a 7a bb 65 73 0f c7 20 e4 1c
px:
a0 a2 14 d6 2e 2e 00 0d b9 01 9a 99 6f b7 49 d5 80 ee 80 4f 2c 6b dc cb 25 63 48 05 d0 a0 8c 45 26 77 00 5f 52 b6 89 e1 8d 32 d0 04 fe cd 4a 12
Lambda^2-px
0e b1 40 7d b7 fd 3e 81 a8 7c 4a 4c 0d 16 d5 47 62 2b 64 ec 56 e1 02 23 58 b7 48 9f b7 2b 70 95 a4 06 fb d5 37 4f 80 98 2d 33 a3 0a c9 52 99 0a
```

OpenSSL (MSB first):

0x0A9952C90AA3332D98804F37D5FB06A495702BB79F48B7582302E156EC642B6247D5160D4C4A7CA8813EFDB77D40B10E

Correct.

Field Subtraction function (-qx):
(LSD first – mpi encoding):

```
Lambda^2-px:
0e b1 40 7d b7 fd 3e 81 a8 7c 4a 4c 0d 16 d5 47 62 2b 64 ec 56 e1 02 23 58 b7 48 9f b7 2b 70 95 a4 06 fb d5 37 4f 80 98 2d 33 a3 0a c9 52 99 0a
qx:
a0 a2 14 d6 2e 2e 00 0d b9 01 9a 99 6f b7 49 d5 80 ee 80 4f 2c 6b dc cb 25 63 48 05 d0 a0 8c 45 26 77 00 5f 52 b6 89 e1 8d 32 d0 04 fe cd 4a 12
Lambda^2-px-qx:
c1 fa 33 d8 9b cf 85 fb 60 95 cd 42 c7 05 5f 1f 05 4e 9a 1c 44 50 d8 69 e6 aa 54 87 f1 fb 12 65 5d d1 e0 c7 63 08 54 c6 9f 00 d3 05 cb 84 4e f8
```

Lambda^2-px-qx: From OpenSSL (MSB first):

0x8507A34DA90B6DC7C6540863C7E0D15D6512FBF18754AAE669D850441C9A4E051F5F05C742CD9560FB85CF9BD833FAC1

Incorrect! It would appear that sometimes our field subtraction function is returning incorrect values, but not every time.
This matches our failure case. In our original **2sx** value from [‘incorrect point double result’].

As the field subtraction function is called many times during execution of the various point arithmetic, it is not unusual that our point multiplication is returns in incorrect and different result based on the point multiplication algorithm used. The alternative coordinate systems also contribute to the differing results by calling this function an different number of times. Any number of executions of this could lead to at least one incorrect answer.

The default assignment of the field subtraction function on brainpoolP384r1 is `ec_GFp_sub_6,` function optimized for generic 384 bit curves, 6 words of 64 bits.

Test with another field subtraction function

To verify that the field subtraction was faulty, and only this, a specific case for brainpool384r1 was created that overrode the optimized function with the generic subtraction function `ec_GFp_sub`. This was done by adding a case: in ecl.c:307 in the section for curve optimizations.

This function calculates r=a-b by calling `mp_sub(b, a, r)` and calling `mp_add(r, &meth->irr, r)`

This generic function returned the correct result for point double, and ultimately passed the `ECPoint_mul(group, &group->order, px, py, &pxt, &pyt)`test from the `ec_GFp_validate_point` function. This curve then correctly validated the point, verified the signature, and negotiated the TLS session.

The generic function ec_GFp_sub calls the mpi function mp_sub(b, a, r) to calculate the value.

What is wrong with ec_GFp_sub_6?

This function is defined as `ec_GFp_sub_6(a, b, r, meth)`, r=a-b, where a b and c are integers (field elements mod p).
Meth is a collection of field arithmetic functions defined for the group.
We are interested in meth->irr, which is **p,** our curve prime.

p for brainpoolP384r1 is:
(LSD first – mpi encoding):

```
53 ec 07 31 13 00 47 87 71 1a 1d 90 29 a7 d3 ac 23 11 b7 7f 19 da b1 12 b4 56 54 ed 09 71 2f 15 df 41 e6 50 7e 6f 5d 0f 28 6d 38 a3 82 1e b9 8c
```

It seems we are getting the wrong answer with inputs:
(LSD first – mpi encoding):

```
a: 0e b1 40 7d b7 fd 3e 81 a8 7c 4a 4c 0d 16 d5 47 62 2b 64 ec 56 e1 02 23 58 b7 48 9f b7 2b 70 95 a4 06 fb d5 37 4f 80 98 2d 33 a3 0a c9 52 99 0a
b: a0 a2 14 d6 2e 2e 00 0d b9 01 9a 99 6f b7 49 d5 80 ee 80 4f 2c 6b dc cb 25 63 48 05 d0 a0 8c 45 26 77 00 5f 52 b6 89 e1 8d 32 d0 04 fe cd 4a 12
```

Our expected result of r= a-b mod p is
(LSD first – mpi encoding):

```
a-b mod p = c1 fa 33 d8 9b cf 85 fb 60 95 cd 42 c7 05 5f 1f 05 4e 9a 1c 44 50 d8 69 e6 aa 54 87 f1 fb 12 65 5d d1 e0 c7 63 08 54 c6 c7 6d 0b a9 4d a3 07
85
```

but it calculates:
(LSD first – mpi encoding):

```
c1 fa 33 d8 9b cf 85 fb 60 95 cd 42 c7 05 5f 1f 05 4e 9a 1c 44 50 d8 69 e6 aa 54 87 f1 fb 12 65 5d d1 e0 c7 63 08 54 c6 9f 00 d3 05 cb 84 4e f8
```

The algorithm implemented is Algorithm 2 from (Brown, Hankerson, J.Lopez, & Menezes, 2000) and this implementation does:

```
copy each mp_digit from the mp_int a, b into an mp_digit object rX,bX (where X is 0-5)
```

```
r0 = sub_borrow(r0, b0)
for i from 1 to 5 do ri = sub_borrow(bi, ri)
if carry bit set then add each respective mp_digit from p to r0 to r4
```

splitting out the results of the for loop when executed with this parameters, we have:

```
r0 = 0x743ecf88a72c0e6e
r1 = 0x728b5e9db2b07aef
r2 = 0x5726762a9ce33ce1
r3 = 0x4fe38ae79a005432
r4 = 0xb6f698e576fa8f7e
r5 = 0xf84e84cb05d3009f
borrow 0x0000000000000001
```

adding each respective mp_digit (8 bytes) from **p** we end up with

```
r0 = 0xfb85cf9bd833fac1
r1 = 0x1f5f05c742cd9560
r2 = 0x69d850441c9a4e05
r3 = 0x6512fbf18754aae6
r4 = 0xc6540863c7e0d15d
r5 = 0xf84e84cb05d3009f
```

with r0-r4 matching our expected correct result.

The line MP_SUB_BORROW(r5, b5, r5, borrow, borrow) with inputs

```
r5 = 0x0a9952c90aa3332d
b5 = 0x124acdfe04d0328d
```

results in `r5 = 0xf84e84cb05d3009f`

This is the incorrect 64 bit word. The expected value to match the correct double point should be:

`0x8507a34da90b6dc7`

Now, if we add the mp_digit component corresponding to the 6th 64 bit word of p to r5

`0x8cb91e82a3386d28 + 0xf84e84cb05d3009f = 0x8507A34DA90B6DC7`

We are missing an add statement in the algorithm

```
r0 = sub_borrow(r0, b0)
for i from 1 to 5 do ri = sub_borrow(bi, ri)
if borrow bit set then add each respective mp_digit from p to r0 to r5
```

The patch is simply:

```
diff --git a/security/nss/lib/freebl/ecl/ecl_gf.c b/security/nss/lib/freebl/ecl/ecl_gf.c
--- a/security/nss/lib/freebl/ecl/ecl_gf.c
+++ b/security/nss/lib/freebl/ecl/ecl_gf.c
@@ -784,16 +784,17 @@ ec_GFp_sub_5(const mp_int *a, const mp_i
               b3 = MP_DIGIT(&meth->irr,3);
               b2 = MP_DIGIT(&meth->irr,2);
               b1 = MP_DIGIT(&meth->irr,1);
               b0 = MP_DIGIT(&meth->irr,0);
               MP_ADD_CARRY(b0, r0, r0, 0,      borrow);
               MP_ADD_CARRY(b1, r1, r1, borrow, borrow);
               MP_ADD_CARRY(b2, r2, r2, borrow, borrow);
               MP_ADD_CARRY(b3, r3, r3, borrow, borrow);
+              MP_ADD_CARRY(b4, r4, r4, borrow, borrow);
       }
       MP_CHECKOK(s_mp_pad(r, 5));
       MP_DIGIT(r, 4) = r4;
       MP_DIGIT(r, 3) = r3;
       MP_DIGIT(r, 2) = r2;
       MP_DIGIT(r, 1) = r1;
       MP_DIGIT(r, 0) = r0;
       MP_SIGN(r) = MP_ZPOS;
@@ -859,16 +860,17 @@ ec_GFp_sub_6(const mp_int *a, const mp_i
               b2 = MP_DIGIT(&meth->irr,2);
               b1 = MP_DIGIT(&meth->irr,1);
               b0 = MP_DIGIT(&meth->irr,0);
               MP_ADD_CARRY(b0, r0, r0, 0,      borrow);
               MP_ADD_CARRY(b1, r1, r1, borrow, borrow);
               MP_ADD_CARRY(b2, r2, r2, borrow, borrow);
               MP_ADD_CARRY(b3, r3, r3, borrow, borrow);
               MP_ADD_CARRY(b4, r4, r4, borrow, borrow);
+              MP_ADD_CARRY(b5, r5, r5, borrow, borrow);
       }

       MP_CHECKOK(s_mp_pad(r, 6));
       MP_DIGIT(r, 5) = r5;
       MP_DIGIT(r, 4) = r4;
       MP_DIGIT(r, 3) = r3;
       MP_DIGIT(r, 2) = r2;
       MP_DIGIT(r, 1) = r1;
```

Execution with this fixed ec_GFp_sub_6 allowed brainpoolP384r1 to operate using the generic GFp functions for 384 bit curves with the generic mont coordinates.
This should allow other curves using group->meth->field_sub = &ec_GFp_sub_5 or &ec_GFp_sub_6 and Most Significant 64 bit Word not equal 0xFFFFFFFFFFFFFFFF to work generically.

Bibliography

Antipa, A., Brown, D., Menezes, A., Struik, R., & Vanstone, S. (2003). Validation of Elliptic Curve Public Keys. *IACR*.

Brown, M., Hankerson, D., J.Lopez, & Menezes, A. (2000). Software Implementation of the NIST Elliptic Curves Over Prime Fields.