**Problem**

The BN\_sqr implementation in OpenSSL before 0.9.8zd, 1.0.0 before 1.0.0p, and 1.0.1 before 1.0.1k does not properly calculate the square of a BIGNUM value, which might make it easier for unauthenticated, remote attackers to defeat cryptographic protection mechanisms via unspecified vectors. Successful exploitation could cause the software to calculate incorrect results. Related to crypto/bn/asm/**mips.pl**, crypto/bn/asm/**x86\_64-gcc.c**, and crypto/bn/**bn\_asm.c**.

<https://access.redhat.com/errata/RHSA-2015:0066.html>

It was found that OpenSSL's BigNumber Squaring implementation could produce incorrect results under certain special conditions. This flaw could possibly affect certain OpenSSL library functionality, such as RSA blinding. Note that this issue occurred rarely and with a low probability, and there is currently no known way of exploiting it.

<https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=1180240>

Bignum squaring (BN\_sqr) may produce incorrect results on some platforms, including x86\_64. This bug occurs at random with a very low probability and is not known to be exploitable in any way, though its exact impact is difficult to determine. The following has been determined:

* The probability of BN\_sqr producing an incorrect result at random is very low: 1/2^64 on the single affected 32-bit platform (MIPS) and 1/2^128 on affected 64-bit platforms.
* On most platforms, RSA follows a different code path and RSA operations are not affected at all. For the remaining platforms (e.g., OpenSSL built without assembly support), pre-existing countermeasures thwart bug attacks [2].
* Static ECDH is theoretically affected: it is possible to construct elliptic curve points that would falsely appear to be on the given curve. However, there is no known computationally feasible way to construct such points with low order, and so the security of static ECDH private keys is believed to be unaffected.
* Other routines known to be theoretically affected are modular exponentiation, primality testing, DSA, RSA blinding, JPAKE and SRP. No exploits are known, and straightforward bug attacks fail - either the attacker cannot control when the bug triggers, or no private key material is involved.

<https://css.csail.mit.edu/6.858/2013/readings/rsa-bug-attacks.pdf>

To read on the syntax : <https://phoenix.goucher.edu/~kelliher/f2009/cs220/mipsir.html#:~:text=SLT%20%2D%2D%20Set%20on%20less%20than%20(signed)&text=It%20gets%20zero%20otherwise>.

Explanation of the importance of BigNum vulnerabilities :

* <https://comsecuris.com/slides/slides-bignum-bhus2015.pdf>

Talk on the CVE : <https://www.youtube.com/watch?v=oje2ZgUzARo&ab_channel=BlackHat>

**buggy\_bn\_asm.c**

#ifdef BN\_LLONG

#define mul\_add\_c(a,b,c0,c1,c2) \

t=(BN\_ULLONG)a\*b; \

t1=(BN\_ULONG)Lw(t); \

t2=(BN\_ULONG)Hw(t); \

c0=(c0+t1)&BN\_MASK2; if ((c0) < t1) t2++; \

c1=(c1+t2)&BN\_MASK2; if ((c1) < t2) c2++;

#define mul\_add\_c2(a,b,c0,c1,c2) \

t=(BN\_ULLONG)a\*b; \

tt=(t+t)&BN\_MASK; \

if (tt < t) c2++; \

t1=(BN\_ULONG)Lw(tt); \

t2=(BN\_ULONG)Hw(tt); \

c0=(c0+t1)&BN\_MASK2; \

if ((c0 < t1) && (((++t2)&BN\_MASK2) == 0)) c2++; \

c1=(c1+t2)&BN\_MASK2; if ((c1) < t2) c2++;

#define sqr\_add\_c(a,i,c0,c1,c2) \

t=(BN\_ULLONG)a[i]\*a[i]; \

t1=(BN\_ULONG)Lw(t); \

t2=(BN\_ULONG)Hw(t); \

c0=(c0+t1)&BN\_MASK2; if ((c0) < t1) t2++; \

c1=(c1+t2)&BN\_MASK2; if ((c1) < t2) c2++;

#define sqr\_add\_c2(a,i,j,c0,c1,c2) \

mul\_add\_c2((a)[i],(a)[j],c0,c1,c2)

#elif defined(BN\_UMULT\_LOHI)

#define mul\_add\_c(a,b,c0,c1,c2) { \

BN\_ULONG ta=(a),tb=(b); \

BN\_UMULT\_LOHI(t1,t2,ta,tb); \

c0 += t1; t2 += (c0<t1)?1:0; \

c1 += t2; c2 += (c1<t2)?1:0; \

}

La vulnérabilité se produit spécifiquement ici!!



Cette section représente le terme

**#define mul\_add\_c2(a,b,c0,c1,c2)** { \

BN\_ULONG ta=(a),tb=(b),t0; \

BN\_UMULT\_LOHI(t0,t1,ta,tb); \

t2 = **t1+t1**; c2 += (t2<t1)?1:0; \ **Overflow de t1+t1 🡪 aucun carry (erreur de calcul)**

t1 = **t0+t0**; t2 += (t1<t0)?1:0; \ **Overflow de t0+t0 🡪 aucun carry (erreur de calcul)**

c0 += t1; t2 += (c0<t1)?1:0; \

c1 += t2; c2 += (c1<t2)?1:0; \

}

#define sqr\_add\_c(a,i,c0,c1,c2) { \

BN\_ULONG ta=(a)[i]; \

BN\_UMULT\_LOHI(t1,t2,ta,ta); \

c0 += t1; t2 += (c0<t1)?1:0; \

c1 += t2; c2 += (c1<t2)?1:0; \

}

#define sqr\_add\_c2(a,i,j,c0,c1,c2) \

mul\_add\_c2((a)[i],(a)[j],c0,c1,c2)

#elif defined(BN\_UMULT\_HIGH)

#define mul\_add\_c(a,b,c0,c1,c2) { \

BN\_ULONG ta=(a),tb=(b); \

t1 = ta \* tb; \

t2 = BN\_UMULT\_HIGH(ta,tb); \

c0 += t1; t2 += (c0<t1)?1:0; \

c1 += t2; c2 += (c1<t2)?1:0; \

}

La vulnérabilité se produit spécifiquement ici!!



Cette section représente le terme

#define mul\_add\_c2(a,b,c0,c1,c2) { \

BN\_ULONG ta=(a),tb=(b),t0; \

t1 = BN\_UMULT\_HIGH(ta,tb); \

t0 = ta \* tb; \

t2 = **t1+t1**; c2 += (t2<t1)?1:0; \ **Overflow de t1+t1 🡪 aucun carry (erreur de calcul)**

t1 = **t0+t0**; t2 += (t1<t0)?1:0; \ **Overflow de t0+t0 🡪 aucun carry (erreur de calcul)**

c0 += t1; t2 += (c0<t1)?1:0; \

c1 += t2; c2 += (c1<t2)?1:0; \

}

#define sqr\_add\_c(a,i,c0,c1,c2) { \

BN\_ULONG ta=(a)[i]; \

t1 = ta \* ta; \

t2 = BN\_UMULT\_HIGH(ta,ta); \

c0 += t1; t2 += (c0<t1)?1:0; \

c1 += t2; c2 += (c1<t2)?1:0; \

}

#define sqr\_add\_c2(a,i,j,c0,c1,c2) \

mul\_add\_c2((a)[i],(a)[j],c0,c1,c2)

#else /\* !BN\_LLONG \*/

#define mul\_add\_c(a,b,c0,c1,c2) \

t1=LBITS(a); t2=HBITS(a); \

bl=LBITS(b); bh=HBITS(b); \

mul64(t1,t2,bl,bh); \

c0=(c0+t1)&BN\_MASK2; if ((c0) < t1) t2++; \

c1=(c1+t2)&BN\_MASK2; if ((c1) < t2) c2++;

#define mul\_add\_c2(a,b,c0,c1,c2) \

t1=LBITS(a); t2=HBITS(a); \

bl=LBITS(b); bh=HBITS(b); \

mul64(t1,t2,bl,bh); \

if (t2 & BN\_TBIT) c2++; \

t2=(**t2+t2**)&BN\_MASK2; \ **Overflow de t2+t2 🡪 aucun carry (erreur de calcul)**

if (t1 & BN\_TBIT) t2++; \

t1=(**t1+t1**)&BN\_MASK2; \ **Overflow de t1+t1 🡪 aucun carry (erreur de calcul)**

c0=(c0+t1)&BN\_MASK2; \

if ((c0 < t1) && (((++t2)&BN\_MASK2) == 0)) c2++; \

c1=(c1+t2)&BN\_MASK2; if ((c1) < t2) c2++;

#define sqr\_add\_c(a,i,c0,c1,c2) \

sqr64(t1,t2,(a)[i]); \

c0=(c0+t1)&BN\_MASK2; if ((c0) < t1) t2++; \

c1=(c1+t2)&BN\_MASK2; if ((c1) < t2) c2++;

#define sqr\_add\_c2(a,i,j,c0,c1,c2) \

mul\_add\_c2((a)[i],(a)[j],c0,c1,c2)

#endif /\* !BN\_LLONG \*/

void bn\_mul\_comba8(BN\_ULONG \*r, BN\_ULONG \*a, BN\_ULONG \*b)

{

#ifdef BN\_LLONG

BN\_ULLONG t;

#else

BN\_ULONG bl,bh;

#endif

BN\_ULONG t1,t2;

BN\_ULONG c1,c2,c3;

c1=0;

c2=0;

c3=0;

**Fix**

t1=lw(t) // It loads 4-byte word from memory at location which address is stored in regist.

//Using the long long type, has to be twice as wide as BN\_ULONG...

# define Lw(t) (((BN\_ULONG)(t))&BN\_MASK2)

# define Hw(t) (((BN\_ULONG)((t)>>BN\_BITS2))&BN\_MASK2)

BN\_MASK2 = 64 bits (toute à 1) ou 32 bits (toute à 1)

**bn\_asm.c**

#ifdef BN\_LLONG

/\*Keep in mind that additions to multiplication result can not overflow, because its high half cannot be all-ones.\*/

#define mul\_add\_c(a,b,c0,c1,c2) do { \

BN\_ULONG hi; \

BN\_ULLONG t = (BN\_ULLONG)(a)\*(b); \

t += c0; /\* no carry \*/ \

c0 = (BN\_ULONG)Lw(t); \

hi = (BN\_ULONG)Hw(t); \

c1 = (c1+hi)&BN\_MASK2; if (c1<hi) c2++; \

} while(0)

#define mul\_add\_c2(a,b,c0,c1,c2) do { \

BN\_ULONG hi; \

BN\_ULLONG t = (BN\_ULLONG)(a)\*(b); \

**BN\_ULLONG tt = t+c0; /\* no carry \*/** \

c0 = (BN\_ULONG)Lw(tt); \

hi = (BN\_ULONG)Hw(tt); \

c1 = (c1+hi)&BN\_MASK2; if (c1<hi) c2++; \

**t += c0; /\* no carry \*/** \

c0 = (BN\_ULONG)Lw(t); \

hi = (BN\_ULONG)Hw(t); \

c1 = (c1+hi)&BN\_MASK2; if (c1<hi) c2++; \

} while(0)

#define sqr\_add\_c(a,i,c0,c1,c2) do { \

BN\_ULONG hi; \

BN\_ULLONG t = (BN\_ULLONG)a[i]\*a[i]; \

t += c0; /\* no carry \*/\

c0 = (BN\_ULONG)Lw(t); \

hi = (BN\_ULONG)Hw(t); \

c1 = (c1+hi)&BN\_MASK2; if (c1<hi) c2++; \

} while(0)

#define sqr\_add\_c2(a,i,j,c0,c1,c2) \

mul\_add\_c2((a)[i],(a)[j],c0,c1,c2)

#elif defined(BN\_UMULT\_LOHI)

/\* Keep in mind that additions to hi can not overflow, because the high word of a multiplication result cannot be all-ones. \*/

#define mul\_add\_c(a,b,c0,c1,c2) do { \

BN\_ULONG ta = (a), tb = (b); \

BN\_ULONG lo, hi; \

BN\_UMULT\_LOHI(lo,hi,ta,tb); \

c0 += lo; hi += (c0<lo)?1:0; \

c1 += hi; c2 += (c1<hi)?1:0; \

} while(0)

#define mul\_add\_c2(a,b,c0,c1,c2) do { \

BN\_ULONG ta = (a), tb = (b); \

BN\_ULONG lo, hi, tt; \

BN\_UMULT\_LOHI(lo,hi,ta,tb); \

**c0 += lo; tt = hi+((c0<lo)?1:0); \**

**c1 += tt; c2 += (c1<tt)?1:0; \**

**c0 += lo; hi += (c0<lo)?1:0; \**

**c1 += hi; c2 += (c1<hi)?1:0; \**

} while(0)

#define sqr\_add\_c(a,i,c0,c1,c2) do { \

BN\_ULONG ta = (a)[i]; \

BN\_ULONG lo, hi; \

BN\_UMULT\_LOHI(lo,hi,ta,ta); \

c0 += lo; hi += (c0<lo)?1:0; \

c1 += hi; c2 += (c1<hi)?1:0; \

} while(0)

#define sqr\_add\_c2(a,i,j,c0,c1,c2) \

mul\_add\_c2((a)[i],(a)[j],c0,c1,c2)

#elif defined(BN\_UMULT\_HIGH)

/\* Keep in mind that additions to hi can not overflow, because the high word of a multiplication result cannot be all-ones.\*/

#define mul\_add\_c(a,b,c0,c1,c2) do { \

BN\_ULONG ta = (a), tb = (b); \

BN\_ULONG lo = ta \* tb; \

BN\_ULONG hi = BN\_UMULT\_HIGH(ta,tb); \

c0 += lo; hi += (c0<lo)?1:0; \

c1 += hi; c2 += (c1<hi)?1:0; \

} while(0)

#define mul\_add\_c2(a,b,c0,c1,c2) do { \

BN\_ULONG ta = (a), tb = (b), tt; \

BN\_ULONG lo = ta \* tb; \

BN\_ULONG hi = BN\_UMULT\_HIGH(ta,tb); \

**c0 += lo; tt = hi + ((c0<lo)?1:0); \**

**c1 += tt; c2 += (c1<tt)?1:0; \**

**c0 += lo; hi += (c0<lo)?1:0; \**

**c1 += hi; c2 += (c1<hi)?1:0; \**

} while(0)

#define sqr\_add\_c(a,i,c0,c1,c2) do { \

BN\_ULONG ta = (a)[i]; \

BN\_ULONG lo = ta \* ta; \

BN\_ULONG hi = BN\_UMULT\_HIGH(ta,ta); \

c0 += lo; hi += (c0<lo)?1:0; \

c1 += hi; c2 += (c1<hi)?1:0; \

} while(0)

#define sqr\_add\_c2(a,i,j,c0,c1,c2) \

mul\_add\_c2((a)[i],(a)[j],c0,c1,c2)

#else /\* !BN\_LLONG \*/

/\*Keep in mind that additions to hi can not overflow, because the high word of a multiplication result cannot be all-ones. \*/

#define mul\_add\_c(a,b,c0,c1,c2) do { \

BN\_ULONG lo = LBITS(a), hi = HBITS(a); \

BN\_ULONG bl = LBITS(b), bh = HBITS(b); \

mul64(lo,hi,bl,bh); \

c0 = (c0+lo)&BN\_MASK2; if (c0<lo) hi++; \

c1 = (c1+hi)&BN\_MASK2; if (c1<hi) c2++; \

} while(0)

#define mul\_add\_c2(a,b,c0,c1,c2) do { \

BN\_ULONG tt; \

BN\_ULONG lo = LBITS(a), hi = HBITS(a); \

BN\_ULONG bl = LBITS(b), bh = HBITS(b); \

mul64(lo,hi,bl,bh); \

tt = hi; \

**c0 = (c0+lo)&BN\_MASK2; if (c0<lo) tt++; \**

**c1 = (c1+tt)&BN\_MASK2; if (c1<tt) c2++; \**

**c0 = (c0+lo)&BN\_MASK2; if (c0<lo) hi++; \**

**c1 = (c1+hi)&BN\_MASK2; if (c1<hi) c2++;** \

} while(0)

#define sqr\_add\_c(a,i,c0,c1,c2) do { \

BN\_ULONG lo, hi; \

sqr64(lo,hi,(a)[i]); \

c0 = (c0+lo)&BN\_MASK2; if (c0<lo) hi++; \

c1 = (c1+hi)&BN\_MASK2; if (c1<hi) c2++; \

} while(0)

#define sqr\_add\_c2(a,i,j,c0,c1,c2) \

mul\_add\_c2((a)[i],(a)[j],c0,c1,c2)

#endif /\* !BN\_LLONG \*/

void bn\_mul\_comba8(BN\_ULONG \*r, BN\_ULONG \*a, BN\_ULONG \*b)

{

BN\_ULONG c1,c2,c3;

c1=0;

c2=0;

c3=0;

…

**Comparisons (unfinished)**

#define mul\_add\_c2(a,b,c0,c1,c2) { \

BN\_ULONG ta=(a),tb=(b),**t0;** \

BN\_UMULT\_LOHI(**t0,t1**,ta,tb); \

t2 = t1+t1; c2 += (t2<t1)?1:0; \ // c2 += (t1+t1 < t1)? 1 (overflow, **t1>moitié**) ou 0.

t1 = t0+t0; t2 += (t1<t0)?1:0; \ // t2 += (t0+t0 < t0)? 1 (overflow t0, **t0>moitié**) ou 0

c0 += t1; t2 += (c0<t1)?1:0; \ // t2 +=(c0+t1 < t1)? 1 (overflow **c0+t1, c0||t1>moitié**)

c1 += t2; c2 += (c1<t2)?1:0; \ // c2 +=(c1+t2<t2) ? 1 (overflow, **c1||t2>moitié**) ou 0

}

//tt = t2+(c0+t2<t1)?1 ou 0

#define mul\_add\_c2(a,b,c0,c1,c2) do { \

BN\_ULONG ta = (a), tb = (b); \

BN\_ULONG **tt**; \

BN\_UMULT\_LOHI(**t1,t2**,ta,tb); \

c0 += t2; tt = t2+((c0<t1)?1:0); \ //tt = t2+1 🡪 c0+t2 < t1 tt = t2 ou t2+1

c1 += tt; c2 += (c1<tt)?1:0; \ //c2 +=

c0 += t1; t2 += (c0<t1)?1:0; \

c1 += t2; c2 += (c1<t2)?1:0; \

} while(0)

#define mul\_add\_c(a,b,c0,c1,c2) \

t=(BN\_ULLONG)a\*b; \

t1=(BN\_ULONG)Lw(t); \

t2=(BN\_ULONG)Hw(t); \

c0=(c0+t1)&BN\_MASK2; if ((c0) < t1) t2++; \

c1=(c1+t2)&BN\_MASK2; if ((c1) < t2) c2++;

#define mul\_add\_c(a,b,c0,c1,c2) do { \

BN\_ULONG hi; \

**BN\_ULLONG** t = (BN\_ULLONG)(a)\*(b); \

t += c0; /\* no carry \*/ \

t1 = (BN\_ULONG)Lw(t); \

t2 = (BN\_ULONG)Hw(t); \

c1 = (c1+hi)&BN\_MASK2; if (c1<hi) c2++; \

} while(0)

#define mul\_add\_c(a,b,c0,c1,c2) do { \

BN\_ULONG hi; \

**BN\_ULLONG** t = (BN\_ULLONG)(a)\*(b); \

t += c0; /\* no carry \*/ \

c0 = (BN\_ULONG)Lw(t); \

hi = (BN\_ULONG)Hw(t); \

c1 = (c1+hi)&BN\_MASK2; if (c1<hi) c2++; \

} while(0)

mflo $t\_1

mfhi $t\_2

slt **$c\_3**,$t\_2,$zero

**$SLL $t\_2,1**

$MULTU **$a\_5,$a\_1** # mul\_add\_c2(a[5],b[1],c1,c2,c3);

slt $a2,$t\_1,$zero

$ADDU $t\_2,$a2

$SLL $t\_1,1

$ADDU $c\_1,$t\_1

sltu $at,$c\_1,$t\_1

$ADDU $t\_2,$at

$ADDU $c\_2,$t\_2

sltu $at,$c\_2,$t\_2

$ADDU $c\_3,$at

mflo $t\_1

mfhi $t\_2

slt **$at**,$t\_2,$zero

**$ADDU $c\_3,$at**

$MULTU **$a\_4,$a\_2** # mul\_add\_c2(a[4],b[2],c1,c2,c3);

**$SLL $t\_2,1**

slt $a2,$t\_1,$zero

$ADDU $t\_2,$a2

$SLL $t\_1,1

$ADDU $c\_1,$t\_1

sltu $at,$c\_1,$t\_1

$ADDU $t\_2,$at

$ADDU $c\_2,$t\_2

sltu $at,$c\_2,$t\_2

$ADDU $c\_3,$at

mflo $t\_1

mfhi $t\_2

slt **$at**,$t\_2,$zero

**$ADDU $c\_3,$at**

$MULTU **$a\_3,$a\_3** # mul\_add\_c(a[3],b[3],c1,c2,c3);

**$SLL $t\_2,1**

slt $a2,$t\_1,$zero

$ADDU $t\_2,$a2

$SLL $t\_1,1

$ADDU $c\_1,$t\_1

sltu $at,$c\_1,$t\_1

$ADDU $t\_2,$at

$ADDU $c\_2,$t\_2

sltu $at,$c\_2,$t\_2

$ADDU $c\_3,$at