Guido Vranken

Notes on OpenSSL remote memory corruption

Posted on June 27, 2022 by guidovranken
OpenSSL version 3.0.4, released on June 21th 2022, is susceptible to remote memory corruption which can be triggered trivially by an attacker. BoringSSL, LibreSSL and the OpenSSL 1.1.1 branch are not affected. Furthermore, only x64 systems with AVX512 support are affected. The bug is fixed (https://github.com/openssl/openssl/pull/18626) in the repository but a new release is still pending.

Somewhat peculiarly, almost nobody is talking about this. If RCE exploitation is possible this makes it worse than Heartbleed in an isolated severity assessment, though the potential blast radius is limited by the fact that many people are still using the 1.1.1 tree rather than 3, libssl has forked into LibreSSL and BoringSSL, the vulnerability has only existed for a week (HB existed for years) and an AVX512-capable CPU is required.

This post gives some background on how this bug came to be and some notes on its exploitability.

On May 31th 2022 I found and reported an issue with constant-time Montgomery modular exponentiation in OpenSSL and BoringSSL (but not LibreSSL). For some values of `B, E, M` for which `B ^ E % M == 0`, some functions would return `M` instead of 0; the result was not fully reduced.

It was found that there are four distinct code paths affected by this:

- RSAZ 1024 (https://github.com/openssl/openssl/blob/b2feb9f0e394da6570346598837f1b01eb58c028/crypto/bn/bn_exp.c#L668-L669)
- RSAZ 512 (https://github.com/openssl/openssl/blob/b2feb9f0e394da6570346598837f1b01eb58c028/crypto/bn/bn_exp.c#L678)
- Dual 1024 RSAZ (https://github.com/openssl/openssl/blob/b2feb9f0e394da6570346598837f1b01eb58c028/crypto/bn/bn_exp.c#L1448-L1452)
- Default constant-time Montgomery modular exponentiation (https://github.com/openssl/openssl/blob/b2feb9f0e394da6570346598837f1b01eb58c028/crypto/bn/bn_exp.c#L687-L1124)

David Benjamin of Google analyzed the issue extensively (https://boringssl.googlesource.com/boringssl/+/13c9d5c69d04485a7a8840c12185c832026c8315) and found that the bug does not constitute a security risk (at least not internally; external callers might end up in incorrect states depending on what they are trying to compute). Interestingly, he also found an apparent bug in the paper (https://boringssl.googlesource.com/boringssl/+/801a801024febe1a33add5ddaa719e257d97aba5) by Shay Gueron upon which the RSAZ code is based.

This took my fuzzer (https://github.com/guidovranken/cryptofuzz) a long time to find, because the odds of finding `B ^ M % E = 0` for large, `N`-bit values of `B, E, M` where those values are semi-random are small, so I proceeded to add a modular exponentation solver, (https://github.com/guidovranken/cryptofuzz/blob/dd20f87e98ca92e9e31f8dd9d0a80efb5c249df7/expmod.cpp) so the bug can now be found quite quickly, and hopefully it helps finding more, similar bugs in the future (in OpenSSL or elsewhere). (Z3's performance in solving equations involving modular exponentiation is quite poor so I had to roll my own).

The fix (https://github.com/openssl/openssl/commit/10d8a109be0fe50315e4eeb0676f6571914cd47a#diff-854689acd04b8f1b65120880bebcd98d519e89b601328820f276ec0e5c164c4f) that was applied to the dual 1024 RSAZ code is wrong because the reduction function is called (https://github.com/openssl/openssl/blob/eea820f3e239a4c11d618741fd5d00a6bc877347/crypto/bn/rsaz_exp_x2.c#L260-L261) with `num` set to the bit size, where it should be number of `BN_ULONG` elements (which are always 8 bytes large, because that is the size of an unsigned long on x64 systems, which is the only architecture which can have AVX512 support). So with the input sizes being 1024 bits, 8192 bytes are accessed (read from or written to) instead of 128.

On to the bug internals. There are 5 distinct arrays involved. 3 arrays are overwritten.

Variable	Description	Allocated size	Over-read/write	Total	Read/write?
`res1`	modexp result 1	128	896	8192	Read, then write
`m1`	Modulus 1	128	896	8192	Read
`res2`	modexp result 1	128	896	8192	Read, then write
`m2`	Modulus 2	128	896	8192	Read
`storage`	Scratch space	1184	7296	8192	Write, then read

- 8192 bytes are read from `res1`, `res2`, `m1`, `m2` and `storage`
- 8192 bytes are written to `res1`, `res2` and `storage` (this is where the memory corruption takes place)
- If we consider `res1_bn` to be a bignum comprising `res1[0..8192]` (where the last byte is the most significant byte) and `m1_bn` to be `m1[0..8192]`, then if `res1_bn < m1_bn`, the contents of `res1[0..8192]` will be left unchanged after being overwritten. The same applies for `res2` and `m2`.
- This implies that if you can set the most significant bit of `m1[8191]` to `1` and the most significant bit of `res1[8191]` to `0`, then `res1[0..8192]` will retain its original state (and no actual corruption takes place). This circumstance may occur by chance. The same applies for `res2` and `m2`.
- Conversely, if `res1_bn >= m1_bn`, then after the write, `res1[N]` will be one of `{res1[N], res1[N] - m1[N], res1[N] - m1[N] - 1}`. The same applies for `res2` and `m2`.
- After the overwrites have occurred, `storage[N]`will be `~(m2[N] - res2[N])` or `~(m2[N] - res2[N])+1`.
- From this it follows that if you control `m2[N]`and `res2[N]`, you mostly control `storage[N]`.
- The original contents of `storage` is never read, so it does not influence the end state in any way.

Summarized:

Variable	Post-write value
res1[N]	`res1[N]` *or* `res1[N] - m1[n]` *or* `res1[N] - m1[N] - 1`
res2[N]	`res2[N]` *or* `res2[N] - m2[n]` *or* `res2[N] - m2[N] - 1`
storage[N]	`~(m2[N] - res2[N])` *or* `~(m2[N] - res2[N]) + 1`

OpenSSL vulnerability post-write states

Each of these lemma's are true independent of what the inputs to the modexp function are, and of any other variable or state.

Here is a fuzzer (https://gist.github.com/guidovranken/b1c46bd9e42e959519009681b261a896) which demonstrates these invariants.

The (wrapping) subtraction mechanics at play here are interesting because you can use them to apply pointer delta's to a function pointer to make it point to a different function and this can help circumvent ASLR (https://en.wikipedia.org/wiki/Address_space_layout_randomization), because while the function addresses are randomized, their spacing is constant for every given binary.

For example, if:

- a single, particular `BN_ULONG` in the `res1` array is known to contain an active pointer to a specific function
- and you can control the `BN_ULONG` in `m1` at the same index (for example via heap spraying)
- and the rest of the state is completely unknown and uncontrolled

then by setting `m1[N]` to `oldfunc - newfunc`, the original function pointer will be exactly `newfunc` after the overflow with a 25% probability:

```c
void goodfunc(void) { printf("good\n"); }
void badfunc(void) { printf("bad\n"); }
int main(void)
{
    /* Indices 0..127 are within allocated bounds */
    /* Beyond that is either unallocated or allocated by different pa
     * of the program.
     */
    struct {
        BN_ULONG storage[1024], res1[1024], m1[1024], res2[1024], m2[
    } vars;

    /* Randomize everything; not known to the attacker */
    FILE* fp = fopen("/dev/urandom", "rb");
    assert(fread(&vars, 1, sizeof(vars), fp));
    fclose(fp);

    /* Assume res1[345] contains a function pointer used internally b
    vars.res1[345] = (BN_ULONG)(&goodfunc);
    /* Assume we control m1[345] */
    vars.m1[345] = (BN_ULONG)(&goodfunc) - (BN_ULONG)(&badfunc);

    bn_reduce_once_in_place(vars.res1, /*carry=*/0, vars.m1, vars.sta
    bn_reduce_once_in_place(vars.res2, /*carry=*/0, vars.m2, vars.sta

    void (*fnptr)(void) = (void*)vars.res1[345];
    fnptr(); /* good or bad? */
    return 0;
}
```

OpenSSL makes heavy use of function pointers. Running `find -name '*.c' -exec grep 'METH. = {' {} \; | grep -v test` from the repository root shows over 130 data structures that encapsulate a set of function pointers. Delta subtraction may be useful in exploiting this circumstance to make OpenSSL misbehave to varying degrees of severity.

Apart from code execution, there can also be scenario's where private data is leaked to the attacker.

Assume:

- R^1 = `res[I..J]`
- M^1 = `m[I..J]`
- I, K >= 128
- J, L <= 1023

Let R^1 be an allocated space which the attacker can read and write (for example an internal TLS state whose value is, to an extent, determined by how the attacker conducts the handshake, and which can be read, to an extent, by how the TLS code subsequently behaves based on its state).

Let M^1 contain some kind of secret information.

Recall that the combination of `res[N]` (before the overwrite) and `m[N]` leaks into `res[N]`. It follows that if the attacker has read-write access to R^1, M^1 can be partially or completely inferred.

This is what I've deduced so far. It's possible there are errors in this post; please e-mail guido@guidovranken.com (mailto:guido@guidovranken.com) and I'll correct them and credit you. You can follow me on Twitter (https://twitter.com/GuidoVranken).