Audit of Grin's secp256k1 extensions

Jean-Philippe Aumasson

23/12/18

1 Summary

Grin is a privacy-enhanced coin that uses MimbleWimble to obfuscate addresses and amount from the public blockchain records.

The goal of this audit is to assess the security of the extensions added to libsecp256k1 in order to support privacy features, in particular the bulletproofs range proofs and the aggsig module.

The section below describe the issues discovered, in perceived order of severity (starting with the highest severity), and proposes mitigations. The last section reports minor observations to improve the application.

The main primitive operations are secp256k1 arithmetic and HMAC-SHA-256. We reviewed the code and looked for issues related to do:

- side-channel leaks (such as timing leaks)
- software safety (such as memory leaks, or abuse of the API)
- usage of the underlying crypto primities
- randomness and pseudorandom generation
- cryptographic security level (such as key lengths)
- decoding of serialized/DER data

Furthremore, we tried to assess the safety of the changes introduced compared to the original Bulletproofs protocol, but have not formally assessed the security of the modified system.

The audit was carried out in limited time (7 hours, including reporting), and consisted in manual code review, and dynamic analysis based on the unit tests included.

2 Potential security issues

2.1 Optimized out dead assignment may leak sensitive data

At ./src/ecmult_gen_impl.h:153, the line `bits = 0` aims to overwrite the value of private exponent bits. However compilers may remove the corresponding instructions, since the variable `bits` is no longer used.

Likewise, the subsequent "clear" functions could be optimized out:

```
    bits = 0;
    secp256k1_ge_clear(&add);
    secp256k1_scalar_clear(&gnb);
  }
```

Furthermore, in aggsig's main_impl.h the following memset may also be optimized out (as evidently noted by the developers):

```
    memset(data, 0, 32);  /* TODO proper clear */
```

There is no simple solution guaranteed to work, the best is to review the binaries generated and check that those operations haven't been removed by the compiler.

2.2 Missing null pointers checks

`secp256k1_aggsig_sign_single()` checks some but not all pointers' nullity. Is this on purpose?

Likewise for `secp256k1_aggsig_verify_single()` and `secp256k1_aggsig_add_signatures_single(()`(where `sigs` is checked but not the `sigs[i]`).

3 Other observations

3.1 Unfreed heap allocations

In `secp256k1_aggsig_verify_single()`, if `secp256k1_ecmult_multi_var()` fails then the `scratch` buffer will not be freed:

```
scratch = secp256k1_scratch_space_create(ctx, 1024*4096);
/* Compute sG - eP, which should be R */
if (!secp256k1_ecmult_multi_var(&ctx->ecmult_ctx, scratch, &pk_sum, &g_sc, secp256k1_aggsig_verify_
    return 0;
}

secp256k1_scratch_space_destroy(scratch);
```

Too, in `secp256k1_bulletproof_rangeproof_prove()`, `secp256k1_scratch_deallocate_frame()` is only called if the function succeeds, hence it may return 0 without deallocating the scratch frame.

3.2 Unchecked heap allocation

The value of `secp256k1_scratch_space_create(ctx, 1024*4096)` in `secp256k1_aggsig_verify_single()` and in `secp256k1_aggsig_build_scratch_and_verify()` is not verified. The `checked_malloc()` would display an error, but would still return NULL to the caller.

Too, in `secp256k1_bulletproof_rangeproof_prove()`, the `tge = malloc(2*sizeof(secp256k1_ge));` are not checked.

These mallocs are unlikely to fail but it's still a bit safer to check them.

3.3 `secp256k1_compute_sighash_single()` always returns 1 with scalar_low_impl.h

Probably a non-issue, just

`secp256k1_compute_sighash_single()` returns the following, given an uninitialized `int overflow`:

```
secp256k1_scalar_set_b32(r, output, &overflow);
 return !overflow;
```

However `overflow` will always be 0 when using scalar_low_impl.h:

```
static void secp256k1_scalar_set_b32(secp256k1_scalar *r, const unsigned char *b32, int *overflow) {
    const int base = 0x100 % EXHAUSTIVE_TEST_ORDER;
    int i;
    *r = 0;
```

```
    for (i = 0; i < 32; i++) {
       *r = ((*r * base) + b32[i]) % EXHAUSTIVE_TEST_ORDER;
    }
    /* just deny overflow, it basically always happens */
    if (overflow) *overflow = 0;
}
```

Hence `secp256k1_compute_sighash_single()` always returns 1 in this case. This does not happen with the 8x32 nor 4x64 implementations.

The same behavior occurs in `secp256k1_computer_sighash()`.

3.4 Unnecessary operations?

At the end of `secp256k1_bulletproof_rangeproof_prove()`:

```
    ret = secp256k1_bulletproof_rangeproof_prove_impl(&ctx->ecmult_ctx, scratch, proof, plen, tau_x, tg

    if (t_one != NULL && tau_x == NULL) {
        secp256k1_pubkey_save(t_one, &tge[0]);
        secp256k1_pubkey_save(t_two, &tge[1]);
    }
    secp256k1_scratch_deallocate_frame(scratch);
    return ret;
```

Two potential improvements here:

- If the proof fails (`ret != 0`), aren't the `secp256k1_pubkey_save()` useless?
- Why isn't there an additional check `t_two != NULL`?

3.5 Unnecessary operation?

I didn't get why the HMAC finalize is done here:

```
void secp256k1_aggsig_context_destroy(secp256k1_aggsig_context *aggctx) {
    if (aggctx == NULL) {
        return;
    }
    memset(aggctx->pubkeys, 0, aggctx->n_sigs * sizeof(*aggctx->pubkeys));
    memset(aggctx->secnonce, 0, aggctx->n_sigs * sizeof(*aggctx->secnonce));
    memset(aggctx->progress, 0, aggctx->n_sigs * sizeof(*aggctx->progress));
    free(aggctx->pubkeys);
    free(aggctx->secnonce);
    free(aggctx->progress);
    secp256k1_rfc6979_hmac_sha256_finalize(&aggctx->rng);
    free(aggctx);
}
```

3.6 Faster rejection of invalid parameters

In `secp256k1_bulletproof_rangeproof_prove()` the only valid values of `nbits` are `< 64` and have a Hamming weight of 1 bit. This may be checked during the `ARG_CHECK()` sequence to avoid doing costly arithmetic operations before `nbits` is eventually rejected in `secp256k1_bulletproof_rangeproof_prove_impl()`.