Dusk BLS and hash reviews

JP Aumasson

2024-07-17

Contents

This document describes our review of Dusk's

- BLS signatures implementation (based on the BLS12-381 curve)
- Circuit-friendly hashing (based on Poseidon and the SAFE mode)

The review was carried out between January and April 2024.

After sharing our results, we reviewed the patches proposed by the Dusk team to address our findings, and ensured that they effectively mitigated the issues report:

For BLS:

- BLS01 Hash to scalar incomplete
- BLS02 Secret values not erased
- BLS03 Missing check in signature verification
- BLS04 Missing check in public key aggregation

Hashing:

- HASH01: Round number off-by-one
- HASH02: Error typos

- HASH03: Cargo test error
- HASH04: Always true condition check

BLS12-381 BLS signature review

We reviewed https://github.com/dusk-network/bls12_381-bls/ based on the goals and scope agreed with Dusk.

We checked the code for generic Rust issues, ran the `clippy`, `geiger`, and `audit` utilities, reviewed the code for cryptographic defects (randomness issues, unsafe primitives, insecure logic, etc.), notably assessing the BLS signatures' functionality implementation against the reference specification from the IETF Internet Draft.

We reviewed all the code under src/, checked the tests in tests/ for obvious issues, and reviewed the choice of dependencies. We also reviewed part of the code of the dusk-network/bls12_381/ repository when necessary.

Below we describe our findings, where the most problematic is the lack of public key identity check (BLS-03 and -04), although the corresponding test function is defined but not used. This issue is easily fixed.

BLS01: Hash to scalar incomplete

The `h()` function in hash.rs hashes to a subgroup element by truncating a 64-byte BLAKE2B hash to `BlsScalar::SIZE=32` bytes, then clearing the top 2 bytes to obtain a value less than 2^{254}.

```rust
/// Hash an arbitrary slice of bytes into a [`BlsScalar`]
fn h(msg: &[u8]) -> BlsScalar {
    let mut digest: [u8; BlsScalar::SIZE] = Blake2b::digest(msg).into();

    // Truncate the contract id to fit bls
    digest[31] &= 0x3f;

    let hash: Option<BlsScalar> = BlsScalar::from_bytes(&digest).into();
    hash.unwrap_or_default()
}
```

However with such an implementation, certain subgroup elements cannot be reached: as the subgroup size

0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001 is approx. $2^{254.857}$.

It means that certain points will never be reached by `h0()`, and certain scalars will never be reached by `h1()` (approx. $2^{253.7}$.)

This appears to us not to be exploitable, but it would be safer to implement the standard `hash_to_field()` function, as defined in the Hashing to Ellpitic Curve RFC and implemented in zkcrypto/bls12_381.

BLS-02: Secret values not erased

Secret values allocated to memory are not securely erased after going out of scope. There is no perfect solution to consistently address that issue, but the zeroize crate is probably the best option out there. It's relatively easy to integrate, by assigning the `Zeroize` trait, and is `no_std` like the library.

BLS-03: Missing check in signature verification

Secure verification of BLS signatures requires several checks, as described in the reference IETF Internet Draft (section 2.7):

```
result = CoreVerify(PK, message, signature)

Inputs:
- PK, a public key in the format output by SkToPk.
- message, an octet string.
- signature, an octet string in the format output by CoreSign.

Outputs:
- result, either VALID or INVALID.

Procedure:
1. R = signature_to_point(signature)
2. If R is INVALID, return INVALID
3. If signature_subgroup_check(R) is INVALID, return INVALID
4. If KeyValidate(PK) is INVALID, return INVALID
5. xP = pubkey_to_point(PK)
6. Q = hash_to_point(message)
7. C1 = pairing(Q, xP)
8. C2 = pairing(R, P)
9. If C1 == C2, return VALID, else return INVALID
```

where `KeyValidate()` checks that a public key is a curve point in the subgroup and is not the identity element.

However, the implementation in keys/public.rs does not do all these required checks:

```rust
impl PublicKey {
    /// Verify a [`Signature`] by comparing the results of the two pairing
    /// operations: e(sig, g_2) == e(H0(m), pk).
    pub fn verify(&self, sig: &Signature, msg: &[u8]) -> Result<(), Error> {
        let h0m = h0(msg);
        let p1 = dusk_bls12_381::pairing(&sig.0, &G2Affine::generator());
        let p2 = dusk_bls12_381::pairing(&h0m, &self.0);

        if p1.eq(&p2) {
            Ok(())
        } else {
            Err(Error::InvalidSignature)
        }
    }
```

Specifically, the public key may be checked to be on the curve and in the subgroup when instantiated with `from_bytes()` or from a secret key, however these do not guarantee that the point is not the identity. To check the public key, the `is_valid()` defined in keys/public.rs may be used.

The signature, as a point, would be checked to be valid (on the curve, in the subgroup) when instantiated via the `from_bytes()` method of the underlying library.

To verify that an invalid public key is accepted in signature generation and verification, we can run the following test:

```
fn pktest() {

    let rng = &mut StdRng::seed_from_u64(0xbeef);
    let msg = random_message(rng);
    let sk = SecretKey::from(BlsScalar::zero());
    let pk = PublicKey::from(&sk);
    let sig = sk.sign_vulnerable(&msg);

    assert!(pk.verify(&sig, &msg).is_ok());
    assert!(pk.is_valid())
}
```

The first assert will pass, only the second will fail:

```
thread 'pktest' panicked at tests/signature.rs:24:5:
assertion failed: pk.is_valid()
```

BLS-04: Missing check in public key aggregation

For the same reasons as in BLS-03, the public key aggregation function omits the identity check, though will do the subgroup check when instantiated with `from_bytes()`:

```
    pub fn aggregate(&mut self, pks: &[PublicKey]) {
        #[cfg(feature = "parallel")]
        let iter = pks.par_iter();
        #[cfg(not(feature = "parallel"))]
        let iter = pks.iter();

        let sum: G2Projective = iter
            .map(|pk| dusk_bls12_381::G2Projective::from(pk.pk_t()))
            .sum();
        (self.0).0 = ((self.0).0 + sum).into();
    }
```

Likewise, `is_valid()` may be used, or just `is_identity()`.

Note that the signature aggregation does not require an identity check.

Poseidon & SAFE security code review

We reviewed the https://github.com/dusk-network/Poseidon252 (crate “dusk-poseidon”, v0.37.0-rc.0) https://github.com/dusk-network/safe (crate “dusk-safe”, v0.2.0-rc.0) based on the goals and scope agreed with Dusk, and assessing the SAFE implementation against the official specification (but recognizing that the implementation reviewed is optimized for authenticate encryption and does not aim to implement all SAFE features).

We checked the code for generic Rust issues, ran the `clippy`, `geiger`, and `audit` utilities, reviewed the code for cryptographic defects (randomness issues, unsafe primitives, insecure logic, etc.). notably assessing the compliance of the SAFE implementation with its specification in https://eprint.iacr.org/2023/522. Note that both crates are `no_std`, thus restricting their attack surface and functionalities.

We reviewed all the code under src/ and reviewed the choice of dependencies. For Hades, we also reviewed the code in assets/, used to generate binary data files. The test cases were not reviewed for security, but used as a basis for various tests. For Poseidon, we also reviewed the code in HOWTO.md, as used to generate constants and MDS matrices.

Below we describe our observations — none of which is a security issue, but only minor quality issues. We found the implementations of Poseidon and SAFE consistent with their specification, and reliable from a security perspective, with proper input validation and error handling.

Poseidon252

The dusk-poseidon crate implements the Poseidon algebraic hash over the BLS12-381 curve’s scalar field. It also defines sponge hashing as well as authenticated encryption and decryption functions leveraging the implementation of these modes in the dusk-safe crate.

HASH01: Round number off-by-one

Table~2 in https://eprint.iacr.org/2019/458 defined 60 partial rounds for a width-5 128-bit Poseidon whereas the implementation does 59 rounds, is there a specific reason for this? This is obviously not a security issue in any case.

As defined in hades.rs:

```
const FULL_ROUNDS: usize = 8;

const PARTIAL_ROUNDS: usize = 59;
```

(We noted that the round number was bumped to 60 in the latest version, after sharing this observation with the authors.)

HASH02: Error typos

In hades.rs, it should be "Absorption":

```
118:            .expect("Absorbtion of the input should work fine");
122:            .expect("Absorbtion of padding should work fine");
```

HASH03: Cargo test error

A `cargo test` without `--features` option will return errors due to the inclusion of documentation in lib.rs (`#![doc = include_str!("../README.md")]`). This may be fixed by reformatting the code in the documentation page.

SAFE

The dusk-safe crate implements the sponge logic as well as SAFE-based authenticated encryption and decryption logic, over some finite field type defined by the caller.

HASH04: Always true condition check

In safe/encryption.rs, the function `decrypt()` must ensure that the cleartext message has length equal to that of the ciphertext minus one (as one element is the authentication tag). As we can see in the excerpt below, `decrypt()` sets a variable `message_len` to `cipher.len() - 1`, and neither of these values is modified throughout the function.

However, the last lines of this excerpt show an `if` statement asserting that `cipher.len()` equals `message_len + 1`, a condition that will always be true by definition of `message_len`. This check can therefore be omitted.

```
    let cipher = cipher.as_ref();
    let message_len = cipher.len() - 1;
    let mut sponge = prepare_sponge(
        safe,
        domain_sep.into(),
        message_len,
        shared_secret,
        nonce,
    )?;

    // construct the message by subtracting sponge.output from the cipher
    let mut message = Vec::from(&sponge.output[..]);
    for i in 0..message_len {
        message[i] = sponge.safe.subtract(&cipher[i], &message[i]);
    }

    // absorb the obtained message
    sponge.absorb(message_len, &message)?;

    // squeeze 1 element
    sponge.squeeze(1)?;

    // assert that the last element of the cipher is equal to the last element
    // of the sponge output
    let s = sponge.output[message_len];
    if !sponge.safe.is_equal(&s, &cipher[message_len]) {
        message.zeroize();
        sponge.zeroize();
        return Err(Error::DecryptionFailed);
    };

    // cipher must yield exactly message_len + 1 elements
    if cipher.len() != message_len + 1 {
        return Err(Error::DecryptionFailed);
    }
```

It follows for example that tests such as the last assert in this excerpt from Poseidon252's tests/encryption_gadget.rs will always pass:

```
  pub fn random(rng: &mut StdRng) -> Self {
        let mut message = [BlsScalar::zero(); L];
        message
            .iter_mut()
            .for_each(|s| *s = BlsScalar::random(&mut *rng));
        let shared_secret =
            GENERATOR_EXTENDED * &JubJubScalar::random(&mut *rng);
        let nonce = BlsScalar::random(&mut *rng);
        let cipher = encrypt(&message, &shared_secret.into(), &nonce)
            .expect("encryption should pass");
```

```
assert_eq!(message.len() + 1, cipher.len());
```