



VetKeys Cryptography Review

DFINITY USA Research
Version 1.0 – October 8, 2025

© 2025 – NCC Group

Prepared by NCC Group Security Services, Inc. for DFINITY USA Research. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

Prepared By
Paul Bottinelli
Marie-Sarah Lacharité
Javed Samuel

Prepared For
Andrea Cerulli
Robin Künzler
Franz-Stefan Preiss

1 Executive Summary

Synopsis

During the spring of 2025, NCC Group was engaged to perform a cryptography review of DFINITY USA Research's verifiably encrypted threshold key derivation (vetKD) protocol implementation. The use of vetKD in the Internet Computer allows users to securely and privately derive keys (vetKeys) tied to their identity.

The protocol itself uses nodes' BLS signature threshold key shares x_i (vetKD master key shares) to sign (a hash of) the user's identity, then ElGamal-encrypt this signature with an ephemeral transport key generated by the user. The encrypted signature is verifiable in the sense that the ciphertext (C_1, C_2, C_3) can be proven to have the form $(g_1^t, g_2^t, H(\text{id})^x \cdot pk^t)$ for some given user identifier id , ElGamal public key pk , BLS signing key x , and random value t . Only the holder of the private key corresponding to pk can decrypt this message to obtain the vetKey (BLS signature $H(\text{id})^x$), from which it can derive keys.

The initial phase of the review was delivered remotely by two consultants over two calendar weeks for a total of ten person-days. In summer 2025, two person-days were allocated to retesting fixes to the findings.

Scope

The following items were in scope with leading priority:

- From [commit 35c233e](#) of the [dfinity ic repository](#):
 - [rs/crypto/internal/crypto_lib/bls12_381/vetkd](#): low-level cryptographic protocol for nodes (creating, serializing/deserializing, and combining encrypted key shares)
 - [rs/crypto/internal/crypto_service_provider/src/vault/local_csp_vault/vetkd](#): Crypto Service Provider Vault (wrapper around encrypted key share creation function)
 - [rs/crypto/src/vetkd/mod.rs](#): high-level cryptographic protocol for nodes (creating, verifying, and combining encrypted key shares; verifying encrypted key)
- From [commit 5885675](#) of the [dfinity vetkeys repository](#):
 - [backend/rs/ic_vetkeys/src/utls/mod.rs](#): Rust client-side helper library for integrating vetKeys into canisters (transport key creation, decrypting encrypted vetKeys, canister key derivation)
 - [frontend/ic_vetkeys/src/utls](#): TypeScript client-side helper library (transport key creation, canister key derivation, vetKey decryption, key derivation from vetKey, AES-GCM encryption/decryption, IBE encryption/decryption)

The following items were in scope with secondary priority:

- From [commit 35c233e](#) of the [dfinity ic repository](#):
 - [rs/consensus/vetkd](#): vetKD payload builder
 - [rs/consensus/dkg/src/payload_builder.rs](#): use of existing DKG implementation for vetKD
 - [rs/consensus/idkg/src/signer.rs](#): threshold signer for vetKD
 - [rs/execution_environment/src/execution_environment.rs](#): canister code for handling requests

To guide this review, NCC Group also reviewed the vetKeys paper *vetKeys: How a Blockchain Can Keep Many Secrets*¹, vetKeys reference², and a snapshot of the Google Doc *IC-1163: vetKeys System Integration* (shared via email on May 19, 2025).

1. <https://eprint.iacr.org/2023/616>

2. <https://internetcomputer.org/docs/references/vetkeys-overview>



Limitations

The vetKD protocol is part of the Internet Computer ecosystem, and it relies on a number of building blocks, such as the underlying distributed key generation or some low-level pairing-friendly elliptic curve operations. The review was constrained solely to the in-scope elements identified above and elements outside of the scope were not reviewed.

Nevertheless, good coverage of the in-scope elements was achieved within the given time frame. Additionally, the development and implementation of state-of-the-art cryptographic algorithms may have subtle issues that a time-boxed review may not necessarily uncover.

Key Findings

NCC Group reported five (5) security findings split between three (3) informational findings, and two (2) low security findings:

- [Finding "Inconsistent Handling of Duplicate Shares"](#), where discrepancies in the result of the share combination protocol depending on the location of duplicate shares may lead to denial-of-service or consensus issues.
- [Finding "Potentially Problematic Missing Upper Bound on Message Size"](#), in which a malicious user can make the target node allocate large amounts of memory, which could lead to a denial of service due to a panic, and where the NCC Group team noted that the implementation may behave differently on 32-bit and 64-bit platforms when attempting to encrypt large messages.

Fixes to these findings were provided by DFINITY USA Research and reviewed by NCC Group in summer 2025. By the end of the retest, three (3) findings were considered fixed, one (1) low finding was considered partially fixed, and DFINITY USA Research accepted the risk of the one (1) remaining informational finding.

Additional comments are collected in [Engagement Notes](#), and an overview of the vetKD protocol is given in [VetKD Protocol Overview](#).

Strategic Recommendations

While the implementation appears to be well-structured, documented and tested, a few outstanding `TODO`s are still present in the code base. Consider addressing these items prior to any wide-scale deployment of the vetKD protocol.

Additionally, the algorithms implemented could be annotated with direct references to the paper, which would help future reviewers and drive community adoption of this novel algorithm.



2 Dashboard

Target Data

Name	vetKD Protocol
Type	Cryptographic library and consensus layer
Platforms	Rust Implementation
Environment	Local


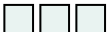
Engagement Data

Type	Cryptography Review
Method	Code-assisted
Dates	2025-05-20 to 2025-06-05
Consultants	2
Level of Effort	12 person-days

Targets

DFINITY ic repository at commit 35c233e	Main priority: implementation of the low-level cryptographic protocol for nodes, Crypto Service Provider Vault, and high-level vetKD protocol. Secondary priority: Consensus-layer use of the vetKD protocol.
DFINITY vetkeys repository at commit 5885675	Rust and TypeScript client-side helper libraries for the integration of vetKeys into canisters.


Finding Breakdown



Critical issues	0
High issues	0
Medium issues	0
Low issues	2 
Informational issues	3 
Total issues	5

Category Breakdown

Cryptography	3 
Data Exposure	1 
Data Validation	1 

Component Breakdown

General	1 
dfinity/ic	2 
dfinity/vetkeys	2 

 Critical  High  Medium  Low  Informational



3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Inconsistent Handling of Duplicate Shares	Fixed	XD6	Low
Potentially Problematic Missing Upper Bound on Message Size	Partially Fixed	RVN	Low
Fragile Check of VetKD Key Type	Fixed	22K	Info
No Identity Check in BLS Verification	Risk Accepted	D7X	Info
Missing Memory Zeroization	Fixed	XPJ	Info



4 Finding Details

Low

Inconsistent Handling of Duplicate Shares

Overall Risk Low

Impact Medium

Exploitability None

Finding ID NCC-E024606-XD6

Component dfinity/ic

Category Cryptography

Status Fixed

Impact

Discrepancies in the result of the share combination protocol depending on the location of duplicate shares may lead to denial-of-service or consensus issues.

Description

The function `combine_valid_shares()` defined in the file `rs/crypto/internal/crypto_lib/bls12_381/vetkd/src/lib.rs` is one of the key features of the vetKD protocol and is used to combine a set of encrypted shares into the final encrypted key (which is subsequently delivered to and decrypted by the original user).

After performing a basic check on the length of the shares passed as parameter (in order to ensure a sufficient threshold can be reached, see line 284 of the excerpt below), the function iterates over the list of shares and appends them to the vector `valid_shares` (see highlighted line 293), provided the share at the current index is valid. The function execution then exits as soon as the number of valid shares reaches the reconstruction threshold, as can be seen highlighted on lines 295-296 below.

This (potentially early) loop exit may be problematic when encountering duplicate shares since the function produces different outcomes based on the duplicate share location. Specifically, if the list of shares contains a duplicate entry at an index located

- past the `reconstruction_threshold`, the function succeeds;
- before the `reconstruction_threshold`, the function fails on line 312 (highlighted below).

This discrepancy may not be expected. Indeed, the documentation preceding the function states that the function

Returns an error if not sufficient shares are given or if not sufficient valid shares are given

However, in the case of duplicate shares within the first `reconstruction_threshold`, an error would be returned *even though* sufficiently valid shares are given. Hence, the order in which the shares are provided to the `combine_valid_shares()` matters; for this function to succeed, potentially duplicate shares must be located past the threshold. This reliance on a specific ordering in the presence of duplicate shares may lead to denial-of-service or consensus issues.

```
270    /// Filters the valid shares from the given ones, and combines them into a valid key,
    /// ↳ if possible.
271    /// The returned key is guaranteed to be valid.
272    /// Returns an error if not sufficient shares are given or if not sufficient valid
    /// ↳ shares are given.
273    /// Takes also the nodes' individual public keys as input, which means the individual
    /// ↳ public keys
```



```

274    /// must be available: calculating them is comparatively expensive. Note that
    ↳ combine_all does not
275    /// take the individual public keys as input.
276    pub fn combine_valid_shares(
277        nodes: &[(NodeIndex, G2Affine, EncryptedKeyShare)],
278        reconstruction_threshold: usize,
279        master_pk: &G2Affine,
280        tpk: &TransportPublicKey,
281        context: &DerivationContext,
282        input: &[u8],
283    ) -> Result<Self, EncryptedKeyCombinationError> {
284        if nodes.len() < reconstruction_threshold {
285            return Err(EncryptedKeyCombinationError::InsufficientShares);
286        }
287
288        // Take the first reconstruction_threshold shares which pass validity check
289        let mut valid_shares = Vec::with_capacity(reconstruction_threshold);
290
291        for (node_index, node_pk, node_eks) in nodes.iter() {
292            if node_eks.is_valid(master_pk, node_pk, context, input, tpk) {
293                valid_shares.push((*node_index, node_eks.clone()));
294
295                if valid_shares.len() >= reconstruction_threshold {
296                    break;
297                }
298            }
299        }
300
301        if valid_shares.len() < reconstruction_threshold {
302            return Err(EncryptedKeyCombinationError::InsufficientValidKeyShares);
303        }
304
305        let c = Self::combine_unchecked(&valid_shares, reconstruction_threshold)?;
306
307        // If sufficient shares are available, and all were valid (which we already
    ↳ checked)
308        // then the resulting signature should always be valid as well.
309        //
310        // This check is mostly to catch the case where the reconstruction_threshold was
    // somehow incorrect.
311        if c.is_valid(master_pk, context, input, tpk) {
312            Ok(c)
313        } else {
314            Err(EncryptedKeyCombinationError::ReconstructionFailed)
315        }
316    }
317 }

```

Fortunately, in the current implementation, the function `combine_valid_shares()` is never called in a way that may allow the `nodes` parameter to contain exact duplicates. The exploitability of this finding was marked as *None* as a result.

Recommendation

Consider updating the function `combine_valid_shares()` to ensure that there are no duplicate entries in the `nodes` parameter. If duplicate entries are allowed by the protocol, prune the `nodes` parameter of duplicate entries before proceeding to share validation.



Location

rs/crypto/internal/crypto_lib/bls12_381/vetkd/src/lib.rs

Retest Results

2025-08-15 – Fixed

In [Pull Request 5459](#) of the ic repository, the function `combine_valid_shares()` was modified to take a set (as a `BTreeMap`) of nodes as input instead of a list. This new type ensures that no duplicate shares (with nodes uniquely identified by their `u32` indices) can be passed to the function, and thus there is no longer the possibility of inconsistent handling of duplicate shares.



Potentially Problematic Missing Upper Bound on Message Size

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E024606-RVN

Component dfinity/vetkeys

Category Data Validation

Status Partially Fixed

Impact

A malicious user can make the target node allocate large amounts of memory, which could lead to a denial of service due to a panic. Additionally, the implementation may behave differently on 32-bit and 64-bit platforms when attempting to encrypt large messages.

Description

The file `vetkeys/backend/rs/ic_vetkeys/src/utils/mod.rs` contains a number of high-level functions to be used by developers to interact with the vetKeys implementation. Among these functions, this file implements an Identity-Based Encryption scheme, which is an extension of the Boneh-Franklin IBE scheme also described in section 6.2 of the reference paper³. This finding highlights some potential concerns with the encryption function when attempting to encrypt large messages. The documentation preceding that function, as well as the function signature, are provided below for reference.

```
530  /// Encrypt a message using IBE
531  ///
532  /// The message can be of arbitrary length
533  ///
534  /// The seed must be exactly 256 bits (32 bytes) long and should be
535  /// generated with a cryptographically secure random number generator. Do
536  /// not reuse the seed for encrypting another message or any other purpose.
537  ///
538  /// To decrypt this message requires using the VetKey associated with the
539  /// provided derived public key (ie the same master key and context string),
540  /// and with an `input` equal to the provided `identity` parameter.
541  pub fn encrypt(
542      dpk: &DerivedPublicKey,
543      identity: &[u8],
544      msg: &[u8],
545      seed: &[u8],
546  ) -> Result<IBECiphertext, String>
```

Of note, the documentation preceding the function states that it does not enforce any bounds on the plaintext size:

└ The message can be of arbitrary length

Correspondingly, the `encrypt()` function does not impose any upper bounds on the length of the `msg` parameter. This comment is slightly misleading in that there exists some limit on the size of the plaintext; trying to encrypt a very large message may lead to issues due to memory exhaustion, and is inherently limited by the maximum size of Rust vectors. Specifically, the plaintext encryption itself is performed by the function `mask_msg()`

3. <https://eprint.iacr.org/2023/616>



excerpted below, which expands a seed passed as parameter with SHAKE256, an extendable-output function (XOF), and XORs the message with the output of that XOF. During the key expansion step, the variable `mask` is instantiated as a Rust Vector of the same size as the input message with the `vec!` macro, see highlighted below.

```
fn mask_msg(msg: &[u8], seed: &[u8; IBE_SEED_BYTES]) -> Vec<u8> {
    fn derive_ibe_ctext_mask(seed: &[u8], msg_len: usize) -> Vec<u8> {
        use sha3::{
            digest::{ExtendableOutputReset, Update, XofReader},
            Shake256,
        };

        let mut shake = Shake256::default();
        shake.update(seed);

        let mut xof = shake.finalize_xof_reset();
        let mut mask = vec![0u8; msg_len];
        xof.read(&mut mask);
        mask
    }

    let domain_sep = IBEDomainSep::MaskMsg(msg.len());

    let shake_seed = derive_symmetric_key(seed, &domain_sep.to_string(), 32);

    let mut mask = derive_ibe_ctext_mask(&shake_seed, msg.len());

    for i in 0..msg.len() {
        mask[i] ^= msg[i];
    }

    mask
}
```

Thus, the encryption of a message will trigger a memory allocation of equivalent size, which may be problematic for large messages and could cause a runtime panic due to insufficient memory. Additionally, the Rust language has inherent limits on the size of allocated memory for `Vector`s. This limitation is detailed in the Rust `vec` module [documentation](#), which states:

Vectors ensure they never allocate more than `isize::MAX` bytes.

An interesting side-effect of that limit is that the maximum size of messages that can be encrypted is architecture-dependent, since the primitive signed integer type `isize` is pointer-sized, as also stated in the [documentation](#):

The size of this primitive is how many bytes it takes to reference any location in memory. For example, on a 32-bit target, this is 4 bytes and on a 64-bit target, this is 8 bytes.

If this code were ever deployed on both 32-bit and 64-bit architectures, some compatibility issues may occur when encrypting large plaintexts.

In summary, contrary to the documentation, messages cannot be of arbitrary length, some compatibility issues may occur if this code is run on both 32-bit and 64-bit platforms, and some potentially large memory allocations may be triggered which could result in panics.



Note that the corresponding function in the TypeScript library (in *frontend/ic_vetkeys/src/utls/utls.ts*) also performs such large memory allocation in the encryption process.

Recommendation

If the encryption of large byte arrays is an operation that is expected to be performed by developers and users, consider updating the implementation of the `encrypt()` function to expose a streaming API. If not, introduce an upper bound on the message size in both the Rust and the TypeScript libraries.

Consider adding documentation to help developers choose what encryption method to use; a hybrid scheme with AES-GCM as provided in the TypeScript library may be more amenable to the encryption of large messages.

Location

vetkeys/backend/rs/ic_vetkeys/src/utls/mod.rs

Retest Results

2025-08-15 – Partially Fixed

In [Pull Request 161](#) of the vetkeys repository, documentation was added to the Rust and TypeScript encryption functions warning users of heap allocations proportional to the size of the message. Library users are thus discouraged from using IBE to directly encrypt large messages. However, no bounds on message size or streaming APIs were introduced.

Client Response

This encryption scheme is not intended for very large messages where heap allocation would be problematic. We have documented this limitation and recommend using IBE for key encapsulation in combination with AES-GCM for encrypting the actual message. We've also decided to defer the introduction of a streaming API until there is a concrete demand for it. Additionally, we opted not to introduce an upper bound on message size, as any such limit would be arbitrary and would not reliably prevent heap allocation issues across all environments. We do however warn users in the documentation that encrypting or decrypting very large messages may result in memory allocation errors.



Fragile Check of VetKD Key Type

Overall Risk	Informational	Finding ID	NCC-E024606-22K
Impact	Undetermined	Component	dfinity/ic
Exploitability	None	Category	Cryptography
		Status	Fixed

Impact

If new non-IDKG master public key types were ever added, keys of that type would be included in the list returned by `VetKdPayloadBuilderImpl::get_enabled_keys_and_expiry()`, potentially allowing unauthorized operations with such keys, since callers may expect vetKD keys.

Description

The function `VetKdPayloadBuilderImpl::get_enabled_keys_and_expiry()` (*rs/consensus/vetkd/src/lib.rs*, line 111) is responsible for fetching vetKD keys enabled for the current subnet. The relevant parts of the function are copied here:

```
/// Return the set of enabled VetKD key IDs and request expiry time according to
/// the chain key config at the registry version corresponding to the given block height.
fn get_enabled_keys_and_expiry(
    &self,
    height: Height,
    context_time: Time,
) -> Result<BTreeSet<MasterPublicKeyId>, RequestExpiry>, PayloadValidationError> {

    // ...

    let config = match self
        .registry
        .get_chain_key_config(self.subnet_id, registry_version)

    // ...

    let key_ids = config
        .key_configs
        .into_iter()
        .map(|key_config| key_config.key_id)
        // Skip keys that don't need to run NIDKG protocol
        .filter(|key_id| !key_id.is_idkg_key())
        // Skip keys that are disabled
        .filter(|key_id| {
            enabled_subnets
                .get(key_id)
                .is_some_and(|subnets| subnets.contains(&self.subnet_id))
        })
        .collect();
    // ...
```

It first gets a list of candidate keys by fetching `KeyConfig`s for the current subnet from the subnet record in the version of the subnet registry at the specified height. From this list of candidate keys, it retains key IDs that *are not IDKG keys*, as determined by `MasterPublicKeyId::is_idkg_key()` (line 2575, highlighted in the preceding code snippet). That function,



copied here, currently returns false only for vetKD keys, so it is effectively checking whether a key is a vetKD key.

```
impl MasterPublicKeyId {  
    /// Check whether this type of [`MasterPublicKeyId`] requires to run on the IDKG protocol  
    pub fn is_idkg_key(&self) -> bool {  
        match self {  
            Self::Ecdsa(_) | Self::Schnorr(_) => true,  
            Self::VetKd(_) => false,  
        }  
    }  
}
```

Currently, there are only three `MasterPublicKeyId` types:

```
pub enum MasterPublicKeyId {  
    Ecdsa(EcdsaKeyId),  
    Schnorr(SchnorrKeyId),  
    VetKd(VetKdKeyId),  
}
```

If new non-IDKG master public key types were ever added, `is_idkg_key()` could return keys that are not vetKD keys, potentially leading to unexpected behavior if the types of the returned `valid_keys` are not checked before being used.

Currently, `get_enabled_keys_and_expiry()` is called in two places. In both cases, there is eventually a call to `reject_if_invalid()` (line 503 of `rs/consensus/vetkd/src/lib.rs`) on the returned set of `valid_keys` and a context with `ThresholdArguments` of type `VetKdArguments` (by construction). `VetKdArguments` contains a key ID of type `VetKdKeyId`. The function will reject the context if its `VetKdKeyId` is not in the set `valid_keys`. Therefore, the current uses of `get_enabled_keys_and_expiry()` are not susceptible to key type confusion even if a new key type were added, so the exploitability of this finding was rated “None”.

Recommendation

Create a new `is_vetkd_key()` function for `MasterPublicKeyId`, and replace the use of `!key_id.is_idkg_key()` with `key_id.is_vetkd_key()` in `get_enabled_keys_and_expiry()`.

Location

`rs/consensus/vetkd/src/lib.rs`

Retest Results

2025-08-15 – Fixed

In [Pull Request 5513](#) of the ic repository, a new `is_vetkd_key()` function was defined and used to filter keys in `get_enabled_keys_and_expiry()`, as recommended.



No Identity Check in BLS Verification

Overall Risk Informational
Impact Undetermined
Exploitability Undetermined

Finding ID NCC-E024606-D7X
Category Cryptography
Status Risk Accepted

Impact

Applications failing to reject received curve points that were maliciously set to the identity may enable practical attacks on the underlying cryptographic schemes.

Description

In elliptic curve cryptography, failing to check that a received elliptic curve (EC) public key is not equal to the point at infinity can lead to serious practical attacks. In the context of the BLS signature scheme, the governing RFC mandates such checks under section 5.2.

[Validating public keys:](#)

All algorithms in Section 2 and Section 3 that operate on public keys require first validating those keys. For the basic and message augmentation schemes, the use of KeyValidate is REQUIRED. For the proof of possession scheme, each public key MUST be accompanied by a proof of possession, and use of PopVerify is REQUIRED.

KeyValidate requires all public keys to represent valid, non-identity points in the correct subgroup. A valid point and subgroup membership are required to ensure that the pairing operation is defined (Section 5.3).

Researchers have demonstrated attacks⁴ on implementations missing that identity check, particularly in the context of aggregate signatures. While technically out of scope of the review, the NCC Group team noticed that some of the underlying BLS signature verification functions were missing that check. After some discussions with the DFINITY team, several locations where this check is potentially missing were identified:

- function `verify_bls_signature()` in `rs/crypto/internal/crypto_lib/bls12_381/type/src/lib.rs`,
- multi-signature verification in the crate `rs/crypto/node_key_validation`, and
- BLS signature verification in the standalone [crate verify-bls-signatures](#).

Being out of scope of the current review, the impact and exploitability of this finding were marked Undetermined, and the overall risk was set to Informational.

The NCC Group team also noted that the current implementation of the vetKD protocol missed such checks on curve points. For example, the implementation does not reject a transport public key sent by a user that is set to the identity, nor does it reject key shares set to the identity. No practical attacks were identified in the time allocated to the project.

Recommendation

Consider reviewing the BLS IETF draft to determine the kinds of checks that could be added (subgroup membership, identity check, etc.) to the low-level BLS implementations, and implement them if deemed necessary. Consider also reviewing the security of the vetKD protocol in the presence of public keys and shares set to the identity.

4. <https://eprint.iacr.org/2021/323.pdf>



Retest Results

2025-08-15 – Partially Fixed

The public key identity check was added to BLS signature verification, as dictated in the RFC, in

- `verify_bls_signature_pt()` in *backend/rs/ic_vetkeys/src/utls/mod.rs* (in [Pull Request 141](#) of the vetkeys repository), which is called when vetKeys are created, deserialized, or decrypted; and
- `verify()` in *src/lib.rs* (in [Pull Request 12](#) of the verify-bls-signatures repository).

Client Response

We did not introduce additional checks in the code related to multi and threshold BLS signatures, as these variants do not claim compatibility with the RFC. Moreover, the instantiation of those schemes rely on standard threshold assumptions, under which the referenced attacks are not expected to apply. Regarding the use of the identity element as a transport public key, this is permitted to support use cases where confidentiality is not required. For example, when using the API purely as a threshold BLS signing API.



Missing Memory Zeroization

Overall Risk	Informational	Finding ID	NCC-E024606-XPJ
Impact	Low	Component	dfinity/vetkeys
Exploitability	Undetermined	Category	Data Exposure
		Status	Fixed

Impact

If regions of memory become accessible to an attacker, perhaps via a core dump, attached debugger or disk swapping, the attacker may be able to extract non-cleared secret values.

Description

Typically, all of a function's local stack variables and heap allocations remain in process memory after the function goes out of scope, unless they are overwritten by new data. This stale data is vulnerable to disclosure through means such as core dumps, an attached debugger and disk swapping. As a result, sensitive data should be cleared from memory once it goes out of scope.

Since the results of memory-clearing functions are not used for functional purposes elsewhere, these functions can become the victim of compiler optimizations and be eliminated. There are a variety of "tricks"⁵ to attempt to avoid compiler optimizations and ensure that a clearing routine is performed reliably. The Rust community has largely adopted the approach provided by the Zeroize⁶ crate.

In general, the vetKD implementation in the *ic/rs/crypto* directory was found to adequately use the `Zeroize` and `ZeroizeOnDrop` traits. For example, the `Scalar` struct (defined in *ic/rs/crypto/internal/crypto_lib/bls12_381/type/src/lib.rs* and excerpted below) which represents the private key shares derives these zeroization traits.

```
67 /// An integer of the order of the groups G1/G2/Gt
68 #[derive(Clone, Eq, PartialEq, Zeroize, ZeroizeOnDrop)]
69 pub struct Scalar {
70     value: ic\_bls12\_381::Scalar,
71 }
```

Similarly, the Rust library for canister developers defined in *vetkeys/backend/rs/ic_vetkeys/src/utils/mod.rs* derives these traits for the `TransportSecretKey` structure, see below.

```
64 #[derive(Clone, Zeroize, ZeroizeOnDrop)]
65 /// Secret key of the transport key pair
66 pub struct TransportSecretKey {
67     secret_key: Scalar,
68 }
```

However, a couple of other secret values within that file fail to properly zeroize memory. For one, the `VetKey` structure does not derive these traits, see below.

```
248 /// A verifiably encrypted threshold key derived by the VetKD protocol
249 ///
250 /// A VetKey is a valid BLS signature created for an input specified
```

5. https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17_slides_zhao_mo_yang.pdf

6. <https://docs.rs/zeroize/1.1.1/zeroize/>




```

251 /// by the user
252 ///
253 #[derive(Clone, Debug, Eq, PartialEq)]
254 pub struct VetKey {
255     pt: G1Affine,
256     pt_bytes: [u8; 48],
257 }

```

Additionally, some sensitive values used in the IBE encryption process are also never zeroized; the `seed` parameter highlighted below is effectively a secret key, and is currently defined as a byte slice, which is thus never zeroized.

```

541 pub fn encrypt(
542     dpk: &DerivedPublicKey,
543     identity: &[u8],
544     msg: &[u8],
545     seed: &[u8],
546 ) -> Result<IBECiphertext, String> {
547     let seed: &[u8; IBE_SEED_BYTES] = seed
548         .try_into()
549         .map_err(|_e| format!("Provided seed must be {} bytes long ", IBE_SEED_BYTES))?;
550 }

```

Similarly, the function `mask_msg()` which performs the actual message encryption derives a pseudo-random stream from the seed; that `mask` is a `Vector` of Rust `u8` and is never zeroized.

```

521 let mut mask = derive_ibe_ctext_mask(&shake_seed, msg.len());
522
523 for i in 0..msg.len() {
524     mask[i] ^= msg[i];
525 }

```

Recommendation

Utilize the `zeroize` crate to derive the `Zeroize` and `ZeroizeOnDrop` trait for all sensitive values, particularly in the `vetkeys` library for canister developers.

Location

`vetkeys/backend/rs/ic_vetkeys/src/utis/mod.rs`

Retest Results

2025-08-15 – Fixed

In [Pull Request 144](#) of the `vetkeys` repository, the `Zeroize` and `ZeroizeOnDrop` traits were added to the structs representing the transport secret key, `vetKey`, and IBE seed. In [Pull Request 152](#) of the `vetkeys` repository, the SHAKE seed was zeroized after use in `mask_msg()`. The mask generated from the seed was not zeroized, since not copied and only modified in place (XORed with the message) to create part of the IBE ciphertext.



5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.



Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.



6 Engagement Notes

This informational section consists of comments and observations from the review that did not warrant standalone security findings, but which may nevertheless be of interest to the DFINITY team.

In response to these comments, DFINITY made the following changes:

- [Pull Request 145](#): corrected `deserialize()` documentation and added details on size of `input` parameter to symmetric key derivation function in Rust and TypeScript; and
- [Pull Request 146](#): replaced constants with integer literals.

Minor Code Base Comments

- The `deserialize()` function in `packages/ic-vetkd-utils/src/lib.rs` is preceded by inaccurate documentation, referring to the serialization process.

```
95  /// Serialize this transport secret key to a bytestring
96  pub fn serialize(&self) -> Vec<u8> {
97      self.secret_key.to_bytes().to_vec()
98  }
99
100  /// Serialize this transport secret key to a bytestring
101  pub fn deserialize(bytes: &[u8]) -> Result<Self, String> {
102      if bytes.len() != 32 {
103          return Err(format!(
104              "TransportSecretKey must be exactly 32 bytes not {}",
105              bytes.len()
106          ));
107      }
108  }
```

- The `VetKdProtocol::verify_encrypted_key_share()` function (line 38, `rs/crypto/src/vetkd/mod.rs`) has an ambiguous name. It is not verifying that the encrypted `vetKey` share is valid with respect to the derived public key; it is verifying the node signature over the entire ciphertext. It could be renamed to something more descriptive, such as `verify_encrypted_key_share_node_sig()`.

Minor Documentation Discrepancy

There is a minor documentation discrepancy between the functions `deriveSymmetricKey()` in the TypeScript library (`frontend/ic_vetkeys/src/utils/utils.ts`) and the equivalent function `derive_symmetric_key()` in the Rust library (`backend/rs/ic_vetkeys/src/utils/mod.rs`).

The TypeScript `deriveSymmetricKey()` function explicitly warns users about the length of the key derivation `input`, though it does not state what *sufficient* length is expected.

```
269  /**
270   * @internal derive a symmetric key from the provided input
271   *
272   * The `input` parameter should be a sufficiently long random input.
273   *
274   * The `domainSep` parameter should be a string unique to your application and
275   * also your usage of the resulting key. For example say your application
276   * "my-app" is deriving two keys, one for usage "foo" and the other for
277   * "bar". You might use as domain separators "my-app-foo" and "my-app-bar".
278   */
279  export function deriveSymmetricKey(
280      input: Uint8Array,
281      domainSep: Uint8Array | string,
282      outputLength: number,
```



```

283 ): Uint8Array {
284     const no_salt = new Uint8Array();
285     return hkdf(sha256, input, no_salt, domainSep, outputLength);
286 }

```

In comparison, the Rust function `derive_symmetric_key()` is missing this warning.

```

279 /**
280  * Derive a symmetric key of the requested length from the VetKey
281  *
282  * The `domain_sep` parameter should be a string unique to your application and
283  * also your usage of the resulting key. For example say your application
284  * "my-app" is deriving two keys, one for usage "foo" and the other for
285  * "bar". You might use as domain separators "my-app-foo" and "my-app-bar".
286  */
287 pub fn derive_symmetric_key(&self, domain_sep: &str, output_len: usize) -> Vec<u8> {
288     derive_symmetric_key(&self.pt_bytes, domain_sep, output_len)
289 }

```

Consider updating that warning to specify an exact minimum length, and mirroring the warning to the Rust library.

Literal Values Instead of Constants

- In `packages/ic-vetkd-utils/src/lib.rs`, the function `mask_msg()` calls the `derive_symmetric_key()` with a literal value `32`, as can be seen below.

```

fn mask_msg(msg: &[u8], seed: &[u8; IBE_SEED_BYTES]) -> Vec<u8> {

    // ...

    let shake_seed = derive_symmetric_key(seed, &domain_sep.to_string(), 32);

```

Consider replacing this with a constant, such as `IBE_SEED_BYTES` defined a few lines above, which is equal to `32`.

- In the function `encryptMessage()` defined in the file `frontend/ic_vetkeys/src/utls/utls.ts`, the IV generation performed via a call to `getRandomValues()` uses an integer literal, see highlighted below.

```

464 async encryptMessage(
465     message: Uint8Array | string,
466     domainSep: Uint8Array | string,
467 ): Promise<Uint8Array> {
468     const gcmKey = await this.deriveAesGcmCryptoKey(domainSep);
469
470     // The nonce must never be reused with a given key
471     const nonce = window.crypto.getRandomValues(new Uint8Array(12));

```

Interestingly, the decryption counterpart defines a constant for that value, see highlighted below.

```

490 async decryptMessage(
491     message: Uint8Array,
492     domainSep: Uint8Array | string,
493 ): Promise<Uint8Array> {
494     const NonceLength = 12;
495     const TagLength = 16;

```



Consider moving that constant out of the `decryptMessage()` function to make it accessible to both functions, and replace the literal value in `encryptMessage()` with it.

Rust Implementation is Missing AES-GCM Encryption

The TypeScript library exposes an AES-GCM encryption function (`encryptMessage()` in the file *frontend/ic_vetkeys/src/utls/utls.ts*), which uses a vetKD-derived key to encrypt data using AES-GCM. The NCC Group team noted that no such function existed in the Rust library. It might be worthwhile to expose such functionality to canister developers.



7 VetKD Protocol Overview

Let $e : G_1 \times G_2 \rightarrow G_T$ be the BLS12-381 pairing, and G_1 , G_2 , and G_T be the appropriate groups of prime order r with generators g_1 , g_2 , and g_t .

VetKD (BLS) Key Generation

In each epoch, each node i of a subnet has a threshold share x_i in \mathbb{Z}_r of a particular BLS signature secret key x , identified by its key ID (e.g. “vetKD production” or “vetKD test”), with corresponding *master public key* $mpk = g_2^x$. Nodes update their shares of the key in each epoch (“resharing”), but the master public key does not change.

The threshold shares are computed using a polynomial $f(i) = a_0 + a_1i + a_2i^2 + \dots + a_{t-1}i^{t-1}$ where the coefficients a_i are integers mod r and the constant term a_0 equals the BLS signature secret key x . The so-called *public coefficients* are $g_2^{a_i}$ for i from 1 to $t - 1$. Node i (between 1 and n) receives the share $x_i = f(i)$ and shares its *master public key share* $mpk_i = g_2^{x_i}$. The public coefficients are stored by all nodes in a *transcript* of the Distributed Key Generation (DKG) protocol that issued the initial shares x_i for the key having that key ID.

The DKG protocol used for vetKD is non-interactive but similar to the interactive DKG protocol for threshold ECDSA, which NCC Group reviewed⁷ in 2022.

VetKD introduces two new functions on the (virtual) *management canister*, which only other canisters may call:

- **VetKdPublicKey**: given a key ID, a canister ID, and a(n optional) context, return the *derived public key* that allows verifying encrypted vetKeys.
- **VetKdDeriveKey**: given a key ID, a(n optional) context, an input, and a *transport public key*, return the *vetKey* for that input, encrypted with the transport public key, and verifiable with the derived public key corresponding to the caller's canister ID (and context).

Canisters can call these two functions to build more complex protocols.

VetKD (BLS) Public Key Derivation

Let mpk be the master public key for a given key ID. Let $H2F(msg, dst)$ be the hash-to-field function from Section 5 of RFC 9380⁸ that outputs one element in the field of scalars for the BLS12-381 subgroups, uses the XMD `expand_message` variant with SHA256, and uses security parameter 128. Its inputs are the message to be hashed, msg , and a data separation tag, dst . Two data separation tags are used in the key derivation implementation, specified here with shorter names for ease of presentation: `DERIV_CANI_DST` (“ic-vetkd-bls12-381-g2-canister-id”) and `DERIV_CONT_DST` (“ic-vetkd-bls12-381-g2-context”). Let $len()$ be the function that returns the length in bytes of its argument, expressed as a `u64`.

- The derived public key for canister ID cid (and an empty context) is computed as $cpk = mpk \cdot g_2^{H2F(len(mpk)||mpk||len(cid)||cid, DERIV_CANI_DST)}$.
- The derived public key for canister cid with non-empty context $context$ is computed as $dpk = cpk \cdot g_2^{H2F(len(cpk)||cpk||len(context)||context, DERIV_CONT_DST)}$.

The value returned by **VetKdPublicKey** is dpk (which equals cpk if no context was provided).

7. <https://www.nccgroup.com/us/research-blog/public-report-threshold-ecdsa-cryptography-review/>.

8. RFC 9380: Hashing to Elliptic Curves, <https://datatracker.ietf.org/doc/html/rfc9380>.



VetKD (BLS) Secret Key Derivation

Node i computes its derived secret key share dsk_i for the canister with ID cid as follows, using the master public key mpk and canister public key cpk :

- The derived secret key share for canister ID cid (and an empty context) is computed as $csk_i = x_i + H2F(\text{len}(mpk)||mpk||\text{len}(cid)||cid, \text{DERIV_CANI_DST})$.
- The derived secret key share for canister cid with non-empty context $context$ is computed as $dsk_i = csk_i + H2F(\text{len}(cpk)||cpk||\text{len}(context)||context, \text{DERIV_CONT_DST})$.

This process parallels the public key derivation process. The corresponding derived public key share is $dpk_i = g_2^{dsk_i}$, where dsk_i equals cpk_i if no context was provided.

VetKey Encrypted Generation and Verification

Nodes compute encrypted vetKeys (the output of `VetKdDeriveKey`) in a distributed manner and then combine their shares. VetKeys are really augmented BLS signatures using the hash-to-curve suite `BLS12381G1_XMD:SHA-256_SSWU_RO_` defined in [Section 8.8.1 of RFC 9380](#), whose main function is denoted $H2G1(msg, dst)$. This hash-to-curve suite is also used by the augmented BLS scheme described in draft-irtf-cfrg-bls-signature-05⁹. The dst argument is passed to hash-to-field, which is the first step of hash-to-curve. Only one DST is used here; `HASH_G1_DST`¹⁰ (`"BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_AUG_"`).

Let $(tsk, tpk = g_1^{tsk})$ be an ElGamal key pair where the secret key tsk is in \mathbb{Z}_r . This transport key pair is used to encrypt the vetKey. Let t_i be an integer mod r . Node i 's encrypted vetKey share for input $input$ is (C_1, C_2, C_3) where $C_1 = g_1^{t_i}$, $C_2 = g_2^{t_i}$, and $C_3 = tpk^{t_i} \cdot (H2G1(dpk||input, \text{HASH_G1_DST}))^{dsk_i}$. This vetKey ciphertext share can be verified by checking the following equations using the node's derived public key share $dpk_i = g_2^{dsk_i}$:

- $e(C_1, g_2) = e(g_1, C_2)$ and
- $e(C_3, g_2) = e(tpk, C_2) \cdot e(H2G1(dpk||input, \text{HASH_G1_DST}), dpk_i)$.

The vetKey ciphertext shares' components are combined in a product using Lagrange basis polynomials, and verification of the combined encrypted vetKey uses the same equations as verification of an encrypted vetKey share.

VetKey Decryption

To decrypt an encrypted vetKey with the transport secret key tsk , the encrypted vetKey is first verified with respect to the derived public key dpk , then decrypted as $k = C_3 / C_1^{tsk}$. Verification when decrypting can be slightly optimized by replacing the second pairing equation with $e(k, g_2) = e(H2G1(dpk||input, \text{HASH_G1_DST}), dpk_i)$. This is used in `EncryptedVetKey::decrypt_and_verify()` in [backend/rs/ic_vetkeys/src/utls/mod.rs](#).

VetKey Uses

Two example uses of VetKeys are provided in the front-end TypeScript library [frontend/ic_vetkeys/src/utls](#): Identity-Based Encryption (IBE) and AES-GCM encryption.

9. IRTF CFRG draft: BLS Signatures, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-05#name-coresign>.

10. The DST was not given a name in the code; `"HASH_G1_DST"` exists only in this report.



The vetKey can be used to implement IBE if it was derived with a user identity as *input*. In this case, the canister would have to authorize the caller with respect to that user identity before calling the `VetKdDeriveKey` method.

The vetKey can also be used as a seed from which to derive a secret key, such as an AES key.

