VeChain JavaScript SDK Cryptography and Security Review

VECHAIN FOUNDATION SAN MARINO SRL
Version 1.2 – March 12, 2025

Prepared By
Paul Bottinelli
Gérald Doussot
Marie-Sarah Lacharité
Eli Sohl

Prepared For
Victor Bibiano
Luca Debiasi
Clayton Neal

1 Executive Summary

Synopsis

In December 2024, VeChain engaged NCC Group's Cryptography Services team to conduct a cryptography and security review of its JavaScript SDK. The SDK allows developers to interact with the VeChain blockchain, and includes essential components such as cryptography- and network-related functions. The review was delivered remotely by 4 consultants with a total effort of 15 person-days, and was followed by a gap and retesting phase of 5 person-days.

The retest found that all findings had been properly acknowledged. A majority of the findings have been *fixed* or *partially fixed*, and a few have been marked as *risk accepted* or slated to fix in an upcoming release.

Scope

The assessment was performed on the vechain-sdk-js v1.0.0 rc4 SDK release. The scope included the following portions of the SDK:

- sdk-core/
  - vechain-sdk-js/packages/core/src/certificate
  - vechain-sdk-js/packages/core/src/hdkey
  - vechain-sdk-js/packages/core/src/keystore
  - vechain-sdk-js/packages/core/src/secp256k1
  - vechain-sdk-js/packages/core/src/vcdm/hash
  - vechain-sdk-js/packages/core/src/vcdm/Mnemonic.ts
- sdk-network/
  - vechain-sdk-js/packages/network/src/signer/signers
  - vechain-sdk-js/packages/network/src/provider (with lower priority on the vechain-sdk-js/packages/network/src/provider/utils/rpc-mapper subdirectory)
- aws-kms/
  - vechain-sdk-js/packages/aws-kms-adapter/src/

In addition, a number of functions were flagged with the remark "Security auditable method" within the code base. While some of these functions were not part of the original scope, the NCC Group team included these functions in the review.

Limitations

The VeChain's JavaScript SDK is a comprehensive project containing many functionalities. The scope of the project was limited to the items explicitly listed above, and the consultants did not attempt to review other portions of the SDK. Similarly, the SDK project dependencies, including the implementation of several low-level cryptographic primitives, were outside of the scope of this review and were not evaluated.

Key Findings

The NCC Group team uncovered a total of 7 findings, among which the most notable were:

- Finding "Underspecified Delegation Process May Lead to Signature Forgery", in which discrepancies between the implementation and the developer documentation may lead to misunderstandings, and where signatures not covering the entirety of the transaction could lead to subtle attacks on the system.
- Finding "Transaction IDs May Collide", where transaction IDs of different transactions may collide, which could lead to unexpected issues if applications expect them to be unique.

A number of informational notes were also captured in the appendix Engagement Notes.

Strategic Recommendations

The SDK release under review contained extensive code comments and function documentation. However, a number of these comments were inaccurate, as noted in section Engagement Notes. Additionally, the fee delegation process exhibited discrepancies and confusion between the documentation and the actual implementation. Consider performing a pass over the code base and the existing documentation to unify naming convention, and to address outdated and inaccurate comments.

The SDK is written in Typescript, which is a statically-typed superset of JavaScript; Typescript introduces a type system which is then checked at compile time. However, once compiled, TypeScript produces plain JavaScript, and no type information remains. As such, type mismatch errors may still occur when the code is executed. Ensure that the SDK properly handles any potential type mismatch errors and consider adding more in-depth parameter validation to SDK functions exposed to developers.

Finally, consider re-thinking the delegation process in order to address the potential signature forgeries identified in finding "Underspecified Delegation Process May Lead to Signature Forgery".

2 Dashboard

Target Data

Name	vechain SDK
Type	Software Development Kit
Platforms	TypeScript
Environment	Local Instance

Engagement Data

Type	Cryptography and Security Assessment
Method	Code-assisted
Dates	2024-12-11 to 2024-12-20
Consultants	4
Level of Effort	20 person-days

Targets

vechain SDK	VeChain SDK repository containing several modules allowing developers to interact with the blockchain. Review performed on v1.0.0 rc4.

Finding Breakdown

Critical issues	0	
High issues	1	🟥
Medium issues	1	🟧
Low issues	4	🟨🟨🟨🟨
Informational issues	1	⬜
Total issues	**7**	

Category Breakdown

Cryptography	2	🟥🟧
Data Exposure	1	🟨
Data Validation	1	🟨
Error Reporting	1	🟨
Other	1	⬜
Patching	1	🟨

🟪 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
Underspecified Delegation Process May Lead to Signature Forgery	Risk Accepted	BNC	High
Transaction IDs May Collide	Risk Accepted	RQY	Medium
Outdated and Vulnerable Dependencies	Partially Fixed	7A9	Low
Potentially Problematic Key Generation Function	Fixed	FR9	Low
Potential Private Key Leak	Fixed	943	Low
Missing Parameter Check in `inflatePublicKey()` Function	Risk Accepted	GWF	Low
Misleading Naming of the `getAddress()` Function	Fixed	NRD	Info

4 Finding Details

High

Underspecified Delegation Process May Lead to Signature Forgery

Overall Risk	High	**Finding ID**	NCC-E020944-BNC
Impact	High	**Category**	Cryptography
Exploitability	Medium	**Status**	Risk Accepted

Impact

Discrepancies between the implementation and the developer documentation may lead to users misunderstanding the fee delegation process. Additionally, signatures do not cover the entirety of the transaction, which could lead to subtle attacks on the system where malicious users could forge delegated signatures.

Description

The VeChain ecosystem supports the concept of Fee Delegation, a process which allows a user to submit a transaction without paying transactions fees; the fees are instead paid by another entity, the gas payer. When the transaction is processed, the fees (i.e., the gas costs) are taken from the gas payer's balance instead of the user who originally submitted the transaction.

This finding describes a number of concerns with the transaction delegation process.

Naming Confusion

There appears to be confusion around the definitions of the originator of the transaction and the gas payer. The code base mostly uses the terms *delegator* and *signer*, evidenced for example by the documentation preceding the function `signWithDelegator()`

> `Signs the transaction using both the signer and the delegator private keys.`

However, some functions incorrectly refer to the signer's address as the delegator's. For example, the `getTransactionHash()` function accepts an optional `Address` parameter, named `delegator` and documented as "Optional delegator's address to include in the hash computation". In practice, this argument is not (and should not be) the delegator's address; it is the signer's.

The online developer resource VIP-191: Designated Gas Payer and the blog post guide How to Setup Fee Delegation for Vechain further add to the confusion by using the terms Designated Gas Payer (or sponsor) for the delegator and user or sender for the original transaction signer.

As also discussed with the VeChain team, the naming convention used in the code base is confusing. The term delegator, defined as being the entity who *delegates*, could be understood as being the original signer, while in practice, the code base uses the term delegator to refer to the gas payer.

Practical Discrepancies

In addition to these naming-related conventions, some confusion also exists with respect to the order in which the two entities sign the transaction, as well as about the content of what is signed.

The blog post guide How to Setup Fee Delegation for Vechain contains a figure aiming to summarize this process. The figure does not accurately represent what the implementation performs. Specifically, the figure (provided below for reference) indicates that the gas payer signs the transaction first, followed by the client.

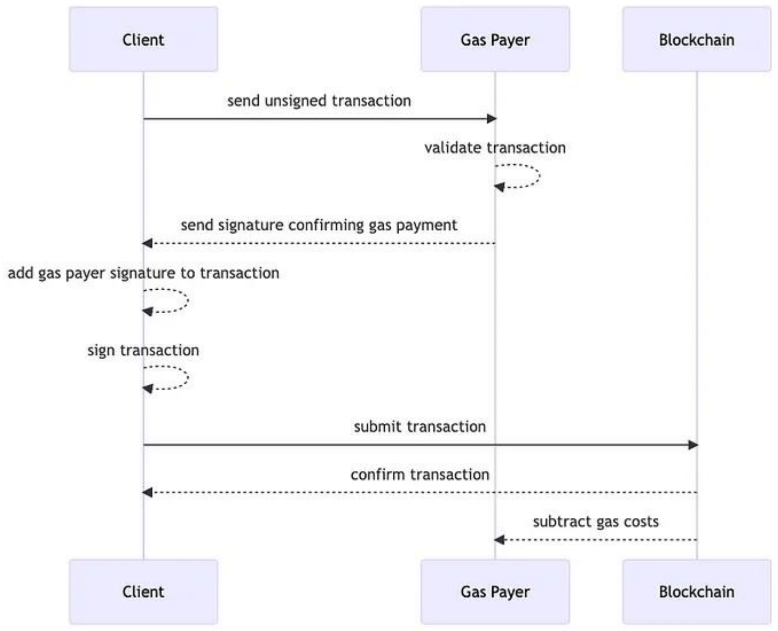

Figure 1: How to Setup Fee Delegation for Vechain

The approach outlined in the figure above could lead to potential misuse of the delegation process. Indeed, with this approach, the gas payer does not explicitly agree to cover the fees for a transaction from a *specific* transaction signer; the delegation signature could be reused for transactions from other signers.

In the file *Transaction.ts*, two implementations supporting the fee delegation process are provided:

- the one-shot function `signWithDelegator()`, which takes in the signer and the delegator private keys as arguments and signs the transaction, and
- the two functions `signForDelegator()` and `signAsDelegator()`, which together are functionally equivalent to the previous function.

In order to sign a transaction, the `signWithDelegator()` function computes the transaction hash and the delegated transaction hash (which incorporates the original signer's address in the hash computation), signs the former with the original signer's private key, and signs the latter with the delegator's private key. This can be seen in the code snippet below, which was excerpted from the function `signWithDelegator()`.

```
const transactionHash = this.getTransactionHash().bytes;
const delegatedHash = this.getTransactionHash(
    Address.ofPublicKey(
        Secp256k1.derivePublicKey(signerPrivateKey)
    )
).bytes;
// Return new signed transaction
return Transaction.of(
    this.body,
    nc_utils.concatBytes(
        Secp256k1.sign(transactionHash, signerPrivateKey),
        Secp256k1.sign(delegatedHash, delegatorPrivateKey)
    )
);
```

Hence, the implementation and the *How to Setup Fee Delegation for Vechain* article diverge in the order in which the transaction is signed.

Delegated Signature Forgery

In order to make sense of what the implementation aims to achieve, the figure below was created to summarize the delegated signature process and aims to highlight a potential shortcoming of the current approach.

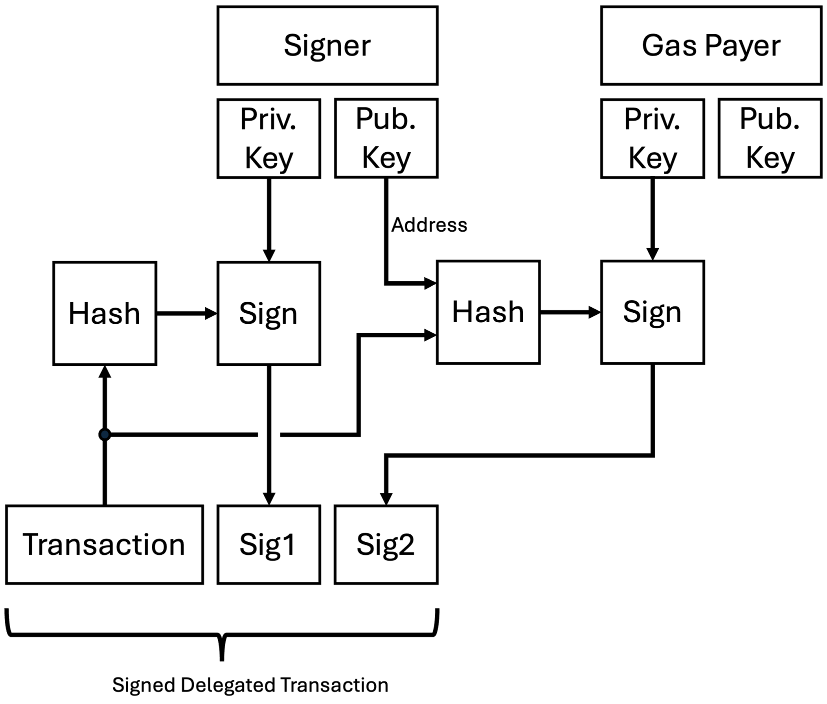

Figure 2: Delegated Signature Process

Importantly, we note that the original signer's signature is *not included* in the hash computation used for the delegator's signature. This means that, in theory (and abstracting away some practical details around transaction contents that might prevent this attack), the current scheme might be vulnerable to the following attack scenario:

- A malicious user creates a message M.
- The user signs this message 10 times, resulting in 10 different signed messages $M_0, \dots, M_9$, where $M_i = (M, \sigma_i)$.
- The user submits one of these messages to be signed by the gas payer, say M_0.
- The gas payer signs the message and returns the signature, say Σ_0.
- Note, the message $M'_0 = (M, \sigma_0, \Sigma_0)$ is a valid delegated message.
- The malicious user can now duplicate the gas payer's signature and append it to every message; all these messages are going to be valid delegated messages. That is $M'_i = (M, \sigma_i, \Sigma_0)$ is a valid delegated message, for all i.

This issue stems partially from the computation of the transaction hash, performed by the function `getTransactionHash()` and provided below, for reference.

```
    public getTransactionHash(delegator?: Address): Blake2b256 {
        const txHash = Blake2b256.of(this.encode(false));
        if (delegator !== undefined) {
            return Blake2b256.of(
                nc_utils.concatBytes(txHash.bytes, delegator.bytes)
            );
        }
        return txHash;
    }
```

In the line highlighted above, the `getTransactionHash()` function encodes the transaction using the `encode()` function, but does so by providing a `false` value to the `isSigned` parameter of that function, even if the transaction already contains a signature. Providing a `false` boolean parameter to the encoding function will essentially discard the signature from the encoding process.

Interestingly, the *Transaction.ts* source file contains the function `encoded()`, which encodes the transaction according to whether it was signed or not, see below.

```
    /**
     * Get the encoded bytes as a Uint8Array.
     * The encoding is determined by whether the data is signed.
     *
     * @return {Uint8Array} The encoded byte array.
     *
     * @see decode
     */
    public get encoded(): Uint8Array {
        return this.encode(this.isSigned);
    }
```

Transaction Hash Computation Collision

The `getTransactionHash()` function excerpted above may also suffer from a vulnerability allowing an attacker to forge a transaction by computing a transaction hash which collides with a delegated transaction hash.

Specifically, the function computes the hash of a transaction tx as $H(tx)$ for a normal transaction, and as $H(H(tx)||address)$ for a delegated transaction, where $address$ corresponds to the original signer's address. Consider a delegated transaction that has been signed by the gas payer, that is, a transaction tx and associated signature σ which was computed on the quantity $H(H(tx)||address)$. Conceptually, the signature by the gas payer on this delegated transaction is no different than a signature by the gas payer on an *encoded* transaction tx' which is equal to $H(tx)||address$. Since the length of the quantity $H(tx)||address$ is equal to 52 bytes (32 bytes for the digest and 20 bytes for the address), and that encoded transactions can be as small as 28 bytes[1], an encoded transaction equal to 52 bytes is well within the realm of possibilities.

1. This number was obtained experimentally by tweaking some existing transaction examples, though it's possible the RLP encoding of a transaction could be even smaller.

As such, the current scheme might be vulnerable to the following attack scenario:

- A malicious user repeatedly creates (potentially meaningless) transactions tx until $H(tx)$ looks like the encoding of a valid transaction.
- Note, the user has some flexibility around their address; once a transaction has been found, they can repeatedly generate private keys and derive addresses until one looks like the encoding of something meaningful.
- The user submits the transaction to the gas payer; the payer signs the payload $H(tx)||address$ and returns the signature σ.
- Now, the transaction $tx' = H(tx)||address$ is a valid transaction under the signature σ; the malicious user essentially forged the transaction tx' signed by the gas payer.

While arguably a little contrived, this attack scenario is not completely impossible and would be serious. The likelihood of this attack hinges upon the probability that hashing a "random" string results in a valid-looking transaction. However, it is amplified by the fact that neither the original signer's address nor the delegator's address are contained in a transaction.

Recommendation

Addressing this finding will require updates to the existing implementation and documentation.

- Update the code and the documentation such that consistent naming conventions are used for the different entities. Consider getting rid of the delegate/delegator terminology.
  - Update the implementation in *packages/core/src/transaction/Transaction.ts* to follow the new naming convention, including the documentation and the variable names of the function `getTransactionHash()`.
  - Also update the implementation under *packages/network/src/signer/signers/* to reflect the new naming convention.
- Update the figure in the *How to Setup Fee Delegation for Vechain* article to reflect what the implementation performs. Go over all existing documentation and developer guides to update outdated terminology to the new naming convention.
- Consider updating the `getTransactionHash()` function such that it incorporates the signature of the original signer when the hash is computed for the delegated signature process by leveraging the `encoded()` function, as follows.

  ```
  const txHash = Blake2b256.of(this.encoded());
  ```

  Note that the impact of this change in the larger context of the VeChain ecosystem is slightly unclear. As such, analyze the impact of this change, particularly with regards to potential signature forgeries and hash collisions.
- Consider adding the signer's address, as well as the gas payer's address, in the transaction structure in order to fix the transaction hash collision computation issue, and ensure the transaction verification process validates the originator of the transaction.
- Additionally, prepending domain separators to the to-be-hashed data (and ensuring these separators could not be interpreted as valid transaction prefixes) might prevent the concern described under "Transaction Hash Computation Collision" above.

Location

- VIP-191: Designated Gas Payer
- How to Setup Fee Delegation for Vechain

- *packages/core/src/transaction/Transaction.ts*
- *packages/network/src/signer/signers/vechain-private-key-signer/vechain-private-key-signer.ts*

Retest Results

2025-02-24 – Partially Fixed

In a series of four commits, the VeChain team addressed the confusing naming of the delegator by replacing instances of “delegator” with “gas payer”:

- commit `f5020f3`,
- commit `148d451`,
- commit `5728313`,
- commit `05fd293`.

This addresses the concerns around “Naming Confusion” listed above.

However, the other matters outlined in this finding, particularly around the “Delegated Signature Forgery” described above, were not addressed. As a result, this finding was marked *Risk Accepted*.

Medium

Transaction IDs May Collide

Overall Risk	Medium	**Finding ID**	NCC-E020944-RQY
Impact	Medium	**Category**	Cryptography
Exploitability	Medium	**Status**	Risk Accepted

Impact

The transaction ID of different transactions may collide, which could lead to unexpected issues if applications expect them to be unique.

Description

The file *packages/core/src/transaction/Transaction.ts* contains functions to operate on transactions. Among them, the `id()` function is defined to compute the transaction ID, which the documentation describes as being the "Blake2b256 hash of the transaction's signature concatenated with the origin's address". The function is provided below, for reference.

```
    /**
     * Get transaction ID.
     *
     * The ID is the Blake2b256 hash of the transaction's signature
     * concatenated with the origin's address.
     * If the transaction is not signed,
     * it throws an UnavailableTransactionField error.
     *
     * @return {Blake2b256} The concatenated hash of the signature
     * and origin if the transaction is signed.
     * @throws {UnavailableTransactionField} If the transaction is not signed.
     *
     * @remarks Security auditable method, depends on
     * - {@link Blake2b256.of}
     */
    public get id(): Blake2b256 {
        if (this.isSigned) {
            return Blake2b256.of(
                nc_utils.concatBytes(
                    this.getTransactionHash().bytes,
                    this.origin.bytes
                )
            );
        }
        throw new UnavailableTransactionField(
            'Transaction.id()',
            'not signed transaction: id unavailable',
            { fieldName: 'id' }
        );
    }
```

A few comments can be made about this function and the documentation preceding it.

1. The function does not actually compute the "Blake2b256 hash of the transaction's signature concatenated with the origin's address". The transaction signature is not incorporated into the hash computation; only the transaction hash as well as the originator's address are taken into account.

2. In the case of delegated transactions, the function above also fails to take the gas payer's (aka delegator's) signature and address into account.

Arguably, IDs should be unique for a given transaction and its associated signature(s). However, if a transaction was signed by a signer multiple times (or in case of different transactions with colliding `getTransactionHash()`), the resulting signed transactions would have the same ID. Even more concerning, if a given to-be-delegated transaction were signed by multiple different gas payers, the resulting IDs would all be equal.

While it initially appeared that this delegated transaction ID would also be equal to the ID of a non-delegated transaction, the VeChain team indicated that this was not the case in practice, since the body of these two types of transactions is slightly different; a *reserved* field is always included in delegated transactions.

As a separate observation, note that the computation of the ID produces the same hash value as the transaction hash computed during the delegated signature process. Specifically, for a given delegated transaction `tx`, `tx.id() === tx.getTransactionHash(this.origin)`, as substantiated by the code excerpt below, from the function `getTransactionHash()`.

```
    if (delegator !== undefined) {
        return Blake2b256.of(
           nc_utils.concatBytes(txHash.bytes, delegator.bytes)
        );
    }
```

Given that nature of the project under review (an SDK), it is unclear whether this last observation may be problematic in practice, but it was deemed worth reporting nonetheless.

Recommendation

Consider modifying the `id()` function such that it includes the signer's signature, and the delegator's address and signature, if the transaction is delegated. Update the documentation to reflect the changes introduced to the function.

Additionally, reflect on whether the `id()` function should product a different digest than the `getTransactionHash()` of a delegated transaction. If the recommendation above is followed, the output of the two functions should be different. If not, consider the potential implications of this type of collision and address it if it has any impact. At the very least, document this behavior.

Location

packages/core/src/transaction/Transaction.ts

Retest Results

2025-02-24 – Not Fixed

The VeChain team indicated:

> We've decided to acknowledge the sub-optimal choice. It's acceptable as duplicated transaction ID would be rejected by every blockchain node.

As such, this finding was marked *Risk Accepted*.

Low

Outdated and Vulnerable Dependencies

Overall Risk	Low	**Finding ID**	NCC-E020944-7A9
Impact	Low	**Category**	Patching
Exploitability	Low	**Status**	Partially Fixed

Impact

An attacker may attempt to identify and utilize vulnerabilities in outdated dependencies to exploit the application.

Description

Incorporating outdated dependencies is one of the most common, serious and exploited application vulnerabilities. The `yarn audit` command can be used to identify potentially vulnerable dependencies in the project. Running this tool on the *vechain-sdk-js* repository outlined 29 vulnerabilities, including one Moderate. A truncated output is provided below, which outlines the Moderate issue identified.

```
yarn audit v1.22.22

...

┌───────────────┬──────────────────────────────────────────────────────────────┐
│ moderate      │ Unpatched `path-to-regexp` ReDoS in 0.1.x                    │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ path-to-regexp                                               │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Patched in    │ >=0.1.12                                                     │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ @vechain/sdk-rpc-proxy                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ @vechain/sdk-rpc-proxy > express > path-to-regexp            │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://www.npmjs.com/advisories/1101081                     │
└───────────────┴──────────────────────────────────────────────────────────────┘

...

29 vulnerabilities found - Packages audited: 1557
Severity: 28 Low | 1 Moderate
Done in 1.07s.
```

Additionally, the SDK contains a number of outdated dependencies. Similar to the tool above, the `yarn outdated` command identifies dependencies that are out-of-date. The truncated output below (from which we also trimmed the Package Type and the URL columns for better visibility) contains a list of the dependencies with "Major Update backward-incompatible updates" and some "Minor Update backward-compatible features".

```
yarn outdated v1.22.22
...
Package                           Current      Wanted      Latest      Workspace
...
@noble/curves                     1.6.0        1.7.0       1.7.0       @vechain/sdk-network
@noble/hashes                     1.5.0        1.6.1       1.6.1       @vechain/sdk-core
@nomicfoundation/hardhat-toolbox 4.0.0        4.0.0       5.0.0       sdk-hardhat-integration
```

```
...
@types/chai                        4.3.20      4.3.20      5.0.1       sdk-hardhat-integration
...
@types/react                       18.3.11     18.3.16     19.0.1      sdk-nextjs-integration
@types/react                       18.3.12     18.3.16     19.0.1      sdk-vite-integration
@types/react-dom                   18.3.1      18.3.5      19.0.2      sdk-nextjs-integration
@types/react-dom                   18.3.1      18.3.5      19.0.2      sdk-vite-integration
@vechain/vebetterdao-contracts     4.0.0       4.1.0       4.1.0       @vechain/sdk-network
...
chai                               4.5.0       4.5.0       5.1.2       sdk-hardhat-integration
...
eslint-plugin-sonarjs              2.0.2       2.0.2       3.0.1       vechain-sdk
...
glob                               10.4.5      10.4.5      11.0.0      vechain-sdk
...
hardhat-gas-reporter               1.0.10      1.0.10      2.2.2       sdk-hardhat-integration
...
react                              18.3.1      18.3.1      19.0.0      sdk-nextjs-integration
react                              18.3.1      18.3.1      19.0.0      sdk-vite-integration
react-dom                          18.3.1      18.3.1      19.0.0      sdk-nextjs-integration
react-dom                          18.3.1      18.3.1      19.0.0      sdk-vite-integration
react-router-dom                   6.27.0      6.28.0      7.0.2       sdk-vite-integration
...
typedoc                            0.26.8      0.26.11     0.27.4      vechain-sdk
typedoc-plugin-missing-exports     3.0.0       3.1.0       3.1.0       vechain-sdk
typescript                         5.6.3       5.7.2       5.7.2       vechain-sdk
...
vite                               5.4.11      5.4.11      6.0.3       sdk-vite-integration
...
vitest-browser-react               0.0.3       0.0.3       0.0.4       sdk-vite-integration
wrangler                           3.84.1      3.95.0      3.95.0      sdk-cloudflare-
↳ integration
Done in 2.77s.
```

A change in the major version number typically signifies breaking changes and may increase the effort to upgrade. As time progresses, the team may be unable to react quickly if issues were to arise.

Recommendation

Update all dependencies to the latest version recommended for production deployment. Add a gating milestone to the development process that involves reviewing all dependencies for outdated or vulnerable versions. Provide instructions on how to build the SDK for a production setting.

Retest Results

2025-02-25 – Partially Fixed

In commit `cbfbac5` and commit `c5834a4`, a number of dependencies were updated and running the `yarn audit` tool only highlights 4 vulnerabilities (3 Low and 1 Moderate), which do not appear to meaningfully impact the codebase.

However, a number of dependencies are still outdated. Specifically, many dependencies with “Major Update backward-incompatible updates” flagged above were not updated. However, the VeChain indicated that they are all related to dev-dependencies and the fact that they are slightly out-of-date is not a concern at the moment. This finding was marked *Partially Fixed* as a result.

Low

Potentially Problematic Key Generation Function

Overall Risk	Low	**Finding ID**	NCC-E020944-FR9
Impact	Medium	**Category**	Error Reporting
Exploitability	Low	**Status**	Fixed

Impact

Generating a key for an algorithm different than the one documented in case of an exception might lead to subtle issues and might obscure problems with the underlying dependency.

Description

The file *packages/core/src/secp256k1/Secp256k1.ts* contains a number of elliptic curve related functionalities, allowing users to generate keys, compress and decompress public keys, etc. The function `generatePrivateKey()`, excerpted below for convenience, generates a random private key. This function leverages the underlying `nc_secp256k1` dependency to generate this key. Upon encountering an exception during the key generation process, the function falls back to the SubtleCrypto interface of the Web Crypto API. However, the process in this case generates a 256-bit AES-GCM key, as can be seen highlighted below.

```
    /**
     * Generates a new random private key.
     * If an error occurs during generation using
     * [nc_secp256k1](https://github.com/paulmillr/noble-secp256k1),
     * an AES-GCM key is generated as a fallback in runtimes not supported
     * by `nc_secp256k1`, if those support {@link {@link global.crypto}.
     *
     * @return {Promise<Uint8Array>} The generated private key as a Uint8Array.
     *
     * @remarks Security auditable method, depends on
     * * {@link global.crypto.subtle.exportKey};
     * * {@link global.crypto.subtle.generateKey};
     * * [nc_secp256k1.utils.randomPrivateKey](https://github.com/paulmillr/noble-
     ↳ secp256k1).
     */
    public static async generatePrivateKey(): Promise<Uint8Array> {
        try {
            return nc_secp256k1.utils.randomPrivateKey();
        } catch (e) {
            // Generate an ECDSA key pair
            const cryptoKey = await global.crypto.subtle.generateKey(
                {
                    name: 'AES-GCM',
                    length: 256
                },
                true,
                ['encrypt', 'decrypt']
            );

            // Export the private key to raw format
            const rawKey = await global.crypto.subtle.exportKey(
                'raw',
                cryptoKey
            );

            // Convert the ArrayBuffer to Uint8Array
            return new Uint8Array(rawKey);
        }
    }
```

In principle, returning a key for a different algorithm than the one requested is a breach of the API contract and can lead to serious issues. However, in the function above, the raw key bytes are exported and eventually returned as a byte array, which is functionally indistinguishable from an ECC private key.

The main issue therefore lies in the fact that a caller would not realize that an exception was ever encountered during key generation. If this exception had occurred because of a failure to locate the necessary dependency, then it is likely that other functions of the SDK leveraging the same dependency might fail as well.

Recommendation

Consider propagating the exception to the caller of the `generatePrivateKey()` in case the underlying dependency fails to generate a private key instead of returning another key. Alternatively, heavily document why the fallback solution works, delete the `Generate an ECDSA key pair` comment, and test the fallback solution with all the other related functions in the SDK.

Location

packages/core/src/secp256k1/Secp256k1.ts

Retest Results

2025-02-24 – Fixed

In commit `dfaa20a`, the function `generatePrivateKey()` was updated according to the recommendation above; it now throws an `InvalidSecp256k1PrivateKey` exception in case there's an issue during the key generation procedure. This finding was marked *Fixed* as a result.

Low

Potential Private Key Leak

Overall Risk	Low	**Finding ID**	NCC-E020944-943
Impact	High	**Category**	Data Exposure
Exploitability	Low	**Status**	Fixed

Impact

An attacker with access to the logs or a stack trace of the SDK execution may obtain the private key material.

Description

The function `ofPrivateKey()` defined in the file *packages/core/src/vcdm/Address.ts* creates an Address from the private key provided as parameter. Upon encountering a generic error during the public key derivation and address computation processes, the function throws an exception and populates the body of the exception with the content of the private key, as can be seen in the highlighted line of the below excerpt.

```
    public static ofPrivateKey(
        privateKey: Uint8Array,
        isCompressed: boolean = true
    ): Address {
        try {
            return Address.ofPublicKey(
                Secp256k1.derivePublicKey(privateKey, isCompressed)
            );
        } catch (error) {
            if (error instanceof InvalidSecp256k1PrivateKey) {
                throw error;
            }
            throw new InvalidDataType(
                'Address.ofPrivateKey',
                'not a valid private key',
                { privateKey: `${privateKey}` },
                error
            );
        }
    }
```

This might allow an attacker to obtain sensitive key material.

In comparison, the function `sign()` in *packages/core/src/secp256k1/Secp256k1.ts* is careful not to disclose the private key content when encountering an invalid key, as can be seen below.

```
    // Check if the private key is valid.
    if (!Secp256k1.isValidPrivateKey(privateKey)) {
        throw new InvalidSecp256k1PrivateKey(
            'Secp256k1.sign',
            'Invalid private key given as input. Ensure it is a valid 32-byte secp256k1
            ↪ private key.',
            undefined
        );
    }
```

Recommendation

Update the content of the exception in the function `ofPrivateKey()` such that it does not include the private key content.

Location

- *packages/core/src/vcdm/Address.ts*

Retest Results

2025-02-24 – Fixed

In commit `b3576a3`, the error message provided in the exception body was modified and no longer includes the private key. This finding was marked *Fixed* as a result.

Low

Missing Parameter Check in `inflatePublicKey()` Function

Overall Risk	Low	**Finding ID**	NCC-E020944-GWF
Impact	Medium	**Category**	Data Validation
Exploitability	Low	**Status**	Risk Accepted

Impact

Encoded public keys with an incorrect length might not be detected by the SDK, which may result in behavior that is difficult to understand and debug and/or potentially undefined.

Description

The function `inflatePublicKey()` in *Secp256k1.ts* is used to “inflate” (i.e., decompress) a compressed Secp256k1 public key to its uncompressed form. The function is excerpted below, for reference.

```
    public static inflatePublicKey(publicKey: Uint8Array): Uint8Array {
        const prefix = publicKey.at(0);
        if (prefix !== Secp256k1.UNCOMPRESS_PREFIX) {
            // To inflate.
            const x = publicKey.slice(0, 33);
            const p = nc_secp256k1.ProjectivePoint.fromAffine(
                nc_secp256k1.ProjectivePoint.fromHex(
                    HexUInt.of(x).digits
                ).toAffine()
            );
            return p.toRawBytes(false);
        } else {
            // Inflated.
            return publicKey;
        }
    }
```

This function, used for example to derive an address from a compressed public key, does not ensure that the public key provided as parameter is of expected length. The function only checks the first byte of the byte array argument to determine whether the key is in compressed or uncompressed form, but does not actually ensure there are enough bytes to make up a key (nor does it actually ensure the first byte is correct; it only checks whether it is different than `4`). If the parameter were of incorrect length, this function might not be able to detect it, which could lead to unexpected issues during execution.

While the entire *noble-secp256k1* dependency was not audited, the NCC Group consultants performed a cursory review of a few key functions used by the Vechain SDK and noted that the functions used in the function above appear to perform the necessary checks on the length and the value of the elliptic curve point passed as parameter. Hence, the problematic edge cases appear to only be able to be triggered when the first byte of the `publicKey` argument is `4`.

An interesting outcome of the cursory review of *noble-secp256k1* is that the point-at-infinity can be exported to hexadecimal, but not *imported from* hexadecimal. Namely, the following line will produce an error.

```
const point = Point.fromHex(Point.ZERO.toHex(false));
```

This highlights an edge case that may be triggered in the underlying library, which may not be expected behavior.

Similarly, the function `ofPublicKey()` in *packages/core/src/vcdm/Address.ts* (which calls the `inflatePublicKey()` function above) does not check the validity of the public key argument. For defense in depth, consider also validating the public key length in that function.

Recommendation

Add appropriate length checks to the `inflatePublicKey()` function above. That is, ensure the key is 33 bytes long when compressed and 65 when uncompressed, and throw an error otherwise. Consider adding similar length checks in the function `ofPublicKey()` in *Address.ts*.

Location

- *packages/core/src/secp256k1/Secp256k1.ts*
- *packages/core/src/vcdm/Address.ts*

Retest Results

2025-02-24 – Not Fixed

The VeChain team indicated:

> Issue acknowledged, will be addressed in next release.

As such, this finding was marked *Risk Accepted*.

Info

Misleading Naming of the `getAddress()` Function

Overall Risk	Informational	**Finding ID**	NCC-E020944-NRD
Impact	Low	**Category**	Other
Exploitability	Low	**Status**	Fixed

Impact

Function names and code comments, in particular the ones documenting functions, represent a form of contract between API developers and end users of these functions. Misleading or incorrect naming may lead to uncaught errors which could break the integrity and correctness of the system.

Description

In *packages/network/src/signer/signers/vechain-private-key-signer/vechain-private-key-signer.ts*, a function named `getAddress()` is defined; this function is excerpted below, for reference.

```
    /**
     * Get the address of the Signer.
     *
     * @returns the address of the signer
     */
    async getAddress(): Promise<string> {
        return Address.checksum(
            HexUInt.of(
                await Promise.resolve(
                    Address.ofPrivateKey(this.privateKey).toString()
                )
            )
        );
    }
```

Unlike its name suggests (and unlike the documentation preceding the function states) that function returns the address' checksum, i.e., the Keccak256 digest of an address, and not the address itself.

Recommendation

Update the function name and preceding documentation to indicate that the function returns the address checksum.

Location

packages/network/src/signer/signers/vechain-private-key-signer/vechain-private-key-signer.ts

Retest Results

2025-02-24 – Fixed

In commit `15e6db1`, the documentation preceding the `getAddress()` function was updated to indicate that the function returned the "address checksum of the Signer", which is a standard Ethereum address format. This finding was marked *Fixed* as a result.

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 highlights a number of observations that the NCC Group team gathered during the engagement and that do not warrant security-related findings on their own.

Misleading Documentation

- In the file *packages/network/src/signer/signers/vechain-private-key-signer/vechain-private-key-signer.ts*, the documentation preceding the function `sendTransaction()` mentions that the function `populateTransaction()` is called first within the function. This does not appear to be the case. The function `populateTransaction()` *does* eventually get called, but only later on, in the function `_signFlow()`.

```
  /**
   *  Sends %%transactionToSend%% to the Network. The
   ↳ ``signer.populateTransaction(transactionToSend)``
   *  is called first to ensure all necessary properties for the
   *  transaction to be valid have been populated first.
   *
   *  @param transactionToSend - The transaction to send
   *  @returns The transaction response
   * @throws {JSONRPCInvalidParams}
   */
  async sendTransaction(
      transactionToSend: TransactionRequestInput
  ): Promise<string> {
```

- In the file *packages/core/src/transaction/Transaction.ts*, the function `isSignatureValid()` is documented to return “true if the signature is valid, otherwise false”. In practice, this function only checks that the signature length is equal to an expected length, but it never actually *verifies* the signature. The fact that this function does not verify the signature in a *cryptographic* sense could be more clearly documented.

```
  /**
   * Return Returns true if the signature is valid, otherwise false.
   *
   * @param {TransactionBody} body - The transaction body to be checked.
   * @param {Uint8Array} signature - The signature to validate.
   * @return {boolean} - Returns true if the signature is valid, otherwise false.
   */
  private static isSignatureValid(
      body: TransactionBody,
      signature: Uint8Array
  ): boolean {
      // Verify signature length
      const expectedSignatureLength = this.isDelegated(body)
          ? Secp256k1.SIGNATURE_LENGTH * 2
          : Secp256k1.SIGNATURE_LENGTH;

      return signature.length === expectedSignatureLength;
  }
}
```

- In the file *packages/core/src/vcdm/hash/Keccak256.ts* (and in the corresponding documentation file *packages/core/dist/index.d.ts*), the comment preceding the computation of the `Keccak256` hash is inaccurate; it purports to compute the SHA256 hash, as can be seen below.

```
* @returns {Sha256} - The [KECCAK 256](https://keccak.team/keccak.html) hash of the
↳ input value.
```

A similar comment applies to the header comment for `Blake2b256.of()` (in *packages/core/src/vcdm/hash/Blake2b256.ts*), which incorrectly claims its return type as `Sha256`.

Unnecessary `isCompressed` Argument

The function `ofPrivateKey()` in *packages/core/src/vcdm/Address.ts* converts a private key to an address. In addition to the private key parameter, the function also takes in a boolean flag `isCompressed` to "indicate if the derived public key should be compressed", as can be seen in the excerpt below.

```
    public static ofPrivateKey(
        privateKey: Uint8Array,
        isCompressed: boolean = true
    ): Address {
        try {
            return Address.ofPublicKey(
                Secp256k1.derivePublicKey(privateKey, isCompressed)
            );
        } catch (error) {
            if (error instanceof InvalidSecp256k1PrivateKey) {
                throw error;
            }
            throw new InvalidDataType(
                'Address.ofPrivateKey',
                'not a valid private key',
                { privateKey: `${privateKey}` },
                error
            );
        }
    }
```

This argument is unnecessary here, since the public key is never actually returned; it is computed from the private key, but then is used to derive the address, which is eventually returned by the function. Consider removing this argument and providing a literal boolean value to the `derivePublicKey()` function call above.

Hexadecimal Value Representation

The file *packages/core/src/vcdm/Hex.ts* defines a class representing hexadecimal values, as well as associated methods to operate on these values. The regular expressions `REGEX_HEX` (equal to `/^-?(0x)?[0-9a-f]*$/i`) and `REGEX_HEX_PREFIX` (equal to `/^-?0x/i`), deem the empty value `0x` valid. This may be slightly confusing in the context of the `isValid0x()` function, which claims that it "Determines whether the given string is a valid hexadecimal number prefixed with '0x'.", and would thus consider `0x` valid.

```
    /**
     * Determines whether the given string is a valid hexadecimal number prefixed with
     ↳ '0x'.
     *
     * @param {string} exp - The string to be evaluated.
     * @return {boolean} - True if the string is a valid hexadecimal number prefixed with
     ↳ '0x', otherwise false.
     */
    public static isValid0x(exp: string): boolean {
        return Hex.REGEX_HEX_PREFIX.test(exp) && Hex.isValid(exp);
    }
```

Ineffective Memory Zeroization

The code contains several attempts to clear secret data from memory after it is used, for example, `privateKey.fill(0)` in *packages/core/src/hdkey/HDkey.ts*.

Memory zeroization is also not used consistently. In the function `fromPrivateKey()` in *packages/core/src/hdkey/HDKey.ts* the function argument `privateKey` is zeroed out after being copied to a new string, the `header` variable in the code excerpt below. However, that `header` variable is then copied to yet another string (the `expandedPrivateKey` variable) and the portion of `header` containing the private key is never zeroed out after this subsequent copy.

```
    const header = nc_utils.concatBytes(
        this.EXTENDED_PRIVATE_KEY_PREFIX,
        chainCode,
        Uint8Array.of(0),
        privateKey
    );
    privateKey.fill(0); // Clear the private key from memory.
    const checksum = Sha256.of(Sha256.of(header).bytes).bytes.subarray(
        0,
        4
    );
    const expandedPrivateKey = nc_utils.concatBytes(header, checksum);
```

Finally, it should be noted that the zeroization of cryptographic secrets in JavaScript (or TypeScript) is not guaranteed to be effective due to the nature of the language and runtime environment. In JavaScript, the memory is managed by the JavaScript engine; one cannot control the memory in the same way as with lower-level languages. Temporary copies of variables may be made by the runtime, which cannot be controlled by the user and sensitive data might thus remain in memory until garbage collection occurs.

Unused Code

Several functions do not appear to be used or documented. Consider removing them.

- The functions `isDerivationPathValid()` in *packages/core/src/hdkey/HDKey.ts*, as well as the function `isDerivationPathComponentValid()`, which is used only by the former function.
- The Bloom filter implementation also appears to be currently unused.

TypeScript Compiler Options

Several compiler options in *tsconfig.json* do not follow best practices. Consider updating them as described.

- `"target"`: The value `"ESNext"` refers to the highest ECMAScript version the local version of TypeScript supports. The tsconfig documentation states the following:

  > _The special ESNext value refers to the highest version your version of TypeScript supports. This setting should be used with caution, since it doesn't mean the same thing between different TypeScript versions and can make upgrades less predictable.

Consider setting a specific target version, such as `es2016` or `es2023`.

- `"moduleResolution"`: The `node` module resolution strategy (also called `node10`) should no longer be used, according to the TypeScript Handbook:

  > `--moduleResolution node` was renamed to `node10` (keeping `node` as an alias for backward compatibility) in TypeScript 5.0. It reflects the CommonJS module resolution algorithm as it existed in Node.js versions earlier than v12. It should no longer be used.

  Instead, consider `node16` (which is the current value of `nodenext`).

  The tsconfig file in *packages/rpc-proxy* also sets `moduleResolution` to `node`.

Certificate Data and Validation

The SDK file *packages/core/src/certificate/index.ts* provides an API to create, sign, and verify self-signed certificates. The `verify()` method validates only the signature of the certificate data and does not perform (or exposes facilities to perform) contextual validation, such as ensuring that the domain field matches expected values. It is the responsibility of the application using the SDK to validate the certificate's contextual data after the signature is verified. It is recommended to document this requirement explicitly to ensure callers perform their own certificate content validation.

Furthermore, if the certificate format is expected to change in the future, consider including a version field. This may enable robust validation by ensuring that the certificate adheres to the expected format for its version, but at the cost of additional complexity (and security bugs if not implemented correctly).

Additionally, certificates are encoded using the function `fastJsonStableStringify()` function prior to being hashed and signed. This function sorts the fields in ascending alphabetic order prior to encoding them. Thus, if the certificate format were ever to change, the results obtained from the encoding process would diverge and lead to interoperability issues due to incorrect signature verification.

It should also be highlighted again that certificates are *self-signed*; they are not actually signed by a trusted entity (such as a CA). Thus, the security they provide is limited and in the absence of a secure public key distribution method, receiving entities cannot assume the authenticity of the sender. A more descriptive name could be used instead, such as "self-signed certificate".

Mnemonic Usage of Custom Random Number Generator

The `Mnemonic` class `of()` method (located at *packages/core/src/vcdm/Mnemonic.ts*) generates BIP39 mnemonic words. It allows SDK users to optionally use a random number generator function parameter, instead of the default one. The method documentation states the following:

```
   * * `randomGenerator` - **Must provide a cryptographic secure source of entropy
     *    else any secure audit certification related with this software is invalid.**
```

It does not specify a method signature, return type and failure mode for this random number generator function. This information should be provided to minimize risk of misuse by callers. In this case, it should accept a byte count with a type domain in `WordListRandomGeneratorSizeInBytes`, and return an `Uint8Array` of requested size in case of success. In case of failure, it should throw an exception.

Pervasive Timing Side-Channels

Most of the cryptographic primitives used by the SDK are implemented by third-party libraries or runtime environments (e.g., the V8 JavaScript engine), which were not in scope for this assessment. During the review, NCC Group observed multiple instances where the SDK, directly or indirectly, invokes code paths whose execution trace may depend on secret values (e.g., mnemonics, private keys). Such data-dependent execution can, in principle, be exploited by an attacker through timing-based side-channel attacks.

Common processors use caches to speed up access to resources such as data and code. Attackers who can monitor the cache may observe changes in speed and cache behavior, when resources that depend on sensitive information are used. This attack can reveal the locations where the victim is accessing data (data flow) , or the code the victim is running (control flow).

While these timing side-channels may not be a concern for all use cases, they could be relevant to SDK users with certain threat models. Therefore, it is advisable to document these potential risks clearly in the SDK's API documentation.

Below is a non-exhaustive list of examples of potentially non-constant-time code identified during the review:

`Keystore` MAC Key Example

In file *packages/core/src/keystore/cryptography/experimental/keystore.ts*, function `encryptKeystore()`, `macPrefix` is the MAC key (`Uint8Array`) that was derived from the user password. It is concatenated with the `ciphertext`, and the result passed to function `Keccak256.of()`:

```
function encryptKeystore(
    privateKey: Uint8Array,
    password: Uint8Array,
    options: EncryptOptions
): Keystore {
     // SNIP
        const macPrefix = key.slice(16, 32);
        // Encrypt the private key: 32 bytes for the Web3 Secret Storage (derivedKey,
        ↪ macPrefix)
     // SNIP
               },
               // Compute the message authentication code, used to check the password.
               mac: Keccak256.of(n_utils.concatBytes(macPrefix, ciphertext))
                   .digits
// SNIP
    }
```

This function will ultimately call `bytesToHex()` in library `noble-curves` on the secret MAC key data. `bytesToHex()` uses an array (`hexes`) to map the MAC key bytes to an hexadecimal string, which is not constant-time:

```
export function bytesToHex(bytes: Uint8Array): string {
  abytes(bytes);
  // pre-caching improves the speed 6x
  let hex = '';
  for (let i = 0; i < bytes.length; i++) {
    hex += hexes[bytes[i]];
  }
  return hex;
}
```

`String`'s `toLowerCase()` JavaScript Runtime Example

In file *packages/core/src/vcdm/Mnemonic.ts* function `toPrivateKey()` calls `HDKey.fromMnemonic(words)`, where `words` is a list of secret words:

```
public static toPrivateKey(
        words: string[],
        path: string = 'm/0'
    ): Uint8Array {
        const root = HDKey.fromMnemonic(words);
```

Function `fromMnemonic()` calls JavaScript's `toLowerCase()`:

```
public static fromMnemonic(
      words: string[],
      path: string = this.VET_DERIVATION_PATH
  ): HDKey {
      let master: s_bip32.HDKey;
      try {
          master = s_bip32.HDKey.fromMasterSeed(
              s_bip39.mnemonicToSeedSync(words.join(' ').toLowerCase())
          );
```

In the V8 engine, this is implemented as follows:

```
BUILTIN(StringPrototypeToLowerCase) {
  HandleScope scope(isolate);
  TO_THIS_STRING(string, "String.prototype.toLowerCase");
  return ConvertCase(string, isolate,
                     isolate->runtime_state()->to_lower_mapping());
}
```

`ConvertCase()` performs table-based conversion, and caching, which is not constant-time.

Base58 Encoding of Private Key

In file *packages/core/src/hdkey/HDKey.ts*, function `fromPrivateKey()` performs base58 encoding of secret keys using lookup values, which is not constant-time:

```
            const expandedPrivateKey = nc_utils.concatBytes(header, checksum);
            try {
                return s_bip32.HDKey.fromExtendedKey(
                    base58.encode(expandedPrivateKey)
                ) as HDKey;
```

The base58 encoding is implemented in third-party library `scure-base`.

IV is not Included in MAC Computation

In file *packages/core/src/keystore/cryptography/experimental/keystore.ts*, functions `encryptKeystore()` and `decryptKeystore()` respectively compute and verify a message authentication code (MAC) derived from the user password. The MAC is used to validate that the encrypted ciphertext has not been tampered with. However, the IV (Initialization Vector) is not included in the MAC computation. This omission allows an attacker to tamper with the IV, potentially causing the decryption to produce a different plaintext that still passes MAC verification. However, the decrypted value is then compared against a hash of the keystore address, a large value as illustrated in the code snippet below. It would be computationally infeasible for an attacker to guess an IV that results in a correct match for the expected hash, given the same ciphertext. Nevertheless, this may open other avenues of attacks, and it would be preferable to compute the MAC over the concatenation of the IV and ciphertext to further strengthen the implementation:

```
function decryptKeystore(
    keystore: Keystore,
    password: Uint8Array
): KeystoreAccount {
// SNIP
        const kdf = decodeScryptParams(keystore);
        const key = scrypt(password, kdf.salt, {
            N: kdf.N,
            r: kdf.r,
            p: kdf.p,
            dkLen: kdf.dkLen
        });
        const ciphertext = n_utils.hexToBytes(keystore.crypto.ciphertext);
        if (
            keystore.crypto.mac !==
            Keccak256.of(n_utils.concatBytes(key.slice(16, 32), ciphertext))
                .digits
        ) {
            // SNIP
        }
        const privateKey = ctr(
            key.slice(0, 16),
            n_utils.hexToBytes(keystore.crypto.cipherparams.iv)
        ).decrypt(ciphertext);
        const address = Address.ofPrivateKey(privateKey).toString();
        if (
            keystore.address !== '' &&
            address !== Address.checksum(Hex.of(keystore.address))
        ) {
            throw new InvalidKeystoreParams(
                '(EXPERIMENTAL) keystore.decryptKeystore()',
                'Decryption failed: address/password mismatch.',
                { keystoreAddress: keystore.address }
            );
        }
      // SNIP
    }
}
```

Suboptimal Bloom Filter Implementation

The file *packages/core/src/vcdm/BloomFilter.ts* provides a basic implementation of a Bloom filter. A number of notes were collected regarding this implementation, and are presented below in no particular order.

Hash Function Usage

Traditionally, a Bloom filter is defined with `k` independent hash functions. This implementation has instead chosen to use a single hash function, `Blake2b256`. This hash is invoked, and the first four bytes of its output are used; the other 28 bytes are discarded. To generate additional “hash” values, this initial value is run through a variant of a linear congruential generator (excerpted from *packages/core/src/vcdm/BloomFilter.ts*):

```
    const delta = ((hash >>> 17) | (hash << 15)) >>> 0;
    for (let i = 0; i < k; i++) {
        const bitPos = hash % m;
        if (!collision(Math.floor(bitPos / 8), 1 << bitPos % 8)) {
            return false;
        }
        hash = addAndWrapAsUInt32(hash, delta);
    }
    return true;
```

In this snippet, a linear offset named `delta` is derived from `hash` using bitshifts. Notably, if `hash` is zero then `delta` will be zero as well. The odds of `hash` being zero by chance are low, but not impossibly low; an attacker could search for inputs with this property easily enough. Under modulo reduction, several further degenerate properties emerge, as we will see.

Moving on, `hash` and `delta` are added together modulo 2^{32}. The result, at each step, is further reduced modulo the buffer size in bits, and the result is used as an index. The series of indices generated in this way are used in place of the independent hash function outputs that a Bloom filter would typically use.

Unfortunately, these indices are not close to being independent, violating the formal analysis that Bloom filters benefit from. This can degrade the filters’ performance in practice.

Given that Blake2b256 generates 32 bytes of high-quality, effectively independent hash output, a clear solution would be to use more of these bytes rather than discarding them, and possibly to take additional domain-separated hashes of the input key if more bytes of hash data are needed.

Inefficiencies in Bloom Filter Generation and Storage

It is trivial to update a Bloom filter in-place. However, in the implementation given here, Bloom filters are immutable. To make up for this, a separate class, `BloomFilterBuilder`, is used to generate these immutable filters. This class keeps track of keys to be inserted by storing them in a `Map<number, boolean>` where each key is mapped to `true` - functioning essentially as a hash set (with extra steps).

As long as an instance of the `BloomFilterBuilder` is preserved, updated versions of a Bloom filter can be generated; however, requiring this parallel data structure seems to fail to leverage the appealingly low storage overhead of the Bloom filter. An additional complication: if the Bloom filter outlives its builder, then further updates would have to be performed through inconvenient means such as merges with new Bloom filters.

Furthermore, each updated version of a Bloom filter generated by a Builder is generated from scratch, rather than reusing the partial results that have already been generated. If the implementation is concerned with performance and minimizing hash calls, then this is a potential area for improvement.

Incorrect Calculations of “Best” Parameterizations

The Bloom filter implementation claims the following:

> Mathematically, `m` is approximated as `(k / ln(2))`
>
> Mathematically, `k` is approximated as `(m * ln(2))`

These equations are incorrect as they ignore the expected number of inserted keys. The value of k which minimizes the false probability chance, for a given size m and for n inserted keys, would be:

$$k \approx \frac{m}{n} \ln 2$$

Rounded, of course, to the nearest integer. In other words, the implemented approximation overestimates k's optimal value by a factor of n.

Pervasive Pattern of Silent Failover Within AWS KMS Adapter

Throughout *KMSVeChainProvider.ts* and *KMSVeChainSigner.ts*, a design pattern of silent fail-over on failure is used. This may result in behavior that is unexpected, surprising, or even problematic to the user. A pair of examples are excerpted below for clarity.

KMS Client Configuration

One clear example is in `KMSVeChainProvider`'s constructor:

```
/**
 * Creates a new instance of KMSVeChainProvider.
 * @param thorClient The thor client instance to use.
 * @param params The parameters to configure the KMS client and the keyId.
 * @param enableDelegation Whether to enable delegation or not.
 **/
public constructor(
    thorClient: ThorClient,
    params: KMSClientParameters,
    enableDelegation: boolean = false
) {
    super(thorClient, undefined, enableDelegation);
    this.keyId = params.keyId;
    this.kmsClient =
        params.endpoint !== undefined
            ? new KMSClient({
                  region: params.region,
                  endpoint: params.endpoint,
                  credentials: params.credentials
              })
            : params.credentials !== undefined
              ? new KMSClient({
                    region: params.region,
                    credentials: params.credentials
                })
              : new KMSClient({ region: params.region });
}
```

In this snippet, a few cases are handled: The case where `params.endpoint` is defined, the case where it is undefined but `params.credentials` is defined, and the case where neither of these are defined.

We first note in passing that there is no adequate handling for the case where `params.endpoint` is defined but `params.credentials` is not; in this case, `params.credentials` would be passed as undefined; this is precisely the case that the following conditionals attempt to prevent, and it could cause issues within KMSClient.

Furthermore, and potentially more severely, this type of system has poor outcomes under misconfiguration. For instance, if a user intends to provide custom credentials but fails to provide them properly, then rather than encountering an error, the system may simply load default credentials instead and proceed as if nothing is wrong, with the user none the wiser. This could result in issues such as transactions being sent from improper addresses; this could potentially also leak social graph data if the intended and actual addresses in use were not intended to be publicly associated with each other.

Delegator Signature Generation

Within *KMSVeChainSigner.ts*, the `signTransaction()` method is provided. This method accepts a transaction to sign, and either returns a signature or an error. The case of delegate signing is handled by a private helper method:

```
const signature =
    await this.concatSignatureIfDelegation(transaction);
```

This helper method works roughly as follows (abridged for clarity):

```
private async concatSignatureIfDelegation(
    transaction: Transaction
): Promise<Uint8Array> {
    // Get the transaction hash
    const transactionHash = transaction.getTransactionHash().bytes;
    // Sign the transaction hash using origin key
    const originSignature =
        await this.buildVeChainSignatureFromPayload(transactionHash);
    // We try first in case there is a delegator provider
    if (this.kmsVeChainDelegatorProvider !== undefined) {
        /* SNIP: Generate the signature using this.kmsVeChainDelegatorProvider */
        return concatBytes(originSignature, delegatorSignature);
    } else if (
        // If not, we try with the delegator URL
        this.kmsVeChainDelegatorUrl !== undefined &&
        this.provider !== undefined
    ) {
        /* SNIP: Generate the signature using this.kmsVeChainDelegatorUrl and
        ↪ this.provider.thorClient.httpClient */
        return concatBytes(originSignature, delegatorSignature);
    }
    return originSignature;
}
```

In this code, we first attempt to use `this.kmsVeChainDelegatorProvider`, and then fall back to `this.kmsVeChainDelegatorUrl` if the former is not available. If neither of these is available, we simply return a single signature, without any delegate signature. Aside from the length of the returned value, the caller receives no indication of which option was chosen.

This is suboptimal in several ways, and again leads to real risks under misconfiguration.

One might wonder what would happen if both `kmsVeChainDelegatorProvider` and `kmsVeChainDelegatorUrl` are defined. In fact, the constructor only defines the `...Url` attribute if the `...Provider` attribute is not defined, but this precedence is undocumented. The user is not alerted to the fact that they have provided a redundant configuration option, and there is no support for using the delegator URL as a fallback if the delegator provider fails for any reason.

Furthermore, if `this.provider` is undefined (which is allowed; the constructor checks for this case and does not throw an error when it happens), then requests to `kmsVeChainDelegatorProvider` would still go through but requests to `kmsVeChainDelegatorUrl` would not. Thus, a user might provide a Delegator URL and end up with the signer failing to delegate as expected, possibly incurring unexpected fees for the user.