Sidechains SDK – Cryptography and Implementation Review

Horizen Labs, Inc

Version 1.0 – May 10, 2023

Prepared By
Parnian Alimi
Elena Bakos Lang
Paul Bottinelli
Gérald Doussot
Aleksandar Kircanski

Prepared For
Giacomo Gussoni
Oleksandr Iozhytsia
Rosario Pabst

1 Table of Contents

2 Executive Summary

Synopsis

Beginning in October 2022, Horizen Labs engaged NCC group to perform a cryptography and implementation review of the Sidechains SDK, also known as “Blaze”. The SDK aims to be a framework supporting the creation of sidechains and their custom business logic, with the Horizen public blockchain as the mainchain.

This engagement was comprised of an initial testing part and a retesting part. Initial testing was split into two phases: the first phase was five weeks long and was delivered remotely by four consultants over 45 person-days, starting mid-October 2022. The second phase was four weeks long, and was delivered remotely by three consultants, over 25 person-days, starting in January 2023. The second phase also included retesting of some of the findings identified in the first phase.

Following these two testing phases, in mid-March 2023, the NCC Group team performed a retest of the findings uncovered during the first two phases. A total of 80 person-days of effort were expended on the project.

Scope

The code under review is located on the GitHub Sidechains-SDK repository, and the review was performed on branch `dev_evm_audit`, at commit ID `8a6bca74` for the first phase, and on branch `dev_evm_audit_phase_2`, at commit ID `d39a7b54` for the second phase. The scope of NCC Group’s evaluation included the following elements:

- sdk: the core development kit, consisting of a Java and a Scala component;
- libevm: a shared library to access a standalone instance of the `go-ethereum` EVM;
- evm: Java Native Interface (JNI) wrappers.

A number of additional resources were shared to support the review, including internal documentation and presentations, as well as the whitepaper Zendoo: a zk-SNARK Verifiable Cross-Chain Transfer Protocol Enabling Decoupled and Decentralized Sidechains, which details the specific cross-chain transfer protocol (CCTP) approach followed by Horizen Labs and a proposal for the underlying sidechain consensus protocol.

Limitations

The Sidechains SDK is, as its name suggests, a development kit. The goal of the review was to identify potential issues within the code of the SDK itself. However, the strength of such a generic platform is also its weakness; developers may define problematic constructions, extend existing ones with buggy logic, or use provided functions in unintended ways. Issues may also arise in the deployment of the node, as careful deployment is integral to the security of the node and for implementing proper access control. These issues cannot be captured by a review of the underlying SDK, and will need to be considered for each deployment independently.

Additionally, a large refactoring of the codebase took place before the retesting part of the engagement. The NCC Group consultants mostly reviewed the fixes in isolation, and did not explore the refactored parts of the codebase in as much depth.

Key Findings

The most notable findings were:

- **Strong Biases in ChaChaPRNG Implementation**, where biases in the random number generation implementation result in predictable randomness which leads to complete failure of its security guarantees.

- **Deterministic Key Generation May Produce Duplicate Private Keys**, where the secret generation procedure may produce duplicate private keys, which could lead to a number of complex, consensus-related issues.
- **Signature Forgery Due to Noncanonical Encoding of `message` to `FieldElement`**, where an attacker may be able to forge signatures for selective messages, potentially allowing them to forge transactions and other sidechain-related constructions.
- **Unauthenticated API Routes May Permit DoS and Eclipse Attacks**, where attackers may cause denial of service, and/or eclipse attacks against the applications using SDK.

This report provides a detailed description of all findings. Additional notes and observations made during the review can be found in the appendix Non-Security Impacting Observations. The findings and notes were discussed in detail at each weekly update meeting and over Slack.

After retesting, NCC Group found that the Horizen Labs team had addressed a number of findings, prioritizing the higher severity items. Specifically, all high severity findings had been fixed, while all but three medium severity findings had been fully fixed. In contrast, all the informational findings were set to be fixed at a later date and were set to *Risk Accepted* as a result. In summary, out of a total of twenty-five (25) original findings, eleven (11) were marked as *Fixed*, two (2) were marked as *Partially Fixed* and twelve (12) were marked as *Risk Accepted* (with reasonable explanations from Horizen Labs). During the retest, the Horizen Labs team requested the review of a PRNG, in which NCC Group found one (1) critical issue, which was promptly fixed.

Strategic Recommendations

Provide a deployment guide that details best practices, especially for security-sensitive features. For instance, clearly document best practices for API authentication, as highlighted in finding "Fragile API Authentication".

Consider reformatting parts of the codebase to reduce code duplication, such as between the different API endpoints. Additionally, review the general code architecture, as some folder names may not accurately depict the files they contain, and some functionalities are spread across multiple folders. These actions will make the code cleaner and easier to follow and reason about it's security properties, making future maintenance and security assessment efforts more efficient. The appendix Non-Security Impacting Observations provides some concrete suggestions.

A number of discrepancies were observed between the Sidechains SDK and equivalent `go-ethereum` methods. Ensure that behaviour is consistent whenever an internal feature is designed to match an external codebase, document any intentional variation, and track updates of `go-ethereum` to ensure any updates get applied to matching functions in a timely manner.

Multiple findings were reported around secret generation, handling, and key usage in general. Ensure these functions are inspected throughout the codebase for additional issues, in addition to implementing the planned addition of support for hardware wallets.

A number of findings were reported around incomplete or unclear validation practices when parsing untrusted input data. Consider reviewing the validation along all input paths for untrusted data, and documenting intentional omissions of validation.

Consider evaluating underlying libraries for code quality and security. In particular, note that the `web3j` library uses non-standard formats for a number of objects, such as representing ECC public keys as `BigInts`, which may introduce issues due to conversions between various data representations.

3 Dashboard

Target Data

Name	Sidechains SDK
Type	Software Development Kit (SDK)
Platforms	Scala, Java, Go
Environment	Local Instance

Engagement Data

Type	Cryptography and Implementation Review
Method	Source Code Review, Dynamic Testing
Dates	2022-10-17 to 2023-03-22
Consultants	5
Level of Effort	80 person-days

Finding Breakdown

Critical issues	1
High issues	3
Medium issues	9
Low issues	4
Informational issues	9
Total issues	**26**

Category Breakdown

Access Controls	1
Authentication	1
Configuration	2
Cryptography	7
Data Exposure	1
Data Validation	7
Denial of Service	1
Error Reporting	1
Other	3
Patching	2

Critical High Medium Low Informational

4 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
Strong Biases in ChaChaPRNG Implementation	Fixed	BBK	Critical
Deterministic Key Generation May Produce Duplicate Private Keys	Fixed	HNB	High
Unauthenticated API Routes May Permit DoS and Eclipse Attacks	Fixed	CRR	High
Potential Signature Forgery Due to Noncanonical Encoding of `message` to `FieldElement`	Fixed	XEX	High
Missing Key Validation Upon Initialization	Fixed	EYU	Medium
Missing Data Length in Hash Computations May Result in Collisions	Risk Accepted	X4D	Medium
Locale-Dependent Charset Encoding of Strings Can Lead to Consensus Breach	Fixed	QY2	Medium
Incorrect Lower Bound in Gas Fee Computation	Fixed	DXB	Medium
Possible Reuse of Randomness When Generating Keys	Fixed	TV2	Medium
Fragile API Authentication	Partially Fixed	4BR	Medium
Potential Null Pointer Dereference	Fixed	MTQ	Medium
Missing Input Validation May Lead to Truncated Input Data or Other Issues	Fixed	QCW	Medium
Missing or Inconsistent Checks in Parsing Functions	Risk Accepted	3QY	Medium
Potentially Incorrect Nonce Results in Failed Transactions	Fixed	DTH	Low
Unprotected Secret Key Storage	Risk Accepted	YBB	Low
Outdated Dependencies	Partially Fixed	RKK	Low
Infinite Loop When Adding Handles	Fixed	DRU	Low
Duplicate Error Codes	Risk Accepted	MUE	Info
Base Fee Computation Potentially Inappropriate with Current Slot Duration	Risk Accepted	YRC	Info
Incorrect API Usage	Risk Accepted	TL9	Info
Overly Permissive Regular Expression	Risk Accepted	QV9	Info
Unsafe Handling of Secret Key Material	Risk Accepted	X4V	Info
Missing In-Memory Merkle Tree Height Validation	Risk Accepted	XA6	Info
Private Key Handling and Memory Zeroization	Risk Accepted	FB4	Info
Unauthenticated Secure Enclave API and DNS Rebinding	Risk Accepted	NK3	Info
Outdated Copy of `go-ethereum` Proof Code	Risk Accepted	U7Q	Info

5 Collateral Issue Details

Critical

Strong Biases in ChaChaPRNG Implementation

Overall Risk	Critical	**Finding ID**	NCC-E005513-BBK
Impact	High	**Component**	ChaChaPRNG
Exploitability	High	**Category**	Cryptography
		Status	Fixed

Impact

Biases in random number generation algorithms result in predictable randomness which can lead to complete failure of the security guarantees of the cryptographic primitives and protocols using these values.

Description

The file ChaChaPrngSecureRandom.java defines a Pseudo-Random Number Generator (PRNG) based on the ChaCha20 cipher. The implementation suffers from a serious vulnerability leading to significant biases in its output distribution.

The cause of this issue stems from an implementation error in the `rotateLeft32()` function, a critical building block of ChaCha20. This function is excerpted below for convenience.

```java
/**
 * @param x The value to be rotated
 * @param k The number of bits to rotate left
 */
private static int rotateLeft32(int x, int k) {
    final int n = 32;

    int s = k & (n - 1);
    return x << s | x >>(n - s);
}
```

Since Java does not have a primitive type for unsigned integers, care has to be taken when performing bitwise operations on integers. Specifically, the `>>` operator (used in the line highlighted above) performs a *signed* right shift in Java. When shifting an integer by one with this operator, the most significant bit (i.e., the leftmost bit) is not *unconditionally* replaced by a zero, but by a bit corresponding to the sign bit of the shifted value (0 for a positive integer, 1 for a negative integer). Since the return value of the `rotateLeft32()` function is computed using a boolean *or* of that shifted quantity, a superfluous 1-bit resulting from shifting a negative input value will be propagated to the output. Hence, the `rotateLeft32()` function may produce incorrect results when performing the bitwise rotation of negative 32-bit integers.

The result of this small difference is a damaging bias in the output distribution. To illustrate this bias, the figure below shows a plot of the output distribution of the `ChaChaPRNG` implementation when seeded with the same seed as in the examples, and used to generate a total of 10k 32-byte samples. The most striking outlier is the value 255, which appears with probability over 20%. Some other values also have significant biases, such as 1 (which appears with probability 2.46%) or 81 (which appears with probability 2.50%). In a truly random distribution, a given byte should appear with probability *1/256 = 0.390625*.

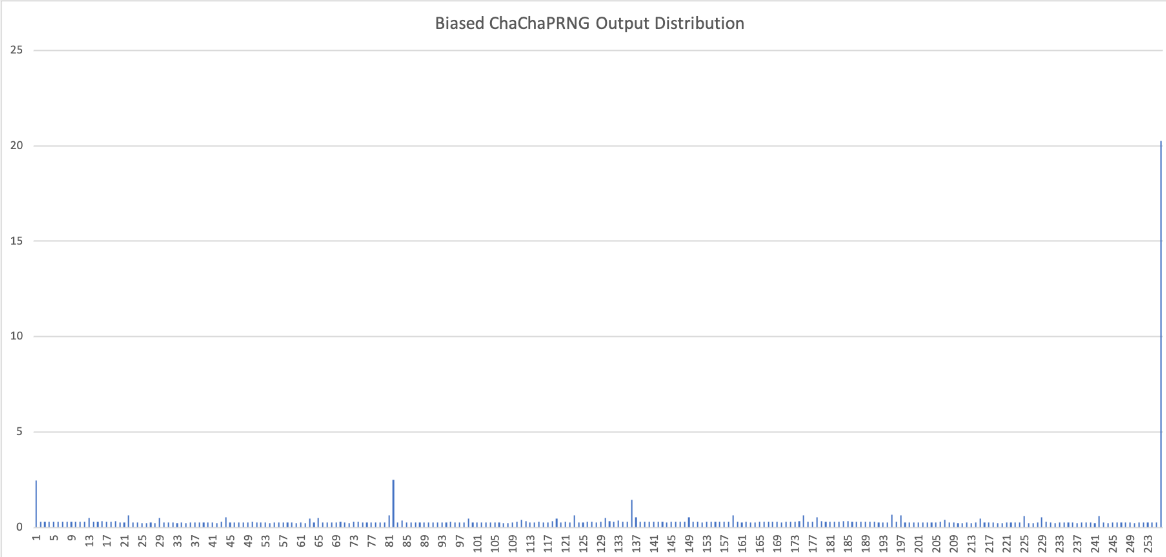

Figure 1: Output distribution of bytes

Research has shown that leaking as little as one bit of an ECDSA nonce could lead to full key recovery [1]. Hence, using the output of this PRNG for cryptographic applications could completely break the security of the systems that rely upon it.

Recommendation

Replace the right-shift operator in the `rotateLeft32()` function by an unsigned right shift (`>>>`). Consider also adding parentheses to ensure operator precedence is clear.

```
return (x << s) | (x >>> (n - s));
```

The figure below shows the output distribution after modifications of the `rotateLeft32()`, with the same number of samples and the same seed as above, which looks much more uniform. Note however that this does not mean the vulnerability described above is the only one in the random number generation procedure.

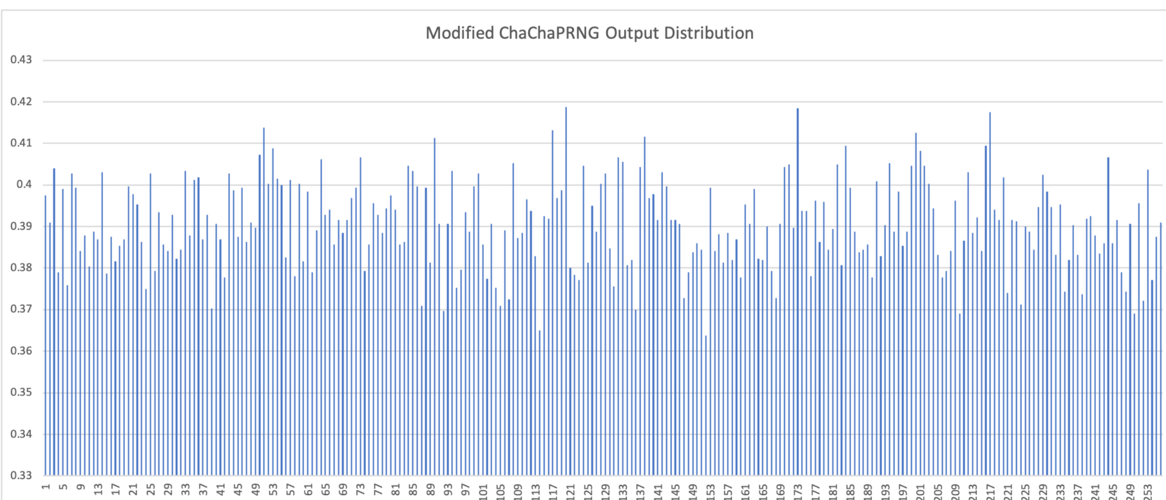

Figure 2: Modified Output Distribution

In addition to fixing the specific issue described in this finding, it is crucial to deeply verify the `ChaChaPRNG` implementation. To do so, ensure the underlying ChaCha20

1. *LadderLeak: Breaking ECDSA With Less Than One Bit Of Nonce Leakage*

implementation is correct. Consider testing against the test vectors defined in RFC 7539. After having ensured the underlying ChaCha20 is correct, perform cross-implementation tests of the `ChaChaPRNG` algorithm by obtaining test vectors from other implementations and testing against them. Writing tests using Known Answer Tests (KAT) that were produced by the same implementation does not effectively test the correctness of the implementation, it only ensures consistency of the algorithm with itself.

In the future, consider testing each new function individually, in order to quickly identify errors during the development process, particularly for complex cryptographic implementations. This might have prevented the issue described in this finding.

Location

ChaChaPrngSecureRandom.java

Retest Results

2023-04-06 – Fixed

In pull request 806, the Horizen Labs team updated the `rotateLeft32()` function to utilize an unsigned right shift along with additional parenthesis per the recommendation:

```
private static int rotateLeft32(int x, int k) {
    return (x << k) | (x >>> (32 - k));
}
```

In addition, the pull request also includes new tests in the Sidechains-SDK/sdk/src/test/java/io/horizen/utils/ChaChaPrngSecureRandomTest.java source file. These tests incorporate both entropy measurements as well as vectors from RFC 7539.

As such, this finding is considered *fixed*.

High

Deterministic Key Generation May Produce Duplicate Private Keys

Overall Risk	High	**Finding ID**	NCC-E005513-HNB
Impact	High	**Component**	SDK
Exploitability	Low	**Category**	Cryptography
		Status	Fixed

Impact

The current secret generation procedure may produce duplicate private keys, which eventually could lead to a number of complex, consensus-related issues, including the generation of duplicate addresses or colliding VRF public keys.

Description

This finding discusses a potentially problematic aspect of the key generation procedures, and is a companion to finding "Possible Reuse of Randomness When Generating Keys".

For a given key type, the current secret key generation procedure starts by fetching all the secrets of a given type from the wallet (see the highlight on line 37 of the excerpt taken from PrivateKey25519Creator.java and shown below). It then generates a seed which is composed of the static wallet seed concatenated with a nonce, the latter being the byte array representation of the number of currently existing secrets (of that same type). This seed computation (see line 39 highlighted below), is then used by the function `generateSecret()` which creates a new key pair based on that deterministic seed. Hence, the generation of new and different keys is entirely reliant on the nonce being different.

```
29 @Override
30 public PrivateKey25519 generateSecret(byte[] seed) {
31     Pair<byte[], byte[]> keyPair = Ed25519.createKeyPair(seed);
32     return new PrivateKey25519(keyPair.getKey(), keyPair.getValue());
33 }
34
35 @Override
36 public PrivateKey25519 generateNextSecret(NodeWalletBase wallet) {
37     List<Secret> prevSecrets = wallet.secretsOfType(PrivateKey25519.class);
38     byte[] nonce = Ints.toByteArray(prevSecrets.size());
39     byte[] seed = Blake2b256.hash(Bytes.concat(wallet.walletSeed(), nonce));
40
41     return generateSecret(seed);
42 }
```

Once such a private key has been successfully generated, it eventually gets added to the list of secrets by means of the `addSecret()` function, defined in AbstractWallet.scala and provided below, for reference.

```
// 1) check for existence
// 2) try to store in SecretStore using SidechainSecretsCompanion
override def addSecret(secret: SidechainTypes#SCS): Try[W] = Try{
  require(secret != null, "AbstractWallet: Secret must be NOT NULL.")
  secretStorage.add(secret).get
  this
}
```

However, a couple of lines below that `addSecret()` function, another interesting function is also defined: `removeSecret()`. Straightforwardly, that function expects a public key as argument, and deletes the corresponding private key from the secret store, as can be seen in the code excerpt below.

```
// 1) check for existence
// 2) remove from SecretStore (note: provide a unique version to SecretStore)
override def removeSecret(publicImage: SidechainTypes#SCP): Try[W] = Try {
  require(publicImage != null, "PublicImage must be NOT NULL.")
  secretStorage.remove(publicImage).get
  this
}
```

While necessary, the fact that this function exists seriously impacts the security of the key generation procedure. Indeed, as we have seen in the function `generateNextSecret()`, the secret seed is completely dependent on the nonce value, which is currently defined as the number of secrets. Therefore, a user generating a first key, then calling the `removeSecret()` function, and finally followed by another call to the key generation procedure will result in the same key being generated, since the size of the underlying secret store is the same between the two key generation invocations.

The only mitigating factor is that the `removeSecret()` function is currently not directly exposed to end-users.

Recommendation

The nonce generation procedure should not depend on the size of a mutable data store. The term “nonce” stands for “number used once” and thus using the size of a flexible structure may result in the same nonce being repeated. Thus, in line with the recommendations provided in finding "Possible Reuse of Randomness When Generating Keys", care should be taken to ensure the same nonce is *never* reused.

This could be done by using a global monotonically increasing nonce counter, or by using algorithm-specific monotonically increasing nonce counters and adding a domain separation tag for each algorithm in the nonce generation process.

More generally, consider updating this key generation functionality to be based on a standardized scheme for deterministic key generation within a wallet, such as BIP 0032[2].

Location

- AbstractWallet.scala
- PrivateKey25519Creator.java
- PrivateKeySecp256k1Creator.java
- VrfKeyGenerator.java
- SchnorrKeyGenerator.java

Retest Results

2023-03-16 – Fixed

In pull request 680, the Horizen Labs team revamped the secret generation procedure in order to address this finding and its sibling finding "Possible Reuse of Randomness When Generating Keys".

The changes introduced in that pull request simplified and standardized the secret generation procedure for the various algorithms. The respective `generateNextSecret()`

2. https://en.bitcoin.it/wiki/BIP_0032

functions defined differently for each algorithm have now been removed, and that functionality has been moved to the *AbstractWallet.scala*. Secret generation is now exclusively performed by the function `generateSecret()` with a *seed* as argument.

That seed is generated by the `generateNextSecret()` function, a generic function defined in *AbstractWallet.scala*, which ensures the seed is unique per-algorithm and per-key, such that duplicate keys cannot be generated. Specifically, the seed is computed by hashing (with Blake2b256) the concatenation of the static wallet seed, the nonce, and a *salt*, a static string representing the key type which acts as a domain separator.

In order to avoid the generation of duplicate keys based on a repeating nonce value (as outlined in this finding), this function now maintains an algorithm-specific nonce counter. Upon generation of a new secret, that nonce is retrieved from storage, after which the execution enters a loop where the nonce is incremented at each iteration. That loop attempts to generate a secret with a seed based on the current nonce value, checks whether that secret already exists and in case it doesn't, returns it while also storing the updated nonce value as well as the secret. This process is aligned with the recommendations outlined above.

As a side note, it is important to highlight that the static wallet seed must be of high-entropy in order for this key generation procedure to be secure.

This finding is considered *fixed* as a result.

High

Unauthenticated API Routes May Permit DoS and Eclipse Attacks

Overall Risk	High	**Finding ID**	NCC-E005513-CRR
Impact	Medium	**Component**	sdk/scala
Exploitability	High	**Category**	Access Controls
		Status	Fixed

Impact

Attackers may cause a Denial of Service (DoS) and/or eclipse attacks against the applications utilizing the Horizen Labs SDK.

Description

NCC Group identified a number of unauthenticated HTTP API routes that may allow attackers to stop a node, force a connection to a node of the attackers choosing, and to drop a node connection to another node. They do not implement the SDK `withAuth` directive, which controls access to resources using an API token, as illustrated below:

```
 def connect: Route = (post & path("connect")) {
    entity(as[ReqConnect]) {
// SNIP
 val port = addressAndPort.group(2).toInt
                 networkController ! ConnectTo(PeerInfo.fromAddress(new InetSocketAddress(host,
                 ↳ port)))
                 ApiResponseUtil.toResponse(RespConnect(host + ":" + port))
             }
 // SNIP
  }
```

```
  def disconnect: Route = (post & path("disconnect")) {
    entity(as[ReqDisconnect]) {
      // SNIP
             peerManager ! RemovePeer(peerAddress)
             // Disconnect the connection if present and active.
             // Note: `Blacklisted` name is misleading, because the message supposed to be used
             ↳ only during peer penalize
             // procedure. Actually inside NetworkController it looks for connection and emits
             ↳ `CloseConnection`.
             networkController ! Blacklisted(peerAddress)
             ApiResponseUtil.toResponse(RespDisconnect(host + ":" + port))
         }
// SNIP
  }
```

```
  def stop: Route = (post & path("stop")) {
    if (app.stopAllInProgress.compareAndSet(false, true)) {
      try {
         // SNIP
           app.sidechainStopAll(true)
           log.info("... core application stop returned")
```

```
        }
      }).start()
// SNIP
  }
```

Adversaries with connectivity to the HTTP API may attempt to isolate and manipulate target nodes by controlling all the peers the target is communicating with to illegitimately influence the network (Eclipse attack) and/or perform a denial of service attack against the network.

Recommendation

Ensure all routes are authenticated by adding the `withAuth` directive.

Location

SidechainNodeApiRoute

Retest Results

2023-03-21 – Fixed

The Horizen Labs team addressed the issue described in this finding and its sibling finding "Fragile API Authentication" in a set of four pull requests, across the SDK and Horizen Labs' *Sparkz* project. The first two pull requests in the list below are for the SDK, while the latter two are updates to the *Sparkz* dependency (and as such were not reviewed in as much depth as the PRs for the SDK).

- Sidechains-SDK PR 647 – Added Basic Authentication inside REST interface
- Sidechains-SDK PR 752 – Updated Bcrypt library
- Sparkz PR 40 – Added Basic Authentication Directive
- Sparkz PR 47 – Update Bcrypt library

In *Sparkz*'s pull request 40, the team replaced the `withAuth` directive with an updated `withBasicAuth` directive, which implements *HTTP Basic Auth*.

Pull request 647 for the SDK then updated all the `withauth` directives to `withBasicAuth` and added the latter to a number of routes that were previously unprotected. In particular, the three routes highlighted in this finding (`connect`, `disconnect` and `stop`) are now protected with the `withBasicAuth` directive.

This finding is considered *fixed* as a result.

High

Potential Signature Forgery Due to Noncanonical Encoding of `message` to `FieldElement`

Overall Risk High
Impact High
Exploitability High

Finding ID NCC-E005513-XEX
Component sdk/java
Category Cryptography
Status Fixed

Impact

An attacker may be able to forge valid signatures for selectively crafted messages, allowing them to potentially forge transactions and other sidechain-related cryptographic constructions.

Description

The function `messageToFieldElement()` defined in the FieldElementUtils.java file is used to convert a byte array, representing a message, to a field element, an object of the `FieldElement` class, which is exposed by the underlying (and out-of-scope) *zendoo-sc-cryptolib* library.

That function expects a byte array as parameter, representing the message to convert to a field element. Currently, the only validation performed on that `message` parameter is a length check, ensuring that byte arrays longer than the field length are rejected. The function then proceeds to call the `FieldElement.deserialize` function, passing as argument a copy of the `message` byte array, of length `fieldElementLength()`, as can be seen in the code excerpt below.

```
public static FieldElement messageToFieldElement(byte[] message) {
    if (message.length > fieldElementLength()) {
        throw new IllegalArgumentException("Message length is exceed allowed message len. Message len " +
                message.length + " but it shall be less than " + fieldElementLength());
    }
    return FieldElement.deserialize(Arrays.copyOf(message, fieldElementLength()));
}
```

However, in case the `message` parameter is of length shorter than the value returned by `fieldElementLength()`, it will be right-padded with zeros until the array has the specified length. This behaviour is described in the documentation of the function `Arrays.copyOf()` defined in *java/util/Arrays.java*, and highlighted below, for reference. One can see that the `copyOf()` function first declares a new byte array of the given length, and then starts copying source bytes at index `0` of the destination, explaining the right-padding behaviour.

```
/**
 * Copies the specified array, truncating or padding with zeros (if necessary)
 * so the copy has the specified length.  For all indices that are
 * <SNIP>
 *
 * @param original the array to be copied
 * @param newLength the length of the copy to be returned
```

```
 * @return a copy of the original array, truncated or padded with zeros
 *     to obtain the specified length
 * @throws NegativeArraySizeException if <tt>newLength</tt> is negative
 * @throws NullPointerException if <tt>original</tt> is null
 * @since 1.6
 */
public static byte[] copyOf(byte[] original, int newLength) {
    byte[] copy = new byte[newLength];
    System.arraycopy(original, 0, copy, 0,
                     Math.min(original.length, newLength));
    return copy;
}
```

Consequently, this implies that the `messageToFieldElement()` encoding of different source byte arrays will result in the same field element. As an example, the byte arrays (in hexadecimal form) `0xAA`, `0xAA00`, `0xAA0000`, ... , `0xAA00...[29 times]..00` will *all* result in the same field element after having been converted by the function `deserialize()` of `FieldElement`.

Since the `messageToFieldElement()` function is used extensively by cryptographic operations, this may allow attackers to forge signatures for new messages. Consider the Schnorr signature verification function, `verify()`, defined in SchnorrFunctionsImplZendoo.java. This function uses the `messageToFieldElement()` to encode the message as a field element, as can be seen in the code excerpt below.

```
public boolean verify(byte[] messageBytes, byte[] publicKeyBytes, byte[] signatureBytes) {
    SchnorrPublicKey publicKey = SchnorrPublicKey.deserialize(publicKeyBytes);
    FieldElement fieldElement = messageToFieldElement(messageBytes);
    SchnorrSignature signature = SchnorrSignature.deserialize(signatureBytes);

    boolean signatureIsValid = publicKey.verifySignature(signature, fieldElement);
```

As a result, a valid signature for the message `0xAA` also constitutes a valid signature for the messages `0xAA00`, `0xAA0000`, etc.

Recommendation

In the Sidechains SDK, a common behaviour is to first hash messages prior to converting them to field elements. Thus, as a first step, consider whether the `messageToFieldElement()` function is ever necessary for messages with lengths that are not equal to the underlying field element size. If no such usage is discovered, reject all input to the function that have lengths different than `fieldElementLength()`. Otherwise, consider explicitly *left-padding* byte arrays prior to calling `FieldElement.deserialize()`.

Location

FieldElementUtils.java

Retest Results

2023-03-20 – Fixed

In pull request 668, the Horizen Labs team updated the `messageToFieldElement()` function such that it throws an exception when the message provided as argument is *not equal to* the value of `fieldElementLength()`:

```
message.length != fieldElementLength()
```

That pull request also updates a few of the `messageToFieldElement()` calls performed as part of the VRF computation to calls to `elementToFieldElement()`, which behaves in the same way the outdated `messageToFieldElement()` function did, namely by silently padding the byte arrays. While this seems to potentially still present a risk, the Horizen Labs team indicated that they are confident that this cannot lead to issues, see the *Client Response* section below.

As such, this finding is considered *fixed*.

Client Response

Since we are sure that in all the places where this utility method is used, the exact length of the input data is `fieldElementLength` we may get rid of Array.copy to avoid leading zeros and make a strict check of message byte array. All the places of creation of message create a 32 bytes array.

The only exception is `buildVrfMessage`, but here we are confident on the source data, every node calculates it by itself in a predetermined way.

Medium

Missing Key Validation Upon Initialization

Overall Risk	Medium	**Finding ID**	NCC-E005513-EYU
Impact	Medium	**Component**	sdk/java
Exploitability	Medium	**Category**	Data Validation
		Status	Fixed

Impact

Invalid public keys or key pairs can be successfully instantiated by the Sidechains SDK, potentially leading to a number of unforeseen issues, such as consensus breaches or invalid computation of cryptographic operations.

Description

The Sidechains SDK defines classes wrapping numerous cryptographic keys and key pairs. Objects of these classes are frequently instantiated by providing a byte array representing the key as a constructor argument. However, in most instances, the key objects are created with limited concern for the validity of the keys themselves. Elliptic curve public keys are points on an elliptic curve, and as such must abide by specific mathematical rules.

To illustrate this issue, consider the `VrfPublicKey` constructor. This VRF public key constructor does not check whether the bytes provided form a valid key; these bytes are simply copied to the member variable `publicBytes`, as can be seen below.

```
public VrfPublicKey(byte[] publicKey) {
    Objects.requireNonNull(publicKey, "Public key can't be null");

    if(publicKey.length != KEY_LENGTH)
        throw new IllegalArgumentException(String.format("Incorrect pubKey length, %d
        ↳ expected, %d found", KEY_LENGTH, publicKey.length));

    publicBytes = Arrays.copyOf(publicKey, publicKey.length);
}
```

Note that there is a `isValid()` function defined a couple of lines below, which checks whether the key bytes represent a valid public key. This function could be called upon key creation.

```
public boolean isValid() {
    return CryptoLibProvider.vrfFunctions().publicKeyIsValid(pubKeyBytes());
}
```

Similarly, in PublicKey25519Proposition.java, the constructor of that function also does not check that the public key is valid, as can be seen below.

```
public PublicKey25519Proposition(byte[] pubKeyBytes)
{
    if(pubKeyBytes.length != KEY_LENGTH)
        throw new IllegalArgumentException(String.format("Incorrect pubKey length, %d
        ↳ expected, %d found", KEY_LENGTH, pubKeyBytes.length));

    _pubKeyBytes = Arrays.copyOf(pubKeyBytes, KEY_LENGTH);
}
```

Additionally, note that this function also only performs a length check, while the `VrfPublicKey()` constructor above ensures the byte array is non-null:

```
Objects.requireNonNull(publicKey, "Public key can't be null");
```

This comment also applies to the `SchnorrProposition()` public key constructor.

Similar observations can also be made about the different private key constructors. Specifically, multiple private key constructors not only require a byte array representation of the private key to be instantiated, but also that of the public key. Consider for example the `PrivateKey25519()` constructor, provided below, which expects two byte arrays for the private and public keys as parameters. The only check performed by that constructor is a length check, but that function never checks whether the public key is valid, or that it matches the corresponding private key.

```
public PrivateKey25519(byte[] privateKeyBytes, byte[] publicKeyBytes)
{
    if(privateKeyBytes.length != PRIVATE_KEY_LENGTH)
        throw new IllegalArgumentException(String.format("Incorrect private key length, %d expected, %d found", PRIVATE_KEY_LENGTH,
                privateKeyBytes.length));
    if(publicKeyBytes.length != PUBLIC_KEY_LENGTH)
        throw new IllegalArgumentException(String.format("Incorrect pubKey length, %d expected, %d found", PUBLIC_KEY_LENGTH,
                publicKeyBytes.length));

    this.privateKeyBytes = Arrays.copyOf(privateKeyBytes, PRIVATE_KEY_LENGTH);
    this.publicKeyBytes = Arrays.copyOf(publicKeyBytes, PUBLIC_KEY_LENGTH);
}
```

Similarly, consider the `VrfSecretKey()` constructor, which also does not check the validity of the key pair. That constructor additionally does not check the length of either key, but only ensures that the pointers are not null.

```
public VrfSecretKey(byte[] secretKey, byte[] publicKey) {
    Objects.requireNonNull(secretKey, "Secret key can't be null");
    Objects.requireNonNull(publicKey, "Public key can't be null");

    secretBytes =  Arrays.copyOf(secretKey, secretKey.length);
    publicBytes = Arrays.copyOf(publicKey, publicKey.length);
}
```

Recommendation

Consider making a pass over all the cryptographic key constructors, ensuring the same level of care is applied to the creation of the different keys. That includes:

- ensuring all byte arrays are non-null,
- ensuring all byte arrays are of correct lengths,
- ensuring all public keys are valid,
- ensuring all private keys are valid, and
- ensuring all public keys match their corresponding private key.

Location

- VrfPublicKey.java
- PublicKey25519Proposition.java
- SchnorrProposition.java

- PrivateKey25519.java
- VrfSecretKey.java

Retest Results

2023-01-26 – Partially Fixed

The updates corresponding to this finding were provided in https://github.com/HorizenOfficial/Sidechains-SDK/pull/679.

The `PrivateKey25519`, `VrfSecretKey` and `SchnorrSecret` constructors were updated to check both that the object is non-null, and that the length is correct. Additionally, a check that the public key matches the private key got added during deserialization, in the `PrivateKey25519Serializer.parse()`, `VrfSecretKeySerializer.parse()` and `SchnorrSecret.parse()` functions respectively. The `parse()` functions call the constructor for their respective class, so the final object is fully validated. However, note that any other direct usage of the constructor will not be fully validated. For instance, the current codebase uses each constructor in the `generateNextSecret()` function as well, although there is a pull request that plans to remove the `generateNextSecret()` methods.

Additionally, note that no validation is present for the `VrfPublicKey`, `PublicKey25519Proposition` or `SchnorrProposition` public key constructors.

Finally, the following method was added for the `PrivateKeySecp256k1` class:

```
public Boolean isPublicKeyValid() {
    return true;
}
```

which does not perform any validation.

2023-03-22 – Fixed

In a subsequent update to the pull request, the Horizen Labs team added optional validation functionalities to the `VrfPublicKey`, `PublicKey25519Proposition` and `SchnorrProposition` public key constructors, toggled by a boolean parameter passed to the constructors.

The team also indicated that shifting the validation responsibility to the parsing logic was a deliberate choice, see section *Client Response* below.

Regarding the function `isPublicKeyValid()` for the `PrivateKeySecp256k1` class highlighted above, consider removing it entirely, since it does not meaningfully perform any validation.

This finding is now considered *fixed*.

Client Response

We never check private->public validity in the constructor, only in the serializer, since the only place where we may have an issue is during the parse method execution, so when we restore the keys from the ledger. Constructor is used only in case of parse and create (here we are confident that everything is fine).

Medium

Missing Data Length in Hash Computations May Result in Collisions

Overall Risk	Medium	**Finding ID**	NCC-E005513-X4D
Impact	High	**Component**	SDK
Exploitability	Low	**Category**	Cryptography
		Status	Risk Accepted

Impact

Non-canonical serialization of data may allow attackers to craft colliding messages with sometimes serious consequences such as signature forgery.

Description

When serializing data, especially for use in cryptographic protocols, care should be taken to ensure that the serialization of different inputs does not result in the same output. If that were the case, attackers could for example craft new messages for which the signature on an existing, but different message would also be valid.

Best practice on this topic is to use a variant of a *TLV* (type-length-value) encoding, where the value to be encoded is prefixed with a non-ambiguous representation of its type, as well as the length of the data to be encoded. A fine example of this is the Recursive Length Prefix (RLP) encoding used by Ethereum and replicated by the Sidechains SDK, for example in the encoding of transactions, as can be seen in the following code snippet taken from *EthereumTransaction.java*.

```
public byte[] encode(boolean accountSignature) {
    return EthereumTransactionEncoder.encodeAsRlpValues(this, accountSignature);
}
```

The NCC Group team noted that data serialization (prior to being fed to hash functions) was in some instances susceptible to non-canonical encodings, potentially giving attackers the ability to forge messages.

The first example occurs in the `messageToSign()` function, in BoxTransaction.java, where a *to-be-signed* message is computed as the concatenation of a few byte arrays. However, the respective lengths of these byte arrays are not prepended before serializing them. Hence, it seems theoretically possible to create two different instances of *messageToSign* that are encoded in the same output.

For illustration purposes only, consider the following example. The encoding of a message containing the values `unlockersStream = {0x00, 0x11}` and `newBoxesStream = {0x22}` would be the same as the encoding of the same message containing the values `unlockersStream = {0x00}` and `newBoxesStream = {0x11, 0x22}` in the function below.

```
public byte[] messageToSign() {
    ByteArrayOutputStream unlockersStream = new ByteArrayOutputStream();
    for(BoxUnlocker<P> u : unlockers()) {
        byte[] boxId = u.closedBoxId();
        unlockersStream.write(boxId, 0, boxId.length);
    }

    ByteArrayOutputStream newBoxesStream = new ByteArrayOutputStream();
```

```java
    for(B box : newBoxes()) {
        byte[] boxBytes = box.bytes();
        newBoxesStream.write(boxBytes, 0, boxBytes.length);
    }

    return Bytes.concat(
            new byte[]{version()},
            unlockersStream.toByteArray(),
            newBoxesStream.toByteArray(),
            Longs.toByteArray(fee()),
            customDataMessageToSign());
}
```

The `messageToSign()` function above is called in the `create()` function in OpenStakeTransaction.java, prior to signing the message, as can be seen in the code excerpt below.

```java
    OpenStakeTransaction unsignedTransaction = new OpenStakeTransaction(from.getKey().id(), output, null, forgerIndex, fee, OPEN_STAKE_TRANSACTION_VERSION);

    byte[] messageToSign = unsignedTransaction.messageToSign();
    Secret secret = from.getValue();

    OpenStakeTransaction transaction = new OpenStakeTransaction(from.getKey().id(), output, (Signature25519) secret.sign(messageToSign), forgerIndex, fee, OPEN_STAKE_TRANSACTION_VERSION);
    transaction.transactionSemanticValidity();

    return transaction;
}
```

Another and somewhat related instance of this problem can also be seen in the AccountStateMetadataStorageView.scala source file, where some data is stored at a particular location indexed by a key, computed as the hash of a static prefix concatenated with some integers, as can be seen in the code excerpt below.

```scala
def calculateKey(key: Array[Byte]): ByteArrayWrapper = {
  new ByteArrayWrapper(Blake2b256.hash(key))
}

// <snip>

private[horizen] def getBlockFeeInfoCounterKey(withdrawalEpochNumber: Int): ByteArrayWrapper = {
  calculateKey(Bytes.concat("blockFeeInfoCounter".getBytes, Ints.toByteArray(withdrawalEpochNumber)))
}

private[horizen] def getBlockFeeInfoKey(withdrawalEpochNumber: Int, counter: Int): ByteArrayWrapper = {
  calculateKey(Bytes.concat("blockFeeInfo".getBytes, Ints.toByteArray(withdrawalEpochNumber), Ints.toByteArray(counter)))
}
```

However, since the fixed prefix used in the call to `getBlockFeeInfoKey()` is a prefix of the one used in `getBlockFeeInfoCounterKey()` (namely `blockFeeInfo` is a prefix of `blockFeeInfoCounter`), there is a possibility that the call to `getBlockFeeInfoKey()` will result in the same value computed as the call to `getBlockFeeInfoCounterKey()`, namely when the encoding of the values `Ints.toByteArray(withdrawalEpochNumber), Ints.toByteArray(counter)` is equal to the byte representation of `"Counter"`.

Recommendation

Perform a pass through the source code and inventory all places where non-canonical encodings may happen, specifically prior to their use in cryptographic operations and when used as storage keys. For each instance identified, consider updating the encoding scheme used to follow a variant of *TLV* encoding, by at least prefixing the byte arrays with their respective lengths.

Location

- BoxTransaction.java
- AccountStateMetadataStorageView.scala

Retest Results

2023-03-20 – Not Fixed

The Horizen Labs team chose not to address this concern, since the perceived risk was deemed to be minimal in the scenarios described in this finding and because data serialization changes would lead to backward compatibility issues.

Client Response

We choose not to fix it, we need to keep in mind the context and usage of the data to be serialized. For example, the encoding of a message containing the concatenated values:

```
unlockersStream = {0x00, 0x11}
newBoxesStream = {0x22}
```

would be the same as the encoding of the same message containing the concatenated values:

```
unlockersStream = {0x00}
newBoxesStream = {0x11, 0x22}
```

Here we need to consider that the creator doesn't control neither unlockers nor new boxes and the meaning/usage of them is completely different. So practically and even theoretically it can't harm us. Moreover, we can't easily change serialization of the data, because it will lead to backward incompatibility, therefore we chose not to fix this.

Database keys are controlled by the Node itself and are not exposed outside, don't participate in any consensus related activities. Similarly as before, we can't easily change serialization of the data, because it will lead to backward incompatibility, therefore we chose not to fix this.

Medium

Locale-Dependent Charset Encoding of Strings Can Lead to Consensus Breach

Overall Risk	Medium	**Finding ID**	NCC-E005513-QY2
Impact	Medium	**Component**	sdk/scala
Exploitability	Low	**Category**	Other
		Status	Fixed

Impact

Non-canonical encoding of Strings may result in clients computing different messages (such as VRF messages), which may eventually lead to consensus breaches and additional, unforeseen adversarial scenarios.

Description

This finding presents a general issue with the encoding of Strings into bytes. The description of the issue starts with a concrete example in the VRF computation, and later on discusses some other problematic instances.

Concrete Example: VRF Computation

The Sidechains SDK defines and uses a *salt* string in order to compute Verifiable Random Function (VRF) messages, as can be seen in the excerpt below, taken from the function `buildVrfMessage()`, located in the file package.scala.

```
def buildVrfMessage(slotNumber: ConsensusSlotNumber, nonce: NonceConsensusEpochInfo): VrfMessage = {
  val slotNumberBytes = Ints.toByteArray(slotNumber)
  val nonceBytes = nonce.consensusNonce

  val resBytes = Bytes.concat(slotNumberBytes, nonceBytes, consensusHardcodedSaltString)
```

This *salt* is defined earlier in the same file, as follows.

```
 val consensusHardcodedSaltString: Array[Byte] = "TEST".getBytes()
```

However, the `getBytes()` method, from the `java.lang.String` package, performs the encoding into bytes with a locale-dependent charset. This is explicitly stated in the function documentation, defined in the source file *java/lang/String.java* and provided below, for reference.

```
/**
 * Encodes this {@code String} into a sequence of bytes using the
 * platform's default charset, storing the result into a new byte array.
 *
 * <p> The behavior of this method when this string cannot be encoded in
 * the default charset is unspecified.  The {@link
 * java.nio.charset.CharsetEncoder} class should be used when more control
 * over the encoding process is required.
 *
 * @return  The resultant byte array
 *
 * @since      JDK1.1
```

```
 */
public byte[] getBytes() {
    return StringCoding.encode(value, 0, value.length);
}
```

As a result, Java deployments using a given platform default charset, for example UTF-16, would compute a different salt value than deployments using UTF-8, which would result in the computation of different VRF messages.

Other Instances

In addition to the VRF computation above, the NCC Group team observed a number of additional instances where the `getBytes()` function was called. A number of these instances happen in the computation of keys for local key-value storage, which should do not pose a problem since they are presumably not shared between peers. There are however other cases where that problematic encoding is used, such as in Equihash.scala:

```
def checkEquihashSolution(msgBytes: Array[Byte], solution: Array[Byte]): Boolean = {
  val b2digest: Blake2bDigest = new Blake2bDigest(null, HashOutputLength, null,
      Bytes.concat("ZcashPoW".getBytes, BytesUtils.reverseBytes(Ints.toByteArray(N)),
      ↳ BytesUtils.reverseBytes(Ints.toByteArray(K))))
  b2digest.update(msgBytes, 0, msgBytes.length)

  checkEquihashSolution(b2digest, solution)
}
```

Recommendation

Consider specifying the exact encoding in the `getBytes()` method call, for *all* calls to that function. Specifically, for the VRF computation example described above, consider replacing the call highlighted with the following.

```
val consensusHardcodedSaltString: Array[Byte] = "TEST".getBytes(StandardCharsets.UTF_8)
```

Reproduction Steps

Consider the following excerpt from a sample run of the Scala REPL.

```
scala> import java.nio.charset.StandardCharsets

scala> "TEST".getBytes(StandardCharsets.UTF_8)
val res2: Array[Byte] = Array(84, 69, 83, 84)

scala> "TEST".getBytes(StandardCharsets.UTF_16)
val res3: Array[Byte] = Array(-2, -1, 0, 84, 0, 69, 0, 83, 0, 84)

scala> "TEST".getBytes(StandardCharsets.US_ASCII)
val res4: Array[Byte] = Array(84, 69, 83, 84)
```

Location

- package.scala
- Equihash.scala

Retest Results

2023-01-26 – Fixed

In commit c8206db, part of pull request 660, the Horizen Labs team addressed the issue outlined in this finding by replacing instances of

```
String.getBytes()
```

with

```
String.getBytes(StandardCharsets.UTF_8)
```

As a result, this finding is considered *fixed*.

Medium

Incorrect Lower Bound in Gas Fee Computation

Overall Risk	Medium	**Finding ID**	NCC-E005513-DXB
Impact	Low	**Component**	sdk/scala/account
Exploitability	Medium	**Category**	Other
		Status	Fixed

Impact

The usage of an incorrect lower bound when decreasing the Base Fee (due to the parent block using less gas than its target) may result in users paying higher fees than necessary; it may also lead to interoperability issues, and potentially have additional, unforeseen consequences in the economic incentive model of the blockchain.

Description

The Ethereum Improvement Proposal (EIP) 1559 introduced an updated transaction pricing mechanism. This updated mechanism now includes a fixed-per-block network fee, the Base Fee, which is always burned and gets updated dynamically based on current network congestion as a function of the gas used and the gas target in the parent block. EIP-1559 summarizes it as follows.

> The algorithm results in the base fee per gas increasing when blocks are above the gas target, and decreasing when blocks are below the gas target.

Concretely, the computation of the Base Fee for the new block depends on how full the parent block was; the updated Base Fee may increase or decrease, up to a maximum of 12.5%.

The NCC Group consultants noticed that the Base Fee computation in the Sidechains-SDK did not fully match the Base Fee computation of other implementations of that proposal.

Specifically, in the transaction fee computation of the SDK, which happens in the file *FeeUtils.scala*, the computation of the Base Fee is lower bounded by 1 in case the parent block used less gas than its target. This can be seen in the highlighted portion of the code excerpt provided below, taken from the `calculateBaseFeeForBlock()` function.

```
val gasDiff = blockHeader.gasUsed - gasTarget

val baseFeeDiff = BigInteger
  .valueOf(Math.abs(gasDiff))
  .multiply(blockHeader.baseFee)
  .divide(BigInteger.valueOf(gasTarget))
  .divide(BASE_FEE_CHANGE_DENOMINATOR)

if (gasDiff.signum == 1) {
  // If the parent block used more gas than its target, the baseFee should increase
  blockHeader.baseFee.add(baseFeeDiff.max(BigInteger.ONE))
} else {
  // Otherwise if the parent block used less gas than its target, the baseFee should
  ↳ decrease
  blockHeader.baseFee.subtract(baseFeeDiff).max(BigInteger.ONE)
}
```

This doesn't follow what `geth`, the Go Ethereum client, performs for this computation. Namely, the computation in that case enforces a lower bound of 0 in the computation performed in eip1559.go, as can be seen in the code excerpt below.

```
if parent.GasUsed > parentGasTarget {
    // If the parent block used more gas than its target, the baseFee should increase.
    // max(1, parentBaseFee * gasUsedDelta / parentGasTarget / baseFeeChangeDenominator)
    num.SetUint64(parent.GasUsed - parentGasTarget)
    num.Mul(num, parent.BaseFee)
    num.Div(num, denom.SetUint64(parentGasTarget))
    num.Div(num, denom.SetUint64(params.BaseFeeChangeDenominator))
    baseFeeDelta := math.BigMax(num, common.Big1)

    return num.Add(parent.BaseFee, baseFeeDelta)
} else {
    // Otherwise if the parent block used less gas than its target, the baseFee should decrease.
    // max(0, parentBaseFee * gasUsedDelta / parentGasTarget / baseFeeChangeDenominator)
    num.SetUint64(parentGasTarget - parent.GasUsed)
    num.Mul(num, parent.BaseFee)
    num.Div(num, denom.SetUint64(parentGasTarget))
    num.Div(num, denom.SetUint64(params.BaseFeeChangeDenominator))
    baseFee := num.Sub(parent.BaseFee, num)

    return math.BigMax(baseFee, common.Big0)
}
```

Recommendation

Consider updating the lower bound in the line highlighted in the Scala excerpt above to 0, as follows:

```
blockHeader.baseFee.subtract(baseFeeDiff).max(BigInteger.ZERO)
```

Location

FeeUtils.scala

Retest Results

2023-01-25 – Fixed

In pull request 661, the Horizen Labs team fixed the issue described in this finding by swapping the line in question with:

```
blockHeader.baseFee.subtract(baseFeeDiff).max(BigInteger.ZERO)
```

As such, this finding is considered *fixed*.

Medium

Possible Reuse of Randomness When Generating Keys

Overall Risk	Medium	**Finding ID**	NCC-E005513-TV2
Impact	Low	**Component**	sdk/java
Exploitability	Low	**Category**	Cryptography
		Status	Fixed

Impact

Reusing randomness when generating keys may lead to weaker security than expected for the resulting private keys.

Description

The function `generateNextSecret` generates public keys on the Ed25519 curve as follows:

```java
public PrivateKey25519 generateNextSecret(NodeWalletBase wallet) {
    List<Secret> prevSecrets = wallet.secretsOfType(PrivateKey25519.class);
    byte[] nonce = Ints.toByteArray(prevSecrets.size());
    byte[] seed = Blake2b256.hash(Bytes.concat(wallet.walletSeed(), nonce));

    return generateSecret(seed);
}
```

It is defined similarly for the curve secp256k1:

```java
public PrivateKeySecp256k1 generateNextSecret(NodeWalletBase wallet) {
    List<Secret> prevSecrets = wallet.secretsOfType(PrivateKeySecp256k1.class);
    byte[] nonce = Ints.toByteArray(prevSecrets.size());
    byte[] seed = Keccak256.hash(Bytes.concat(wallet.walletSeed(), nonce));

    return generateSecret(seed);
}
```

Since the nonce depends only on the number of secrets of a particular type, if the number of existing Ed25519 key pairs matches the number of existing secp256k1 key pairs, then the next key generated for each curve will use the same seed as randomness.

The seed is then used to generate private keys as follows. In the case of the Ed25519 curve, it computes a hash of the seed, and uses this hash to form a private key.

```java
public static Pair<byte[], byte[]> createKeyPair(byte[] seed) {
    Ed25519PrivateKeyParameters privateKey = new Ed25519PrivateKeyParameters(Sha256.hash(seed), 0);
    Ed25519PublicKeyParameters publicKey = privateKey.generatePublicKey();

    return new Pair<>(privateKey.getEncoded(), publicKey.getEncoded());
}
```

In the case of curve secp256k1, the seed is used to instantiate a `SecureRandom` instance:

```java
public PrivateKeySecp256k1 generateSecret(byte[] seed) {
    try {
        ECKeyPair keyPair = Keys.createEcKeyPair(new SecureRandom(seed));
        // keyPair private key can be 32 or 33 bytes long
```

```
        byte[] privateKey = Arrays.copyOf(keyPair.getPrivateKey().toByteArray(),
        ↪ Secp256k1.PRIVATE_KEY_SIZE);
        return new PrivateKeySecp256k1(privateKey);
    } catch (Exception e) {
        // TODO handle it
        System.out.println("Exception: "+ e.getMessage());
        return null;    }
}
```

Since this seed is used to generate the keys using two different non-reversible methods, it does not appear that reusing the seed currently results in weaknesses in either of the keys. However, reusing randomness is generally a bad idea, and could lead to issues if the details of the implementation were to change.

Additionally, similar methods are also used to generate a seed in the case of VRF and Schnorr secret key generation, but the seed is not currently used in the key generation process. However, the VRF key generation function includes the following note (in the VrfFunctionsImplZendoo.java file):

```
    //@TODO Seed shall be supported from JNI side
```

which suggests the seed might be used in the future.

Recommendation

Ensure the same seed is never reused for security sensitive purposes. This could be done by using a global monotonically increasing nonce counter, or by using algorithm-specific monotonically increasing nonce counters and adding a domain separation tag for each algorithm in the nonce generation process. In the cases where the seed is not used, remove unnecessary seed computations or document plans for future usage.

Additionally, consider aligning the key generation processes to use a more standardized approach for each of the primitives. For example, the `Ed25519PrivateKeyParameters` method from the `org.bouncycastle.crypto` library also supports key generation from a `SecureRandom` [3] instance, which would allow aligning the implementations of the `Ed25519` and `Secp256k1` private key generation functions.

More generally, consider updating this key generation functionality to be based on a standardized scheme for deterministic key generation within a wallet, such as BIP 0032[4].

Location

- PrivateKey25519Creator.java
- PrivateKeySecp256k1Creator.java
- VrfKeyGenerator.java
- SchnorrKeyGenerator.java

Retest Results

2023-03-21 – Fixed

In pull request 680, the Horizen Labs team revamped the secret generation procedure in order to address this finding and its sibling finding "Deterministic Key Generation May Produce Duplicate Private Keys".

3. https://javadoc.io/static/org.bouncycastle/bcprov-ext-jdk15on/1.67/org/bouncycastle/crypto/params/Ed25519PrivateKeyParameters.html
4. https://en.bitcoin.it/wiki/BIP_0032

The changes introduced in that pull request simplified and standardized the secret generation procedure for the various algorithms. The respective `generateNextSecret()` functions defined differently for each algorithm have now been removed, and that functionality has been moved to the *AbstractWallet.scala*. Secret generation is now exclusively performed by the function `generateSecret()` with a *seed* as argument.

That seed is generated by the `generateNextSecret()` function, a generic function defined in *AbstractWallet.scala*, which ensures the seed is unique per-algorithm and per-key, such that duplicate keys cannot be generated. Specifically, the seed is computed by hashing (with Blake2b256) the concatenation of the static wallet seed, the nonce, and a *salt*, a static string representing the key type which acts as a domain separator. This is aligned with the recommendations outlined in this finding.

As a side note, it is important to highlight that the static wallet seed must be of high-entropy in order for this key generation procedure to be secure.

This finding is considered *fixed* as a result.

Medium

Fragile API Authentication

Overall Risk	Medium	**Finding ID**	NCC-E005513-4BR
Impact	High	**Component**	sdk/scala
Exploitability	Medium	**Category**	Authentication
		Status	Partially Fixed

Impact

The current authentication/authorization mechanism provided by the Sidechains SDK does not follow best practices for API authentication, which may allow attackers to obtain sensitive material or perform other attacks on a target node.

Description

The HTTP API exposed by the Sidechains SDK allows users to interact with nodes and perform various types of operations, such as generating keys, connecting to other nodes, or querying the blockchain for specific blocks. Some of these operations being more sensitive than others, a programmatic gateway has been put in place to limit access to these operations to authorized parties only. That protection mechanism is realized by the `withAuth` block, as can be seen in the code snippet below, excerpted from the SidechainWalletApiRoute.scala source file.

```
  /**
   * Perform a dump on a file of all the secrets inside the wallet.
   */
  def dumpSecrets: Route = (post & path("dumpSecrets")) {
    withAuth {
      entity(as[ReqDumpWallet]) { body =>
```

The `withAuth` block is an API directive defined in the *Sparkz* dependency, in ApiDirectives.scala more specifically. The implementation of the `withAuth` block checks whether the incoming request includes an “api_key” field, in which case it hashes it using Blake2b and compares that result to the value stored in its configuration, as also shown in the implementation below.

```
  lazy val withAuth: Directive0 = optionalHeaderValueByName(apiKeyHeaderName).flatMap {
    case _ if settings.apiKeyHash.isEmpty => pass
    case None => reject(AuthorizationFailedRejection)
    case Some(key) =>
      val keyHashStr: String = encoder.encode(Blake2b256(key))
      if (settings.apiKeyHash.contains(keyHashStr)) pass
      else reject(AuthorizationFailedRejection)
  }
```

The documentation of the Sidechains SDK for the SimpleApp comparably describes how developers should add authentication to endpoints. The following excerpt, taken verbatim out of the documentation, outlines exactly the current authentication process.

How to add authentication to the endpoints

If you want to add authentication to your HTTP endpoints, you can add an api key hash inside the config file in the section: *restApi.apiKeyHash*. The api key hash should be an Hash of another string (api key) that it’s used in the HTTP request. You can calculate this

Hash using the ScBootstrapping tool with the command `encodeString` (e.g. `encodeString {"string": "Horizen"}`). In the HTTP request you need to add an additional custom header: "api_key":

Example:

HTTP request:

```
"api_key": "Horizen"
```

Config file:

```
restApi {
    "apiKeyHash": "aa8ed2a907753a4a7c66f2aa1d48a0a74d4fde9a6ef34bae96a86dcd7800af98"
}
```

The mechanism used by the Sidechains SDK and the example highlighted in the documentation do not follow best practices for API authentication, which may allow attackers to obtain sensitive material or perform other attacks on a target node. Selected recommendations from online resources[5][6] include:

- Always using Transport Layer Security (TLS).
- Generating long, high entropy API keys.
- Encourage using good secrets management for API keys.
- Configuring different permissions for different API keys.
- Require API keys for every request to the protected endpoint.
- Supporting the ability to revoke API keys.
- Restricting the validity period of API keys, so as not to issue tokens that never expire.
- Do not rely exclusively on API keys to protect sensitive, critical or high-value resources.

In light of these best practices items, the current authentication mechanism does not provide a high level of security. Specifically, there are currently no mechanisms in place for API key expiration and revocation, or more granular permissions for specific API keys. Additionally, the use of TLS is currently not supported by the SDK, high-value resources are exclusively protected using one API key, and not all endpoints are enclosed within a `withAuth` block. Finally, the example above does not promote safe usage and generation of a secure API key; the `api_key` used as example (`"Horizen"`) is easily guessable by an adversary. The StackOverflow blog article also linked above suggests generating such key in the following way.

```
var token = crypto.randomBytes(32).toString('hex');
```

Finally, while most sensitive API routes are protected by the `withAuth` directive, finding "Unauthenticated API Routes May Permit DoS and Eclipse Attacks" describes some specific attacks that may apply to two such unprotected ones. The NCC Group team observed four additional unprotected routes that should presumably be protected as well: `enableCertificateSubmitter`, `disableCertificateSubmitter`, `enableCertificateSigner`, and `disableCertificateSigner`.

5. https://stackoverflow.blog/2021/10/06/best-practices-for-authentication-and-authorization-for-rest-apis/
6. https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html

Recommendation

As a first step, update the example documentation to force, or at least strongly encourage, users to set up API authentication. Promote secure deployments by updating the documentation to use a long, randomly-generated API key. Similar to the *ScBootstrapping* described in the documentation above, a tool could be developed to generate such secure API keys.

Secondly, consider adding the `withAuth` directive to all API routes to decrease attack surface by default. Additionally, consider adding mechanisms to support TLS in applicable scenarios.

Finally, consider limiting the validity period of API keys, providing users with the ability to periodically issue new API keys, and implementing revocation methods for the existing key in case it has been compromised.

Location

- Sidechains-SDK/sdk/src/main/scala/com/horizen/account/api/http/
- Sidechains-SDK/sdk/src/main/scala/com/horizen/api/http/
- Sparkz/src/main/scala/sparkz/core/api/http/ApiDirectives.scala
- Sidechains-SDK/examples/simpleapp/README.md

Retest Results

2023-03-21 – Partially Fixed

The Horizen Labs team addressed the issue described in this finding and its sibling finding "Unauthenticated API Routes May Permit DoS and Eclipse Attacks" in a set of four pull requests, across the SDK and Horizen Labs' *Sparkz* project. The first two pull requests in the list below are for the SDK, while the latter two are updates to the *Sparkz* dependency (and as such were not reviewed in as much depth as the PRs for the SDK).

- Sidechains-SDK PR 647 – Added Basic Authentication inside REST interface
- Sidechains-SDK PR 752 – Updated Bcrypt library
- Sparkz PR 40 – Added Basic Authentication Directive
- Sparkz PR 47 – Update Bcrypt library

In *Sparkz*'s pull request 40, the team replaced the `withAuth` directive with an updated `withBasicAuth` directive, which implements *HTTP Basic Auth*.

Pull request 647 for the SDK then updated all the `withauth` directives to `withBasicAuth` and added the latter to a number of routes that were previously unprotected.

The API key is now computed using `BCrypt`, which has been introduced with the pull requests 47 for *Sparkz* and 752 for the SDK. It should be noted that `BCrypt`'s cost factor is currently set to 8, which might be too low for best security practices. NCC Group recommends to evaluate whether this cost factor could be increased with minimal impact on usability.

The Horizen Labs team additionally indicated that a number of additional improvements in line with the recommendations above were going to be investigated and implemented in future versions, see section *Client Response* below. As such, the status of this finding was updated to *Partially Fixed*.

Client Response

Not all the endpoints should be protected in the same way but instead we want to implement a sort of an Access Control List (ACL) that let us to select which endpoint should be protected and which is the level of protection.

Based on our needs we also want to limit the call to our API from a localhost origin or from a whitelist of allowed IPs. In the future we also want to investigate the possibility to bind the API server to different network interfaces (at the moment only one can be used).

Since the RPC server is implemented using a single HTTP endpoint that receive all the RPC requests and based on the method included in the body, processes the requests using the correct handler, we are not able right now to give different permissions to different RPC calls.

Medium

Potential Null Pointer Dereference

Overall Risk	Medium	**Finding ID**	NCC-E005513-MTQ
Impact	Medium	**Component**	sdk/java/account
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

Dereferencing a null pointer typically causes a crash or exit, which could be used as a denial of service attack vector against a target node.

Description

In the file TransactionArgs.java, the `toMessage()` function converts a set of transaction arguments into a *Message*, which is the type internally used by the system. The documentation additionally states that this function reimplements similar logic from the Go Ethereum client, *geth*; see below for the relevant code portion and function signature.

```
* Reimplementation of the same logic in GETH.
*/
public Message toMessage(BigInteger baseFee) throws RpcException {
```

However, the BigInteger `baseFee` passed as a parameter is never checked to be non-null and is subsequently used a few lines below, at TransactionArgs.java:

```
if (gasFeeCap.bitLength() > 0 || gasTipCap.bitLength() > 0) {
    effectiveGasPrice = baseFee.add(gasTipCap).min(gasFeeCap);
}
```

Dereferencing a null pointer, by calling the `add()` function on it, would result in a crash due to a `java.lang.NullPointerException` being triggered.

In comparison, the *geth* client performs this null check in its `ToMessage()` function, as can be seen in the code snippet provided below and excerpted from transaction_args.go:

```
if baseFee == nil {
    // If there's no basefee, then it must be a non-1559 execution
    gasPrice = new(big.Int)
    if args.GasPrice != nil {
        gasPrice = args.GasPrice.ToInt()
    }
    gasFeeCap, gasTipCap = gasPrice, gasPrice
}
```

Recommendation

Add a null pointer check on the `baseFee` parameter and set the `effectiveGasPrice` variable to the appropriate value in case the `baseFee` is null (presumably, `effectiveGasPrice` should be set to `gasPrice`).

Location

TransactionArgs.java

Retest Results

2023-01-25 – Fixed

The Horizen Labs team addressed this issues in pull request 657. It was noted that the `baseFee` field cannot be `null` since it is coming from the contextual block header and not from RPC input. The commit adds a check that treats the a `null` `baseFee` value as an invariant violation.

As a result, this finding is considered *fixed*.

Medium

Missing Input Validation May Lead to Truncated Input Data or Other Issues

Overall Risk	Medium	**Finding ID**	NCC-E005513-QCW
Impact	Low	**Component**	sdk/scala/account
Exploitability	Medium	**Category**	Data Validation
		Status	Fixed

Impact

Parsing of structured input data without any validation can lead to truncated inputs or unclear error codes.

Description

The `sendRawTransaction` API defined in the file AccountTransactionApiRoute.scala takes as input an RLP encoded transaction, decodes and possibly signs it, before sending it out:

```
def sendRawTransaction: Route = (post & path("sendRawTransaction")) {
  withAuth {
    entity(as[ReqRawTransaction]) { body =>
      // lock the view and try to create CoreTransaction
      applyOnNodeView { sidechainNodeView =>
        var signedTx = new EthereumTransaction(EthereumTransactionDecoder.decode(body.payload))
        if (!signedTx.isSigned) {
          val txCost = signedTx.getValue.add(signedTx.getGasPrice.multiply(signedTx.getGasLimit))

          val secret =
            getFittingSecret(sidechainNodeView, body.from, txCost)
          secret match {
            case Some(secret) =>
              signedTx = signTransactionWithSecret(secret, signedTx)
              validateAndSendTransaction(signedTx)
            case None =>
              ApiResponseUtil.toResponse(ErrorInsufficientBalance("ErrorInsufficientBalance", JOptional.empty()))
          }
        }
        else
          validateAndSendTransaction(signedTx)
      }
    }
  }
}
```

To decode the RLP encoded byte string, `sendRawTransaction` calls `EthereumTransactionDecoder.decode`, which, in the case of an EIP-1559 transaction, calls `decodeEIP1559Transacti` `on` defined in *EthereumTransactionDecoder.java*, and outputs a decoded transaction structure:

```
22 public static RawTransaction decode(String hexTransaction) {
23     byte[] transaction = Numeric.hexStringToByteArray(hexTransaction);
24     return getTransactionType(transaction) == TransactionType.EIP1559 ? decodeEIP1559Transaction(transaction) : decodeLegacyTransaction(transaction);
25 }
```

```
32 private static RawTransaction decodeEIP1559Transaction(byte[] transaction) {
33     byte[] encodedTx = Arrays.copyOfRange(transaction, 1, transaction.length);
34     RlpList rlpList = RlpDecoder.decode(encodedTx);
35     RlpList values = (RlpList)rlpList.getValues().get(0);
36     long chainId = ((RlpString)values.getValues().get(0)).asPositiveBigInteger().longValueExact();
37     BigInteger nonce = ((RlpString)values.getValues().get(1)).asPositiveBigInteger();
38     BigInteger maxPriorityFeePerGas = ((RlpString)values.getValues().get(2)).asPositiveBigInteger();
39     BigInteger maxFeePerGas = ((RlpString)values.getValues().get(3)).asPositiveBigInteger();
40     BigInteger gasLimit = ((RlpString)values.getValues().get(4)).asPositiveBigInteger();
41     String to = ((RlpString)values.getValues().get(5)).asString();
42     BigInteger value = ((RlpString)values.getValues().get(6)).asPositiveBigInteger();
43     String data = ((RlpString)values.getValues().get(7)).asString();
44     if (((RlpList)values.getValues().get(8)).getValues().size() > 0) throw new IllegalArgumentException("Access list is not supported");
45     RawTransaction rawTransaction = RawTransaction.createTransaction(chainId, nonce, gasLimit, to, value, data, maxPriorityFeePerGas, maxFeePerGas);
46     if (values.getValues().size() == 9) {
47         return rawTransaction;
48     } else {
49         byte[] v = Sign.getVFromRecId(Numeric.toBigInt(((RlpString)values.getValues().get(9)).getBytes()).intValueExact());
50         byte[] r = Numeric.toBytesPadded(Numeric.toBigInt(((RlpString)values.getValues().get(10)).getBytes()), 32);
51         byte[] s = Numeric.toBytesPadded(Numeric.toBigInt(((RlpString)values.getValues().get(11)).getBytes()), 32);
52         Sign.SignatureData signatureData = new Sign.SignatureData(v, r, s);
53         return new SignedRawTransaction(rawTransaction.getTransaction(), signatureData);
54     }
55 }
```

The call to `RlpDecoder.decode()` on line 34 uses functions from the web3j library to parse an RLP encoded byte array into the decoded data. The remainder of the function then parses the returned data into a `rawTransaction` or `SignedRawTransaction` object.

However, the `RlpDecoder.decode()` function only checks that the input is correctly RLP encoded[7], and not whether the list it outputs encodes a properly EIP-1559 formatted transaction[8]. An improperly formatted signature (that is correctly RLP encoded) could thus cause problems during the parsing steps. For example, an insufficient number of elements in the list will throw an out of bounds exception. While any thrown exceptions will be caught and handled by the API and RPC functions, the returned error message may not be very informative in the case of an incorrectly formatted transaction. Additionally, items past

7. https://ethereum.org/en/developers/docs/data-structures-and-encoding/rlp/
8. https://eips.ethereum.org/EIPS/eip-1559

the 12th element of the list will be truncated and silently ignored, which may not coincide with desired or expected behaviour.

A similar lack of input validation is found in the `decodeLegacyTransaction()` function that is called from `EthereumTransactionDecoder.decode()` in the case of a legacy transaction. The `EthereumTransactionDecoder.decode()` function is also called from the `sendRawTransaction` RPC and API endpoints, which are thus susceptible to similar problems. Finally, the functions `EvmLogUtils.rlpDecode()` and `EthereumConsensusDataReceipt.decodeLegacy()` also lack input validation when decoding transaction receipts, but are currently unused.

Recommendation

When parsing structured data, validate that the data has the expected structure before parsing it into the final data structures, and return invalid formatting exceptions otherwise. In particular, verify that there is no trailing data in an RLP stream before declaring a transaction as valid and further processing it.

Location

- AccountTransactionApiRoute.scala
- EthService.scala
- EthereumTransactionDecoder.java
- EvmLogUtils.scala
- EthereumConsensusDataReceipt.scala

Retest Results

2023-03-21 – Fixed

In pull request 719, the Horizen Labs team added a number of length checks and parameter validation steps to the different functions highlighted in this finding. Specifically, both functions `decodeEIP1559Transaction()` and `decodeLegacyTransaction()` (through their helper functions `RlpList2EIP1559Transaction()` and `RlpList2LegacyTransaction()`, respectively) in *EthereumTransactionDecoder.java*, were augmented with a number of checks, ensuring for example that the RLP list obtained after decoding is of expected size.

A few locations in the codebase have also been updated to throw an exception if trailing data is present after having completed the decoding step, see SidechainAccountTransactionsCompanion.scala, EthereumConsensusDataReceipt.scala and EthereumTransactionDecoder.java)

Finally, the pull request also added more comprehensive error checking in the file *RlpStreamDecoder.java*.

As a result, this finding is considered *fixed*.

Medium

Missing or Inconsistent Checks in Parsing Functions

Overall Risk	Medium	**Finding ID**	NCC-E005513-3QY
Impact	Low	**Component**	SDK
Exploitability	Medium	**Category**	Data Validation
		Status	Risk Accepted

Impact

Parsing of structured input data with insufficient validation can lead to truncated inputs, unnecessary processing of invalid inputs, or unclear error codes. Additionally, extraneous input may indicate malicious behavior and failure to check and log it may prevent early detection.

Description

In a number of parsing functions, the performed validation checks appear to be missing or inconsistent. A few possible issues are highlighted below.

Truncated Trailing Data

First, similarly to finding "Missing Input Validation May Lead to Truncated Input Data or Other Issues", various API endpoints do not check whether any trailing data remains in the buffer after parsing. As a concrete example, the `sendTransaction` API endpoint (excerpted below) is designed to parse, validate, and send the transaction provided as input; the `companion.parseBytesTry()` function highlighted below detects the type of transaction provided, and uses the appropriate parsing function from `MC2SCAggregatedTransactionSerializer`, `SidechainCoreTransactionSerializer`, `OpenStakeTransactionSerializer`, or `CertificateKeyRotationTransactionSerializer`.

```
/**
  * Validate and send a transaction, given its serialization as input.
  * Return error in case of invalid transaction or parsing error, otherwise return the id of the transaction.
  */
def sendTransaction: Route = (post & path("sendTransaction")) {
  withAuth {
    entity(as[ReqSendTransactionPost]) { body =>
      val transactionBytes = BytesUtils.fromHexString(body.transactionBytes)
      companion.parseBytesTry(transactionBytes) match {
        case Success(transaction) =>
         validateAndSendTransaction(transaction)
        case Failure(exception) =>
         ApiResponseUtil.toResponse(GenericTransactionError("GenericTransactionError", JOptional.of(exception)))
      }
    }
  }
}
```

However, the various parsing methods do not validate whether all the data has been consumed. For instance, the `OpenStakeTransactionSerializer.parse()` method does not validate that the buffer is empty after reading in the `forgerIndex`, silently truncating any additional data that may be present, as highlighted in the code excerpt below.

```
public OpenStakeTransaction parse(Reader reader) {
    byte version = reader.getByte();
    if (version != OPEN_STAKE_TRANSACTION_VERSION) {
        throw new IllegalArgumentException(String.format("Unsupported transaction
        ↳ version[%d].", version));
    }

    // <snip>

    Signature25519 proof = Signature25519Serializer.getSerializer().parse(reader);
    int forgerIndex = reader.getInt();

    return new OpenStakeTransaction(inputsId, output, proof, forgerIndex, fee, version);
}
```

Missing or Inconsistent Validation

Additionally, the validation in various parsing methods appears to be done in an inconsistent manner, and may omit some desirable checks. For instance, the three parsing methods called by `companion.parseBytesTry()` from the `sendTransaction` API endpoint do not perform all the validity checks that appear in the respective `SemanticValidity()` functions. This may lead to unnecessary processing of invalid transactions until the invalid transaction is caught.

In particular, the `OpenStakeTransactionSerializer.parse()` method only validates that the version is correct; no other validation is performed. However, the `OpenStakeTransaction.transactionSemanticValidity()` method also throws an exception if `forgerIndex < 0`, but this method is not called during the parsing process.

Similarly, the `MC2SCAggregatedTransaction.semanticValidity()` method also checks that the transaction is shorter than the `MAX_TRANSACTION_SIZE`; this check does not appear to be performed during the parsing process.

Finally, the `SidechainCoreTransaction.semanticValidity()` method checks that the number of input ids, proofs, and boxes match, and throws an exception otherwise, see below:

```
// check that we have enough proofs and try to open each box only once.
if (inputsIds.size() != proofs.size() || inputsIds.size() != boxIdsToOpen().size())
        throw new TransactionSemanticValidityException(String.format("Transaction [%s] is
        ↳ semantically invalid: " +
                "inputs number is not consistent to proofs number.", id()));
```

Again, this check does not appear to be performed during the parsing process.

Missing Length Validation

As a final example, in a number of parsing functions, a length value is read from a buffer, and is then used to inform the number of bytes to be read from the buffer. The lack of validation on the length may lead to unexpected issues in the parsing/validation of the remainder of the message.

For instance, in SignatureSecp256k1Serializer.java, the `parse` function reads an integer `vl`, and then reads off `vl` bytes, see the highlighted lines below.

```
    public SignatureSecp256k1 parse(Reader reader) {
        var vl = reader.getInt();
        var v = reader.getBytes(vl);
        var r = reader.getBytes(Secp256k1.SIGNATURE_RS_SIZE);
        var s = reader.getBytes(Secp256k1.SIGNATURE_RS_SIZE);
        return new SignatureSecp256k1(v, r, s);
    }
```

Note that no validation is performed, so a negative or oversized `vl` may trigger an exception during parsing. However, note that various lengths are validated in the `SignatureSecp256k1` constructor, which calls the `checkSignatureDataSizes()` function excerpted below, although this check will never be reached in case of an exception during parsing.

```
    private static boolean checkSignatureDataSizes(byte[] v, byte[] r, byte[] s) {
        return (v.length > 0 && v.length <= Secp256k1.SIGNATURE_V_MAXSIZE) &&
                (r.length == Secp256k1.SIGNATURE_RS_SIZE && s.length ==
                ↳ Secp256k1.SIGNATURE_RS_SIZE);
    }
```

A similar lack of length validation may be observed in the SidechainCoreTransactionSerializer and EthereumTransactionSerializer classes.

Finally, note that malicious inputs may (in principle) cause enormous memory allocations before being caught. In extreme cases, this could lead to Denial of Service (DoS) attacks.

Recommendation

Consider adding additional validation checks during the parsing of input data, including checking no trailing data is present, and additional semantic validation checks specific to each class.

Additionally, document the validation process for all paths parsing untrusted input data, and confirm that all desired validation is performed at an appropriate stage.

Location

- SidechainTransactionApiRoute.scala
- OpenStakeTransactionSerializer.java
- SidechainCoreTransaction.java
- SignatureSecp256k1Serializer.java
- SignatureSecp256k1.java
- SidechainCoreTransactionSerializer
- EthereumTransactionSerializer

Retest Results

2023-03-20 – Partially Fixed

In pull request 765, the Horizen Labs team addressed the first issue outlined in this finding (i.e., *Truncated Trailing Data*) by implementing a `CheckedCompanion` trait which throws an exception in case spurious bytes are present in the stream reader after parsing.

The other two issues outlined in this finding (namely *Missing or Inconsistent Validation* and *Missing Length Validation*) were not addressed. The Horizen Labs team indicated that they preferred to keep the parsing and validation functionalities separate.

As a result, the status of this finding has been updated to *Risk Accepted*.

Client Response

We prefer keeping parsing and validation as separate phases because in the correct path, which is expected to be the vast majority of cases, it leads to a cleaner code flow and separation of concerns. In case of errors, deemed as unlikely occurrences, we are willing to pay this choice with a slightly more expensive error catching.

Low

Potentially Incorrect Nonce Results in Failed Transactions

Overall Risk	Low	**Finding ID**	NCC-E005513-DTH
Impact	Low	**Component**	sdk/scala/account
Exploitability	Medium	**Category**	Other
		Status	Fixed

Impact

Failure to use the correct nonce in an Ethereum transaction will result in a failed transaction and the sender will be charged for the fee.

Description

The following code snippet from sdk/account/api/http/AccountTransactionApiRoute.scala is used to handle the RPC to create an EIP1559[9] transaction[10] from the request's body:

```
/**
  * Create and sign a core transaction, specifying regular outputs and fee. Search for and spend proper amount of regular coins. Then validate and send the transaction.
  * Return the new transaction as a hex string if format = false, otherwise its JSON representation.
  */
def createEIP1559Transaction: Route = (post & path("createEIP1559Transaction")) {
  withAuth {
    entity(as[ReqEIP1559Transaction]) { body =>
      // lock the view and try to create CoreTransaction
      applyOnNodeView { sidechainNodeView =>
        val secret = getFittingSecret(sidechainNodeView, body.from, body.value)

        val nonce = body.nonce.getOrElse(sidechainNodeView.getNodeState.getNonce(secret.get.publicImage.address))

        var signedTx: EthereumTransaction = new EthereumTransaction(
          params.chainId,
          body.to.orNull,
          nonce,
          body.gasLimit,
          body.maxPriorityFeePerGas,
          body.maxFeePerGas,
          body.value,
          body.data,
          if (body.signature_v.isDefined)
            new SignatureData(
              body.signature_v.get,
              body.signature_r.get,
              body.signature_s.get)
          else
            null
        )
        if (!signedTx.isSigned) {
```

9. https://eips.ethereum.org/EIPS/eip-1559
10. This RPC is the equivalent of eth_sendtransaction.

```
216         val txCost = signedTx.getValue.add(signedTx.getGasPrice.multiply(signedTx.getGasLimit))
217
218         val secret =
219           getFittingSecret(sidechainNodeView, body.from, txCost)
220         secret match {
221           case Some(secret) =>
222             signedTx = signTransactionWithSecret(secret, signedTx)
223             validateAndSendTransaction(signedTx)
224           case None =>
225             ApiResponseUtil.toResponse(ErrorInsufficientBalance("ErrorInsufficientBalance", JOptional.empty()))
226         }
227       }
228       else
229         validateAndSendTransaction(signedTx)
230     }
231   }
232 }
233 }
```

The *first* call to the `getFittingSecret()` function, on line 194, finds the first account (`Account1`) in the node's wallet that has at least `body.value` in its balance (**assuming `body.from` is empty**). Then the `nonce` for this account is retrieved from the node view, and the Ethereum transaction is constructed from the request's body and this nonce. If the body of the request does not contain a signature, the conditional statement on line 215 will attempt to sign the transaction. However, since it recalculates the transaction cost (`txCost`) the *second* call to `getFittingSecret()`, on line 219, can result in a different account (`Account2`).

Looking further at the `getFittingSecret()`'s implementation, depending on whether `ReqEIP1559Transaction`'s `body.from` parameter is an empty string or not[11], two scenarios are possible:

```
86  def getFittingSecret(nodeView: AccountNodeView, fromAddress: Option[String], txValueInWei: BigInteger)
87  : Option[PrivateKeySecp256k1] = {
88
89    val wallet = nodeView.getNodeWallet
90    val allAccounts = wallet.secretsOfType(classOf[PrivateKeySecp256k1])
91    val secret = allAccounts.find(
92      a => (fromAddress.isEmpty ||
93        BytesUtils.toHexString(a.asInstanceOf[PrivateKeySecp256k1].publicImage
94          .address) == fromAddress.get) &&
95        nodeView.getNodeState.getBalance(a.asInstanceOf[PrivateKeySecp256k1].publicImage.address).compareTo(txValueInWei) >= 0 // TODO account for gas
96    )
97
98    if (secret.nonEmpty) Option.apply(secret.get.asInstanceOf[PrivateKeySecp256k1])
99    else Option.empty[PrivateKeySecp256k1]
100 }
```

1. If the `from` address is not empty the logic statement on lines 93-94 will force `Account1` and `Account2` to be equal.

11. ReqEIP1559Transaction's input validation logic requires `from` to be empty or a 40 character address.

2. If the `from` address is empty there is a non-zero probability that different values of `txValueInWei` between the two calls will result in `Account1` and `Account2` not being the same.

Since `Account1` and `Account2` may have different nonces, the latter case will lead to a failed transaction, and the sender will be charged for the transaction fee.

Recommendation

Adapt the logic to use the signing account's nonce when `body.signature_v.isDefined` is false.

Location

AccountTransactionApiRoute.scala

Retest Results

2023-01-25 – Fixed

Before the fix, the `createEIP1559Transaction` function fetched the signing secret from the node's view *twice*: once to determine and set the `nonce` field in the transaction that's being created and the second time to be able to sign the transaction. When processing unsigned transaction content, `createEIP1559Transaction` made sure that the signing account had sufficient funds to cover the maximum transaction cost.

The fix removes the second call to `getFittingSecret`; the secret that was fetched the first time is used both to fetch the nonce and to sign the transaction if it is not already signed. There no longer exists potential for a nonce/signature mismatch. It is ensured that the fetched account has sufficient balance to cover the transaction cost.

This fix was implemented in pull request 697 (see also commit 7aba5628). This finding is considered *fixed* as a result.

Low

Unprotected Secret Key Storage

Overall Risk	Low	**Finding ID**	NCC-E005513-YBB
Impact	High	**Component**	sdk/scala/account
Exploitability	Low	**Category**	Configuration
		Status	Risk Accepted

Impact

An attacker obtaining access to a node's storage would be able to access all of the node's cryptographic key material.

Description

The Sidechains SDK keeps track of a number of cryptographic keys for different purposes, such as for signature generation and verification, or for Verifiable Random Functions (VRFs).

The different private keys (namely, Ed25519, VRF, Schnorr, and Secp256K1) are currently maintained using the `SidechainSecretsCompanion` class, which stores and retrieves these secrets from storage.

However, this sensitive key material is stored on disk, as plaintext, from the same general location as the history, the state and the consensus data, as can be seen in AccountSidechainApp:

```
val dataDirAbsolutePath: String = sidechainSettings.sparkzSettings.dataDir.getAbsolutePath
val secretStore = new File(dataDirAbsolutePath + "/secret")
val metaStateStore = new File(dataDirAbsolutePath + "/state")
val historyStore = new File(dataDirAbsolutePath + "/history")
val consensusStore = new File(dataDirAbsolutePath + "/consensusData")

// Init all storages
protected val sidechainSecretStorage = new SidechainSecretStorage(
  registerStorage(new VersionedLevelDbStorageAdapter(secretStore)),
  sidechainSecretsCompanion)
```

The keys are never encrypted before being stored. In a scenario where an attacker obtains access to a participating node's storage (assuming a partial read-only compromise of the node, e.g., a stolen backup), these keys are left unprotected.

Recommendation

Consider encrypting the keys before storing them, at the very least by deriving an encryption key from a user password, which the user would then need to supply when starting a node. Use of hardware-backed security modules could also be implemented to handle key material.

Additionally, consider restricting the file permissions such that only allowed users may read and write to the file containing cryptographic key material. This could also be implemented for the other files storing blockchain-related data.

Location

AccountSidechainApp.scala

Retest Results

2023-03-14 – Not Fixed

The Horizen Labs team indicated that keys should never be stored in the SDK Node storage, but in dedicated third-party wallets and that the risk of storing the keys inappropriately was accepted.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

We assume that keys are never stored in the SDK Node storages for the Mainnet chains but instead are operated by third-party wallets like Metamask; secure enclave or cold wallets. Moreover, the Node that keeps any keys must be isolated from the network (except P2P messaging) and from other software. We accept all the risks of holding the keys in the unprotected way. Anyway, we keep in mind for the future possible improvements here according to the business and security needs. No code changes at the moment.

Low

Outdated Dependencies

Overall Risk	Low	**Finding ID**	NCC-E005513-RKK
Impact	Low	**Component**	SDK
Exploitability	Low	**Category**	Patching
		Status	Partially Fixed

Impact

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

Description

Using outdated dependencies with discovered vulnerabilities is one of the most common and serious routes of application exploitation. Many of the most severe breaches have relied upon exploiting known vulnerabilities in dependencies[12].

The OWASP Dependency-Check package can be used to check for vulnerable dependencies in Java projects, using the command `mvn org.owasp:dependency-check-maven:check`. When run on the repository, it highlights the following vulnerable dependencies for the repository:

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count
akka-http-circe_2.12-1.39.2.jar	cpe:2.3:a:akka:akka:1.39.2:*:*:*:*:*:*:*	pkg:maven/de.heikoseeberger/akka-http-circe_2.12@1.39.2	HIGH	1
jackson-databind-2.13.3.jar	cpe:2.3:a:fasterxml:jackson-databind:2.13.3:*:*:*:*:*:*:* cpe:2.3:a:fasterxml:jackson-modules-java8:2.13.3:*:*:*:*:*:*:*	pkg:maven/com.fasterxml.jackson.core/jackson-databind@2.13.3	HIGH	2
kotlin-stdlib-1.4.10.jar	cpe:2.3:a:jetbrains:kotlin:1.4.10:*:*:*:*:*:*:*	pkg:maven/org.jetbrains.kotlin/kotlin-stdlib@1.4.10	MEDIUM	2
kotlin-stdlib-common-1.4.0.jar	cpe:2.3:a:jetbrains:kotlin:1.4.0:*:*:*:*:*:*:*	pkg:maven/org.jetbrains.kotlin/kotlin-stdlib-common@1.4.0	HIGH	3
okhttp-4.9.0.jar	cpe:2.3:a:squareup:okhttp:4.9.0:*:*:*:*:*:*:* cpe:2.3:a:squareup:okhttp3:4.9.0:*:*:*:*:*:*:*	pkg:maven/com.squareup.okhttp3/okhttp@4.9.0	HIGH	1

Figure 3: Vulnerable Dependencies for Sidechains-SDK

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count
jackson-annotations-2.9.0.jar	cpe:2.3:a:fasterxml:jackson-modules-java8:2.9.0:*:*:*:*:*:*:*	pkg:maven/com.fasterxml.jackson.core/jackson-annotations@2.9.0	MEDIUM	1
jackson-databind-2.9.9.jar	cpe:2.3:a:fasterxml:jackson-databind:2.9.9:*:*:*:*:*:*:* cpe:2.3:a:fasterxml:jackson-modules-java8:2.9.9:*:*:*:*:*:*:*	pkg:maven/com.fasterxml.jackson.core/jackson-databind@2.9.9	CRITICAL	51
log4j-core-2.17.0.jar	cpe:2.3:a:apache:log4j:2.17.0:*:*:*:*:*:*:*	pkg:maven/org.apache.logging.log4j/log4j-core@2.17.0	MEDIUM	1

Figure 4: Vulnerable Dependencies for EVM

In particular, note that the `jackson-databind` package is used in both EVM and Sidechains-SDK, and has a number of reported vulnerabilities, especially for the older version used in EVM. The other highlighted vulnerabilities are from higher-order dependencies, but should still be monitored so that they can be updated as soon as a patch is available.

Also note that the function `objectMapper.disable(MapperFeature.DEFAULT_VIEW_INCLUSION)` from the package `jackson-databind` called on line 54 of ApplicationJsonSerializer.java is deprecated[13]. It has been replaced with the function `JsonMapper.builder().disable(...)`.

12. https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug/
13. https://javadoc.io/doc/com.fasterxml.jackson.core/jackson-databind/latest/index.html

Additionally, the Versions Maven Plugin[14] can be used to check for outdated dependencies, using the command `mvn versions:display-dependency-updates`. While it is common for a few packages to be slightly out of date, a subset of dependencies are concerning due to a lagging major version number. A change in the major version number typically signifies breaking changes and may increase a developer's reluctance to upgrade. When `mvn versions:display-dependency-updates` is run on the repository, it highlights the following packages that are out of date by a major version:

```
[INFO] --------------------------< com.horizen:evm >---------------------------
[INFO] Building evm 0.5.0                                                  [3/8]
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- versions-maven-plugin:2.13.0:display-dependency-updates (default-cli) @ evm ---
[INFO] The following dependencies in Dependencies have newer versions:
[INFO]   ...
[INFO]   org.web3j:utils ....................................... 4.9.2 -> 5.0.0
[INFO]
[INFO]
[INFO] ---------------------< io.horizen:sidechains-sdk >----------------------
[INFO] Building io.horizen:sidechains-sdk 0.5.0                            [4/8]
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- versions-maven-plugin:2.13.0:display-dependency-updates (default-cli) @ sidechains-
↳ sdk ---
[INFO] The following dependencies in Dependencies have newer versions:
[INFO]   ...
[INFO]   org.glassfish.tyrus:tyrus-container-grizzly-server ..... 1.18 -> 2.1.1
[INFO]   org.glassfish.tyrus:tyrus-server ....................... 1.18 -> 2.1.1
[INFO]   org.glassfish.tyrus.bundles:tyrus-standalone-client .... 1.18 -> 2.1.1
[INFO]   ...
[INFO]   org.scala-lang:scala-library ...................... 2.12.12 -> 2.13.10
[INFO]   ...
[INFO]   org.web3j:core ........................................ 4.9.2 -> 5.0.0
```

It was noted by the Horizen Labs team that version `5.0.0` of the `web3j` package is an old version, and should not be used. The latest version for both `web3j:core` and `web3j:utils` is `4.9.5`.

Finally, note that the scala version used by the project is out of date. Currently, the latest Scala 2 version is 2.13.10[15] and there is also a version 3 of the language, currently at 3.2.0[16], which is available as `scala3-library` through Maven.

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, deprecated or vulnerable versions.

Location

Sidechains-SDK/pom.xml

14. https://www.mojohaus.org/versions-maven-plugin/
15. https://www.scala-lang.org/download/2.13.10.html
16. https://www.scala-lang.org/download/3.2.0.html

Retest Results

2023-03-21 – Partially Fixed

As of pull request 704, the Horizen Labs team updated a number of vulnerable dependencies, mostly in EVM. However, one vulnerability still exists in that repository, see the figure below.

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count
guava-31.1-jre.jar	cpe:2.3:a:google:guava:31.1:*:*:*:*:*:*:*	pkg:maven/com.google.guava/guava@31.1-jre	MEDIUM	1

Figure 5: Vulnerable Dependencies for EVM (as of pull request 704)

Similarly, checking vulnerabilities for the Sidechains-SDK shows that the vulnerable dependencies in that repository have not been addressed, and that a few new ones were introduced, see below.

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count
akka-http-circe_2.12-1.39.2.jar	cpe:2.3:a:akka:akka:1.39.2:*:*:*:*:*:*:*	pkg:maven/de.heikoseeberger/akka-http-circe_2.12@1.39.2	HIGH	1
commons-net-3.8.0.jar	cpe:2.3:a:apache:commons_net:3.8.0:*:*:*:*:*:*:*	pkg:maven/commons-net/commons-net@3.8.0	MEDIUM	1
evm-0.6.0-SNAPSHOT.jar	cpe:2.3:a:evm_project:evm:0.6.0:snapshot:*:*:*:*:*:*	pkg:maven/com.horizen/evm@0.6.0-SNAPSHOT	CRITICAL	3
guava-31.1-jre.jar	cpe:2.3:a:google:guava:31.1:*:*:*:*:*:*:*	pkg:maven/com.google.guava/guava@31.1-jre	MEDIUM	1
jackson-databind-2.13.3.jar	cpe:2.3:a:fasterxml:jackson-databind:2.13.3:*:*:*:*:*:*:* cpe:2.3:a:fasterxml:jackson-modules-java8:2.13.3:*:*:*:*:*:*:*	pkg:maven/com.fasterxml.jackson.core/jackson-databind@2.13.3	HIGH	2
kotlin-stdlib-1.4.10.jar	cpe:2.3:a:jetbrains:kotlin:1.4.10:*:*:*:*:*:*:*	pkg:maven/org.jetbrains.kotlin/kotlin-stdlib@1.4.10	MEDIUM	2
kotlin-stdlib-common-1.4.0.jar	cpe:2.3:a:jetbrains:kotlin:1.4.0:*:*:*:*:*:*:*	pkg:maven/org.jetbrains.kotlin/kotlin-stdlib-common@1.4.0	HIGH	3
okhttp-4.9.0.jar	cpe:2.3:a:squareup:okhttp:4.9.0:*:*:*:*:*:*:* cpe:2.3:a:squareup:okhttp3:4.9.0:*:*:*:*:*:*:*	pkg:maven/com.squareup.okhttp3/okhttp@4.9.0	HIGH	1

Figure 6: Vulnerable Dependencies for Sidechains-SDK (as of pull request 704)

The Scala version of the project was also not updated.

As a result, the status of this finding has been updated to *Partially Fixed*.

Low

Infinite Loop When Adding Handles

Overall Risk	Low	**Finding ID**	NCC-E005513-DRU
Impact	High	**Component**	libevm
Exploitability	Low	**Category**	Denial of Service
		Status	Fixed

Impact

An infinite loop can be triggered when adding objects to the `Handles` structure, potentially leading to a Denial of Service (DoS) against users of `libevm`.

Description

In `libevm`, the file *libevm/lib/handles.go* defines an object store, a data structure used to store arbitrary objects, and a few related functions used to add, remove, or fetch particular objects associated with a handle. The data store is represented by the `Handles` type, a structure comprising a Golang `map` to store objects, as well as an integer keeping track of the *current* (and most likely) index where a new object shall be added; this can be seen in the type definition excerpted below.

```
type Handles[T comparable] struct {
  used    map[int]T
  current int
}
```

In case a `Handles` object fills up to capacity, adding a new object will cause the execution to hang indefinitely. Adding new objects is performed by the `Add()` function, provided below for reference. Upon adding a new object, the function tries to find the next unused handle by increasing the current index (see highlighted line 31 below). It then inserts the object at that index, provided it is currently empty. In case an object was already present at that index, the function will loop (see the `for`-statement on line 24), check whether the current index is equal to the maximum, predefined capacity of the `Handles` object (namely, `math.MaxInt32`) and set the index to 0 if that's the case, before increasing it again and checking if there is a slot available in the `used` map. As a result, if the `used` map is full, the `Add()` function will loop indefinitely when trying to add new objects.

```
23 func (h *Handles[T]) Add(obj T) int {
24   for {
25     // wrap around
26     if h.current == math.MaxInt32 {
27       h.current = 0
28     }
29     // find the next unused handle:
30     // this will never give a handle of 0, which is on purpose - we might consider a handle
       ↳ of 0 as invalid
31     h.current++
32     if _, exists := h.used[h.current]; !exists {
33       h.used[h.current] = obj
34       return h.current
35     }
36   }
37 }
```

The risk rating of this finding is set to *Low*, since the current usages of `Handles` objects are never expected to come close to filling up the `used` map to capacity.

Recommendation

Consider modifying the `Add()` function such that it detects when all indices in the range have been tried. The function should additionally be modified to return a tuple of `(error, int)` to indicate whether the object was successfully inserted or not.

Location

libevm/lib/handles.go

Retest Results

2023-03-16 – Fixed

In commit 16b7d241, the Horizen Labs team added the following conditional statement ensuring that there is at least one empty slot if the execution proceeds to the `for`-loop.

```
// pedantic safety check for overflow: this is highly unlikely because there are 2**31-1 available handles,
// just the memory consumption alone will likely cause problems before this happens
if len(h.used) == math.MaxInt32 {
  panic(fmt.Sprintf("out of handles, unable to add %T", obj))
}
```

Note that the function now panics if it reaches capacity (instead of returning an error, as recommended). But since this event is unlikely to occur, this finding is considered *fixed*.

Info

Duplicate Error Codes

Overall Risk	Informational	**Finding ID**	NCC-E005513-MUE
Impact	Undetermined	**Component**	sdk/scala
Exploitability	Low	**Category**	Error Reporting
		Status	Risk Accepted

Impact

The inability to distinguish between different errors with sufficient granularity may pose a usability problem and may prevent correct automated handling of these errors by software clients.

Description

The Sidechains SDK exposes some of its functionalities with an API to be accessed over HTTP. Application-specific errors are returned to callers upon failure; these errors are conveniently defined within the relevant Scala objects, in specialized case classes and with distinct error codes.

For example, the *AccountBlockApiRoute.scala* source file defines the following `SidechainBlockErrorResponse`, which may be a case of `ErrorNoBlockFound`. That particular error has a numeric error code of “0401”, as can be seen in the code excerpt below, taken from AccountBlockApiRoute.scala:

```
object SidechainBlockErrorResponse {
  case class ErrorNoBlockFound(description: String, exception: JOptional[Throwable]) extends ErrorResponse {
    override val code: String = "0401"
  }
```

The NCC Group team noted that the integer code associated with the `ErrorNoBlockFound` above is duplicated; indeed, the case class `ErrorInvalidHost` of the `SidechainNodeErrorResponse` object uses the same code, as can be seen in the code excerpt below, taken from SidechainNodeApiRoute.scala:

```
object SidechainNodeErrorResponse {

  case class ErrorInvalidHost(description: String, exception: JOptional[Throwable]) extends ErrorResponse {
    override val code: String = "0401"
  }
```

Additionally, the error code “0302” is also duplicated. It is first defined in the `ErrorCouldNotGetBalance` case class of the object `AccountWalletErrorResponse` in AccountWalletApiRoute.scala, see below:

```
object AccountWalletErrorResponse {
  case class ErrorCouldNotGetBalance(description: String, exception: JOptional[Throwable]) extends ErrorResponse {
    override val code: String = "0302"
  }
```

And it is also defined in the `ErrorSecretAlreadyPresent` case class of the object `SidechainWalletErrorResponse` in SidechainWalletApiRoute.scala, see below:

```
object SidechainWalletErrorResponse {

  case class ErrorSecretAlreadyPresent(description: String, exception:
  ↳ JOptional[Throwable]) extends ErrorResponse {
    override val code: String = "0302"
  }
```

Recommendation

Update the numeric error codes of `ErrorNoBlockFound` and of `ErrorCouldNotGetBalance` such as to avoid any collision. Additionally, consider defining a new file conveniently recording all of the error codes as private static variables with adequate names, and initializing the corresponding error codes with these externally-defined error codes.

Location

- AccountBlockApiRoute.scala
- SidechainNodeApiRoute.scala
- AccountWalletApiRoute.scala
- SidechainWalletApiRoute.scala

Retest Results

2023-03-15 – Not Fixed

The Horizen Labs team indicated that this finding does not have a significant impact for the moment and that it will be fixed at a later date.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

We are going to proceed in the way proposed by NCC. This issue is considered to not have a significant impact, so was put to the backlog and will be addressed later. No code changes at the moment.

Info

Base Fee Computation Potentially Inappropriate with Current Slot Duration

Overall Risk	Informational	**Finding ID**	NCC-E005513-YRC
Impact	Undetermined	**Component**	sdk/scala
Exploitability	Low	**Category**	Configuration
		Status	Risk Accepted

Impact

The Base Fee computation follows the same incentive model as Ethereum, which may not be appropriate with the block rate of the Sidechains. This incentive model may deter some users to participate in the system and add latency to operations.

Description

As also described in finding "Incorrect Lower Bound in Gas Fee Computation", the Ethereum Improvement Proposal (EIP) 1559 introduced an updated transaction pricing mechanism, including a fixed-per-block network fee, the *Base Fee*, which is always burned. This fee gets updated dynamically based on current network congestion and can be concisely summarized as follows.

> The algorithm results in the base fee per gas increasing when blocks are above the gas target, and decreasing when blocks are below the gas target.

Concretely, the computation of the base fee for the new block depends on how full the parent block was; the updated base fee may increase or decrease up to a maximum of 12.5%.

However, the fee computation defined in EIP-1559 is inherently tied to the economic model of Ethereum. Specifically, this maximum increase (resp. decrease) value has been chosen in such a way that the base fee may *double* in value only if 6 consecutive blocks are full, since $1.125^{6} = 2.027$. With Ethereum's current block rate of approximately 12 seconds[17], this means that the base fee value could theoretically double in about 72 seconds.

Consequently, platforms dealing with Ethereum introduced EIP-1559-compliant gas fee estimators leveraging simple heuristic in order to calculate the recommended *Max Fee* such that a given transaction is likely to make it on the blockchain. Best practices seem to be to calculate the max fee of a transaction with the following formula, as described in a blog post by blocknative[18].

> Max Fee = (2 * Base Fee) + Max Priority Fee

While this finding does not highlight a specific vulnerability, it should be seen as an incentive to carefully review whether the numbers and computations presented above and based on Ethereum's specific incentive model apply to the Horizen Sidechains.

17. https://ethereum.org/en/developers/docs/blocks/#block-time
18. https://www.blocknative.com/blog/eip-1559-fees

For example, the current slot number defined in MainNetParams.scala indicate that a block will be produced every 120 seconds.

```
override val sidechainGenesisBlockTimestamp: Block.Timestamp = 720 * 120,
override val consensusSecondsInSlot: Int = 120,
override val consensusSlotsInEpoch: Int = 720,
```

Given that the same maximum base fee percentage increase as Ethereum is used, see FeeUtils.scala, the base fee will double only after 12 minutes if the last consecutive 6 blocks are full. This may not be appropriate with the stated goals of the Sidechains SDK, which may deter some users from participating in the system and add latency to the different operations.

Recommendation

Carefully review whether the per-block increase percentage of the base fee is appropriate with the current block rate of the Sidechains. Consider giving the ability to Sidechains-SDK users to define their own incentive models.

Location

- MainNetParams.scala
- FeeUtils.scala

Retest Results

2023-03-15 – Not Fixed

The Horizen Labs team indicated that the slot time was now configurable at the application level and that the fee model was going to be improved to better fit users' needs in future versions of the SDK.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

In the recent version of the SDK we made the slot time configurable on application level and kept it equal to 12 seconds, that more or less equals the block rate in eth (12.5s). Note: slot rate is not equal to the block rate and in the real network with many independent Forgers actual block rate may be from 15 to 18 seconds.

At the moment we consider Base fee computation strategy acceptable for the Account based sidechains as an intermediate step. In the next versions we are going to improve the fee model significantly to let it fit the forgers/verifiers/users needs.

Info

Incorrect API Usage

Overall Risk	Informational	**Finding ID**	NCC-E005513-TL9
Impact	None	**Component**	sdk/java
Exploitability	None	**Category**	Cryptography
		Status	Risk Accepted

Impact

High-level APIs are usually stable, but underlying libraries and implementation details may change unexpectedly. Calling lower-level intermediate functions instead of the exposed API differs from the usage intended by the developer and could become a source of bugs.

Description

In the file *Ed25519.java*, a number of constants, as well as the `sign` and `verify` algorithms, are implemented using the methods from the `org.bouncycastle.math.ec.RFC8032`:

```java
public static int privateKeyLength() {
    return org.bouncycastle.math.ec.rfc8032.Ed25519.SECRET_KEY_SIZE;
}
public static int publicKeyLength() {
    return org.bouncycastle.math.ec.rfc8032.Ed25519.PUBLIC_KEY_SIZE;
}
public static int signatureLength() {
    return org.bouncycastle.math.ec.rfc8032.Ed25519.SIGNATURE_SIZE;
}
```

```java
public static boolean verify(byte[] signature, byte[] message, byte[] publicKey) {
    try {
        return org.bouncycastle.math.ec.rfc8032.Ed25519.verify(signature, 0, publicKey, 0,
        message, 0, message.length);
    } catch (Exception e) {
        return false;
    }
}

public static byte[] sign(byte[] privateKey, byte[] message, byte[] publicKey) {
    byte[] signature = new byte[64];
    org.bouncycastle.math.ec.rfc8032.Ed25519.sign(privateKey, 0, publicKey, 0, (byte[])
    null, message, 0, message.length, signature, 0);
    return signature;
}
```

The methods defined in the `bouncycastle` package are intended to be called using the `Signer` and `Ed25519PrivateKeyParameters` classes from the `org.bouncycastle.crypto` package. Example usage can be found in the `org.bouncycastle.crypto.test` method for Ed25519[19][20]. Instead, the SDK relies on lower-level functions that are not intended to be called directly.

19. https://www.bouncycastle.org/documentation.html
20. https://github.com/bcgit/bc-java/blob/master/core/src/test/java/org/bouncycastle/crypto/test/Ed25519Test.java#L48

Specifically, the SDK uses the `org.bouncycastle.math.ec.RFC8032` package in the two code excerpts above, which provides a low-level implementation of the Ed25519 instantiation of the Edwards-Curve Digital Signature Algorithm specified in RFC 8032[21]. An interface to the signing and verifying functions is provided through the `org.bouncycastle.crypto` methods `signer.generateSignature` and `signer.verifySignature`, which should be preferred. Additionally, the `Ed25519.SECRET_KEY_SIZE`, `Ed25519.PUBLIC_KEY_SIZE` and `Ed25519.SIGNATURE_SIZE` constants are available from the `org.bouncycastle.crypto` package as `Ed25519PrivateKeyParameters.KEY_SIZE`, `Ed25519PublicKeyParameters.PUBLIC_KEY_SIZE` and `Ed25519PrivateKeyParameters.SIGNATURE_SIZE` respectively.

High-level APIs are usually pretty stable, but underlying libraries and implementation details may change unexpectedly. This may lead to missing security checks or optimizations, or simply return incorrect results in the future.

Recommendation

Remove all direct dependencies to the `org.bouncycastle.math.ec` package. In particular, modify the `sign` and `verify` methods to use the `org.bouncycastle.crypto` `signer.generateSignature` and `signer.verifySignature` methods, and replace the constants from `org.bouncycastle.math.ec` with the corresponding constants from the `org.bouncycastle.crypto` package.

Location

Ed25519.java

Retest Results

2023-03-15 – Not Fixed

The Horizen Labs team indicated that a ticket had been opened and that this finding will be addressed at a later date.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

We are going to address the ticket in the way suggested by NCC. At the moment the task was put to the backlog and will be done later. So far, no code changes.

21. https://www.bouncycastle.org/docs/docs1.8on/

Info

Overly Permissive Regular Expression

Overall Risk	Informational	**Finding ID**	NCC-E005513-QV9
Impact	Undetermined	**Component**	sdk/scala
Exploitability	Medium	**Category**	Data Validation
		Status	Risk Accepted

Impact

A regular expression that accepts values for IP addresses and port numbers that are not supported by the application may result in unexpected issues for legitimate users.

Description

The Sidechains SDK exposes a number of HTTP API routes, such as `connect` and `disconnect` in SidechainNodeApiRoute.scala. These two functions make use of the following regular expression to ensure the address and port number provided in the body of the request are roughly following the expected structure `<IP address>:<port>`.

```
private val addressAndPortRegexp = "([\\w\\.]+):(\\d{1,5})".r
```

That regex is overly permissive and does not very accurately define the proper format of the expected address and port. Specifically, this regex is very generic and will match one or more repetitions of a word character (`\w` representing the set `[a-zA-Z_0-9]`) or periods, followed by a number composed of 1 to 5 digits[22].

As such, this regular expression does not prevent malformed IP addresses from being entered into the system, and it also allows any port number of up to 5 digits, including numbers larger than 65535.

The regex will also prevent usage of IPv6. Indeed, IPv6 addresses being separated by colons, the regex will try to find a match where an IPv6 *quartet* starting with at least one digit is preceded by another quartet, and match the first as the address, and the second as the port number, erroneously parsing that IPv6 address.

Additionally, it is unclear why the parsing of address and port number are performed as they are in the `connect()` and `disconnect()` functions. Namely, the address and port are first *combined* (see line 86 highlighted below), matched by the regular expression (see line 87 highlighted below), before being finally separated again and used to initialized an `InetAddress` object, and assigned to the variable `port`, respectively.

```
83  def connect: Route = (post & path("connect")) {
84    entity(as[ReqConnect]) {
85      body =>
86        val address = body.host + ":" + body.port
87        val maybeAddress = addressAndPortRegexp.findFirstMatchIn(address)
88        maybeAddress match {
89          case None =>
90            ApiResponseUtil.toResponse(ErrorInvalidHost("Incorrect host and/or port.",
               ↳ JOptional.empty()))
91          case Some(addressAndPort) =>
92            Try(InetAddress.getByName(addressAndPort.group(1))) match {
93              case Failure(exception) => SidechainApiError(exception)
```

22. The Oracle Documentation describes the syntax of the *java/util/regex/Pattern* construction used here.

```
            case Success(host) =>
              val port = addressAndPort.group(2).toInt
              networkController ! ConnectTo(PeerInfo.fromAddress(new InetSocketAddress(host, port)))
              ApiResponseUtil.toResponse(RespConnect(host + ":" + port))
          }
        }
    }
  }
```

Note that the impact of this oversight is limited, since the `InetAddress` will return an error in case the address is invalid.

Recommendation

Consider updating the `addressAndPortRegexp` expression to be less permissive, such that it only matches URLs that are currently supported by the application. Additionally, consider adding logic to validate the port number, and review whether the current execution logic (*combining-matching-splitting*) is superfluous, in which case a simplification would be recommended.

Location

SidechainNodeApiRoute.scala

Retest Results

2023-03-15 – Not Fixed

The Horizen Labs team indicated that a ticket had been opened and that this finding will be addressed at a later date.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

We are going to address the ticket in the way suggested by NCC. At the moment the task was put to the backlog and will be done later. Will impact SDK and Sparkz. So far, no code changes.

Info

Unsafe Handling of Secret Key Material

Overall Risk	Informational	**Finding ID**	NCC-E005513-X4V
Impact	High	**Component**	sdk/java/account
Exploitability	Undetermined	**Category**	Cryptography
		Status	Risk Accepted

Impact

An attacker may obtain part, or all, of the private key material used to sign transactions. The API may facilitate unsafe usage of the SDK.

Description

The Java class `PrivateKeySecp256k1` implements a number of methods to handle secret keys. NCC Group identified several issues with these methods.

Method `equals()` perfoms comparison of secret key values:

```
public boolean equals(Object obj) {
    if (obj == null) return false;
    if (!(obj instanceof PrivateKeySecp256k1)) return false;
    if (obj == this) return true;
    var other = (PrivateKeySecp256k1) obj;
    return Arrays.equals(privateKey, other.privateKey);
}
```

The comparison resorts to Java's method `Arrays.equals()`, which is not performed in constant-time, and is therefore amenable to timing side channel attacks.

Java method `hashCode()` returns a hash code value for the object, and in this case, the secret key. The `hashCode()` method was probably introduced in tandem with the `equals()` method, as implementing the latter requires implementing the former. The `hashCode()` method is also typically used in Java to support hash tables to index contents.

```
public int hashCode() {
    return Arrays.hashCode(privateKey);
}
```

The pattern of using a secret value as an indexing key may leak sensitive information, contingent of the SDK use case. For instance, it may leak what secrets were used (access to memory based on a secret index).

Method `toString()` provides a string representation of key material.

```
public String toString() {
        return String.format("PrivateKeySecp256k1{privateKey=%s}", Numeric.toHexString(privateKey));
    }
```

The presence of such a method is likely to facilitate unsecure dissemination of secret key material in logs, or otherwise. The implementation of the method is also unlikely to be constant-time (mapping from bytes to hexadecimal characters).

Recommendation

Consider comparing the public keys instead of the private keys, by computing the public keys from the private keys (or maintaining a reference to the public keys from the private keys). Remove the `equals()` and `hashCode()` methods for the private keys.

Otherwise, ensure that all secrets are compared in constant-time. That is, the comparison operation should not return until after all bytes of the input have been evaluated. Ensure that private keys are not used as indexes to Java structures in this case.

Consider removing the `toString()` method from the implementation, or returning the public key string built from the private key, if outputting the private key is used for troubleshooting purposes. If the private key is meant to be exported, it should probably be encrypted before leaving the system.

Location

PrivateKeySecp256k1.java

Retest Results

2023-03-15 – Not Fixed

The Horizen Labs team indicated that a ticket had been opened and that this finding will be addressed at a later date.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

We are going to address the ticket in the way suggested by NCC. At the moment the task was put to the backlog and will be done later. So far, no code changes.

Info

Missing In-Memory Merkle Tree Height Validation

Overall Risk	Informational	**Finding ID**	NCC-E005513-XA6
Impact	Undetermined	**Component**	sdk/java
Exploitability	None	**Category**	Data Validation
		Status	Risk Accepted

Impact

Failure to validate the height of Merkle Trees may allow attackers (or careless users) to define arbitrarily large trees, potentially resulting in a DoS by memory exhaustion as a result.

Description

The Sidechains SDK defines an `InMemorySparseMerkleTreeWrapper` class, which, as its name suggests, generates a Merkle Tree on the fly and stores it in memory. An object of the `InMemorySparseMerkleTreeWrapper` class is instantiated by calling its constructor with a height parameter. This integer height is then used to initiate an `InMemorySparseMerkleTree` (provided by the underlying Rust *zendoo-sc-cryptolib* library) and also generates a number of leaves corresponding to 2^{height} leaves.

```
public InMemorySparseMerkleTreeWrapper(int height) {
    merkleTree = InMemorySparseMerkleTree.init(height);
    leavesNumber = 1L << height;
    emptyLeaves = TreeRangeSet.create(Arrays.asList(Range.closedOpen(0L, leavesNumber)));
}
```

The NCC Group team noted that no validity check is performed on the `height` parameter provided to the constructor of the `InMemorySparseMerkleTreeWrapper`. Specifically, the height could be negative, zero, or it could be arbitrarily large, leading to large memory usage since the object will create a `RangeSet` containing 2^{height} elements.

The severity of this finding is set as *Informational* since this issue does not seem to be currently exploitable, as `InMemorySparseMerkleTreeWrapper` s are only ever instantiated from a fixed, bounded height. Nevertheless, the target of this review being an SDK, care should be taken so as not to allow users to misuse existing constructions.

Recommendation

Consider adding validation checks of the height parameter, in order to ensure only strictly positive values are accepted, and further ensure that they are upper-bounded by a reasonable maximum height.

Location

InMemorySparseMerkleTreeWrapper.java

Retest Results

2023-03-15 – Not Fixed

The Horizen Labs team indicated that this finding will be addressed at a later date.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

We will add a check for the height parameter 0 <= height <= 22 and throw an exception otherwise. The maximum value was chosen because of the business needs for this Merkle tree in our SDK. At the moment core code controls the creation of the tree and always passes a height value equal to 22 and passed to us from *zendoo-sc-cryptolib*. Activity is scheduled for one of the next releases.

Info

Private Key Handling and Memory Zeroization

Overall Risk	Informational	**Finding ID**	NCC-E005513-FB4
Impact	High	**Component**	sdk/java
Exploitability	Undetermined	**Category**	Data Exposure
		Status	Risk Accepted

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, especially if the protocol was run on external infrastructure. As a result, sensitive data should be cleared from memory once it goes out of scope. In the context of cryptography, the action of erasing sensitive data is usually referred to as zeroization.

The Sidechains SDK does not exhibit particular care for memory zeroization; in no instance was it observed to erase sensitive data. Additionally, private keys are commonly stored as byte arrays within the SDK, and these byte arrays are copied on repeated occasions, thereby increasing the potential for disclosure of sensitive material.

As an example, consider the file PrivateKey25519.java. The class constructor copies the private key byte array passed as a parameter to the class member variable `privateKeyBytes`, as highlighted in the code excerpt below.

```
public PrivateKey25519(byte[] privateKeyBytes, byte[] publicKeyBytes)
{
    if(privateKeyBytes.length != PRIVATE_KEY_LENGTH)
        throw new IllegalArgumentException(String.format("Incorrect private key length, %d expected, %d found", PRIVATE_KEY_LENGTH,
                privateKeyBytes.length));
    if(publicKeyBytes.length != PUBLIC_KEY_LENGTH)
        throw new IllegalArgumentException(String.format("Incorrect pubKey length, %d expected, %d found", PUBLIC_KEY_LENGTH,
                publicKeyBytes.length));

    this.privateKeyBytes = Arrays.copyOf(privateKeyBytes, PRIVATE_KEY_LENGTH);
    this.publicKeyBytes = Arrays.copyOf(publicKeyBytes, PUBLIC_KEY_LENGTH);
}
```

Similarly, the function `privateKey()` of the same file returns a copy of these private key bytes, as can be seen below.

```
public byte[] privateKey() {
    return Arrays.copyOf(privateKeyBytes, PRIVATE_KEY_LENGTH);
}
```

An example usage of this function is in the CswCircuitImplZendoo.java source file, where one can see that the content of the byte array holding the private key material is never zeroized.

```java
@Override
public byte[] privateKey25519ToScalar(PrivateKey25519 pk) {
    byte[] pkBytes = pk.privateKey();

    byte[] hash;
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-512");
        digest.update(pkBytes, 0, pkBytes.length);
        hash = digest.digest();
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(e);  // Cannot happen.
    }

    // Only the lower 32 bytes are used
    byte[] lowerBytes = Arrays.copyOfRange(hash, 0, 32);

    // Pruning:
    // The lowest three bits of the first octet are cleared
    lowerBytes[0] &= 0b11111000;
    // The highest bit of the last octet is cleared, and the second-highest bit of the last octet is set.
    lowerBytes[31] &= 0b01111111;
    lowerBytes[31] |= 0b01000000;

    return lowerBytes;
}
```

It should be noted that memory zeroization in Java is a tricky topic since the Garbage Collector may move objects in memory and cause the creation of multiple hidden copies of the sensitive data in the process address space. As such, this finding was marked as *Informational*.

Recommendation

Consider performing more thorough private key zeroization throughout the SDK codebase. 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 approaches[23] to attempt to avoid compiler optimizations and ensure that a clearing routine is performed reliably.

Location

- PrivateKey25519.java
- CswCircuitImplZendoo.java

Retest Results

2023-03-15 – Not Fixed

The Horizen Labs team indicated that private keys should never be stored in the SDK Node storage, but in dedicated third-party wallets and that the risk of storing the keys inappropriately was accepted.

The status of this finding was updated to *Risk Accepted* as a result.

23. https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17_slides_zhaomo_yang.pdf

Client Response

We accept the possible risks and issues considering that nodes should not deal with the private keys by themself but delegate it to the third party wallets. No code changes.

Info

Unauthenticated Secure Enclave API and DNS Rebinding

Overall Risk	Informational	**Finding ID**	NCC-E005513-NK3
Impact	Undetermined	**Component**	SDK
Exploitability	Undetermined	**Category**	Data Validation
		Status	Risk Accepted

Impact

If the Secure Enclave API functionality is exposed as a REST API service on `localhost` without authentication, DNS rebinding attacks may apply, potentially leaking signing keys.

Description

Applications running on a user's host commonly communicate with each other through REST interfaces exposed on `localhost`; for example, application A may ask another wallet application B to generate blockchain transactions. Sometimes, APIs exposed by the target application are left unauthenticated, as they are not accessible outside the host they're running on.

A DNS rebinding attack[24] occurs when an attacker abuses the Domain Name System to bypass a victim browser's same-origin policy[25] to access restricted resources. The user's browser performs an action on the said service and delivers the result to the attacker[26]. By controlling the DNS server, the browser origin URL's IP address is silently modified and the browser's same origin policy is bypassed. This allows a site to make requests and read responses of the target service.

In the case of the Sidechain application, most of the critical wallet-related endpoints (eg. `/wallet/dumpSecrets`) are authenticated and as such the attack does not apply, assuming a strong password is used. There are some exceptions previously noted in finding "Unauthenticated API Routes May Permit DoS and Eclipse Attacks", but the endpoints do not appear to allow leaking significant secrets and may not be worth a rebinding attack in practice.

This finding notes that the Sidechain source code and tests support an additional service, called the Secure Enclave API (and also referred to as the Remote Keys Manager service). This service exposes APIs such as the `createSignature` function, which implements signing with the Schnorr signature scheme.

When it comes to implementation and usage of this service, inside the Java code, there exists a command line tool which implements the endpoints and takes JSON-formatted command line arguments. There also exists a `SecureEnclaveClient` class, which implements a client that calls the service. Finally, inside the qa/ directory, several files call out to the same service, exposed on `localhost`.

DNS rebinding are a concern in this case, as (1) the endpoints are able to generate signatures and therefore are sensitive and (2) the implementations identified in the source code do not appear to support authentication. This finding is set to informational and it may

24. https://crypto.stanford.edu/dns/
25. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
26. https://blog.ret2.io/2019/08/28/sia-coin-dns-rebinding/

be updated after discussion with the Horizen Team (depending on the relevance of the Secure Enclave API service).

Recommendation

Authenticate the Secure Enclave API if it is enabled. When it comes to DNS rebinding defense-in-depth, consider validating the `Host` header of every HTTP request made to any of the services exposed on `localhost`. Finally, if the JAR command line tool is used for implementing the Secure Enclave API, ensure that command injection is mitigated by only allowing a whitelist of characters inside the parameters passed to the command line.

Location

- SigningToolCommandProcessor.java
- SecureEnclaveApiClient.scala
- Sidechains-SDK/qa/

Retest Results

2023-03-15 – Not Fixed

The Horizen Labs team acknowledged the recommendations provided above and indicated that this finding will be addressed at a later date.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

The Secure Enclave service will not be exposed on localhost but through a private docker network. Even there, DNS rebinding attacks could still be applicable, but the attacker would need system level access already. Adding support to the SDK to use authentication on the Enclave API would be a useful improvement though.

At the moment the task was put to the backlog and will be done later. So far, no code changes.

Info

Outdated Copy of `go-ethereum` Proof Code

Overall Risk	Informational	**Finding ID**	NCC-E005513-U7Q
Impact	Undetermined	**Component**	libevm
Exploitability	Low	**Category**	Patching
		Status	Risk Accepted

Impact

Discrepancies between the current implementation of `GetProof()` and the corresponding code in `go-ethereum` (as well as calls from `libevm` to functions whose signatures have changed in updated versions of `go-ethereum`) may lead to potential consensus breach, and future unforeseen issues when upgrading to newer versions of the `go-ethereum` dependency.

Description

In `libevm`, the file *libevm/lib/geth_internal/proof.go* defines functions and structures to support proof creation. This file is heavily inspired by a similar implementation in `go-ethereum`, as also indicated by a comment on line 11 of that file. (As an aside, note that the link provided in the comment below leads to a page "Not Found" error.)

```
11 // The contents here are copied from an internal GETH package which we cannot import, see:
12 // github.com/ethereum/go-ethereum@v1.10.26/internal/ethapi/api.go:657
```

The NCC Group team noted that the code in the Sidechains SDK's *proof.go* was based on slightly outdated code of `go-ethereum` and that more recent commits had introduced a few improvements to that code. Specifically, the algorithms in *proof.go* follow the implementation and API defined in `go-ethereum` v1.10.26 (which is the latest stable version dated Nov 3, 2022). However, code improvements made since that release have introduced more robust error handling, in at least two locations.

As an initial example, consider the first operation in the function `GetProof()` which is a call to `state.StorageTrie()`, as can be seen below.

```go
// GetProof returns the Merkle-proof for a given account and optionally some storage keys.
func GetProof(state *state.StateDB, address common.Address, storageKeys []string) (*AccountResult, error) {
  storageTrie := state.StorageTrie(address)
  storageHash := types.EmptyRootHash
  codeHash := state.GetCodeHash(address)
  storageProof := make([]StorageResult, len(storageKeys))

  // if we have a storageTrie, (which means the account exists), we can update the storagehash
  if storageTrie != nil {
    storageHash = storageTrie.Hash()
  } else {
```

The `StorageTrie()` function is defined in *go-ethereum@v1.10.26/core/state/statedb.go* and may return `nil`:

```
// StorageTrie returns the storage trie of an account.
// The return value is a copy and is nil for non-existent accounts.
func (s *StateDB) StorageTrie(addr common.Address) Trie {
  stateObject := s.getStateObject(addr)
  if stateObject == nil {
    return nil
  }
```

While a `nil` value assigned to the `storageTrie` variable is correctly handled by the `GetProof()` function (see highlighted line in the first code snippet), a recent commit in `go-ethereum`[27] changed the implementation of `StorageTrie()` to return a tuple with an error, a more idiomatic and robust method of handling potential issues.

```
  storageTrie, err := state.StorageTrie(address)
  if err != nil {
    return nil, err
  }
```

As a second example, consider the process of converting a key from its hexadecimal representation to a hash value, computed by the call to `HexToHash()` in *proof.go*.

```
  // create the proof for the storageKeys
  for i, key := range storageKeys {
    if storageTrie != nil {
      proof, storageError := state.GetStorageProof(address, common.HexToHash(key))
      if storageError != nil {
        return nil, storageError
      }
```

This call leverages the function `HexToHash()` defined in *go-ethereum@v1.10.26/common/types.go* and which explicitly allows lengths to be larger that the hash length, as described in the inline documentation preceding the function.

```
// HexToHash sets byte representation of s to hash.
// If b is larger than len(h), b will be cropped from the left.
func HexToHash(s string) Hash { return BytesToHash(FromHex(s)) }
```

In comparison, consider the corresponding `GetProof()` computations perform in go-ethereum since a commit dated Sep 16, 2022[28].

```
  // create the proof for the storageKeys
  for i, hexKey := range storageKeys {
    key, err := decodeHash(hexKey)
    if err != nil {
      return nil, err
    }
    if storageTrie != nil {
```

27. https://github.com/ethereum/go-ethereum/commit/01808421e20ba9d19c029b64fcda841df77c9aff
28. https://github.com/ethereum/go-ethereum/commit/8ade5e6c144afb7ac389ae28b50cb68a0dbec7b8

```
    proof, storageError := state.GetStorageProof(address, key)
    if storageError != nil {
      return nil, storageError
    }
```

This function calls `decodeHash()`, which explicitly rejects lengths larger than `32` (see the highlighted code below) and additionally returns an error if the string contains invalid hexadecimal characters.

```
// decodeHash parses a hex-encoded 32-byte hash. The input may optionally
// be prefixed by 0x and can have an byte length up to 32.
func decodeHash(s string) (common.Hash, error) {
  if strings.HasPrefix(s, "0x") || strings.HasPrefix(s, "0X") {
    s = s[2:]
  }
  if (len(s) & 1) > 0 {
    s = "0" + s
  }
  b, err := hex.DecodeString(s)
  if err != nil {
    return common.Hash{}, fmt.Errorf("hex string invalid")
  }
  if len(b) > 32 {
    return common.Hash{}, fmt.Errorf("hex string too long, want at most 32 bytes")
  }
  return common.BytesToHash(b), nil
}
```

Recommendation

Update the logic in `GetProof()` to ensure that changes in the `go-ethereum` library (such as by upgrading the version of the dependency used by `libevm`) does not lead to potential consensus issues, for example by handling erroneous conversions from `string` to `hash` differently. Additionally, consider closely following the release cycle of `go-ethereum`, and ensuring that improvements made to that library are correctly applied to `libevm`.

Location

libevm/lib/geth_internal/proof.go

Retest Results

2023-03-15 – Not Fixed

The Horizen Labs team indicated that the current implementation is in line with `go-ethereum` v1.10.26 and that updating parts of the local code copy in isolation would be illogical.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

As of now, we are using go-ethereum v1.10.26 and our implementation of getProof is in line with that. We had to copy this code snippet because it is from an internal package that we can't import. If we decide to update our dependency to a newer version we need to update our code accordingly. It has no sense to update parts of getProof without updating the version of go-ethereum we're using. Additionally, everything here has absolutely zero chance of "consensus breach", this is just a utility used for the RPC function eth_getProof. No code changes at the moment.

6 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.
Medium	

Rating	Description
	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.

7 Non-Security Impacting Observations

This informal section contains notes and observations generated during the project. There are no security issues that are not already reported in the preceding findings, but the following content may be useful for discussion purposes and to improve the overall quality and consistency of the codebase. This section is not intended to be exhaustive.

Methodology Notes

The primary testing methodology involved static code review. The consultant team began with a bottom-up review of the implementation within a line-by-line context. This was subsequently complemented by a top-down effort involving planned use cases, documented functionality, system requirements, and desired assurances in the larger context. The two approaches overlap to achieve robust coverage. The consultants also leveraged the different test suites to try and identify shortcomings, to understand the system in more depth and to validate hypotheses.

General Notes

- The following comment in the PrivateKeySecp256k1Creator.java source file is slightly misleading; the Secp256k1 private key is only ever 32 bytes long, and additionally, the `Secp256k1.PRIVATE_KEY_SIZE` constant is 32 bytes. Hence, the function always copies 32 bytes into the `privateKey` byte array, and never 33.

```
ECKeyPair keyPair = Keys.createEcKeyPair(new SecureRandom(seed));
// keyPair private key can be 32 or 33 bytes long
byte[] privateKey = Arrays.copyOf(keyPair.getPrivateKey().toByteArray(), Secp256k1.PRIVATE_K
↪ EY_SIZE);
return new PrivateKeySecp256k1(privateKey);
```

- A number of deserialization functions use hardcoded length literals to parse inner fields. Consider for example the function `parse()` of the `ForwardTransferCswDataSerializer` object, and provided below, for reference.

```
override def parse(r: Reader): ForwardTransferCswData = {
val boxId = r.getBytes(32)

val amount = r.getLong()

val receiverPubKeyReversedLength = r.getInt()
val receiverPubKeyReversed = r.getBytes(receiverPubKeyReversedLength)

val paybackAddrDataHash = r.getBytes(20)
val txHash = r.getBytes(32)
val outIdx = r.getInt()

// <snip>
```

  Another example (which does not happen in a deserialization procedure) can be seen in the function `vrfPublicKeyToAbi()` implemented in the ForgerStakeMsgProcessor.scala source file:

```
private[horizen] def vrfPublicKeyToAbi(vrfPublicKey: Array[Byte]): (Bytes32, Bytes1) = {
  val vrfPublicKeyFirst32Bytes = new Bytes32(util.Arrays.copyOfRange(vrfPublicKey, 0, 32))
  val vrfPublicKeyLastByte = new Bytes1(Array[Byte](vrfPublicKey(32)))
  (vrfPublicKeyFirst32Bytes, vrfPublicKeyLastByte)
}
```

  While the respective lengths of these fields are not expected to change regularly, it is recommended to define constants associated with these fields and structures.

- In the `consensus/package.scala` source file, the package object `consensus` defines a `consensusHardcodedSaltString` variable, whose value is currently set to "TEST".

```
val consensusHardcodedSaltString: Array[Byte] = "TEST".getBytes()
val consensusPreForkLength: Int = 4 + 8 + consensusHardcodedSaltString.length
```

  Consider updating this salt value to something more appropriate for production deployments.
- In the TransactionArgs.java source file, the following global RPC gas cap could be specified as a configuration variable, as in *geth*.

```
// global RPC gas cap (in geth this is a config variable)
var gasLimit = BigInteger.valueOf(50_000_000);
```

- In the FieldElementUtils.java source file, the naming of the following function is slightly misleading, as it actually does not *hash* its input, but expects a String, corresponding to the output of a call to the Poseidon hash function. Additionally, this function does not currently seem to be used anywhere.

```
public static FieldElement hashToFieldElement(String hexByte) {
    byte[] hashBytes = BytesUtils.fromHexString(hexByte);
    if (hashBytes.length > fieldElementLength()) {
        throw new IllegalArgumentException("Hash length is exceed Poseidon hash len. Hash
        ↳ len " +
                hashBytes.length + " but it shall be " + fieldElementLength());
    }
    return FieldElement.deserialize(Arrays.copyOf(hashBytes, fieldElementLength()));
}
```

- Throughout the ThresholdSignatureCircuitImplZendoo.java source file, a number of assumptions are made about the underlying Finite Field and the implementation of elements of that field, particularly around the size of such elements. One example can be observed in the `reconstructUtxoMerkleTreeRoot()` function, where the function first deserializes two field elements, and then joins them by assuming that their respective length is 16, without performing any length checks. Instead of using the hardcoded value `16`, consider using a constant coming from the underlying zendoo-sc-cryptolib, such as what is returned by `FIELD_ELEMENT_LENGTH()`.

```
@Override
public byte[] reconstructUtxoMerkleTreeRoot(byte[] fe1Bytes, byte[] fe2Bytes) {
    FieldElement fe1 = FieldElement.deserialize(fe1Bytes);
    if(fe1 == null)
        return new byte[0];
    FieldElement fe2 = FieldElement.deserialize(fe2Bytes);
    if(fe2 == null) {
        fe1.freeFieldElement();
        return new byte[0];
    }

    FieldElement utxoMerkleTreeRootFe = FieldElement.joinAt(fe1, 16, fe2, 16);
    byte[] utxoMerkleTreeRoot = utxoMerkleTreeRootFe.serializeFieldElement();

```

- In the Ed25519.java source file, the signature length is hardcoded to `64` even though there is a function returning a constant with that value a few lines above.

```
public static int signatureLength() {
    return org.bouncycastle.math.ec.rfc8032.Ed25519.SIGNATURE_SIZE;
}

// <snip>

public static byte[] sign(byte[] privateKey, byte[] message, byte[] publicKey) {
    byte[] signature = new byte[64];
```

- In the ByteArrayWrapper.java source file, the comments preceding the implementation of the `hashCode()` function provides an explanation for choosing a multiplier different than 31, stating

> `//do not use Arrays.hashCode, it generates too many collisions (31 is too low)`

This reasoning does not seem correct; the main reason Java uses 31 is mostly because it is relatively prime to 2^{32} and because, since it's small and close to 2^5, it can be very efficiently computed using bitwise shift operations and a subtraction[29].

```
@Override
public int hashCode() {
    //do not use Arrays.hashCode, it generates too many collisions (31 is too low)
    int h = 1;
    for (byte b : data) {
        h = h * (-1640531527) + b;
    }
    return h;
}
```

The value used in the `hashcode()` function is `-1640531527`, which is equal to `2654435769` modulo 2^{32}, a highly composite number (namely the product of `3 * 5 * 11 * 17 * 23 * 29 * 35 * 41 * 47 * 53 * 59 * 65 * 71 * 77 * 83 * 89`). Being also relatively prime to 2^{32}, this value is suitable but it is unclear that less collisions would be generated than by using 31.

If collisions were an issue in that case, a cryptographic hash function should be considered instead.

- There are small discrepancies between the way receipts for unsupported transaction types are handled within the codebase and within geth, the Go Ethereum client: The `EthereumConsensusDataReceipt.rlpEncode()` function will throw a `MatchError` if the transaction is not a `LegacyTxType, AccessListTxType` or `DynamicFeeTxType`:

```
def getTxType: ReceiptTxType = {
  transactionType match {
    case 0 =>
      ReceiptTxType.LegacyTxType
    case 1 =>
      ReceiptTxType.AccessListTxType
    case 2 =>
```

29. See the book *Effective Java* by Joshua Bloch, under Chapter 3: *Always override hashcode when you override equals*.

```
      ReceiptTxType.DynamicFeeTxType
  }
}

def rlpEncode(r: EthereumConsensusDataReceipt): Array[Byte] = {
  val values = asRlpValues(r)
  val rlpList = new RlpList(values)
  val encoded = RlpEncoder.encode(rlpList)
  if (!(r.getTxType == ReceiptTxType.LegacyTxType)) { ... }
```

The geth equivalent `EncodeIndex` will ignore an unsupported transaction type with a note that it will be caught when matching the derived root to the block, see go-ethereum/core/types/receipt.go

Similarly, the `EthereumConsensusDataReceipt.rlpDecode()` function checks whether it's one of two defined EIP-2718 types (although it doesn't use the types defined in the `ReceiptTxType` object), and otherwise tries to decode the input as a legacy receipt, which will fail and return an incorrect RLP encoding error:

```
val b0 = rlpData(0)
if (b0 == 1 || b0 == 2) {
  val rt = ReceiptTxType(b0).id
  decodeTyped(rt, util.Arrays.copyOfRange(rlpData, 1, rlpData.length))
} else decodeLegacy(rlpData)
```

In contrast, the geth implementation function `DecodeRLP()` first checks whether it's a legacy receipt, and returns an unsupported type error if the type is not recognized.

```
case kind == rlp.List:
   // It's a legacy receipt.
   ...
default:
   // It's an EIP-2718 typed tx receipt.
   ...
   return r.decodeTyped(b)
}
```

The `EthereumConsensusDataReceipt.rlpDecode()` function is currently unused, but the `EthereumConsensusDataReceipt.rlpEncode()` function is used to compute the receipt root hash using the function `HashRoot()`, which has a comment noting it is heavily based on the `types.DeriveSha()` function from geth.

- When instantiating a Merkle Tree Path in the MerklePath.java source file, the validity of the index is not checked when iterating through the list of pairs, which is provided as argument. Arguably, this allows supporting more than only binary Merkle Trees. However, only binary trees are used in practice. Hence, index validation would be sugested here.

```
public MerklePath(List<Pair<Byte, byte[]>> merklePath) {
    if(merklePath == null)
        throw new IllegalArgumentException("Merkle path object is not defined.");
    for(Pair<Byte, byte[]> pair : merklePath) {
        if(pair == null || pair.getValue() == null)
           throw new IllegalArgumentException("Merkle path contains broken item inside");
        if(pair.getValue().length != Utils.SHA256_LENGTH)
           throw new IllegalArgumentException("Some of merkle path nodes contains broken
           ↪ bytes. Bytes expected to be SHA256 hash of length 32.");
```

```
    }

    this.merklePath = merklePath;
}
```

- In the Pair.java source file, consider setting the multiplier to 31 for consistency with the rest of the code.

```
@Override
public int hashCode() {
    return key.hashCode() * 13 + (value == null ? 0 : value.hashCode());
}
```

- In the ByteArrayWrapper.java source file, a `compare()` function is defined, which compares two byte arrays provided as argument. There is no documentation surrounding this function indicating that the intent of this method is to perform a comparison using *lexicographic ordering* (which is what the function does). Consider adding documentation clearly indicating the intended ordering assumed by this function.
- When a `Secp256k1` signature is provided to an API endpoint as part of an Ethereum Transaction, for instance in the `sendRawTransaction` API endpoint, the signature gets decoded during the call to `EthereumTransactionDecoder.decode()`, and the `r` and `s` values of the signature get left-padded before being processed:

```
byte[] v = getVFromRecId(Numeric.toBigInt(((RlpString)
↪values.getValues().get(9)).getBytes()).intValueExact());
byte[] r = Numeric.toBytesPadded(Numeric.toBigInt(((RlpString)
↪values.getValues().get(10)).getBytes()), 32);
byte[] s = Numeric.toBytesPadded(Numeric.toBigInt(((RlpString)
↪values.getValues().get(11)).getBytes()), 32);
SignatureSecp256k1 signature = new SignatureSecp256k1(v, r, s);
```

On the other hand, if the signature is provided directly to the API endpoint, such as for the `createEIP1559Transaction` API endpoint, the `r` and `s` elements of the signature get passed to the `SignatureSecp256k1` constructor directly:

```
if (body.signature_v.isDefined)
  new SignatureSecp256k1(
    body.signature_v.get,
    body.signature_r.get,
    body.signature_s.get)
```

However, note that the `SignatureSecp256k1` constructor only accepts signatures as valid if the `r` and `s` elements are contained in a byte array of `SIGNATURE_RS_SIZE`:

```
private static boolean checkSignatureDataSizes(byte[] v, byte[] r, byte[] s) {
    return (v.length > 0 && v.length <= Secp256k1.SIGNATURE_V_MAXSIZE) &&
           (r.length == Secp256k1.SIGNATURE_RS_SIZE && s.length ==
           ↪Secp256k1.SIGNATURE_RS_SIZE);
}
```

As a consequence, non-padded signatures will be accepted by some, but not all, API endpoints.

Insufficient Parameter Validation

- In the SidechainStateUtxoMerkleTreeProvider.scala source file, the `version` parameter should be checked to be non-null; the other two arguments of that function are correctly validated.

```
override def update(version: ByteArrayWrapper,
                    boxesToAppend: Seq[SidechainTypes#SCB],
                    boxesToRemoveSet: Set[ByteArrayWrapper]): Try[SidechainStateUtxoMerkleTre
                    ↪ eProvider] = Try {
  require(boxesToAppend != null, "List of boxes to add must be NOT NULL. Use empty List
  ↪ instead.")
  require(boxesToRemoveSet != null, "List of Box IDs to remove must be NOT NULL. Use empty
  ↪ List instead.")
  val coinBoxesToAppend = boxesToAppend.filter(box => box.isInstanceOf[CoinsBox[_ <: PublicK
  ↪ ey25519Proposition]])

  SidechainUtxoMerkleTreeProviderCSWEnabled(utxoMerkleTreeStorage.update(version,
  ↪ coinBoxesToAppend, boxesToRemoveSet).get)
}
```

- In the `semanticValidity()` function of the `SidechainBlockHeaderBase` source file a few byte arrays are checked to be of correct length, without checking first that these arrays are non-null.

```
def semanticValidity(params: NetworkParams): Try[Unit] = Try {
  if(parentId.length != 64
    || sidechainTransactionsMerkleRootHash.length != 32
    || mainchainMerkleRootHash.length != 32
    || ommersMerkleRootHash.length != 32
    || ommersCumulativeScore < 0
    || feePaymentsHash.length != 32
    || timestamp <= 0)
    throw new InvalidSidechainBlockHeaderException(s"SidechainBlockHeaderBase $id contains
    ↪ out of bound fields.")
}
```

  In comparison, the function `semanticValidity()` of the `MainchainHeader` class performs this null-check prior to the length check.

```
def semanticValidity(params: NetworkParams): Try[Unit] = Try {
  if(hashPrevBlock == null || hashPrevBlock.length != 32
    || hashMerkleRoot == null || hashMerkleRoot.length != 32
    || hashScTxsCommitment == null || hashScTxsCommitment.length != 32
    || nonce == null || nonce.length != 32
```

- In the function `semanticValidity()` of *sdk/src/main/java/com/horizen/account/transaction/EthereumTransaction.java*, the value returned by the call to `getData()` (highlighted below) should first be checked to be non-null before ensuring its length is nonzero.

```
//TODO why this check is not in Geth (maybe)?
if (getTo() == null && getData().length == 0)
    throw new TransactionSemanticValidityException(String.format("Transaction [%s] is
    ↪ semantically invalid: " +
            "smart contract declaration transaction without data", id()));
```

Discrepancies Between RPC and HTTP Endpoints

There seem to be a few discrepancies between the functionalities exposed by the RPC and the HTTP endpoints.

The first one being in the function `signTransactionWithSecret()`, which is implemented in both files. In the EthService.scala source file, that function includes a conditional statement which generates an EIP-155 signature according to the conditions highlighted in the code excerpt below:

```
private def signTransactionWithSecret(secret: PrivateKeySecp256k1, tx: EthereumTransaction):
↳ EthereumTransaction = {
  val signature = secret.sign(tx.messageToSign())
  var signatureData = new SignatureData(signature.getV, signature.getR, signature.getS)
  if (!tx.isEIP1559 && tx.isSigned && tx.getChainId != null) {
    signatureData = TransactionEncoder.createEip155SignatureData(signatureData, tx.getChainId)
  }
  new EthereumTransaction(new SignedRawTransaction(tx.getTransaction.getTransaction,
  ↳ signatureData))
}
```

In comparison, the AccountTransactionApiRoute.scala source file defines two specialized functions, `signTransactionWithSecret()` and `signTransactionEIP155WithSecret()`, which are called from the `sendCoinsToAddress()` according to whether the transaction is of type EIP-155, as shown below.

```
val isEIP155 = body.EIP155.getOrElse(false)
val response = if (isEIP155) {
  // <snip>
  validateAndSendTransaction(signTransactionEIP155WithSecret(secret, tmpTx))
} else {
  // <snip>
  validateAndSendTransaction(signTransactionWithSecret(secret, tmpTx))
}
```

However, the two other checks (namely, `tx.isSigned && tx.getChainId != null`) performed in the *EthService.scala* source file are missing in the function `signTransactionEIP155WithSecret()` in AccountTransactionApiRoute.scala:

```
def signTransactionEIP155WithSecret(secret: PrivateKeySecp256k1, tx: EthereumTransaction):
↳ EthereumTransaction = {
  val messageToSign = tx.messageToSign()
  val msgSignature = secret.sign(messageToSign)
  new EthereumTransaction(
    new SignedRawTransaction(
      tx.getTransaction.getTransaction,
      createEip155SignatureData(new SignatureData(msgSignature.getV, msgSignature.getR,
      ↳ msgSignature.getS), params.chainId)
    )
  )
}
```

The other notable discrepancy is in the `sendRawTransaction()` function, which in *EthService.scala* doesn't seem to perform extensive checks, as can be seen below.

```
def sendRawTransaction(signedTxData: String): String = {
    val tx = new EthereumTransaction(EthereumTransactionDecoder.decode(signedTxData))
    implicit val timeout: Timeout = new Timeout(5, SECONDS)
```

```
    // submit tx to sidechain transaction actor
    val submit = (sidechainTransactionActorRef ?
    ↳ BroadcastTransaction(tx)).asInstanceOf[Future[Future[ModifierId]]]
    // wait for submit
    val validate = Await.result(submit, timeout.duration)
    // wait for validation of the transaction
    val txHash = Await.result(validate, timeout.duration)
    Numeric.prependHexPrefix(txHash)
}
```

Comparatively, the `sendRawTransaction()` function in *AccountTransactionApiRoute.scala* performs more in-depth validation of the transactions, as shown in the excerpt below.

```
var signedTx = new EthereumTransaction(EthereumTransactionDecoder.decode(body.payload))
if (!signedTx.isSigned) {
  val txCost = signedTx.getValue.add(signedTx.getGasPrice.multiply(signedTx.getGasLimit))

  val secret =
    getFittingSecret(sidechainNodeView, body.from, txCost)
  secret match {
    case Some(secret) =>
      signedTx = signTransactionWithSecret(secret, signedTx)
      validateAndSendTransaction(signedTx)
    case None =>
        ApiResponseUtil.toResponse(ErrorInsufficientBalance("ErrorInsufficientBalance",
        ↳ JOptional.empty()))
  }
```

Minor Cosmetic Notes

- The following `todo` comment in the AbstractSidechainNodeViewHolder.scala source file could probably be removed, since it appears that it has been addressed.

```
val branchingPoint = progressInfo.branchPoint.get //todo: .get
```

- In the AbstractHistory.scala source file, there appears to be an incongruously indented pattern matching case.

```
  case Some(commonBlockId) =>
    storage.activeChainAfter(commonBlockId, Some(size + 1)).tail.map(id =>
    ↳ (SidechainBlockBase.ModifierTypeId, id))      case None =>
    //log.warn("Found chain without common block ids from remote")
    Seq()
}
```

- In the CryptoLibProvider.scala source file, the following variable is defined to convert ZEN to Zennies.

```
val ZEN_COINS_DIVISOR: BigDecimal = new BigDecimal(100000000)
```

  Consider updating that assignment to using a *power of ten* notation, as in the ZenWeiConverter.scala source file, as shown below.

```
val ZENNY_TO_WEI_MULTIPLIER: BigInteger = BigInteger.TEN.pow(10)
```

- Another instance of incongruous indentation is found in the EthService.scala source file; the highlighted code below should be on a new line.

```
@RpcMethod("eth_sendRawTransaction") def sendRawTransaction(signedTxData: String): String =
↳{
  val tx = new EthereumTransaction(EthereumTransactionDecoder.decode(signedTxData))
```

Remaining TODOs and commented code

The reviewed codebase is currently still interspersed with a number of `TODO`s and other remnants of commented code, indicating that significant development effort is still to be performed on the project. The Horizen Labs team should aim to addressing these open items prior to upcoming release cycles.

Additionally, the NCC Group consultants noted that the codebase used a number of different spellings for `TODO`s, namely:

- `todo` in ThresholdSignatureCircuit.java
- `to-do` in WebSocketCommunicationClient.scala
- `to do` in AbstractHistory.scala
- `TODO` for example in ClosableResourceHandler.scala
- `TO-DO` in SidechainTransactionApiRoute.scala
- `TO DO` in SparkzEncoding.java

Hence, when searching through the codebase for such instances, consider including all these different spellings.

Notes on `libevm`

- In the libevm/lib/evm.go source file, the function `setDefaults()` is used to assign default values to parameters that were omitted. In that function, `GasLimit` is set as the maximum signed 64-bit integer `math.MaxInt64`, as shown below.

```
// setDefaults for parameters that were omitted
func (c *EvmContext) setDefaults() {
  if c.GasLimit == 0 {
      c.GasLimit = (hexutil.Uint64)(math.MaxInt64)
  }
      // <snip>
```

  Whereas in a comparable file in `go-ethereum`, in *core/vm/runtime/runtime.go*, it is set as the max *unsigned*.

```
if cfg.GasLimit == 0 {
  cfg.GasLimit = math.MaxUint64
}
```

  In the same file (i.e., *libevm/lib/evm.go*), consider also checking whether the following signed `AvailableGas` default value is correct.

```
if p.AvailableGas == 0 {
  p.AvailableGas = (hexutil.Uint64)(math.MaxInt64)
}
```

- In the libevm/lib/refund.go source file, the documentation preceding the `setStateRefunds()` function states that it follows an implementation located in *gas_table.go:178*.

```
// setStateRefunds replicates the refund part of the original SSTORE gas consumption logic.
// Original implementation can be found in go-ethereum, function "gasSStoreEIP2200":
// github.com/ethereum/go-ethereum@v1.10.26/core/vm/gas_table.go:178
```

  This does not appear to be true. After discussion with the Horizen Labs team, it was revealed that this function follows an implementation of go-ethereum/core/vm/operations_acl.go instantiated by go-ethereum/core/vm/operations_acl.go.

- In `libevm`, the function `StateGetStorageBytes()` in the libevm/lib/storage.go source file may silently fail. Specifically, the call to `GetState()` highlighted below may return an empty hash, which will then be interpreted as the zero value by the function `hashToInt()`. In turn, this function will then create an empty byte array, and will return it with an error value equal to `nil`, representing a success.

```
func (s *Service) StateGetStorageBytes(params StorageParams) (error, []byte) {
  err, statedb := s.statedbs.Get(params.Handle)
  if err != nil {
      return err, nil
  }
  length := hashToInt(statedb.GetState(params.Address, params.Key))
  data := make([]byte, length)
  for start := 0; start < length; start += common.HashLength {
      chunkIndex := start / common.HashLength
      end := start + common.HashLength
      if end > length {
         end = length
      }
      chunk := statedb.GetState(params.Address, getChunkKey(params.Key, chunkIndex))
      copy(data[start:end], chunk.Bytes())
  }
  return nil, data
}
```

  In comparison, the function `StateSetStorageBytes()` defined a couple of lines further down handles this in the following way:

```
if statedb.Empty(params.Address) {
  // if the account is empty any changes would be dropped during the commit phase
  return fmt.Errorf("%w: %v", ErrEmptyAccount, params.Address)
}
// get previous length of value stored, if any
oldLength := hashToInt(statedb.GetState(params.Address, params.Key))
```

- In the same storage.go source file, the function `hashToInt()` is used to convert a hash value into an int

```
// convert 256-bit hash value to int, takes care of deserialization and padding
func hashToInt(hash common.Hash) int {
  return int(hash.Big().Int64())
}
```

  The result of the call to `Int64()` is undefined if the big integer representation of the hash is larger than an `int64`. Fortunately, this does not seem to happen in any of the current call to `hashToInt()`, but it might still be advisable to explicitly handle this edge-case.

- In the same source file, the function `StateGetStorageBytes()` is used to fetch an arbitrary long value stored in a number of leaves of the Trie. It starts by fetching the length of the data to be retrieved (highlighted), before deterministically iterating over the leaves and retrieving the expected chunks of data.

```
func (s *Service) StateGetStorageBytes(params StorageParams) (error, []byte) {
  err, statedb := s.statedbs.Get(params.Handle)
  if err != nil {
    return err, nil
  }
  length := hashToInt(statedb.GetState(params.Address, params.Key))
  data := make([]byte, length)
```

If the `length` value retrieved were larger than an `int` (be it through a malicious process or not), the function would crash when trying to instantiate a byte array of that length with an `(untyped int constant) overflows int` error. Consider validating that parameter prior to the creation of the byte array.

- Still in the same source file, the function `bytesToHash()` (provided below for reference) assumes that the length of the byte array will be *at most* equal to the length of a Hash in the `else`-clause of the conditional statement. Consider updating the function to ensure that in no case `len() > hashlen`, which would be evidence of misuse.

```
// make sure we add trailing zeros, not leading zeroes
func bytesToHash(bytes []byte) common.Hash {
    if len(bytes) < common.HashLength {
        tmp := make([]byte, common.HashLength)
        copy(tmp, bytes)
        return common.BytesToHash(tmp)
    } else {
        return common.BytesToHash(bytes)
    }
}
```

- While not in direct scope for this engagement, `zend` appears vulnerable to Slowloris-type attacks. During stress testing the connections through which sidechains and `zend` communicate, it was found that sidechains appear to be resistant to attacks with the following flavor: opening a huge number of connections and passing large amounts of data. This is likely thanks to the Akka framework. This is not the case with `zend`; if flooded with roughly more than 50000 socket openings, the corresponding port gets closed. In particular, the following steps were used:
    - Run the `qa/sc_mem_usage.py` test
    - Use `nmap localhost -p-` to find open ports and find ports owned by `zend` using `netstat -ano -p tcp`. This port is typically the highest open port, eg. 13794
    - Run a script that opens a large number of connections (eg. 50000) keeps them open and occasionally writes a small amount of data

```
zend: pthread_mutex_lock.c:450: __pthread_mutex_lock_full: Assertion `e != ESRCH || !
↳robust' failed.
[2023-01-31 00:05:48,399] : [ERROR] : Unexpected exception caught during testing: [Errno
↳111] Connection refused
File "/home/aleks/Sidechains-SDK/qa/SidechainTestFramework/sc_test_framework.py", line 171,
↳in main
```

```
  self.run_test()
File "/home/aleks/Sidechains-SDK/qa/./sc_mem_usage.py", line 126, in run_test
  while mc_node.getmempoolinfo()["size"] == 0 and attempts > 0:
...
```

Notes on the Reference Paper

The Zendoo reference paper states:

> Following this design, we also introduce withdrawal epochs in a sidechain which coincide with the mainchain withdrawal epochs. A WE is defined as a range of SC blocks where the first and last blocks of the range are determined by references to the first and last MC blocks in the corresponding withdrawal epoch in the MC (see Fig. 8).

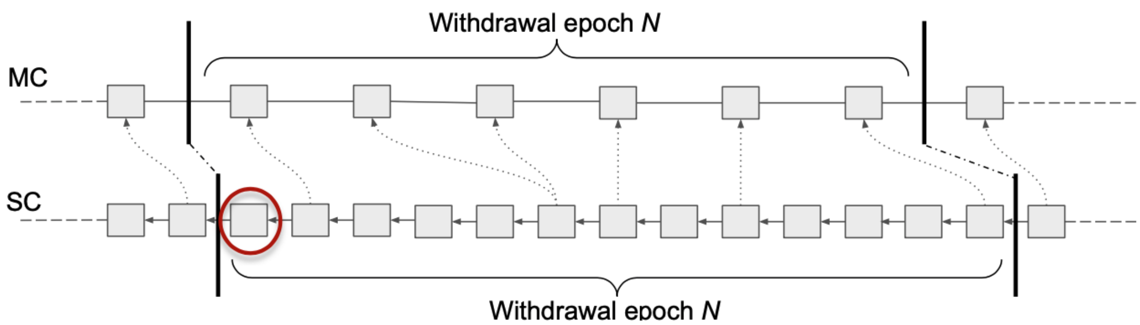

Figure 8: An example of a withdrawal epoch in the sidechain.

However, the illustration does not accurately reflect that; the first block in the *Sidechain Withdrawal Epoch N* (circled in red) does not point to the first block in the Mainchain Withdrawal Epoch *N*.