Security Audit Report

Stroom Network Security Audit Report

Prepared by:

The Arcadia Group

Prepared for:

Rostyslav Shvets. Stroom Network (Current Limited)

Date: July 16, 2025

https://arcadia.agency/

Email: audits@arcadiamgroup.com

Telegram: https://t.me/thearcadiagroup

Auditor Information

Company: Arcadia

Address: 955 W John Carpenter Frwy, Ste. 100, Irving, Texas 75039

Website: https://arcadia.agency/

Email: audits@arcadiamgroup.com

Auditors:

- Van Cam Pham, PhD - Lead Security Engineer
- Toni Dong - Security Engineer
- Minh Khai Do - Security Engineer
- Hernán Darío Vanegas Madrigal - Cryptography Research Engineer

About Us: Arcadia is a software security and product development company building and securing software.

Client Information

Company Name: Current Limited

Website: https://stroom.network/

Contact Person: Rostyslav Shvets

Project Name: Stroom Network

Revision History

Date	Reason	Commit
01/10/2025	Initial Audit Scope	
2/24/2025	Initial Delivery of Smart Contract and Cryptography Findings	
3/14/2025	Additional Cryptography Findings	
6/12/2025	Review of Smart Contracts Rebase	#bed2cb09
6/12/2025	Review of Frost and go-stroom remediation	#547a98eb, #ca35e3df
7/4/2025	Review of 4626 Token Modification	
8/5/2025	Review of Review of go-stroom remediation	
8/10/2025	Review of Production Deployments and Notes for Deprecated Contracts	

Contents

1 Introduction

Stroom Network (Current Limited) engaged Arcadia to perform security audits of their various project repositories.

The primary objective of this audit is to ensure the security, integrity, and robustness of Stroom's codebases, focusing on the custom cryptographic implementations, smart contracts, and critical components that support the protocol's bridge implementation. Given the complexity and critical nature of these components, the audit will specifically address potential vulnerabilities related to the multi-party computation (MPC) protocol, threshold signature schemes (TSS), and the interaction between Bitcoin and Ethereum (EVM) chains. Additionally, the audit will assess the synchronization mechanisms between Bitcoin and Ethereum to ensure consistency and correctness across both networks.

1.1 Audit Scope

The audit covered multiple components of the Stroom Network ecosystem, focusing on critical security aspects across different repositories and technologies.

1.1.1 Custom Cryptographic Libraries

- **FROST Implementation (frost)**: Review the FROST (Flexible Round-Optimized Schnorr Threshold) MPC protocol for security flaws in the threshold signing process, focusing on the Distributed Key Generation (DKG) and its integration with the broader protocol. Special attention will be given to the correctness of cryptographic operations, resistance to known attack vectors, and the secure management of key shares during distributed signing operations.
- **Key Generation (keygen)**: Evaluate the key generation process for Schnorr signatures, ensuring compliance with BIP-0340. The audit will verify that the public keys generated are secure, resistant to potential vulnerabilities, and correctly integrated with the FROST protocol.

1.1.2 Bridge and Validator Codebases

- **Main Bridge Node (go-stroom)**: Assess the main bridge codebase, which includes validators and executors monitoring Bitcoin and Ethereum chains for mints and redeems. This component is critical for the cross-chain operations that underpin Stroom's protocol. The audit will focus on ensuring that the integration of the FROST library within the bridge is secure and that the cross-chain operations are resilient to both network-level and protocol-level attacks. Additionally, the audit will cover data validation processes for cross-chain monitoring and the synchronization of the BTC and lnBTC balances, including the handling of reordering, forks and finalization issues.

- **Blockchain Tools (blockchain-tools)** : Examine the Solidity library for Bitcoin-related functionality, focusing on the encoding and decoding of Bitcoin addresses and the implementation of Segwit v0 and v1 (Taproot) addresses. The review will ensure that the Solidity optimizations and error-handling mechanisms are secure and function as intended.
- **Smart Contracts (stroom-contracts) - Deprecated**: Analyze the smart contracts used for managing bridged assets on Ethereum and other EVM chains. The audit will include a review of the contract logic, security against typical DeFi attacks (such as reentrancy and overflow/underflow)
  - **lnBTC Smart Contract**: This ERC-20 token represents a 1-to-1 convertible asset with Bitcoin. The audit will review its minting, rewards minting, and redemption processes, particularly the Schnorr signature verification process conducted by the smart contract. The audit will also evaluate whether the ERC-7265 standard should be implemented and how minting and redemption operations are secured and correctly handled.
  - **slnBTC Smart Contract** : This ERC-4626 compliant contract represents users' shares in a pool of staked lnBTCs. The audit will assess the staking and redemption processes, ensuring that the calculations for lnBTC and slnBTC conversions are accurate and secure. Additionally, the audit will review the direct minting and redemption features to ensure they are correctly implemented and secure.
- **Smart Contracts (stroom-contracts-rebase)**: Analyze the rewritten smart contracts used for managing bridged assets on Ethereum and other EVM chains. The audit will include a review of the contract logic, security against typical DeFi attacks (such as reentrancy and overflow/underflow)

1.1.3 Infrastructure Components

- **Infrastructure-As-Code (IAAC)** : Review the monitoring scripts, deployment scripts and Docker configurations for Stroom services to ensure the monitoring is sufficiently granular and covers potential missing edge cases. Arcadia will additionally review security groups and permissions of the EC2 deployment pipeline as well as pod-specific permissions. Review and assess the current AWS infrastructure, including architecture, security, scalability, and performance, with recommendations for improvements and best practices.review and assess the current AWS infrastructure, including architecture, security, scalability, and performance, with recommendations for improvements and best practices.

1.1.4 Source Code Repositories

- Primary repositories audited across multiple commits
- Focus on commits: #bed2cb09, #547a98eb, #ca35e3df
- Version control via GitHub with continuous integration

1.2 Review Team

Our Engagement Specific Team

- **Van Cam Pham** - Lead Security Engineer
- **Tony Dong** - Security Engineer
- **Minh Khai Do** - Security Engineer
- **Hernán Darío Vanegas Madrigal** - Cryptography Research Engineer

1.3 Project Background

Stroom is a Bitcoin Liquid Staking protocol that enables users to earn yield through the Lightning Network without requiring complex infrastructure. Stroom's protocol simplifies access to the Lightning Network's capabilities, allowing for scalable and efficient staking within the broader cryptocurrency ecosystem.

1.4 Coverage

For this audit, we performed research, test coverage, investigation, and review of the below listed repositories, followed by issue reporting and mitigation and remediation instructions as outlined in this report. The following code repositories, files, and/or libraries are considered in scope for the review.

- **frost** - FROST (Flexible Round-Optimized Schnorr Threshold) MPC protocol implementation
  - Distributed Key Generation (DKG) components
  - Threshold signing process implementation
  - Cryptographic operations and key share management
- **keygen** - Schnorr signature key generation library
  - BIP-0340 compliant key generation
  - Public key security validation
  - FROST protocol integration components
- **go-stroom** - Main bridge node implementation
  - Bridge validators and executors

  - Bitcoin and Ethereum chain monitoring
  - Cross-chain mint and redeem operations
  - FROST library integration
  - Balance synchronization mechanisms
- **blockchain-tools** - Solidity library for Bitcoin functionality
  - Bitcoin address encoding/decoding
  - Segwit v0 and v1 (Taproot) address implementation
  - Solidity optimizations and error handling
- **stroom-contracts** - The now deprecated smart contracts for bridged asset management
  - lnBTC Smart Contract (ERC-20)
    - Minting and redemption processes
    - Schnorr signature verification
    - Rewards minting functionality
  - slnBTC Smart Contract (ERC-4626)
    - Staking and redemption processes
    - lnBTC/slnBTC conversion calculations
    - Direct minting and redemption features
- **stroom-contracts-rebase** - The reimplemented smart contracts for bridged asset management
- **Infrastructure-As-Code (IAAC)** - Deployment and monitoring configurations
  - Monitoring scripts and configurations
  - Docker deployment configurations
  - AWS infrastructure components
  - EC2 deployment pipeline and security groups
  - Pod-specific permissions and access controls

Repository Coverage Summary:

Description	GitHub Repository Path	Commit Hash
blockchain-tools	stroomnetwork/blockchain-tools	
frost	stroomnetwork/frost	1ba6e32
go-stroom	stroomnetwork/go-stroom	

iaac	stroomnetwork/iaac	
stroom-app	stroomnetwork/stroom-app	
stroom-contracts	stroomnetwork/stroom-contracts	
btcwallet	stroomnetwork/btcwallet	Removed
keygen	stroomnetwork/keygen	
stroom-contracts-rebase	stroomnetwork/strom-contracts-rebase	Added 77e04bb

2 Report Summary

This audit identified a total of **35 security findings** across the Stroom Network smart contracts and supporting infrastructure. The findings are categorized by severity level as follows:

Severity Level	Count	Finding IDs
Critical	1	SN-3
High	9	SN-1, SN-11, SN-12, SN-15, SN-29, SN-30, SN-32, SN-33, SN-34
Medium	6	SN-7, SN-10, SN-17, SN-22, SN-26, SN-35
Low	10	SN-2, SN-5, SN-6, SN-18, SN-19, SN-23, SN-24, SN-27, SN-28, SN-31
Informational	9	SN-4, SN-8, SN-9, SN-13, SN-14, SN-16, SN-20, SN-21, SN-25

2.1 Key Security Concerns

The audit revealed several critical areas requiring immediate attention:

- **Access Control Issues**: Critical finding SN-3 identified that the `setSeed` function can be called by anyone, potentially compromising validator key management
- **Centralization Risks**: Multiple high-severity findings (SN-11, SN-15, SN-35) highlight risks associated with upgradeable contracts and centralized minting capabilities
- **Signature Replay Vulnerabilities**: High-severity finding SN-12 identified potential signature replay attacks in validator key updates
- **Token Supply Discrepancies**: High-severity findings (SN-1, SN-32, SN-34) identified potential mismatches between deposited BTC and minted tokens
- **Infrastructure Security**: Multiple findings address AWS security configurations and network controls

2.2 Remediation Status

Of the 35 findings identified:

- **Resolved**: 34 findings have been addressed by the development team
- **Acknowledged**: 1 finding has been marked as acknowledged
- **Pending**: 0 findings require remediation

The development team has demonstrated strong responsiveness to security concerns, with the majority of critical and high-severity issues either resolved or acknowledged with planned remediation strategies.

3 Audit Findings

The audit identified several issues, categorized by severity: Critical, High, Medium, Low, and Informational. Each finding includes a description, impact, and recommended mitigation.

3.1 [SN-1] mintRewards and updateTotalSupply mint new stBTC with a self created depositId

Risk Level
Severity: High, Likelihood: High

Code Segment

```
function mintRewards(
      uint nonce,
      uint delta
  ) external nonReentrant onlyOwner {
      require(
          nonce == totalSupplyUpdateNonce,
          "Invalid update total supply nonce"
      );
      totalSupplyUpdateNonce += 1;

      if (delta == 0) {
          revert("delta is 0");
      }

      address asset = address(this.asset());
      bytes32 mintId = getTotalSupplyUpdateHash(nonce, delta);
      IStBTC(asset).updateTotalSupply(delta, mintId);

      uint256 _totalSupply = IERC20(asset).totalSupply();
      uint256 _totalShares = this.totalSupply();

      emit TotalSupplyUpdatedEvent(nonce, _totalSupply,
          _totalShares);
  }

  // ========== VALIDATOR ONLY ==========

  /**
```

```
28      * @notice Mints rewards to update the total pooled BTC supply
            based on a signed validator message.
29      * @param nonce The unique identifier for the supply update,
            ensuring the correct order of updates.
30      * @param delta The amount of BTC to be added to the total pooled
            supply as rewards.
31      * @param signature The validator's signature, verifying the
            authenticity of the update request.
32      */
33      function mintRewards(
34          uint nonce,
35          uint delta,
36          bytes calldata signature
37      )
38          external
39          nonReentrant
40          onlyValidator(
41              MESSAGE_UPDATE_TOTAL_SUPPLY,
42              encodeTotalSupplyUpdate(nonce, delta),
43              signature
44          )
45      {
46          require(
47              nonce == totalSupplyUpdateNonce,
48              "Invalid update total supply nonce"
49          );
50          totalSupplyUpdateNonce += 1;
51
52          if (delta == 0) {
53              revert("delta is 0");
54          }
55
56          address asset = address(this.asset());
57          bytes32 mintId = getTotalSupplyUpdateHash(nonce, delta);
58          IStBTC(asset).updateTotalSupply(delta, mintId);
59
60          uint256 _totalSupply = IERC20(asset).totalSupply();
61          uint256 _totalShares = this.totalSupply();
62
63          emit TotalSupplyUpdatedEvent(nonce, _totalSupply,
                _totalShares);
64      }
65
66  /**
67       * @dev Update total supply.
68       * Only the minter can call this function.
69       * @param _amount The amount of tokens to mint.
70       * @param _btcDepositId The id of the BTC deposit =
```

```
        keccak256(txHash, vout)
71      */
72      function updateTotalSupply(
73          uint256 _amount,
74          bytes32 _btcDepositId
75      ) public whenNotPaused {
76          require(msg.sender == minter, "stBTC: only minter allowed to
                mint");
77          _updateTotalSupply(_amount, _btcDepositId);
78      }
79
80      /**
81       * @dev Update total supply.
82       * @param _amount The amount of tokens to mint.
83       * @param _btcDepositId The id of the BTC deposit =
             keccak256(txHash, vout)
84       */
85      function _updateTotalSupply(
86          uint256 _amount,
87          bytes32 _btcDepositId
88      ) internal {
89          require(_amount > 0, "MINT_AMOUNT_ZERO");
90          require(_amount < 21_000_000 * BTC, "MINT_AMOUNT_TOO_BIG");
91          require(
92              btcDepositIds[_btcDepositId] == false,
                    "UPDATE_ALREADY_PROCESSED"
93          );
94          btcDepositIds[_btcDepositId] = true;
95
96          _mint(msg.sender, _amount);
97      }
```

Description

- stBTC must be minted only upon real bitcoin deposit transactions to ensure 1-1 mapping between stBTC and the deposited amount of BTC on the bitcoin blockchain
- The two functions mintRewards of bstBTC contract can create "unreal" depositId and call the corresponding updateTotalSupply functions of stBTC to mint new stBTC
- These functions have no verification on the delta amount of stBTC to be minted and how does the to-be-minted amount have any relation to the previously minted stBTC. This could potentially create a mismatch between deposited BTC on the bitcoin network and the amount of stBTC minted

- These functions can also create a transparency issue as there is no on-chain information to verify how the delta amount to be minted is computed

Code Location

- `stroom-contracts/src/stBTC.sol` Lines **198-229**
- `stroom-contracts/src/bstBTC.sol` Lines **43-66, 70-107**

Recommendation

- The delta amount to be minted by these functions must be verified in relation to verifiable on-chain transactions
- If the delta amount to be minted is the fee amount when bridging from bitcoin network to stBTC on Ethereum, the delta amount must be exposed and verifiable through the fee amount
- The following is a possible code implementation: The following code shows unmintedFee increment when new stBTC is minted and any one can call a publicly verifiable function to mint all remaining unminted fee. This ensures that at any point in time the 1-1 mapping between BTC and stBTC is always ensured

```
uint256 public unmintedFee;
   uint256 const FEE_BPS = 30; // 0.3%
   address public feeReceiver;
   /**
    * @dev Mint new tokens.
    * @param _amount The amount of tokens to mint.
    * @param _recipient The address that will receive the minted
        tokens.
    * @param _btcDepositId The id of the BTC deposit =
        keccak256(txHash, vout)
    */
   function _mint(
       uint256 _amount,
       address _recipient,
       bytes32 _btcDepositId
   ) internal {
       require(_amount > 0, "MINT_AMOUNT_ZERO");
       require(_amount < 21_000_000 * BTC, "MINT_AMOUNT_TOO_BIG");

       require(_recipient != address(this),
           "MINT_TO_THE_CONTRACT_ADDRESS");

       require(
           btcDepositIds[_btcDepositId] == false,
```

```
            "MINT_ALREADY_PROCESSED"
        );
        btcDepositIds[_btcDepositId] = true;

        uint256 fee = FEE_BPS * _amount / 10000;
        _mint(_recipient, _amount - fee);
        // accumulate fee each time new stBTC is minted
        unmintedFee += fee;

        emit MintBtcEvent(_recipient, _amount, _btcDepositId);
    }

    function _mintFee() external {
        if (unmintedFee > 0) {
            _mint(feeReceiver, unmintedFee);
            unmintedFee = 0;
            // emit events for minting fee
        }
    }
```

Remediation Status
Resolved, this contract version was deprecated for a new contract.

3.2 [SN-2] Specify a deterministic solidity version in your code

Risk Level
Severity: Low

Code Segment

```
pragma solidity ^0.8.24;

import {Deriver} from "./Deriver.sol";
import {Bech32m} from "./Bech32m.sol";
import {BitcoinNetworkEncoder} from "./BitcoinNetworkEncoder.sol";

error SeedWasNotSetYet();
error UnsupportedBtcAddress(string btcAddress);
error CannotParseBtcAddress(
    string btcAddress,
    string hrp,
    Bech32m.DecodeError err
);

// Types of Bitcoin Network
```

```
contract BTCDepositAddressDeriver {
```

Code Segment

```
[profile.default]
solc = "0.8.25"
src = "src"
out = "out"
libs = ["lib"]
fs_permissions = [
  { access = "read", path = "./broadcast" },
  { access = "read-write", path = "./temp" }
]
# See more config options
    https://github.com/foundry-rs/foundry/blob/master/crates/config/README.md#all-optio
```

Code Segment

```
pragma solidity ^0.8.27;

import
    "@openzeppelin/contracts-upgradeable/contracts/utils/ReentrancyGuardUpgradeable.sol
import
    "@openzeppelin/contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgrad

import "./IBstBTC.sol";
import "./lib/ValidatorRegistry.sol";
import "./lib/ValidatorMessageReceiver.sol";

contract bstBTC is
    ERC4626Upgradeable,
    ReentrancyGuardUpgradeable,
    ValidatorMessageReceiver
{
```

Description

- Specifying a deterministic Solidity compiler version in your Solidity code is highly recommended
- This ensures that your code is compiled consistently across different environments and avoids potential issues caused by differences between compiler versions
- By specifying the compiler version, you ensure that your Solidity code behaves as expected and is less prone to issues caused by compiler differences

- Using caret (^) notation allows for minor version updates that may introduce unexpected behavior changes

Code Location

- Across multiple source code files in `stroom-contracts/`
- Files using `pragma solidity ^0.8.20;`

Recommendation

- **Pin the Exact Version**: For production contracts, it's best to pin the exact compiler version to avoid unexpected changes
- **Test with the Specified Version**: Always test your contracts with the exact compiler version specified in the pragma directive
- **Update Carefully**: When updating the compiler version, thoroughly test your contracts to ensure compatibility and security
- Replace `pragma solidity ^0.8.20;` with `pragma solidity 0.8.20;` for deterministic compilation

Remediation Status
Resolved

3.3 [SN-3] setSeed function can be called by anyone

Risk Level
Severity: Critical

Code Segment

```
function setSeed(
     string calldata _btcAddr1,
     string calldata _btcAddr2,
     BitcoinNetworkEncoder.Network _network
  ) public virtual {
     string memory _hrp =
         BitcoinNetworkEncoder.getNetworkPrefix(_network);

     networkHrp = _hrp;

     (p1x, p1y) = parseBTCTaprootAddress(_hrp, _btcAddr1);
     (p2x, p2y) = parseBTCTaprootAddress(_hrp, _btcAddr2);

```

```
13        btcAddr1 = _btcAddr1;
14        btcAddr2 = _btcAddr2;
15
16        wasSeedSet = true;
17        emit SeedChanged(_btcAddr1, _btcAddr2, _hrp);
18    }
19
20  function getBTCDepositAddress(
21        address ethAddr
22    ) public view returns (string memory) {
23
24        if (!wasSeedSet) {
25            revert SeedWasNotSetYet();
26        }
27
28        return
29            Deriver.getBtcAddressFromEth(
30                p1x,
31                p1y,
32                p2x,
33                p2y,
34                bytes(networkHrp),
35                ethAddr
36            );
37    }
```

Description

- The function is used to set up the public keys of validators by parsing the input taproot addresses of validators
- The function updates the validators' keys each time the function is called
- The updated keys of validators are used in the function `getBTCDepositAddress` for generating bitcoin deposit addresses from a user ethereum address
- As anyone can update the keys of validators, the contract functions could generate unexpected Bitcoin deposit addresses, which cause users to lose deposited bitcoin

Code Location

- `blockchain-tools/src/BTCDepositAddressDeriver.sol`

Recommendation

- The function `setSeed` should have permission control that only allows authorized addresses to update the keys of validators

- Or the function `setSeed` should only allow updating the keys if the keys were not set before
- If the function `setSeed` is not intended for other contracts to inherit, it is recommended to set the function visibility to internal

Remediation Status
Resolved.
The contract is meant to be inherited and the function setSeed will be written by the child contract. Specifically the contract UserActivator will inherit BTCDepositAddressDeriver and implement necessary access control.

3.4 [SN-4] Consistency and code readability for the use of prefix and suffix in for loop

Risk Level
Severity: Informational, Likelihood: Low

Code Segment
No specific code segment - affects multiple for loop implementations throughout the codebase

Description

- The code uses a mixture of both suffix and prefix increments in its for loops
- Even though the gas costs for these increments are the same for Solidity version 0.8.0, the inconsistency of suffix or prefix increment usage in the code will affect code readability
- Although there is no performance difference, using prefix increment is considered a best practice in Solidity. This is because:
  - It aligns with the behavior of other languages (e.g., C++, JavaScript) where `++i` can be more efficient
  - It makes the code more consistent and easier to read for developers familiar with other programming languages

Code Location

- `blockchain-tools` - Various for loop implementations

Recommendation

- It is recommended to use either prefix or suffix increments consistently across the source code files to increase source code readability
- Adopt prefix increment (`++i`) as the standard practice for consistency with best practices

Remediation Status
Resolved

3.5 [SN-5] Optimize gas cost for searching for separator position

Risk Level
Severity: Low, Likelihood: Low

Code Segment

```
bytes memory bechLow = toLower(bech);
     int delimiterPos = int256(bechLow.length) - 1;
     while (true) {
         delimiterPos -= 1;
         if (delimiterPos < 0) {
             return (
                 new bytes(0),
                 new bytes(0),
                 BechEncoding.UNKNOWN,
                 DecodeError.NoDelimiter
             );
         }
         // 0x31 is '1'
         if (bechLow[uint256(delimiterPos)] == 0x31) {
             break;
         }
     }
```

Description

- On mainnet, a bitcoin taproot address has the separator ("1") at position 2
- The above while loop searches for the separator position by iterating the address from the last byte to the first byte
- As the separator position is at 2, the while loop should iterate from the beginning of the address to minimize gas usage

Code Location

- `blockchain-tools/src/Bech32m.sol` Lines **537-553**

Recommendation

- It is recommended to search for the separator position starting at the beginning of the address
- Or directly check whether the byte at position 2 is the separator character ("1") to minimize gas cost as Taproot addresses on Bitcoin Mainnet should have the separator at position 2

Remediation Status
Resolved

3.6 [SN-6] Calculating array length on each loop consumes additional gas

Risk Level
Severity: Low, Likelihood: Low

Code Segment

```
function areBytesEqual(
     bytes memory a,
     bytes memory b
  ) internal pure returns (bool) {
     if (a.length != b.length) {
         return false;
     }
     for (uint i = 0; i < a.length; i += 1) {
         if (a[i] != b[i]) {
             return false;
         }
     }
     return true;
  }
```

Description

- Using `array.length` directly in the condition of a for loop in Solidity can have performance and gas efficiency drawbacks

- While it may seem convenient, it is generally better to cache the length of the array outside the loop
- **Repeated Storage Reads**: In Solidity, accessing `array.length` involves reading from storage or memory, which incurs gas costs. If `array.length` is accessed in every iteration of the loop, the gas cost increases because the length is read repeatedly
- For large arrays, the repeated access to `array.length` can significantly increase the gas cost of the transaction
- For storage arrays, accessing `array.length` in each iteration is particularly expensive because storage reads are much more costly than memory reads

Code Location

- `blockchain-tools/src/Base58.sol` - Lines 24, 63, 79, 90, 151
- `blockchain-tools/src/Bech32m.sol` - Lines 124, 141, 163, 166, 195, 198, 230, 304, 445, 474, 492, 596, 609, 645
- `blockchain-tools/src/BitcoinUtils.sol` - Lines 137, 237, 261, 285, 335, 343, 370
- `blockchain-tools/src/Tools.sol` - Line 37
- `stroom-contracts/src/lib/Base58.sol` - Lines 23, 62, 78, 89, 150

Recommendation

- Cache the length of the array outside the loop to avoid repeated reads and reduce gas costs
- Use the cached length in the loop condition for better gas efficiency
- Example: `uint256 len = a.length; for (uint i = 0; i < len; i += 1)`

Remediation Status
Resolved

3.7 [SN-7] Conflicting comments and code implementation

Risk Level
Severity: Medium, Likelihood: High

Code Segment
No specific code segment - affects function documentation throughout the codebase

Description

- The comments of these functions state that the function `getBtcAddressTaprootNoScriptFr` should be used to generate a user bitcoin deposit address from a recipient address for stBTC on Ethereum
- However, across the source code, only the function `getBtcAddressFromEth` is used for generating deposit addresses
- Having comments that conflict with the source code can lead to several issues. Comments are meant to provide clarity and context for the code, but if they are inaccurate, outdated, or misleading, they can cause confusion, bugs, and maintenance challenges
- During code reviews, conflicting comments can mislead reviewers, causing them to miss potential issues or approve incorrect code

Code Location

- `blockchain-tools/src/Deriver.sol` - Lines 153-181, 106-125
- `blockchain-tools/src/BTCDepositAddressDeriver.sol` - Function `getBTCDepositAddre`

Recommendation

- **Keep Comments Updated**: Determine which function between the two functions should be used and ensure comments are updated whenever the code changes
- **Write Clear and Concise Comments**: Avoid ambiguous or overly complex comments

Remediation Status
Resolved

3.8 [SN-8] Numerous unresolved TODOs

Risk Level
Severity: Informational, Likelihood: Low

Code Segment
Multiple TODO comments throughout the codebase, some related to security

checks

Description

- The code contains many ambiguous unresolved TODOs, which are also related to security checks
- TODOs in production code can indicate incomplete functionality or missing security validations
- Unresolved TODOs can lead to maintenance issues and potential security oversights

Code Location

- `blockchain-tools/src/Bech32m.sol` - Line 156
- `blockchain-tools/src/BitcoinUtils.sol` - Lines 7, 11, 163, 221
- `blockchain-tools/src/Deriver.sol` - Lines 27, 73, 123, 172
- `stroom-contracts/src/bstBTC.sol` - Line 43

Recommendation

- Track and resolve all the pending TODOs
- Prioritize TODOs related to security checks for immediate resolution
- Implement a process to prevent TODOs from reaching production code

Remediation Status

Resolved

3.9 [SN-9] Misleading function name

Risk Level

Severity: Informational, Likelihood: Low

Code Segment

```
function hexStringToAddress(
      string memory s
  ) internal pure returns (bytes memory) {
      bytes memory ss = bytes(s);
      require(ss.length % 2 == 0, "string with hex encoded data
          should have even length"); // length must be even
```

```
6       bytes memory r = new bytes(ss.length / 2);
7       for (uint i = 0; i < ss.length / 2; ++i) {
8           r[i] = bytes1(
9               fromHexChar(uint8(ss[2 * i])) *
10                  16 +
11                  fromHexChar(uint8(ss[2 * i + 1]))
12          );
13      }
14
15      return r;
16  }
```

Description

- The function converts a hex string to bytes
- However, the function name leads to understanding that the function is converting a hex string to Solidity address type
- Misleading function names can cause confusion during development and code review

Code Location

- `blockchain-tools/src/Tools.sol` - Lines 31-46

Recommendation

- Rename the function to a more meaningful name, e.g., `hexStringToBytes`
- Ensure function names accurately reflect their functionality

Remediation Status

Resolved

3.10 [SN-10] Determine specific commits of submodules dependencies

Risk Level

Severity: Medium, Likelihood: Medium

Code Segment

Blockchain-tools and stroom-contracts submodule dependencies in .gitmodules files

Description

- In Foundry, managing dependencies like submodules is crucial for ensuring reproducibility, stability, and security in your project
- When using submodules, it is generally recommended to pin them to a specific commit rather than relying on a branch
- **Pin Submodules to a Specific Commit** has the following advantages:
  - **Reproducibility**: Pinning to a specific commit ensures that everyone working on the project uses the exact same version of the submodule
  - **Stability**: A specific commit represents a stable, tested state of the submodule. Using a branch can introduce breaking changes
  - **Security**: Pinning to a commit ensures that you are using a known, reviewed version of the submodule, reducing the risk of introducing vulnerabilities

Code Location

- `blockchain-tools/.gitmodules`
- `stroom-contracts/.gitmodules`

Recommendation

- It is highly recommended to pin all the dependencies in submodule files to specific commits
- Regularly review and update pinned commits to maintain security while preserving stability

Remediation Status
Resolved.

3.11 [SN-11] Risks associated with upgradeable token contracts

Risk Level
Severity: High, Likelihood: Medium

Code Segment
stBTC and bstBTC token contracts implemented as upgradeable contracts

Description

- Tokens stBTC and bstBTC are both implemented and will be deployed as upgradeable contracts
- Upgradeable token contracts allow their logic to be upgraded after deployment. While upgradeability offers flexibility and the ability to fix bugs or add features, it also introduces several consequences and risks:
  - **Centralization Risks**: Admin control introduces a single point of failure
  - **Security Risks**: Proxy vulnerabilities and upgrade exploits can compromise the token
  - **User Trust Issues**: Users may distrust upgradeable tokens due to their mutable nature
  - **Regulatory Risks**: Upgrades may introduce non-compliant behavior, leading to legal issues
  - **Upgrade Process Risks**: Failed upgrades or governance challenges can disrupt the token's functionality
  - **Loss of Immutability**: Upgradeability sacrifices the immutability of blockchain-based systems

Code Location

- `stroom-contracts/src/stBTC.sol`
- `stroom-contracts/src/bstBTC.sol`
- `stroom-contracts/script/Deploy.s.sol`

Recommendation

Upgradeable token contracts should follow recommended best practices as follows:

- **Implement Access Control**: Restrict upgrade permissions to a multisig wallet or decentralized governance mechanism to reduce centralization risks. It is also strictly recommended to protect the upgradeable contracts by having time lock contract control over the token contracts
- **Clearly document** the purpose and impact of each upgrade to maintain transparency to maintain user trust and to avoid regulatory risks
- **Limit Upgrade Frequency**: Avoid frequent upgrades to maintain user trust and reduce the risk of introducing vulnerabilities
- It is recommended to have a clear timeline for when to revoke the contracts' upgradeability

Remediation Status
Resolved, within deploy scripts.

3.12 [SN-12] Potential signature replaying for updating jointPublicKey

Risk Level
Severity: High, Likelihood: High

Code Segment

```
// Function to update the validator public key with a signature
  function setJointPublicKeySigned(
      bytes32 _jointPublicKey,
      bytes calldata signature
  ) public {
      require(
          validateMessage(MESSAGE_UPDATE_JOINT_PUBLIC_KEY,
              abi.encodePacked(_jointPublicKey), signature),
          "ValidatorRegistry: INVALID_SIGNATURE"
      );
      jointPublicKey = _jointPublicKey;
      emit JointPublicKeyUpdated(_jointPublicKey);
  }

function validateMessage(
      bytes memory prefix,
      bytes memory data,
      bytes calldata signature
  ) public view returns (bool) {
      bytes32 hash = getMessageHash(prefix, data);
      return validateMessageHash(hash, signature);
  }

// Function to get the hash of a message based on its type
  function getMessageHash(bytes memory prefix, bytes memory data)
      public pure returns (bytes32) {
      return keccak256(abi.encodePacked(keccak256(prefix), data));
  }
```

Description

- The combined multisignature key of validators `jointPublicKey` can be updated by calling the function `setJointPublicKeySigned`

- The function verifies that the signature is a valid Schnorr signature generated by `jointPublicKey` to update a new `_jointPublicKey`
- The function however does not validate whether the signature has been verified to update `jointPublicKey` before
- A signature replaying to the function can be reproduced as follows:
  - Step 1: `jointPublicKey` is initialized as K1
  - Step 2: Validators sign to update the new `jointPublicKey` as K2 with signature S
  - Step 3: Validators sign to update the validator key from K2 back to K1
  - Step 4: Any one can replay the signature S to the function `setJointPublicKeySigned` to update validator from K1 to K2 without agreement from the validators

Code Location

- `stroom-contracts/src/lib/ValidatorRegistry.sol`

Recommendation

- Generate a random nonce and attach to the data for the function `validateMessage`
- Add a state mapping variable to save verified nonce to validate whether a signature has been verified
- Example implementation: `mapping(bytes32 => bool) public usedNonces;`

Remediation Status
Resolved

3.13 [SN-13] Use custom error to reduce gas cost

Risk Level
Severity: Informational, Likelihood: Medium

Code Segment

```
modifier onlyValidator(
      bytes memory prefix,
      bytes memory data,
      bytes calldata signature
  ) {
```

```
        require(
            validatorRegistry.validateMessage(prefix, data, signature),
            "ValidatorMessageReceiver: INVALID_SIGNATURE"
        );
        _;
    }
```

Description

- Introduced in Solidity 0.8.4, custom errors are a gas-efficient way to revert transactions with specific error data.
- Custom errors reduce gas costs because they do not store string error messages on-chain.
- They are preferable for complex conditions and reusable error definitions.

Code Location

- `stroom-contracts/src/lib/ValidatorRegistry.sol` — Lines 35, 53
- `stroom-contracts/src/lib/ValidatorMessageReceiver.sol` — Line 28
- `stroom-contracts/src/lib/UserActivator.sol` — Line 24
- `stroom-contracts/src/stBTC.sol` — Lines 119, 148, 182, 183, 185, 187, 208, 221, 222, 223, 242, 246
- `stroom-contracts/src/bstBTC.sol` — Lines 48, 59

Recommendation

- Replace `require(..., "...")` statements with custom errors to improve gas efficiency and clarity.
- Define reusable custom error types that can be leveraged across multiple contracts and modifiers.

Remediation Status
Resolved.

3.14 [SN-14] Misleading comment for function activateUser

Risk Level
Severity: Informational, Likelihood: Medium

Code Segment

```
/**
    * @dev Activates a user by setting their address as activated.
    * @param _userAddress The address of the user to be activated.
    * Emits a 'UserAddressActivated' event with the user address and
       their BTC deposit address.
    * Reverts if the user address is already activated.
    */
   function activateUser(address _userAddress) public {
       require(
           activatedAddresses[_userAddress] == false,
           "User is already activated"
       );

       activatedAddresses[_userAddress] = true;

       emit UserAddressActivated(_userAddress);
   }
```

Description

- The comment states emitting event UserAddressActivated with the user address and the user's bitcoin deposit address
- The event however only has user address as event parameter

Code Location

- `stroom-contracts/src/lib/UserActivator.sol`
  Lines 20

Recommendation

Update the comment for avoiding inconsistency of the function and the comment

Remediation Status

Resolved.

3.15 [SN-15] stBTC minter exposes centralization risks

Risk Level

Severity: High, Likelihood: High

Code Segment

```
/**
    * @dev Sets the minter address.
    * @param _minter The address of the minter.
    * @notice Only the contract owner can call this function.
    */
   function setMinter(address _minter) public onlyOwner {
       minter = _minter;
   }

   /**
    * @dev Mint new tokens.
    * Only the minter can call this function.
    * @param _amount The amount of tokens to mint.
    * @param _recipient The address that will receive the minted
        tokens.
    * @param _btcDepositId The id of the BTC deposit =
        keccak256(txHash, vout)
    */
   function mint(
       uint256 _amount,
       address _recipient,
       bytes32 _btcDepositId
   ) public whenNotPaused {
       require(msg.sender == minter, "stBTC: only minter allowed to
           mint");
       _mint(_amount, _recipient, _btcDepositId);
   }

/**
   * @notice Mints new stBTC tokens based on a valid validator
       message and stakes them into the vault.
   * @param invoice The 'MintInvoice' containing details of the mint
       operation (recipient, amount, BTC deposit ID).
   * @param signature The validator's signature, validating the mint
       operation.
   */
   function mintAndStake(
       IStBTC.MintInvoice calldata invoice,
       bytes calldata signature
   )
       public
       onlyValidator(
           MESSAGE_MINT,
           IStBTC(address(this.asset())).encodeInvoice(invoice,
               address(this.asset())),
           signature
```

```
        )
    {
        address asset = address(this.asset());

        IERC20(asset).approve(address(this), invoice.amount);

        uint256 assets = invoice.amount;
        address receiver = invoice.recipient;

        // ERC4626-deposit(uint256 assets, address receiver)
        uint256 maxAssets = maxDeposit(receiver);
        if (assets > maxAssets) {
            revert ERC4626ExceededMaxDeposit(receiver, assets,
                maxAssets);
        }

        // it must mint shares first, diluting everyone in the pool,
            then it does mint assets
        // to address(this), fixing the dilution
        uint256 shares = previewDeposit(assets);
        _mint(receiver, shares);

        IStBTC(asset).mint(invoice.amount, address(this),
            invoice.btcDepositId);
    }

/**
    * @notice Mints rewards to update the total pooled BTC supply
        based on a signed validator message.
    * @param nonce The unique identifier for the supply update,
        ensuring the correct order of updates.
    * @param delta The amount of BTC to be added to the total pooled
        supply as rewards.
    * @param signature The validator's signature, verifying the
        authenticity of the update request.
    */
    function mintRewards(
        uint nonce,
        uint delta,
        bytes calldata signature
    )
        external
        nonReentrant
        onlyValidator(
            MESSAGE_UPDATE_TOTAL_SUPPLY,
            encodeTotalSupplyUpdate(nonce, delta),
            signature
        )
```

```
81      {
82          require(
83              nonce == totalSupplyUpdateNonce,
84              "Invalid update total supply nonce"
85          );
86          totalSupplyUpdateNonce += 1;
87
88          if (delta == 0) {
89              revert("delta is 0");
90          }
91
92          address asset = address(this.asset());
93          bytes32 mintId = getTotalSupplyUpdateHash(nonce, delta);
94          IStBTC(asset).updateTotalSupply(delta, mintId);
95
96          uint256 _totalSupply = IERC20(asset).totalSupply();
97          uint256 _totalShares = this.totalSupply();
98
99          emit TotalSupplyUpdatedEvent(nonce, _totalSupply,
                _totalShares);
100     }
101
102 // TODO: remove onlyOwner after testing
103     function mintRewards(
104         uint nonce,
105         uint delta
106     ) external nonReentrant onlyOwner {
107         require(
108             nonce == totalSupplyUpdateNonce,
109             "Invalid update total supply nonce"
110         );
111         totalSupplyUpdateNonce += 1;
112
113         if (delta == 0) {
114             revert("delta is 0");
115         }
116
117         address asset = address(this.asset());
118         bytes32 mintId = getTotalSupplyUpdateHash(nonce, delta);
119         IStBTC(asset).updateTotalSupply(delta, mintId);
120
121         uint256 _totalSupply = IERC20(asset).totalSupply();
122         uint256 _totalShares = this.totalSupply();
123
124         emit TotalSupplyUpdatedEvent(nonce, _totalSupply,
                _totalShares);
125     }
```

Description

- stBTC token should be only minted by the other mint function with validator signature.
- The minter state variable exposes centralization risks that if this address is exploited, stBTC can be minted with an unlimited amount.
- Due to the decentralized nature of Bitcoin and stBTC, the contract should not have a centralized minter that can mint stBTC without validator signature
- Even though the minter address is meant to be set as the bstBTC token address, having updatable minter is still risky to the decentralization of stBTC.

Code Location

- `stroom-contracts/src/stBTC.sol`
  Lines 127-150
- `stroom-contracts/src/bstBTC.sol`
  Lines 43-66, 70-107, 109-144

Recommendation

- The stBTC contract should not have the minter state variable and the functions minting new stBTC without validator signature.
- The functions mintRewards and mintAndStake of bstBTC having validator signature inputs should call the mint function with signature of stBTC that verifies the input validator signature and mints proper amount of stBTC, instead of verifying validator signature by itself.
- The function updateTotalSupply of stBTC should accept input validator signature and verify it before minting new stBTC and updating stBTC total supply.
- It is recommended to not having the mintRewards function without validator signature to ensure that all functions that mint new stBTC should have validator signature

Remediation Status

Resolved. This contract was deprecated.

3.16 [SN-16] Lacks zero-check on input address for minter

Risk Level
Severity: Informational

Code Segment

```
/**
 * @dev Sets the minter address.
 * @param _minter The address of the minter.
 * @notice Only the contract owner can call this function.
 */
function setMinter(address _minter) public onlyOwner {
    minter = _minter;
}
```

Description
Lacks zero-check on input address

Code Location

- `stroom-contracts/src/stBTC: Lines 132-133`

Recommendation
Add zero-check for the input address

Remediation Status
Resolved.

3.17 [SN-17] Lack of zeroization of secret values in the internal state of the node after finishing a signing session

Risk Level
Severity: Medium

Code Segment
(Not explicitly shown)

Description
There is no zeroization of the node's state after the signing session ends. Memory zeroization ensures no residual information in the RAM once the signing session ends. Hence, it is important to zeroize all the secret information, for example, the shares. Specifically, the values that we recommend to zeroize is:
In the struct `SignDescriptor`, zeroize the following fields: `RPrivs`, `RPrivsCombined`, `PartialSigns`, and `SignSharesMap`.

In `BindingInfo`, zeroize the following fields: `privKey`, and `boundPrivKey`.
In `SoloSigner`, zeroize the following fields: `seedPrivKey`, and `privKeyMap`.
Although Go has a garbage collector that removes unused memory, it is important to either trigger the garbage collector to remove those values, allocate them in the stack if possible, or zeroize them manually in the code after the session is finished. A common practice in heap-stack programming languages is to zeroize the heap-allocated values and bring the rest of the values to the stack for an automatic removal.
Impact: The lack of memory zeroization allows an adversary to look at the state of the RAM for sensitive information and use it to tamper with another signing session.

Code Location

- `frost/sign_session.go`
- `frost/crypto/sign_descriptor.go`
- `frost/config/binding_info.go`
- `frost/solo_signer.go`

Recommendation
A possible implementation would be to create a `cleanState` function that changes the values mentioned in the description to zero values in the corresponding field. This function should be called in the `finish` function of the `signSession` struct.

Remediation Status
This issue was resolved in https://github.com/stroomnetwork/frost/pull/159.

3.18 [SN-18] InversePrivKey does not panic when the argument is zero

Risk Level
Low

Description
The function InversePrivKey does not panic or return an error when the argument is zero. Although there is a TODO that expresses the intention to implement this behavior, the current version of the function is not implementing it.

To reproduce the error, consider to include and execute the following test:

```
1 func TestInversePrivKey(t *testing.T) {
```

```
  // Creates a key that is zero.
  privKey := PrivKeyFromInt(0)
  assert.True(t, IsPrivKeyZero(privKey))

  // Invert the key.
  invPrivKey := InversePrivKey(privKey)
  assert.False(t, IsPrivKeyZero(invPrivKey))
}
```

Impact: The lack of handling of this situation may cause a panic, which can cause a denial of service attack.

Code Location

- `frost/crypto/crypto.go`

Recommendation

A possible implementation of this is to return an error. Consider the following version of the function:

```
func InversePrivKey(privKey *btcec.PrivateKey) (*btcec.PrivateKey,
    error){
  if IsPrivKeyZero(privKey) {
    return nil, fmt.Errorf("The private key to be inverted is zero.")
  }

  k1 := privKey.Key
  var k3 btcec.ModNScalar
  k3.Set(&k1)
  k3.InverseNonConst()
  var rezPrivKey btcec.PrivateKey
  rezPrivKey.Key = k3
  return &rezPrivKey, nil
}
```

Remediation Status

This issue was resolved in https://github.com/stroomnetwork/frost/pull/156.

3.19 [SN-19] Missing checks in Lagrange coefficients computation

Risk Level

Severity: Low

Description

The function GetLagrangeCoefficient presented next does not check that all

the points in points are different:

```
func GetLagrangeCoefficient(points []*btcec.PrivateKey, valuePoint,
    desiredPoint *btcec.PrivateKey) *btcec.PrivateKey {
    fVals := make([]*btcec.PrivateKey, 0)
    for _, p := range points {
        if !ArePrivKeysEqual(p, valuePoint) {
            fVals = append(fVals, SubPrivKey(desiredPoint, p))
        }
    }
    f := MulPrivKeys(fVals...)

    gVals := make([]*btcec.PrivateKey, 0)
    for _, p := range points {
        if !ArePrivKeysEqual(p, valuePoint) {
            gVals = append(gVals, SubPrivKey(valuePoint, p))
        }
    }
    g := MulPrivKeys(gVals...)

    return DivPrivKey(f, g)
}
```

There are three edge cases:
The list of points has duplicate points, in which case, the function should return an error.
The list of points has just one point, and in that case, the Lagrange coefficient should be 1.
The value point is not in the list of points, in which case, the function should return an error

To reproduce this error, you may execute the following test:

```
func TestEqualPonintsLagrangeCoefficients(t *testing.T) {
    // Two equal private keys.
    privKey1 := PrivKeyFromInt(1)
    privKey2 := PrivKeyFromInt(1)
    privKey3 := PrivKeyFromInt(1)
    points := []*btcec.PrivateKey{privKey1, privKey2, privKey3}

    desiredPoint := PrivKeyFromInt(1)

    // Generate the Lagrange Coefficients and test for panic.
    assert.Panics(t, func() { GetLagrangeCoefficient(points,
        privKey2, desiredPoint) })
}
```

Impact: If the points are not all different, it is possible that the function reaches a point in which it executes a division by zero, causing a possible panic.

Code Location

- `frost/crypto/crypto.go`

Recommendation

A possible solution to this issue is to create a helper function to check whether there are repeated points and use it into the GetLagrangeCoefficient function:

```
func allUniquePrivKey(keys []*btcec.PrivateKey) bool {
    found := make(map[string]bool)
    for _, privKey := range keys {
        if found[hex.EncodeToString(privKey.Serialize())] {
            return false
        }
        found[hex.EncodeToString(privKey.Serialize())] = true
    }
    return true
}

func GetLagrangeCoefficientV2(points []*btcec.PrivateKey,
    valuePoint, desiredPoint *btcec.PrivateKey) (*btcec.PrivateKey,
    error) {
    if !allUniquePrivKey(points) {
        return nil, fmt.Errorf("the points in the provided list are
            not unique")
    }

    // Check if valuePoint is in the list of points
    contained := false
    for _, point := range points {
        if point.Key.Equals(&valuePoint.Key) {
            contained = true
            break
        }
    }

    if !contained {
        return nil, fmt.Errorf("the value point is not contained in
            the list of points")
    }

    if len(points) == 1 {
        result := new(btcec.ModNScalar).SetInt(1)
        return btcec.PrivKeyFromScalar(result), nil
    }

    fVals := make([]*btcec.PrivateKey, 0)
    for _, p := range points {
        if !ArePrivKeysEqual(p, valuePoint) {
```

```
            fVals = append(fVals, SubPrivKey(desiredPoint, p))
        }
    }
    f := MulPrivKeys(fVals...)

    gVals := make([]*btcec.PrivateKey, 0)
    for _, p := range points {
        if !ArePrivKeysEqual(p, valuePoint) {
            gVals = append(gVals, SubPrivKey(valuePoint, p))
        }
    }
    g := MulPrivKeys(gVals...)

    return DivPrivKey(f, g), nil
}
```

Remediation Status
The issue was resolved in https://github.com/stroomnetwork/frost/pull/157

3.20 [SN-20] The Start function for NetworkSigner does not return errors

Risk Level
Severity: Informational

Description
The Start function in the NetworkSigner does not return errors, but stops the execution when there is a failure producing a silent stop of the execution.

Impact: Although this issue does not represent a risk for the security of the implementation, it would be appropriate to include logging errors to inform any possible failure to the user.

Code Location

- `frost/signer.go`

Remediation Status
Resolved in https://github.com/stroomnetwork/frost/commit/ca35e3df113010a57310df430

3.21 [SN-21] Missing documentation

Risk Level
Severity: Informational

Description
There are some functions that do not have documentation. In particular, the consensus package does not have significant documentation. Although the consensus package has functions whose name is self-contained and understandable, it may be relevant for library maintainers and contributors to be aware of possible edge cases or rationale inside those functions.

Other files that have some function with missing documentation are:
`frost/session.go`
`frost/signer.go`
`frost/solo_signer.go`
`frost/crypto/crypto.go`
`frost/crypto/linear_combination.go`
`frost/crypto/linear_sig.go`

Recommendation
It is important to add proper documentation to functions in order to clarify the purpose or the rationale of them for maintainers and contributors

Remediation Status
Resolved in https://github.com/stroomnetwork/frost/commit/ca35e3df113010a57310df430

3.22 [SN-22] Lack of randomness in the nonce generation

Risk Level
Severity: Medium

Description
As recommended in RFC 9591, Section 4.1, the best practice to avoid predictable nonces would be to compute a hash of the secret key along with random bytes. This will increase the security of the protocol.

Currently, the generation of rPrivs is done by hashing the private key, the public keys, and the message, along with some deterministic tags. This makes the key generation deterministic and predictable.

As expressed in RFC 9591, Section 7.1, the use of predictable nonces could lead an adversary to conduct a replay attack

Recommendation

Following the suggestion of RFC 9591, they recommend the following algorithm in pseudo-code to implement the nonce generation:

```
Inputs:
- secret, a Scalar.
Outputs:
- nonce, a Scalar.

def nonce_generate(secret):
    random_bytes = random_bytes(32)
    secret_enc = G.SerializeScalar(secret)
    return H3(random_bytes || secret_enc)
```

Here, the secret corresponds to the secret key of the party generating the nonce.

Remediation Status
The issue was resolved in https://github.com/stroomnetwork/frost/pull/160/

3.23 [SN-23] Removal of public commitments

Risk Level
Severity: Low

Description
As specified in RFC 9591, Section 5.2, and the reference FROST paper, Section 5.2, "Handling Ephemeral Outstanding Shares", the commitments to secret values must be removed at the end of a signing session. In the context of the implementation, the struct saving the rPubs must be cleaned up after the signing session is finished.

According to the FROST paper, "the reuse of the nonces can lead to the exposure of the secret key used for signing". The accidental use of the commitments could also produce a DoS if an adversary constantly uses a commitment of a previous section that does not match the discrete logarithm of the current session.

Affected Files:
`frost/sign_session.go`
`frost/crypto/sign_descriptor.go`

Recommendation
In line with the issue STR-1, the function clearState that is proposed in the mentioned issue should include a line that clears up the struct holding the rPubs. Specifically, the instances that need to be cleared are:
`RPubsCombined`

`RPubsPerSignature`
`RPubsMap`

Remediation Status
Resolved in https://github.com/stroomnetwork/frost/commit/ca35e3df113010a57310df430

3.24 [SN-24] Improve the error handling of the functions

Risk Level
Severity: Low

Description
Some functions can improve their error handling. Some of the functions return a boolean and others return silently. We suggest more informational error handling. Here we will present the current behavior of the function, and later, we will recommend an improvement to each function:

The following functions return a boolean in case of an error. Although the potential error is informed using a logging function. We think that this should also return an error to be handled up in the stack:
`calculateSignatures`
`generateRPubsPerParticipant`
`combineRs`

The following functions return nothing in case of an error. Similarly, the addition of an informative error handling will improve the quality of the code:
`progressToFinished`
`handleSignatureMessage`
`progressToCollectSignShares`
`progressToFinished`
`handleSignShareMessage`
`handleSignRefuseMessage`

The function `isValid` is a special case, which can be rewritten to remove the call to the `validationError` function. It is recommended to remove the `validationError` function completely.

Affected Files
`frost/sign_session.go`

Recommendation
We will add a possible suggestion for the `calculateSignatures` function. The other functions can be fixed similarly:

```
func (ss *signSession) calculateSignatures() ([]*schnorr.Signature, error) {
    signs := make([]*schnorr.Signature, len(ss.sd.Msd.SignDescriptors))

    for i := 0; i < len(ss.sd.Msd.SignDescriptors); i += 1 {
        combinedSs := crypto.MulPrivKey(ss.sd.HS[i], ss.sd.Msd.SignDescriptors[i].LC.GetAddTweak())
        for _, signShares := range ss.sd.SignSharesMap {
            combinedSs = crypto.AddPrivKey(combinedSs, signShares[i])
        }

        sigBytes := crypto.JoinBytes(crypto.GetPubKey32Bytes(ss.sd.RPubsCombined[i]), combinedSs.Serialize())
        sign, err := schnorr.ParseSignature(sigBytes)
        if err != nil {
            returnError := fmt.Errorf("cannot parse signature after creation: %w", err)
            ss.error(returnError)
            return nil, returnError
        }

        if !sign.Verify(ss.sd.Msd.SignDescriptors[i].MsgHash, ss.sd.Msd.SignDescriptors[i].LC.GetCombinedPubKey()) {
            returnError := fmt.Errorf("generated signature is not valid for pubkey: %x", ss.sd.Msd.SignDescriptors[i].LC.GetCombinedPubKey()
            .SerializeCompressed())
            ss.error(returnError)
            return nil, returnError
        }
        signs[i] = sign
    }
    return signs, nil
}
```

Remediation Status
Resolved in https://github.com/stroomnetwork/frost/commit/ca35e3df113010a57310df430

3.25 [SN-25] grpc.DialContext function is depreciated

Risk Level
Informational

Description

Function DialContext of grpc package is depreciated as noted at https://github.com/grpc/grpc-go/blob/a51009d1d7074ee1efcd323578064cbe44ef87e5/clientconn.go#L237
Although there are no identified flaws of using DialContext, it is recommended to use the new function NewClient to always receive related optimizations and bug fixes.

```
1 func RegisterExecutorApiHandlerFromEndpoint(ctx context.Context, mux
      *runtime.ServeMux, endpoint string, opts []grpc.DialOption) (err
      error) {
2     conn, err := grpc.DialContext(ctx, endpoint, opts...)
3     if err != nil {
4         return err
5     }
6   //...
7 }
```

Code Location
pkg/shared/http_server.go

Recommendation
Update the code for new function NewClient
or always monitor grpc package to receive any update related to the DialContext function

Remediation Status
Resolved.

3.26 [SN-26] Function potentially returns without terminating pending underlying resources

Risk Level
Medium

Description
The function FetchAllAddresses fetch all activated user addresses by fetching all UserAddressActivated events emitted on Ethereum
A new Ethereum event subscription is created to receive new activated user
When there is an error with the subscription, the go routine can return without closing the subscription to release any pending underlying resources
Not closing many subscriptions for long time could make high usage of memory

```
1 go func() {
2     batchSize := uint64(1000)
3     startHeight := w.config.StartHeight
4
```

```
    endHeight, err := w.repository.GetHeight(ctx)
    if err != nil {
        errCh <- fmt.Errorf("cannot get Ethereum height from
            repository %w", err)
        return
    }
    w.Log.Infof("Fetching user registrations between %d - %d",
        startHeight, endHeight)
    for startHeight < endHeight {
        iter, err := w.ethereumManager.FetchUserRegistrations(ctx,
            startHeight, batchSize)
        if err != nil {
            errCh <- fmt.Errorf("RECOVERY: Failed to fetch user
                registrations: %w", err)
            return
        }
        for iter.Next() {
            if iter.Error() != nil {
                errCh <- fmt.Errorf("RECOVERY: Error while fetching
                    user registrations: %w", iter.Error())
                return
            }
            //
        }
        iter.Close()
        startHeight += batchSize
    }
    close(resultCh)
}()
```

Code Location

`pkg/shared/node/watcher.go` function `FetchAllAddresses`

Recommendation

Defer the subscription close right after its creation

```
iter, err := w.ethereumManager.FetchUserRegistrations(ctx,
    startHeight, batchSize)
defer iter.Close()
if err != nil {
    errCh <- fmt.Errorf("RECOVERY: Failed to fetch user
        registrations: %w", err)
    return
}
```

Remediation Status

Resolved.

3.27 [SN-27] GetRedeemEvents function call is redundant in function IsValidRedeem

Risk Level
Low

Description
The function verifies the validity of an input redeem by fetching on-chain transaction and verify the existence of the input redeem
The function fetches the block containing the transaction of the input redeem, then fetch all RedeemBtcEvent events in the block and check the existence of the input redeem
The second step seems to be redundant as all RedeemBtcEvent events of the input transaction can be directly parsed based the transaction receipt
There is no known identified flaw with the code, the issue is related to code optimization to avoid unnecessary calls to the Ethereum node

Code Location
`pkg/shared/node/manager.go` function `IsValidRedeem`

Recommendation
Parse the transaction receipt for RedeemBtcEvent events and compare with the input redeem

```
receipt, err := m.gateway.Client.TransactionReceipt(ctx,
    ethCommon.HexToHash(signatureData.TxHash))
if err != nil || receipt == nil {
    return fmt.Errorf("cannot get transaction receipt: %w", err)
}

for _, log := range receipt.Logs {
    redeem, err := m.contracts.StBtc.ParseRedeemBtcEvent(*log)
    if err == nil && redeem.From.String() ==
        signatureData.EthereumAddress &&
        redeem.BTCAddress == signatureData.BitcoinAddress &&
        redeem.Value.Int64() == signatureData.Amount {
        return nil
    }
}
```

Remediation Status
Resolved.

3.28 [SN-28] Code maintenance: Duplicate imports and unused functions

Risk Level
Low

Description
Function `queueRedeem` in `pkg/executor/redeem/` package is unused
Unused function `boolToStr` in `pkg/shared/node/bitcoind_wallet.go`
Duplicate import of `github.com/stroomnetwork/frost/crypto` in `pkg/shared/crypto/btc.go`
Duplicate import of `"github.com/lib/pq"` in `pkg/shared/persistence/postgres.go`

Code Location

- `pkg/executor/redeem/` package
- `pkg/shared/node/bitcoind_wallet.go`
- `pkg/shared/crypto/btc.go`
- `pkg/shared/persistence/postgres.go`

Recommendation
Remove unused functions and duplicate imports to improve code quality

Remediation Status
Resolved.

3.29 [SN-29] Potential missing of mints

Risk Level
High

Description
The function `getBitcoinMints` parses mints by querying BTC receiver addresses (in the BTC deposit transactions outputs) in the local UserManager repository of the local database.
An output is accepted as a mint if the receiver address matches a record in the UserManager repository.
BTC deposits made to unactivated addresses or addresses activated after deposits will not be indexed.
Therefore, mints of stBTC for those users may be missing. This is unexpected, as the network should recognize any sufficient BTC deposit regardless of address activation timing.

Steps to reproduce the issue

1. Go to UserActivation contract on Sepolia and call `getBTCDepositAddress` for a user Ethereum address
2. Deposit BTC testnet4 to the above-generated address
3. Wait 2-3 hours, then activate the user address via https://testnet.stroom.network/mint
4. The user will not see an unfinished mint

Code Location
`pkg/executor/mint` package

Recommendation
Store all unmatched outputs in an additional repository.
When a user activates, check if any previously unmatched mints now match and process them.

Remediation Status
Resolved.

3.30 [SN-30] Insufficient Network Security Group Controls

Risk Level
Severity: High, Likelihood: High

Code Segment

```
this.asg.addSecurityGroup(new ec2.SecurityGroup(this,
    "SecurityGroup", {
    securityGroupName: `${properties.infraName}-asg-sg`,
    vpc: this.vpc,
    allowAllOutbound: true,
}))
```

Description
The Auto Scaling Group security group configuration allows all outbound traffic (`allowAllOutbound: true`), which violates the principle of least privilege. This could potentially allow unauthorized data exfiltration, malicious outbound connections and or unintentional data leakage.

Code Location
`lib/infra-stack.ts:150-155`

Recommendation
Implement specific outbound rules based on required traffic patterns.
Limit outbound traffic to expected Bitcoind, LND, stroom-go, and logging ports.

Remediation Status
Resolved

3.31 [SN-31] Insecure Secret Management

Risk Level
Severity: Low

Code Segment

```
this.secret = aws_secretsmanager.Secret.fromSecretPartialArn(this,
    "SM",
    `arn:aws:secretsmanager:eu-north-1:CENSOREDFORPUBLICATION:secret:${secretKey}`)
```

Description
The code uses hardcoded AWS account IDs and region in the secret ARN. Additionally, there's no rotation policy defined for the secrets, which could lead to long-lived credentials.

Code Location
`lib/infra-stack.ts:45-46`

Recommendation
Use environment variables or CDK context for account IDs and regions

Remediation Status
Resolved

3.32 [SN-32] Potential discrepancies between deposited BTC and redeemed strBTC when minting rewards

Risk Level
Severity: High

Code Segment

```
function mintRewards(uint256 nonce, uint256 delta, bytes calldata
    signature)
       external
       whenNotPaused
       onlyValidator(MESSAGE_UPDATE_TOTAL_SUPPLY,
           encodeTotalSupplyUpdate(nonce, delta), signature)
{
```

```
    if (nonce != totalSupplyUpdateNonce) revert
        InvalidTotalSupplyNonce();
    totalSupplyUpdateNonce += 1;

    if (delta == 0) revert DeltaIsZero();

    bytes32 rewardId = getTotalSupplyUpdateHash(nonce, delta);
    if (btcDepositIds[rewardId]) revert UpdateAlreadyProcessed();
    btcDepositIds[rewardId] = true;

    _totalPooledBTC += delta;

    emit TotalSupplyUpdatedEvent(nonce, _totalPooledBTC,
        _totalShares);
}
```

```
function redeem(uint256 _amount, string calldata BTCAddress) public
    whenNotPaused {
    if (_amount < minWithdrawAmount) revert AmountBelowMinWithdraw();
    if (!network.validateBitcoinAddress(BTCAddress)) revert
        InvalidBTCAddress();

    _burn(msg.sender, _amount);

    redeemCounter += 1;
    emit RedeemBtcEvent(msg.sender, BTCAddress, _amount,
        redeemCounter);
}
```

Description

Users deposit BTC on the Bitcoin blockchain and mint a corresponding amount of strBTC on the Ethereum blockchain through the strBTC smart contract with function mint.

The stroom network executes rebase transactions to reward users by making transactions to function mintRewards of the strBTC smart contract. Note the mintRewards function does not mint new strBTC to a specific address, instead it creates new strBTC by increasing the total supply of strBTC, thus increase the strBTC balance of all users (thus called rebase token)

When redeeming, an amount of strBTC is burnt and an exact amount of BTC on the Bitcoin blockchain will be sent to the recipient of the redeem transaction.

Example:

- User A deposits 5 BTC and mints 5 strBTC
- User B deposits 5 BTC and mints 5 strBTC
- Total supply of strBTC is 10, total BTC is 10
- mintRewards is called with delta = 2, increasing supply to 12

- User A and B now both hold 6 strBTC
- They each redeem 6 strBTC for 6 BTC, which totals 12 BTC
- This exceeds the 10 BTC actually deposited

Code Location
`stroom-contracts-rebase/src/strBTC.sol`

Recommendation

- Clarify how mintRewards input is computed and make it verifiable on-chain
- Add a threshold for rewards to avoid backend computation errors
- If possible, record mint rewards per invoice on-chain

Remediation Status
Resolved by adding checks for mint reward frequency and max percentage compared to current supply.

3.33 [SN-33] Insecure Container Configuration

Risk Level
Severity: High, Likelihood: Medium

Code Segment

```
const loadBalancer = new lb.NetworkLoadBalancer(this, `${l.name}LB`,
    {
    vpc: this.vpc,
    internetFacing: true,
    loadBalancerName: `${properties.infraName}-${l.name}`,
})
```

Description
The infrastructure exposes services directly to the internet without proper WAF (Web Application Firewall) protection.

Code Location
`lib/infra-stack.ts:180-185`

Recommendation

- Implement AWS WAF for internet-facing services
- Add container-specific security configurations

- Run containers as non-root users

Remediation Status
Resolved

3.34 [SN-34] Potential discrepancies between deposited BTC and redeemed strBTC when minting from converter and redeem

Risk Level
Severity: High

Code Segment

```
function convertWBTCToStrBTC(uint256 wbtcAmount) external
    whenNotPaused returns (uint256) {
    if (wbtcAmount == 0) revert AmountMustBeGreaterThanZero();

    uint256 strbtcAmount = (wbtcAmount * incomingRateNumerator) /
        incomingRateDenominator;
    if (strbtcAmount == 0) revert ConversionResultedInZeroTokens();

    if (currentlyMinted() + strbtcAmount > mintingLimit) revert
        MintingLimitExceeded();

    wbtc.transferFrom(msg.sender, address(this), wbtcAmount);

    strbtc.converterMint(msg.sender, strbtcAmount);

    totalMinted += strbtcAmount;

    emit WBTCConverted(msg.sender, wbtcAmount, strbtcAmount);

    return strbtcAmount;
}
```

Description
Users can mint strBTC by locking WBTC via a converter contract. This minted strBTC can be redeemed for BTC. If too much strBTC is minted via WBTC and redeemed for BTC, but there's insufficient BTC locked in the network, this creates a shortfall.
For example:

- User A deposits 5 BTC, mints 5 strBTC
- User B deposits 5 BTC, mints 5 strBTC

- User C converts 20 WBTC into strBTC
- Total supply is now 30 strBTC, but only 10 BTC are locked
- User C redeems 20 strBTC, BTC shortfall of 10

Code Location
`stroom-contracts-rebase/src/strBTC.sol`
`stroom-contracts-rebase/src/wBTCConverter.sol`

Recommendation

- Monitor WBTC locked vs. BTC deposits
- If manual bridging from WBTC to BTC is required, show clear messages to users and note that redemption may take longer

Remediation Status
Acknowledged.

3.35 [SN-35] Default admin role must be secured in production

Risk Level
Severity: Medium, Likelihood: Medium

Code Segment

```
function addConverter(address converter) external
    onlyRole(DEFAULT_ADMIN_ROLE) {
    grantRole(CONVERTER_ROLE, converter);
}

function removeConverter(address converter) external
    onlyRole(DEFAULT_ADMIN_ROLE) {
    revokeRole(CONVERTER_ROLE, converter);
}

function converterMint(address recipient, uint256 amount) external
    whenNotPaused onlyRole(CONVERTER_ROLE) {
    if (recipient == address(0)) revert CannotMintToZeroAddress();

    _mint(recipient, amount);

    emit ConverterMint(msg.sender, recipient, amount);
}
```

Description

- The default admin role (DEFAULT_ADMIN_ROLE) is granted permissions to execute sensitive administrative functions, including adding and removing WBTC converter, which allows minting of strBTC.
- If the default admin role is compromised, an attacker can mint unlimited tokens or manipulate roles without restriction.

Code Location

`stroom-contracts-rebase/src/strBTC.sol`

Recommendation

- Assign the Default Admin Role to a Multisignature Wallet
- Deploy a timelock contract (e.g., OpenZeppelin's TimelockController) and assign the *DEFAULT_ADMIN_ROLE* to it

Remediation Status

Resolved as of 5PM CET on August 10th, 2025* on Ethereum, the following addresses were reviewed with the following on-chain state.

Contract Address	Control Mechanism	Notes / Transactions
0xb2723d5df98689eca6a4e7321121662ddb9b3017	Timelock (0x22bc85...)	- Added `0x5619...1352` as `CONVERTER_ROLE` (tx) - Added Timelock to `ADMIN_ROLE` (tx) - Removed old admin (tx)
0xa3ca88cfb7bbe9cfbd47df053ffa2130c7e6f770	None	No privileged roles or ownership set.
0x8cdb1eab36096f4891299c7d1b8dee777ae10e36	Timelock (0x22bc85...)	Ownership transferred to Timelock (tx)
0xdc738b2f70e72904f365bd97e694d56b53430aa3	Timelock (0x22bc85...)	Ownership transferred to Timelock (tx)
0x56192f14c1d84e41db3d5d4c5d407efdb5cb1352	None	No owner or privileged roles. Has a cap of 500 WBTC.
0xe386a2a6d97eb9bd65ec42587af019047353d89d	Timelock (0x22bc85...)	Ownership transferred to Timelock (tx)

Table 3: On-chain verification of privileged control for audited contracts.

* Access control is a fluid element and can be modified.

.

4 Unit Tests

4.1 stroom-contracts Repository

All unit tests have passed successfully. A total of 31 tests were executed across multiple test suites with 0 failures and 0 skipped tests.

- **Test Summary:** 31 tests passed, 0 failed, 0 skipped
- **Test Files:** `Bip340Mint.t.sol`, `UserActivator.t.sol`, `stBTC.t.sol`

Test Coverage

While deployment and setup scripts (e.g., `Deploy.s.sol`, `SetSeed.s.sol`) are excluded from the analysis, the remaining smart contract files show strong test coverage:

- `src/bstBTC.sol`: 92.5% lines, 94.5% statements
- `src/lib/UserActivator.sol`: 100% lines, 100% statements
- `src/stBTC.sol`: 58.82% lines, 59.57% statements

Conclusion: Excluding scripts used for deployment, the smart contracts in the stroom-contracts repo are generally well covered by unit tests.

4.2 blockchain-tools Repository

A total of 65 tests passed with 0 failures and 0 skips across 7 test suites. All tests completed successfully.

- **Test Summary:** 65 tests passed, 0 failed, 0 skipped
- **Test Files:** `BitcoinUtils`, `Bech32m`, `BTCDepositAddressDeriver`, etc.

Test Coverage

Excluding scripts, the test coverage is robust across core files:

- `src/BTCDepositAddressDeriver.sol`: 88.46% lines
- `src/BitcoinUtils.sol`: 85.71% lines
- `src/Deriver.sol`: 100% lines
- `src/Bech32m.sol`: 75.80% lines

Conclusion: Deployment scripts remain uncovered, but the actual application logic is well-tested.

4.3 FROST Library Coverage

Package	Coverage (%)
approver	61.10
config	83.00
consensus	68.50
crypto	83.40
network	82.10
storage	60.80
Total	**79.20**

Table 4: FROST library test coverage by package

Observation:
The overall test coverage for the FROST library is 79.20%. While packages such as `config`, `crypto`, and `network` exceed 80%, important modules like `consensus`, `storage`, and `approver` fall below this threshold.

Recommendation:
Increase test coverage in lower-covered packages by including tests for failure paths, edge cases, and unintended behaviors.

5 Citations

- Komlo, C., & Goldberg, I. (2021). *FROST: Flexible Round-Optimized Schnorr Threshold Signatures*. In O. Dunkelman, M. J. Jacobson Jr, & C. O'Flynn (Eds.), *Selected Areas in Cryptography* (pp. 34–65). Cham: Springer International Publishing.
- Katz, J., & Lindell, Y. (2021). *Introduction to Modern Cryptography* (3rd ed.). CRC Press.
- Cramer, R., Damgård, I., & Nielsen, J. B. (2015). *Secure Multiparty Computation and Secret Sharing*.

5.1 Web Resources and Repositories

1. Stroom Network FROST Repository - Pull Request #159. https://github.com/stroomnetwork/frost/pull/159
2. Stroom Network FROST Repository - Pull Request #156. https://github.com/stroomnetwork/frost/pull/156
3. Stroom Network FROST Repository - Pull Request #157. https://github.com/stroomnetwork/frost/pull/157
4. Stroom Network FROST Repository - Pull Request #160. https://github.com/stroomnetwork/frost/pull/160/
5. Stroom Network FROST Repository - Commit ca35e3d. https://github.com/stroomnetwork/frost/commit/ca35e3df113010a57310df430d4ceb942f3d0be5
6. gRPC Go ClientConn Documentation. https://github.com/grpc/grpc-go/blob/a51009d1d7074ee1efcd323578064cbe44ef87e5/clientconn.go#L237
7. Stroom Network Testnet Mint Interface. https://testnet.stroom.network/mint

6 Contact Information

For further inquiries or to schedule a follow-up audit, please contact:
Arcadia Agency

Email: audits@arcadiamgroup.com
Website: https://arcadia.agency/
Telegram: https://t.me/thearcadiagroup

7 Disclaimer

While best efforts and precautions have been taken in preparing this document, Arcadia and the Authors assume no responsibility for errors, omissions, or damages resulting from the use of the provided information. Additionally, Arcadia would like to emphasize that the use of Arcadia's services does not guarantee the security of a smart contract or set of smart contracts and does not guarantee against attacks. One audit on its own is not enough for a project to be considered secure; that categorization can only be earned through extensive peer review and battle testing over an extended period.