// Security Assessment

01.19.2026 - 02.20.2026

VerifiedX-Core

VerifiedX

Submit Feedback

HALBORN

VerifiedX-Core - VerifiedX

Prepared by: HALBORN

Last Updated 02/27/2026

Date of Engagement: January 19th, 2026 - February 20th, 2026

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
23	6	5	10	2	0

TABLE OF CONTENTS

1. Introduction

`VerifiedX` engaged Halborn to conduct a security assessment on their vBTC V2 system from January 19th, 2026 to February 20th, 2026. The security assessment scope was limited to the system components provided to Halborn. Commit hashes and further details can be found in the Scope section of this report.

`VerifiedX vBTC V2` is a tokenized Bitcoin system built on the VerifiedX (VFX) blockchain. It uses FROST threshold Schnorr signatures (via a Rust FFI library) to coordinate distributed key generation (DKG) for Taproot (P2TR) deposit addresses and to collectively sign Bitcoin withdrawals. The system integrates on-chain consensus/state transitions (VFX transactions and state tree updates) with Bitcoin coordination (validator discovery, MPC ceremony orchestration, ElectrumX-based Bitcoin network interactions).

2. Assessment Summary

Halborn was allocated 25 days for this engagement and assigned one full-time security engineer to conduct a comprehensive review of the system components within scope. The engineer(s) have expertise in blockchain protocol security, Bitcoin coordination security, and cryptographic integration, with advanced skills in penetration testing and exploitation of node-side APIs and MPC systems.

The objectives of this assessment were to:

- Identify potential security vulnerabilities within the system.
- Verify that the system functionality operates as intended.

In summary, Halborn identified several areas for improvement to reduce the likelihood and impact of risks, which were fully addressed by the `VerifiedX team`. The main recommendations were:

- `Implement and enforce end-to-end MPC correctness: ensure DKG yields valid Bech32m P2TR addresses derived from the group key, and ensure signing produces verifiable Schnorr witnesses over correct BIP341 sighashes (per-input as required), failing closed on any mismatch.`
- `Ensure consensus/state coverage matches the protocol for all vBTC V2 transaction types, binding state transitions to consensus-validated transaction properties and preventing “no-op” or divergent behavior.`
- `Harden validator/MPC networking surfaces (validator discovery, FROST endpoints, ElectrumX I/O) with strict input validation, bounded resource usage, timeouts, and clear operational runbooks to prevent liveness failures and abuse.`

3. Test Approach And Methodology

Halborn conducted a combination of manual code review and targeted automated security checks to balance efficiency, timeliness, practicality, and accuracy within the scope of this assessment. While manual testing is crucial for identifying flaws in logic, processes, and implementation, targeted automation enhances coverage and helps detect deviations from established security best practices.

The following phases were employed throughout the term of the assessment:

- Research into the platform's architecture, purpose, and intended multi-node operational model.
- Manual code review and walkthrough of scoped system components to identify logical issues and security risks.
- Review of critical cryptographic integration points (FROST DKG/signing flows, Taproot transaction signing assumptions, FFI boundaries).
- Review of networked coordination surfaces and input validation (validator discovery, FROST endpoints, ElectrumX interactions), with emphasis on liveness and DoS resilience.
- Consistency analysis between documentation/runbooks and implementation behavior for multi-node execution.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: VerifiedX-Core

(b) Assessed Commit ID: 7a2e38b

(c) Items in scope:

- ReserveBlockCore/Controllers/ActionFilterController.cs
- ReserveBlockCore/Bitcoin/Controllers/VBTCController.cs
- ReserveBlockCore/Bitcoin/Services/VBTCService.cs
- ReserveBlockCore/Bitcoin/Services/VBTCThresholdCalculator.cs
- ReserveBlockCore/Bitcoin/Services/BitcoinTransactionService.cs
- ReserveBlockCore/Services/BlockTransactionValidatorService.cs
- ReserveBlockCore/Services/BlockValidatorService.cs
- ReserveBlockCore/Data/StateData.cs
- ReserveBlockCore/Models/SmartContractStateTrei.cs
- ReserveBlockCore/Models/SmartContractStateTreiTokenizationTX.cs
- ReserveBlockCore/Services/TransactionValidatorService.cs
- ReserveBlockCore/Models/Transaction.cs
- ReserveBlockCore/Data/AccountData.cs
- ReserveBlockCore/Bitcoin/Models/VBTCContractV2.cs
- ReserveBlockCore/Bitcoin/Models/VBTCTransferInput.cs
- ReserveBlockCore/Bitcoin/Models/VBTCValidator.cs
- ReserveBlockCore/Bitcoin/Models/VBTCWithdrawalRequest.cs
- ReserveBlockCore/Bitcoin/Models/VBTCWithdrawalCancellation.cs
- ReserveBlockCore/Bitcoin/Models/MPCCeremonyState.cs
- ReserveBlockCore/Bitcoin/Services/FrostMPCService.cs
- ReserveBlockCore/Services/ValidatorService.cs
- ReserveBlockCore/Bitcoin/FROST/FrostStartup.cs
- ReserveBlockCore/Bitcoin/FROST/FrostServer.cs
- ReserveBlockCore/Bitcoin/FROST/FrostNative.cs
- ReserveBlockCore/Bitcoin/FROST/Models/FrostMessages.cs
- ReserveBlockCore/Bitcoin/FROST/Models/FrostSessions.cs
- ReserveBlockCore/Bitcoin/Integrations/MempoolSpaceTestnet4.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Client.cs
- ReserveBlockCore/Bitcoin/ElectrumX/ClientService.cs
- ReserveBlockCore/Bitcoin/ElectrumX/ClientSettings.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Converter.cs
- ReserveBlockCore/Bitcoin/ElectrumX/IdCounter.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Subscriber.cs

- ReserveBlockCore/Bitcoin/ElectrumX/Request/RequestBase.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/BlockchainEstimateFeeRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/BlockchainBlockHeaderGetRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/BlockchainHeadersSubscribeRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/BlockchainScripthashGetBalance.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/BlockchainScripthashGetHistoryRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/BlockchainScripthashListunspentRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/BlockchainScripthashSubscribeRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/BlockchainTransactionBroadcastRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/BlockchainTransactionGetMerkleRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/BlockchainTransactionGetRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/ServerPingRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Request/ServerVersionRequest.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/ResponseBase.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/Error.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainEstimatefeeResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainBlockHeaderGetResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainHeadersSubscribeResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainScripthashGetBalanceResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainScripthashGetHistoryResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainScripthashListunspentResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainScriphashSubscribeResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainTransactionBroadcastResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainTransactionGetConfirmsResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainTransactionGetMerkleResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/BlockchainTransactionGetResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/ServerPingResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Response/ServerVersionResponse.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/BlockchainEstimatefeeResult.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/BlockchainBlockHeaderGetResult.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/BlockchainHeadersSubscribeResult.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/BlockchainScripthashGetBalanceResult.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/BlockchainScripthashGetHistoryResult.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/BlockchainScripthashListunspentResult.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/BlockchainTransactionBroadcastResult.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/BlockchainTransactionGetConfirmsResult.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/BlockchainTransactionGetMerkleResult.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/BlockchainTransactionGetResult.cs
- ReserveBlockCore/Bitcoin/ElectrumX/Results/ServerVersionResult.cs

Out-of-Scope: Third party dependencies.

REMEDIATION COMMIT ID:

- f7bf16c
- 8187862
- 6cadea9
- 45d59fc
- 3b142ba
- 331c854
- 9a007b3
- 2accf60
- b16762a
- 372c027
- f576bc1
- a6aff5a
- d57ff37
- a659575
- 4222bd4
- 87c2fde
- 5a1a169
- 2bab0a9
- bd73934
- a621c46
- 427604c
- d29a78a
- 549e809

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
6	5	10	2	0

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT BLOCK-LEVEL TRANSACTION PARSING CAN MAKE VALID VBTC V2 VALIDATOR LIFECYCLE TRANSACTIONS UNMINEABLE	CRITICAL	SOLVED - 02/06/2026
A BROKEN VALIDATOR HEARTBEAT CHECK CAN CAUSE THE ACTIVE VALIDATOR SET TO DECAY TO ZERO AND BREAK VBTC V2 LIVENESS	CRITICAL	SOLVED - 02/06/2026
INCONSISTENT CLIENT–SERVER API CONTRACTS CAN BREAK FROST DKG AND SIGNING CEREMONIES AND PREVENT CORE VBTC V2 FLOWS	CRITICAL	SOLVED - 02/15/2026
NON-DETERMINISTIC VALIDATOR SERVICE STARTUP CAN PREVENT THE FROST SERVER FROM RUNNING AND BREAK VBTC V2 MPC LIVENESS	CRITICAL	SOLVED - 02/18/2026
INCORRECT TAPROOT SIGNING SEMANTICS CAN PRODUCE INVALID BITCOIN WITHDRAWALS AND LEAVE FUNDS STUCK	CRITICAL	SOLVED - 02/16/2026
INSUFFICIENT CONSENSUS VALIDATION OF TRANSFER DATA ENABLES ARBITRARY VBTC V2 BALANCE MANIPULATION	CRITICAL	SOLVED - 01/29/2026
MISSING SENDER BINDING IN THE VBTC V2 WITHDRAWAL LIFECYCLE ENABLES UNAUTHORIZED STATE TRANSITIONS AND UNBACKED BURNS	HIGH	SOLVED - 01/29/2026
INSUFFICIENT VALIDATION OF VALIDATOR NETWORK ENDPOINTS CAN ENABLE SSRF AND INTERNAL NETWORK TARGETING	HIGH	SOLVED - 02/06/2026
INSUFFICIENT VALIDATION OF FROST PROTOCOL INPUTS CAN EXPOSE NATIVE FFI TO CRASH AND RESOURCE-EXHAUSTION RISKS	HIGH	SOLVED - 02/15/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
WEAK DKG SESSION AUTHORIZATION CAN ENABLE CEREMONY SABOTAGE AND QUORUM MANIPULATION	HIGH	SOLVED - 02/15/2026
MISSING CONSENSUS SUPPORT FOR WITHDRAWAL CANCELLATION CAN LEAVE FUNDS STUCK AND PREVENT RECOVERY GOVERNANCE	HIGH	SOLVED - 02/16/2026
INCORRECT REPLAY-PREVENTION KEYING CAN ENABLE CROSS-USER COLLISIONS AND DENIAL OF SERVICE	MEDIUM	SOLVED - 01/29/2026
NON-CANONICAL ADDRESS ENCODINGS CAN ENABLE ADDRESS ALIASING AND USER-ASSISTED FUND LOSS	MEDIUM	SOLVED - 02/06/2026
MISSING STATE-TRANSITION HANDLING CAN CAUSE VBTC V2 TRANSFERS TO BURN FEES WITHOUT APPLYING ON-CHAIN BALANCE UPDATES	MEDIUM	SOLVED - 02/13/2026
MISSING CONSENSUS-STATE HANDLING CAN CAUSE VBTC V2 CONTRACT CREATION TO SUCCEED LOCALLY BUT HAVE NO ON-CHAIN EFFECT	MEDIUM	SOLVED - 02/16/2026
AMBIGUOUS LOCAL CONTRACT RECORDS CAN BYPASS WITHDRAWAL AUTHORIZATION AND ENABLE UNAUTHORIZED VBTC V2 WITHDRAWALS	MEDIUM	SOLVED - 01/29/2026
PLACEHOLDER MPC ARTIFACTS CAN BREAK DEPOSIT ADDRESS GENERATION AND WITHDRAWAL SIGNING FOR VBTC V2	MEDIUM	SOLVED - 02/24/2026
HARDCODED DEPOSIT ADDRESS OUTPUTS CAN MISDIRECT VBTC DEPOSITS AND CAUSE IRREVERSIBLE FUND LOSS	MEDIUM	SOLVED - 02/20/2026
UNRESTRICTED FROST SESSION CREATION CAN ENABLE REMOTE RESOURCE EXHAUSTION AND DEGRADE MPC AVAILABILITY	MEDIUM	SOLVED - 02/15/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
VBTC V2 VALIDATOR REGISTRATION AND EXIT FLOW BLOCKED BY ZERO FEE REQUIREMENT CONFLICTING WITH GLOBAL TRANSACTION VALIDATION	MEDIUM	SOLVED - 01/29/2026
INSUFFICIENT TRUST AND I/O HARDENING FOR ELECTRUMX CAN ALLOW CHAIN-DATA TAMPERING AND WITHDRAWAL DISRUPTION	MEDIUM	SOLVED - 02/16/2026
WITHDRAWAL STATE MACHINE CAN ENTER A TERMINAL STATE THAT BLOCKS FUTURE WITHDRAWALS	LOW	SOLVED - 01/29/2026
INACCURATE DEPLOYMENT DOCUMENTATION CAN CAUSE MISCONFIGURATION AND BREAK VBTC V2 MPC LIVENESS	LOW	SOLVED - 02/18/2026

7. FINDINGS & TECH DETAILS

7.1 INCORRECT BLOCK-LEVEL TRANSACTION PARSING CAN MAKE VALID VBTC V2 VALIDATOR LIFECYCLE TRANSACTIONS UNMINEABLE

// CRITICAL

Description

vBTC V2 validator participation depends on on-chain registration and exit transactions (`TransactionType.VBTC_V2_VALIDATOR_REGISTER` / `VBTC_V2_VALIDATOR_EXIT`). These transactions are explicitly constructed and accepted by `TransactionValidatorService.VerifyTX()` (so they can enter the mempool).

However, during block validation, `BlockValidatorService.ValidateBlock()` runs a generic "duplicate smart-contract tx" guard on most non-basic transaction types. That guard calls `TransactionUtility.GetSCTXFunctionAndUID()` and **fails the entire block** (`return false`) if `tx.Data` does not contain `ContractUID` and `Function`.

Because vBTC V2 validator register/exit transaction payloads do not include `ContractUID` or `Function` fields (they carry `ValidatorAddress`, `IPAddress`, etc.), `GetSCTXFunctionAndUID()` returns false and blocks that include these transactions become invalid on nodes that run this code. Practically, this makes vBTC V2 validator register/exit transactions **unmineable**, preventing the vBTC V2 validator set from forming (and undermining MPC flows that depend on it).

Code Reference

- `ValidatorService` constructs `VBTC_V2_VALIDATOR_REGISTER` without `ContractUIDFunction` keys in `Data`:

Copy Code

```
// Create VBTC_V2_VALIDATOR_REGISTER transaction
var registerTx = new Transaction
{
    Timestamp = TimeUtil.GetTime(),
    FromAddress = validator.Address,
    ToAddress = validator.Address,  // Self transaction
    Amount = 0M,
    Fee = 0M,  // FREE transaction
    Nonce = sTreiAcct.Nonce,
    TransactionType = TransactionType.VBTC_V2_VALIDATOR_REGISTER,
    Data = JsonConvert.SerializeObject(new
    {
        ValidatorAddress = validator.Address,
        IPAddress = ipAddress,
        FrostPublicKey = "PLACEHOLDER_FROST_PUBLIC_KEY",  // TODO: Replace with actual FROST key generat
        RegistrationBlockHeight = Globals.LastBlock.Height,
        Signature = signature
    })
};
```

- `TransactionUtility.GetSCTXFunctionAndUID()` returns false unless `tx.Data` contains both `ContractUID` and `Function` (either as JSON array element 0, or as a JSON object):

Copy Code

```
public static (bool, bool, string, string, JArray?) GetSCTXFunctionAndUID(Transaction tx)
{
    string scUID = "";
    string function = "";
    bool skip = false;
    JToken? scData = null;
    JArray? scDataArray = null;
    try
    {
        scDataArray = JsonConvert.DeserializeObject<JArray>(tx.Data);
        scData = scDataArray[0];

        function = (string?)scData["Function"];
        scUID = (string?)scData["ContractUID"];
        skip = true;
    }
    catch { }

    try
    {
        if (!skip)
        {
            var jobj = JObject.Parse(tx.Data);
            scUID = jobj["ContractUID"]?.ToObject<string?>();
            function = jobj["Function"]?.ToObject<string?>();
            if (function == "TransferCoinMulti()")
                scUID = "NA"; // this is so the process tx mempool doesn't fail.
        }
    }
    catch { }

    if(!string.IsNullOrEmpty(scUID) && !string.IsNullOrEmpty(function))
    {
        return (true, skip, scUID, function, scDataArray);
    }

    return (false, skip, "FAIL", "FAIL", scDataArray);
}
```

- `BlockValidatorService.ValidateBlock()` applies this function parsing guard to most non-basic transaction types and returns false if parsing fails:

Copy Code

```
//check for duplicate tx
if (blkTransaction.TransactionType != TransactionType.TX &&
    blkTransaction.TransactionType != TransactionType.ADNR &&
    blkTransaction.TransactionType != TransactionType.VOTE &&
    blkTransaction.TransactionType != TransactionType.VOTE_TOPIC &&
    blkTransaction.TransactionType != TransactionType.DSTR &&
    blkTransaction.TransactionType != TransactionType.RESERVE &&
    blkTransaction.TransactionType != TransactionType.NFT_SALE)
{
    if (blkTransaction.Data != null)
    {
        try
        {
            AccountStateTrei? stateTreiAcct = null;
            stateTreiAcct = StateData.GetSpecificAccountStateTrei(blkTransaction.FromAddress);
            var scInfo = TransactionUtility.GetSCTXFunctionAndUID(blkTransaction);
            if (!scInfo.Item1)
                return false;
```

Impact

This creates a consensus-level availability failure for vBTC V2 validator lifecycle transactions:

- **Validator set formation blocked**: registration/exit transactions cannot be included in blocks accepted by the network, so the validator registry never reaches a usable on-chain state.
- **MPC flows degraded/blocked**: FROST coordination depends on a reliable validator set; inability to register/exit creates stale or empty validator lists and breaks DKG/signing availability assumptions.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:M/Y:M (10.0)

Recommendation

It is recommended to align block validation with the actual schema of vBTC V2 validator lifecycle transaction types so that smart-contract parsing assumptions are not applied to non-smart-contract transaction families. It is also recommended to implement any required duplicate or consistency checks for `VBTC_V2_VALIDATOR_REGISTER` and `VBTC_V2_VALIDATOR_EXIT` using schema-aware validation that does not depend on ContractUID/Function parsing, and to ensure mempool admission and block validation enforce consistent rules so transactions accepted into the mempool remain mineable.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by exempting `VBTC_V2_VALIDATOR_REGISTER`, `VBTC_V2_VALIDATOR_EXIT` and `VBTC_V2_VALIDATOR_HEARTBEAT` from smart-contract parsing in `BlockValidatorService.ValidateBlock()`, making them mineable without requiring ContractUID or Function fields.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/f7bf16cde530d66a802e508068cb69e9fcaabeff

7.2 A BROKEN VALIDATOR HEARTBEAT CHECK CAN CAUSE THE ACTIVE VALIDATOR SET TO DECAY TO ZERO AND BREAK VBTC V2 LIVENESS

// CRITICAL

Description

vBTC V2 relies on an “active validator set” concept (validators are considered active if their `LastHeartbeatBlock` is within the last 1000 blocks). This active set is used for critical operations like the FROST DKG ceremony (contract creation) and is exposed via ValAPI for non-validator nodes.

However, the background heartbeat loop (`VBTCValidatorHeartbeatService`) attempts to ping each validator using a URL path that does not exist in the ValAPI routing (`/validatorapi/ValidatorController/Ping`). As a result, heartbeats for remote validators are never updated, so after ~1000 blocks from registration the “active” set will naturally decay toward zero, breaking vBTC V2 operations that require active validators.

This is reachable in normal operation (no attacker required), and results in a systemic availability failure for vBTC V2 MPC ceremonies and any flow that depends on “active validators within last 1000 blocks”.

Code Reference

- Heartbeat loop calls a non-existent endpoint path (`/validatorapi/ValidatorController/Ping`):

Copy Code

```
try
{
    // Ping validator's ValAPI
    var url = $"http://{validator.IPAddress}:{Globals.ValAPIPort}/validatorapi/ValidatorControlle
    var response = await httpClient.GetAsync(url);

    if (response.IsSuccessStatusCode)
    {
        // Validator is alive - update heartbeat
        validator.LastHeartbeatBlock = Globals.LastBlock.Height;
        VBTCValidator.SaveValidator(validator);
```

- ValAPI uses the `valapi/` route prefix, and `ValidatorController` exposes `GetActiveValidators` but no `Ping` route:

Copy Code

```
[Route("valapi/[controller]")]
[ApiController]
public class ValidatorController : ControllerBase
```

Copy Code

```
[HttpGet]
[Route("GetActiveValidators")]
[ProducesResponseType(typeof(object), StatusCodes.Status200OK)]
public async Task<string> GetActiveValidators()
{
    // Get validators with recent heartbeat (within 1000 blocks)
```

```
    var currentBlock = Globals.LastBlock.Height;
    var activeValidators = Bitcoin.Models.VBTCValidator.GetActiveValidatorsSinceBlock(currentBlock - 100
    // ...
}
```

- "Active" is defined in code as `IsActive && LastHeartbeatBlock >= currentBlock - 1000`:

Copy Code

```
public static List<VBTCValidator>? GetActiveValidatorsSinceBlock(long blockHeight)
{
    var validators = GetDb();
    if (validators != null)
    {
        var validatorList = validators.Find(x => x.IsActive && x.LastHeartbeatBlock >= blockHeight).ToLi
        if (validatorList.Any())
        {
            return validatorList;
        }
    }

    return null;
}
```

- Contract creation (FROST DKG ceremony) depends on this "active since last 1000 blocks" filter; if it returns empty, the ceremony fails:

Copy Code

```
// Step 1: Get list of active validators
var currentBlock = Globals.LastBlock.Height;
List<VBTCValidator>? activeValidators;

if (!string.IsNullOrEmpty(Globals.ValidatorAddress))
{
    activeValidators = VBTCValidator.GetActiveValidatorsSinceBlock(currentBlock - 1000);
}
else
{
    activeValidators = await VBTCValidator.FetchActiveValidatorsFromNetwork();
}

if (activeValidators == null || !activeValidators.Any())
{
    ceremony.Status = CeremonyStatus.Failed;
    ceremony.ErrorMessage = "No active validators available for vBTC V2 contract creation.";
    // ...
    return;
}
```

Impact

- **vBTC V2 liveness failure**: after ~1000 blocks from initial validator registration, validators will no longer be considered "active" unless their heartbeat is updated, which breaks MPC ceremonies that require a minimum active set.
- **Operational fragility**: nodes may disagree on validator "activeness" depending on local DB state and whether the broken heartbeat loop is running, increasing inconsistent behavior across deployments.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:M/Y:M (10.0)

Recommendation

We recommend fixing the heartbeat mechanism to call an actually implemented endpoint under the ValAPI routing (for example, add a simple `GET /valapi/ValidatorController/Ping` that returns 200, or update the heartbeat loop to call an existing route). Additionally, ensure heartbeat updates are based on a consistent, protocol-defined signal (and consider marking validators inactive after a grace period of failed checks, rather than leaving `IsActive` true indefinitely) and add an integration test that runs a heartbeat cycle against a real ValAPI instance and asserts `LastHeartbeatBlock` advances for reachable validators.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by updating the heartbeat URL in VBTCValidatorHeartbeatService to call an existing ValAPI endpoint.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/818786292dfb3d02b8c5a4aaf199c57247bf4400

7.3 INCONSISTENT CLIENT–SERVER API CONTRACTS CAN BREAK FROST DKG AND SIGNING CEREMONIES AND PREVENT CORE VBTC V2 FLOWS

// CRITICAL

Description

vBTC V2 relies on FROST MPC ceremonies coordinated by `FrostMPCService` (client/orchestrator) and executed by each validator's `FrostServer` + `FrostStartup` HTTP endpoints. The current implementation contains multiple protocol integration mismatches:

- `FrostMPCService` calls endpoints that are not implemented by `FrostStartup` (`POST /frost/dkg/round2/{sessionId}`), causing ceremony progression to fail even for honest validators.
- `FrostMPCService` deserializes `GET /frost/dkg/round1/{sessionId}` responses into a `FrostDKGRound1Message`, but the server returns an object containing a `Commitments` dictionary and metadata fields, so commitment collection yields empty/invalid results.
- Similar method/path mismatches exist in the signing ceremony flow (coordinator expects `GET` endpoints with `{sessionId}` route parameters that are not provided by the server).

As a result, DKG (deposit address generation) and signing (withdrawal signing) ceremonies cannot reliably complete, preventing vBTC V2 contract creation and withdrawals from functioning.

Code Reference

- Coordinator expects a `FrostDKGRound1Message` from `GET /frost/dkg/round1/{sessionId}` and extracts `CommitmentData` from it:

Copy Code

```
private static async Task<Dictionary<string, string>?> CollectDKGRound1Commitments(
    string sessionId,
    List<VBTCValidator> validators)
{
    try
    {
        var commitments = new Dictionary<string, string>();
        // ...
        foreach (var validator in validators)
        {
            try
            {
                var url = $"http://{validator.IPAddress}:{Globals.FrostValidatorPort}/frost/dkg/round1/{
                var response = await _httpClient.GetAsync(url);

                if (response.IsSuccessStatusCode)
                {
                    var message = await response.Content.ReadFromJsonAsync<FrostDKGRound1Message>();
                    if (message != null && message.SessionId == sessionId)
                    {
                        commitments[validator.ValidatorAddress] = message.CommitmentData;
                    }
                }
            }
            catch (Exception ex)
```

```
            {
                // ...
            }
        }
        // ...
        return commitments.Count > 0 ? commitments : null;
    }
    catch (Exception ex)
    {
        // ...
        return null;
    }
}
```

- Server returns a different response schema for `GET /frost/dkg/round1/{sessionId}` (wrapper object with `Commitments` dictionary and metadata), not a `FrostDKGRound1Message`:

Copy Code

```
endpoints.MapGet("/frost/dkg/round1/{sessionId}", async context =>
{
    try
    {
        var sessionId = context.Request.RouteValues["sessionId"] as string;
        // ...
        var commitments = session.Round1Commitments.ToDictionary(kvp => kvp.Key, kvp => kvp.Value);
        var requiredCount = (int)Math.Ceiling(session.ParticipantAddresses.Count * (session.RequiredThre

        context.Response.StatusCode = StatusCodes.Status200OK;
        await context.Response.WriteAsync(JsonConvert.SerializeObject(new
        {
            Success = true,
            Message = "Round 1 commitments retrieved",
            SessionId = sessionId,
            Commitments = commitments,
            CommitmentCount = commitments.Count,
            RequiredCount = requiredCount,
            ThresholdReached = commitments.Count >= requiredCount
        }, Formatting.Indented));
    }
    catch (Exception ex)
    {
        // ...
    }
});
```

- Coordinator attempts to drive Round 2 via a non-existent endpoint `POST /frost/dkg/round2/{sessionId}`):

Copy Code

```
private static async Task<bool> CoordinateShareDistribution(
    string sessionId,
    List<VBTCValidator> validators,
    Dictionary<string, string> commitments)
{
    try
    {
        // ...
        var tasks = validators.Select(async validator =>
        {
            try
            {
                var url = $"http://{validator.IPAddress}:{Globals.FrostValidatorPort}/frost/dkg/round2/{
                var content = new StringContent(commitmentPayload, Encoding.UTF8, "application/json");
                var response = await _httpClient.PostAsync(url, content);
                return response.IsSuccessStatusCode;
            }
            catch
            {
```

```
                return false;
            }
        });
        // ...
        return successCount >= (validators.Count * 2 / 3);
    }
    catch (Exception ex)
    {
        // ...
        return false;
    }
}
```

- Server provides `/frost/dkg/share` and `/frost/dkg/round3`, but does not implement `/frost/dkg/round2/{sessionId}`:

Copy Code

```
endpoints.MapPost("/frost/dkg/share", async context =>
{
    try
    {
        using (var reader = new StreamReader(context.Request.Body))
        {
            var body = await reader.ReadToEndAsync();
            var share = JsonConvert.DeserializeObject<FrostDKGShareMessage>(body);
            // ... placeholder ...
            context.Response.StatusCode = StatusCodes.Status200OK;
            await context.Response.WriteAsync(JsonConvert.SerializeObject(new
            {
                Success = true,
                Message = "Share received and verified",
                SessionId = share.SessionId,
                FromValidator = share.FromValidatorAddress
            }, Formatting.Indented));
        }
    }
    catch (Exception ex)
    {
        // ...
    }
});

endpoints.MapPost("/frost/dkg/round3", async context =>
{
    // ...
});
```

Impact

- **vBTC V2 contract creation can be blocked**: `VBTCController.InitiateMPCCeremony()` depends on a successful DKG ceremony to produce a Taproot deposit address; mismatched endpoints/schemas cause DKG to fail and contract creation to be unusable.
- **Withdrawals can be blocked**: signing ceremony coordination depends on compatible signing endpoints; mismatches prevent reaching a valid aggregated signature, blocking BTC withdrawal execution.
- **Operational fragility and partial state**: callers can initiate ceremonies that will fail mid-flight, leading to repeated retries, degraded UX, and potential inconsistent operator assumptions about ceremony completion.

Attack Scenario

An attacker (or simply normal users) repeatedly triggers `InitiateMPCCeremony` on nodes configured for vBTC V2. Ceremonies consistently fail due to protocol mismatches, consuming coordinator and validator resources (HTTP requests, logs, background tasks) and effectively preventing vBTC V2 contract creation and any withdrawal flows dependent on signing.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:M/D:H/Y:M (10.0)

Recommendation

It is recommended to define a single authoritative FROST HTTP API contract and enforce it on both the coordinator and validator server, ensuring endpoint paths, HTTP methods, and request/response schemas remain consistent and versioned. It is also recommended to treat schema mismatches as explicit failures and to validate responses before proceeding so deserialization issues cannot silently degrade into null or default values.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by aligning the FROST client/server API contract, adding missing server endpoints, and updating FrostMPCService to correctly deserialize the server's wrapper response schemas.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/6cadea93ca94643b289d65f6e5c6e649ac5be2fa

7.4 NON-DETERMINISTIC VALIDATOR SERVICE STARTUP CAN PREVENT THE FROST SERVER FROM RUNNING AND BREAK VBTC V2 MPC LIVENESS

// CRITICAL

Description

vBTC V2's MPC protocol depends on each validator running the FROST HTTP server `FrostServer` + `FrostStartup`) so the coordinator can execute DKG and signing ceremonies over the network. However, `FrostServer.Start()` is gated by `Globals.IsFrostValidator`, which defaults to `false` and is not set to `true` anywhere in the codebase snapshot. As a result, the validator process attempts to start FROST but the server never binds its port, and the validator startup loop repeatedly reports the FROST API port as closed.

This breaks multi-node execution by preventing the coordinator from reaching validator FROST endpoints, which directly prevents DKG (deposit address generation / contract creation) and signing ceremonies (withdrawals) from completing.

Code Reference

- The FROST server only starts when `Globals.IsFrostValidator` is true:

Copy Code

```
public static async Task Start()
{
    try
    {
        if (Globals.IsFrostValidator)
        {
            var builder = Host.CreateDefaultBuilder()
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseKestrel(options =>
                {
                    options.ListenAnyIP(Globals.FrostValidatorPort + 1, listenOption => { listenOption.U
                    options.ListenAnyIP(Globals.FrostValidatorPort);
                })
                .UseStartup<FrostStartup>()
                .ConfigureLogging(loggingBuilder => loggingBuilder.ClearProviders());
            });

            _ = builder.RunConsoleAsync();

            Console.WriteLine($"FROST Validator Server started on port {Globals.FrostValidatorPort}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"FROST Server Error: {ex}");
    }
}
```

- But the flag is defined with a default value of `false`:

```
public static bool IsBlockCaster = false;
public static bool IsWardenMonitoring = false;
public static bool IsFrostValidator = false;
public static bool IsValidatorPortOpen = false;
public static bool IsValidatorAPIPortOpen = false;
public static bool IsFROSTAPIPortOpen = false;
```

- Validator startup attempts to start FROST and then checks whether the FROST port is open; if not, it retries forever:

```
_ = StartCasterAPIServer();
_ = StartValidatorServer();
_ = FrostServer.Start();
_ = StartupValidators();
_ = Task.Run(BlockHeightCheckLoop);
_ = VBTCValidatorHeartbeatService.VBTCValidatorHeartbeatLoop();  // Start vBTC V2 validator heartbeat lo

// ...

Globals.IsFROSTAPIPortOpen = PortUtility.IsPortOpen(myIP, Globals.FrostValidatorPort);

if(!Globals.IsFROSTAPIPortOpen || !Globals.IsValidatorAPIPortOpen || !Globals.IsValidatorPortOpen)
{
    Console.WriteLine("Validator Ports Not Open Please open ports and try again. Retrying in 30 seconds.
    // ... prints FROST API port status and loops ...
    await Task.Delay(new TimeSpan(0, 0, 30));
    continue;
}
```

Impact

If validators do not actually start the FROST server, then:

- **DKG ceremonies fail**: the coordinator cannot complete deposit address generation, blocking vBTC V2 contract creation flows that depend on MPC.
- **Signing ceremonies fail**: withdrawals that depend on FROST signing cannot be completed.
- **Validator runtime readiness degrades**: the validator startup loop explicitly checks the FROST port and will continue retrying while the port is closed, creating persistent operational failures in multi-node setups.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:H/Y:H (10.0)

Recommendation

It is recommended to make FROST server activation deterministic whenever validator mode is active so `FrostServer.Start()` cannot be skipped due to an unset `Globals.IsFrostValidator` gate. It is also recommended to align deployment documentation with the port selection logic in `Config.cs` and to add a startup health signal that clearly indicates when FROST is expected but not running.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by updating validator startup so the FROST server is deterministically enabled in validator mode by setting `Globals.IsFrostValidator` during `ValidatorService.StartupValidatorProcess()`, ensuring `FrostServer.Start()` binds and MPC ceremonies can proceed.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/45d59fcdebb0d0023465eb3bafbf20e2ce829b08

7.5 INCORRECT TAPROOT SIGNING SEMANTICS CAN PRODUCE INVALID BITCOIN WITHDRAWALS AND LEAVE FUNDS STUCK

// CRITICAL

Description

`BitcoinTransactionService.SignTransactionWithFROST()` attempts to produce a Taproot key-path witness by asking the FROST MPC layer to sign `unsignedTx.GetHash().ToString()` (the transaction ID). For Taproot, validators must sign the **BIP341 signature hash** for each input, which commits to specific preimage data including prevouts and the input index. Signing the txid is not equivalent to signing the Taproot sighash.

The implementation also reuses the same aggregate signature for all inputs by attaching a single witness to every input. In Bitcoin, each input requires its own valid signature (and the sighash differs per input index), so multi-input withdrawals will be invalid even if the MPC signature were otherwise correct.

Operationally, vBTC V2 withdrawals can become permanently stuck at the “Bitcoin execution” step: the system can broadcast transactions that are rejected by the Bitcoin network, while the vBTC-side withdrawal state may still progress (or users may repeatedly attempt completion and burn fees / time).

Code Reference

- Message-to-sign is set to txid (“simplified for now”) even though Taproot requires BIP341 sighash:

Copy Code

```
public static async Task<(bool Success, string SignedTxHex, string TxHash, string ErrorMessage)>
    SignTransactionWithFROST(
        NBitcoin.Transaction unsignedTx,
        string scUID,
        List<VBTCValidator> validators,
        int threshold)
{
    try
    {
        // ... omitted ...

        // Prepare the message to sign (transaction sighash)
        // For Taproot, we use BIP 341 signature hash
        // Use the transaction ID as the message to sign (simplified for now)
        string messageToSign = unsignedTx.GetHash().ToString();

        // Coordinate FROST signing ceremony
        var signingResult = await FrostMPCService.CoordinateSigningCeremony(
            messageToSign,
            scUID,
            validators,
            threshold);

        // ... omitted ...

        // Attach witness to all inputs (they all spend from same Taproot address)
        for (int i = 0; i < unsignedTx.Inputs.Count; i++)
        {
            unsignedTx.Inputs[i].WitScript = witness;
```

```
        }

        string signedTxHex = unsignedTx.ToHex();
        string txHash = unsignedTx.GetHash().ToString();

        return (true, signedTxHex, txHash, string.Empty);
    }
```

- vBTC V2 withdrawal completion depends on `ExecuteFROSTWithdrawal(...)` to build/sign/broadcast the BTC transaction:

Copy Code

```
public static async Task<(bool Success, string TxHash, string ErrorMessage)>
    ExecuteFROSTWithdrawal(
        string taprootAddress,
        string destinationAddress,
        decimal amountBTC,
        long feeRateSatsPerVByte,
        string scUID,
        List<VBTCValidator> validators,
        int threshold)
{
    try
    {
        // Step 1: Build unsigned transaction
        var buildResult = await BuildUnsignedTaprootTransaction(
            taprootAddress,
            destinationAddress,
            amountBTC,
            feeRateSatsPerVByte);

        // Step 2: Sign with FROST
        var signingResult = await SignTransactionWithFROST(
            buildResult.UnsignedTx,
            scUID,
            validators,
            threshold);

        // Step 3: Broadcast to Bitcoin network
        var broadcastResult = await BroadcastTransaction(buildResult.UnsignedTx);
```

Impact

This can cause vBTC V2 withdrawals to fail at the Bitcoin layer even when validators are online and the contract state is otherwise consistent.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:H (9.4)

Recommendation

It is recommended to implement Taproot signing in a BIP341-correct manner by presenting the MPC with the per-input Taproot signature hash rather than the transaction ID, and by ensuring each input receives its own valid witness when the signature hash differs per input. It is also recommended to add a fail-closed verification step prior to broadcast so an invalid or unverifiable signed transaction cannot be broadcast and cannot advance the vBTC V2 withdrawal lifecycle.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding
by updating BitcoinTransactionService.SignTransactionWithFROST() to compute BIP341 Taproot signature hashes per input instead of signing the txid, perform per-input FROST signing with unique Schnorr witnesses, and fail closed via pre-broadcast Schnorr verification.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/3b142bae07562cab4aed92ba11b28c9c6e2be0ba

7.6 INSUFFICIENT CONSENSUS VALIDATION OF TRANSFER DATA ENABLES ARBITRARY VBTC V2 BALANCE MANIPULATION

// CRITICAL

Description

The vBTC v2 balance accounting relies on tokenization records stored in `SmartContractStateTreiTokenizationTXes`, where credits/mints and debits/burns are represented as entries with sentinel addresses `FromAddress = "+"` or `ToAddress = "−"`). The `StateData.TransferVBTCV2()` state transition appends such entries by reading `ContractUID`, `FromAddress`, `ToAddress`, and `Amount` directly from `tx.Data`.

However, the transition does not bind `tx.Data.FromAddress` to `tx.FromAddress`, and does not validate that the transfer `Amount` is positive.

Importantly, this issue is reachable through an **alternative state-transition routing path**: `StateData.UpdateTreis()` executes `TransferVBTCV2()` based on the untrusted `Function` string inside `tx.Data` for certain non`TX` transaction types (smart-contract/token transaction families). This is distinct from the intended vBTC v2 transfer flow that uses `TransactionType.VBTC_V2_TRANSFER` and is handled separately during transaction processing. As a result, an attacker can submit a crafted transaction that bypasses the vBTC v2 transfer path and still triggers `TransferVBTCV2()` to arbitrarily mint/burn/redistribute vBTC balances inside a contract's state, by choosing attacker-controlled `toAddress` (or using negative amounts to flip the accounting).

Code Reference

- `ReserveBlockCore/Data/StateData.cs` (reachable state transition selector)

Copy Code

```
if (tx.TransactionType != TransactionType.TX)
{
    if (tx.TransactionType == TransactionType.NFT_TX
        || tx.TransactionType == TransactionType.NFT_MINT
        || tx.TransactionType == TransactionType.NFT_BURN
        || tx.TransactionType == TransactionType.FTKN_MINT
        || tx.TransactionType == TransactionType.FTKN_TX
        || tx.TransactionType == TransactionType.FTKN_BURN
        || tx.TransactionType == TransactionType.TKNZ_MINT
        || tx.TransactionType == TransactionType.TKNZ_TX
        || tx.TransactionType == TransactionType.TKNZ_BURN
        || tx.TransactionType == TransactionType.SC_MINT
        || tx.TransactionType == TransactionType.SC_TX
        || tx.TransactionType == TransactionType.SC_BURN
        || tx.TransactionType == TransactionType.TKNZ_WD_ARB
        || tx.TransactionType == TransactionType.TKNZ_WD_OWNER)
    {
        // ... parse function ...
        if (!string.IsNullOrWhiteSpace(function))
        {
            switch (function)
            {
```

```
                // ... other cases ...
                case "TransferVBTCV2()":
                    TransferVBTCV2(tx);
                    break;
                // ... other cases ...
            }
        }
    }
```

- `ReserveBlockCore/Data/StateData.cs` (untrusted `tx.Data` drives balance updates; no sender binding; no amount validation)

Copy Code

```
private static void TransferVBTCV2(Transaction tx)
{
    try
    {
        // Parse transaction data
        var jobj = JObject.Parse(tx.Data);
        var scUID = jobj["ContractUID"]?.ToObject<string?>();
        var fromAddress = jobj["FromAddress"]?.ToObject<string?>();
        var toAddress = jobj["ToAddress"]?.ToObject<string?>();
        var amount = jobj["Amount"]?.ToObject<decimal?>();

        if (string.IsNullOrEmpty(scUID) || !amount.HasValue)
        {
            ErrorLogUtility.LogError($"TransferVBTCV2 failed: Missing required fields", "StateData.Trans
            return;
        }

        var scStateTreiRec = SmartContractStateTrei.GetSmartContractState(scUID);

        if (scStateTreiRec != null)
        {
            List<SmartContractStateTreiTokenizationTX> tknTxList = new List<SmartContractStateTreiTokeni
            {
                new SmartContractStateTreiTokenizationTX
                {
                    Amount = amount.Value,
                    FromAddress = "+",
                    ToAddress = toAddress
                },
                new SmartContractStateTreiTokenizationTX
                {
                    Amount = amount.Value * -1.0M,
                    FromAddress = fromAddress,
                    ToAddress = "-"
                }
            };
            // ...
```

Impact

An attacker can tamper with vBTC balances for any contract that has a `SmartContractStateTrei` record:

- **Unauthorized redistribution / theft**: set `toAddress` to an attacker-controlled address and `fromAddress` to a victim address, creating a “mint to attacker + burn from victim” pair that changes balances without the victim’s authorization.
- **Accounting manipulation via negative amounts**: if negative values are accepted in `tx.Data.Amount`, the sign flip can be abused to invert credits/debits, enabling additional balance manipulation patterns.
- **Protocol integrity break**: balance calculations and withdrawal eligibility derived from `SmartContractStateTreiTokenizationTXes` become untrustworthy, undermining vBTC v2’s core

accounting model.

Attack Scenario

1. The attacker creates a valid VFX transaction (proper signature/nonce/fee) of a transaction type that is routed through the `Function`-based state transition dispatcher (a non`TX` type that reaches the `switch (function)` inside `StateData.UpdateTreis()`).
2. The attacker sets `tx.Data` so it parses into a `Function = "TransferVBTCV2()"` call (either as a JSON object or as the first element of a JSON array, depending on the parsing branch), and includes attacker-chosen fields:
    1. `ContractUID`: the target vBTC v2 contract UID
    2. `FromAddress`: a victim address
    3. `ToAddress`: the attacker address
    4. `Amount`: a positive amount to shift balances (or negative for inverted effects)
3. The transaction is accepted under generic validation rules and processed by `StateData.UpdateTreis()`, which dispatches into `TransferVBTCV2()` solely because the `Function` string matches.
4. `StateData.TransferVBTCV2()` appends tokenization entries that directly change balances, without validating the caller or enforcing a positive amount.

BVSS

AO:A/AC:L/AX:L/R:P/S:C/C:N/A:L/I:C/D:H/Y:H (9.0)

Recommendation

It is recommended to enforce strict invariants for vBTC V2 balance transitions at consensus and within `TransferVBTCV2()` by binding debits to `tx.FromAddress`, rejecting non-positive transfer amounts, and preventing any implicit mint or burn behavior through transfer operations unless explicitly authorized by design. It is also recommended to ensure vBTC V2 transfer routing is unambiguous so only the intended transaction type and execution path can trigger vBTC V2 balance updates.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by implementing consensus-level TransferVBTCV2() validation with sender binding, positive-amount enforcement, and balance checks.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/331c854b986102bf124fa2f119792e15c5865be3

7.7 MISSING SENDER BINDING IN THE VBTC V2 WITHDRAWAL LIFECYCLE ENABLES UNAUTHORIZED STATE TRANSITIONS AND UNBACKED BURNS

// HIGH

Description

The vBTC v2 withdrawal flow relies on two transaction types (`VBTC_V2_WITHDRAWAL_REQUEST` and `VBTC_V2_WITHDRAWAL_COMPLETE`) that are applied by `StateData.UpdateTreis()`. However, the state transition logic does not bind critical fields (such as `OwnerAddress`, `WithdrawalRequestHash`, and `Amount`) to the transaction sender (`tx.FromAddress`) or to the active withdrawal state stored for the contract.

In addition, consensus-level transaction validation (`TransactionValidatorService.VerifyTX`) does not apply vBTC v2–specific constraints for these transaction types (unlike other smart-contract/tokenization transaction types), meaning an attacker can submit syntactically valid vBTC v2 withdrawal transactions that pass signature/fee/nonce checks.

As a result, any account can:

- Set a contract's local withdrawal state to `Requested` with attacker-chosen parameters, and
- Finalize a "withdrawal complete" transition and burn vBTC supply using a burn amount taken from `tx.Data` (not from the contract's active withdrawal amount), without proving that a corresponding Bitcoin payout occurred.

This breaks the integrity of vBTC accounting and can cause unbacked burns (supply reduction without redemption), lock legitimate withdrawals, and permanently damage user balances.

Code Reference

- `TransactionValidatorService.VerifyTX` does not parse/enforce vBTC v2 withdrawal transaction data (vBTC types are not in the transaction-type list that triggers smart-contract/tokenization function validation)

Copy Code

```
if (txRequest.TransactionType != TransactionType.TX)
{
    if (txRequest.TransactionType == TransactionType.NFT_TX
        || txRequest.TransactionType == TransactionType.NFT_MINT
        || txRequest.TransactionType == TransactionType.NFT_BURN
        || txRequest.TransactionType == TransactionType.FTKN_MINT
        || txRequest.TransactionType == TransactionType.FTKN_TX
        || txRequest.TransactionType == TransactionType.FTKN_BURN
        || txRequest.TransactionType == TransactionType.TKNZ_MINT
        || txRequest.TransactionType == TransactionType.TKNZ_TX
        || txRequest.TransactionType == TransactionType.TKNZ_BURN
        || txRequest.TransactionType == TransactionType.SC_MINT
        || txRequest.TransactionType == TransactionType.SC_TX
        || txRequest.TransactionType == TransactionType.SC_BURN
        || txRequest.TransactionType == TransactionType.TKNZ_WD_ARB
```

```
        || txRequest.TransactionType == TransactionType.TKNZ_WD_OWNER)
    {
        // parses ContractUID/Function and enforces constraints for those flows
        // ...
    }
}
```

- `StateData.RequestVBTCV2Withdrawal` trusts `tx.Data.OwnerAddress` and does not bind it to `tx.FromAddress` or to the contract owner; it directly sets active withdrawal fields

Copy Code

```
private static void RequestVBTCV2Withdrawal(Transaction tx)
{
    try
    {
        // Parse transaction data
        var jobj = JObject.Parse(tx.Data);
        var scUID = jobj["ContractUID"]?.ToObject<string?>();
        var ownerAddress = jobj["OwnerAddress"]?.ToObject<string?>();
        var btcAddress = jobj["BTCAddress"]?.ToObject<string?>();
        var amount = jobj["Amount"]?.ToObject<decimal?>();
        var feeRate = jobj["FeeRate"]?.ToObject<int?>();
        // ...
        var contract = VBTCContractV2.GetContract(scUID);
        // ...
        contract.WithdrawalStatus = VBTCWithdrawalStatus.Requested;
        contract.ActiveWithdrawalRequestHash = tx.Hash;
        contract.ActiveWithdrawalAmount = amount.Value;
        contract.ActiveWithdrawalBTCDestination = btcAddress;
        contract.ActiveWithdrawalFeeRate = feeRate.Value;
        contract.ActiveWithdrawalRequestTime = tx.Timestamp;
        VBTCContractV2.UpdateContract(contract);
    }
    catch (Exception ex)
    {
        // ...
    }
}
```

- `StateData.CompleteVBTCV2Withdrawal` does not validate `WithdrawalRequestHash` against the active request, and burns the amount taken from `tx.Data.Amount` (not from the active withdrawal amount)

Copy Code

```
private static void CompleteVBTCV2Withdrawal(Transaction tx)
{
    try
    {
        // Parse transaction data
        var jobj = JObject.Parse(tx.Data);
        var scUID = jobj["ContractUID"]?.ToObject<string?>();
        var withdrawalRequestHash = jobj["WithdrawalRequestHash"]?.ToObject<string?>();
        var btcTxHash = jobj["BTCTransactionHash"]?.ToObject<string?>();
        var amount = jobj["Amount"]?.ToObject<decimal?>();
        var destination = jobj["Destination"]?.ToObject<string?>();
        // ...
        var contract = VBTCContractV2.GetContract(scUID);
        // ...
        contract.WithdrawalStatus = VBTCWithdrawalStatus.Completed;
        // ...
        VBTCContractV2.UpdateContract(contract);

        // CRITICAL: Burn the withdrawn tokens in state trei
        var scStateTreiRec = SmartContractStateTrei.GetSmartContractState(scUID);
        if (scStateTreiRec != null && amount.HasValue)
        {
            List<SmartContractStateTreiTokenizationTX> tknTxList = new List<SmartContractStateTreiTokeni
```

```
            {
                new SmartContractStateTreiTokenizationTX
                {
                    Amount = amount.Value * -1.0M,  // Negative amount = burn
                    FromAddress = contract.OwnerAddress,
                    ToAddress = "-"  // "-" indicates burn/withdrawal
                }
            };
            // ...
            SmartContractStateTrei.UpdateSmartContract(scStateTreiRec);
        }
    }
    catch (Exception ex)
    {
        // ...
    }
}
```

Impact

This flaw allows unauthorized vBTC v2 withdrawal state transitions and supply changes:

- **Unbacked burns**: vBTC balances can be reduced on VFX without an actual BTC payout occurring, breaking the 1:1 redemption assumptions and harming token holders.
- **Denial of service**: an attacker can place a contract into an in-progress or completed state, preventing legitimate withdrawals and disrupting operations.
- **State corruption**: because the burn amount is taken from `tx.Data.Amount`, an attacker can burn an arbitrary amount (including mismatching the active withdrawal amount), creating inconsistent accounting and undermining the integrity of vBTC supply.

Attack Scenario

1. An attacker crafts a `VBTC_V2_WITHDRAWAL_REQUEST` transaction targeting a victim `ContractUID`, setting `OwnerAddress`, `BTCAddress`, and `Amount` arbitrarily in `tx.Data`, and signs it with their own key `tx.FromAddress` is the attacker).
2. `TransactionValidatorService.VerifyTX` accepts the transaction (no vBTC v2–specific checks are applied), and `StateData.RequestVBTCV2Withdrawal` records the withdrawal as `Requested` using the attacker-provided parameters.
3. The attacker then crafts a `VBTC_V2_WITHDRAWAL_COMPLETE` transaction for the same `ContractUID` with an arbitrary `BTCTransactionHash` and an arbitrary `Amount` in `tx.Data`.
4. `StateData.CompleteVBTCV2Withdrawal` finalizes the withdrawal state and burns `tx.Data.Amount` from the tokenization state three, without validating authorization or that the BTC payout happened.

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:H/I:C/D:H/Y:C (8.1)

Recommendation

It is recommended to bind `VBTC_V2_WITHDRAWAL_REQUEST` and `VBTC_V2_WITHDRAWAL_COMPLETE` to the transaction sender and the contract's canonical owner record to prevent unauthorized withdrawal

actions. It is also recommended to validate withdrawal completion against the active withdrawal state and to derive the burn amount from the tracked active withdrawal amount rather than from tx.data. Finally, it is recommended to add explicit consensus-level validation rules for vBTC V2 transaction types so malformed or unauthorized transitions cannot be included in blocks.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by implementing per-user withdrawal tracking with consensus-level sender binding.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/9a007b36bb0464d13ac0f0bae99ba54b8c88006c

7.8 INSUFFICIENT VALIDATION OF VALIDATOR NETWORK ENDPOINTS CAN ENABLE SSRF AND INTERNAL NETWORK TARGETING

// HIGH

Description

vBTC V2 relies on a validator registry (`VBTCValidator`) that stores each validator's network location (`IPAddress`) and is then used to coordinate FROST ceremonies via HTTP calls (DKG/signing). The validator registry is populated from on-chain transactions (`VBTC_V2_VALIDATOR_REGISTER`) during block processing.

However, the registration flow does not validate that the provided `IPAddress` is well-formed, safe, or restricted to expected formats (an IP literal). As a result, a registering validator can supply an arbitrary host value (including loopback, link-local, internal RFC1918 ranges, or attacker-controlled domains). Downstream components then interpolate this value into URLs and issue HTTP requests, creating an SSRF-like "forced outbound request" primitive from nodes that participate in MPC coordination or validator discovery.

Important constraint: the destination **port and path are fixed by the implementation** (`:{Globals.FrostValidatorPort}/frost/...` and `:{Globals.ValAPIPort}/valapi/...`). This significantly reduces the typical SSRF blast radius (no arbitrary port/path selection).

This is primarily an **off-chain** risk (HTTP calls from node processes), but it is **triggered by on-chain data** because the malicious `IPAddress` is written based on an included transaction.

Code Reference

- `TransactionValidatorService.VerifyTX()` validates `VBTC_V2_VALIDATOR_REGISTER` but does not validate the `IPAddress` beyond non-empty:

Copy Code

```
// VBTC V2 Validator Registration
if (txRequest.TransactionType == TransactionType.VBTC_V2_VALIDATOR_REGISTER)
{
    var txData = txRequest.Data;
    if (txData != null)
    {
        try
        {
            var jobj = JObject.Parse(txData);
            var validatorAddress = jobj["ValidatorAddress"]?.ToObject<string>();
            var ipAddress = jobj["IPAddress"]?.ToObject<string>();

            if (string.IsNullOrEmpty(validatorAddress) || string.IsNullOrEmpty(ipAddress))
                return (txResult, "Validator address and IP address cannot be null.");
            // ...
        }
        catch (Exception ex)
        {
            // ...
        }
    }
```

- `BlockValidatorService` persists `IPAddress` from the transaction into the `VBTCValidator` database record:

```
// Process vBTC V2 validator registration/exit transactions
if (block.Transactions.Count() > 0)
{
    foreach (var tx in block.Transactions)
    {
        // Process VBTC V2 Validator Registration
        if (tx.TransactionType == TransactionType.VBTC_V2_VALIDATOR_REGISTER)
        {
            try
            {
                var jobj = JObject.Parse(tx.Data);
                var validatorAddress = jobj["ValidatorAddress"]?.ToObject<string>();
                var ipAddress = jobj["IPAddress"]?.ToObject<string>();
                // ...
                var vbtcValidator = new Bitcoin.Models.VBTCValidator
                {
                    ValidatorAddress = validatorAddress,
                    IPAddress = ipAddress,
                    // ...
                };

                Bitcoin.Models.VBTCValidator.SaveValidator(vbtcValidator);
            }
            catch (Exception ex)
            {
                // ...
            }
        }
```

- `FrostMPCService` uses `validator.IPAddress` to build URLs and issues HTTP requests during DKG/signing:

```
var tasks = validators.Select(async validator =>
{
    try
    {
        var url = $"http://{validator.IPAddress}:{Globals.FrostValidatorPort}/frost/dkg/start";
        var response = await _httpClient.PostAsJsonAsync(url, startRequest);
        return response.IsSuccessStatusCode;
    }
    catch (Exception ex)
    {
        LogUtility.Log($"[FROST MPC] Failed to contact validator {validator.ValidatorAddress}: {ex.Messa
        return false;
    }
});
```

- `VBTCValidator.FetchActiveValidatorsFromNetwork()` also uses stored `IPAddress` to issue HTTP requests to validator APIs:

```
foreach (var validator in knownValidators)
{
    try
    {
        var url = $"http://{validator.IPAddress}:{Globals.ValAPIPort}/valapi/ValidatorController/GetActi
        LogUtility.Log($"Fetching validator list from {validator.ValidatorAddress} ({validator.IPAddress
            "VBTCValidator.FetchActiveValidatorsFromNetwork()");

        var response = await httpClient.GetAsync(url);
```

```
if (response.IsSuccessStatusCode)
{
    // ...
}
```

Impact

If an attacker can register as a vBTC V2 validator (or otherwise cause malicious validator records to be stored), other nodes may be induced to send HTTP requests to attacker-chosen targets, which can lead to:

- **Internal network reachability from validator nodes (constrained SSRF)**: requests to loopback/internal hosts are possible, but only on the fixed validator ports and fixed endpoint paths used by the protocol.
- **Network scanning / DoS**: repeated MPC coordination calls can generate high-volume requests to an internal host/port.
- **Operational disruption**: DKG/signing ceremonies may fail or hang due to unreachable/malicious endpoints, reducing availability of withdrawals/contract creation.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:L/A:M/I:N/D:M/Y:M (8.1)

Recommendation

It is recommended to treat IPAddress provided during validator registration as a security-sensitive input by restricting it to an IP literal format, rejecting unsafe destinations that are incompatible with the deployment model, and enforcing canonical normalization to avoid URL parsing ambiguities. It is also recommended to add defense-in-depth controls for outbound MPC and validator-discovery requests so a single malicious validator record cannot repeatedly disrupt ceremonies or trigger repeated requests to unsafe targets.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by introducing strict validator IP address validation at the consensus level to reject dangerous destinations and adding defense-in-depth checks before outbound HTTP calls to validator endpoints.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/2accf60cad5a566d9c1b14b116991cbe11a3f94d

7.9 INSUFFICIENT VALIDATION OF FROST PROTOCOL INPUTS CAN EXPOSE NATIVE FFI TO CRASH AND RESOURCE-EXHAUSTION RISKS

// HIGH

Description

The vBTC V2 FROST validator server accepts DKG messages over HTTP and stores them in memory (`Round1Commitments`, `Round3Verifications`). When enough Round 3 "verified" responses are received, the server finalizes the ceremony by calling `GeneratePlaceholderGroupPublicKey()`, which serializes all Round 1 commitment strings into JSON and passes that JSON to the Rust FROST FFI via `FrostNative.DKGRound3Finalize()`.

Because Round 1 commitments are supplied by remote HTTP clients and are not validated for:

- participant membership (any `ValidatorAddress` can be used as the dictionary key),
- size/format (any string is accepted), or
- reasonable `RequiredThresholdParticipantAddresses` constraints (both are attacker-controlled via `/frost/dkg/start`),

an attacker can inject arbitrarily large or malformed data that is then forwarded into the native FFI boundary. Native libraries commonly fail "hard" (process crash/abort) on unexpected inputs, and even safe error handling can still cause significant CPU/memory pressure during JSON serialization and FFI parsing. This creates a realistic remote availability attack against validator nodes and, by extension, MPC ceremonies.

Code Reference

- Untrusted commitment strings are accepted over HTTP and stored under attacker-chosen `ValidatorAddress` keys:

Copy Code

```
endpoints.MapPost("/frost/dkg/round1", async context =>
{
    // ...
    var commitment = JsonConvert.DeserializeObject<FrostDKGRound1Message>(body);

    if (commitment == null || string.IsNullOrEmpty(commitment.CommitmentData))
        // ...

    // Get session
    if (!FrostSessionStorage.DKGSessions.TryGetValue(commitment.SessionId, out var session))
        // ...

    // Validate validator signature (presence only)
    if (string.IsNullOrEmpty(commitment.ValidatorSignature))
        // ...

    // Store commitment
    session.Round1Commitments[commitment.ValidatorAddress] = commitment.CommitmentData;
```

```
    // ...
});
```

- Attacker-controlled finalization trigger: once enough “verified” values are recorded, the server calls `GeneratePlaceholderGroupPublicKey(session.Round1Commitments)`:

Copy Code

```
if (verifiedCount >= requiredCount && !session.IsCompleted)
{
    // Aggregate DKG result with placeholder crypto
    session.GroupPublicKey = GeneratePlaceholderGroupPublicKey(session.Round1Commitments);
    session.TaprootAddress = GeneratePlaceholderTaprootAddress(session.GroupPublicKey);
    session.DKGProof = GeneratePlaceholderDKGProof(session.SessionId, session.GroupPublicKey);
    session.IsCompleted = true;
}
```

- Commitments are serialized and passed to the native FFI boundary:

Copy Code

```
private static string GeneratePlaceholderGroupPublicKey(System.Collections.Concurrent.ConcurrentDictiona
{
    try
    {
        // Serialize commitments to JSON for FROST library
        var commitmentsJson = JsonConvert.SerializeObject(commitments.Values.ToList());

        var (groupPubkey, keyPackage, pubkeyPackage, errorCode) = FrostNative.DKGRound3Finalize(
            round2SecretPackage: "{}",
            round1PackagesJson: commitmentsJson,
            round2PackagesJson: "[]"
        );
        // ...
    }
    catch (Exception ex)
    {
        // ...
    }
}
```

- Native FFI accepts raw `string` inputs (ANSI marshaling) for JSON packages:

Copy Code

```
[DllImport(DllName, CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Ansi)]
public static extern int frost_dkg_round3_finalize(
    [MarshalAs(UnmanagedType.LPStr)] string round2SecretPackage,
    [MarshalAs(UnmanagedType.LPStr)] string round1PackagesJson,
    [MarshalAs(UnmanagedType.LPStr)] string round2PackagesJson,
    out IntPtr outGroupPubkey,
    out IntPtr outKeyPackage,
    out IntPtr outPubkeyPackage);
```

Impact

- **Validator process crash / abort risk**: malformed or adversarial JSON passed into native code can cause a hard crash depending on Rust FFI error handling and assumptions.
- **Resource exhaustion**: large commitment strings inflate JSON serialization size and increase CPU/memory use during serialization and FFI parsing, degrading validator responsiveness.
- **MPC ceremony disruption**: degraded validators reduce participation and can prevent DKG/signing from reaching required thresholds.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (7.5)

Recommendation

It is recommended to treat all FROST HTTP inputs as security-sensitive and to enforce strict validation before any values can influence native FFI calls such as `FrostNative.DKGRound3Finalize`. It is also recommended to enforce participant-only submission, bound commitment and share sizes, and reject invalid RequiredThreshold values so finalization only occurs once protocol invariants are satisfied.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by enforcing authenticated, participant-only FROST DKG message submission and by bounding commitment and share payload sizes via `MAX_COMMITMENT_DATA_LENGTH` before data can be stored or reach the native FFI boundary.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/b16762af8f4ac98233ef537b14f892625f22ac30

7.10 WEAK DKG SESSION AUTHORIZATION CAN ENABLE CEREMONY SABOTAGE AND QUORUM MANIPULATION

// HIGH

Description

During FROST DKG, validators are supposed to participate only if they are selected for the ceremony (the `ParticipantAddresses` list created at `/frost/dkg/start`). However, the DKG message submission endpoints `/frost/dkg/round1` and `/frost/dkg/round3`) do not enforce that `ValidatorAddress` belongs to the session's participant list, and they allow overwriting previously stored values for a given address.

As implemented:

- Any caller can submit a Round 1 commitment and be counted toward the quorum, even if they are not in `ParticipantAddresses`.
- A caller can submit a message using an existing participant's `ValidatorAddress` key and overwrite previously stored data (the latest write wins).
- Quorum is computed using the count of stored keys, not the count of valid participants, so accepting non-participant keys can make the ceremony appear to reach threshold prematurely.

This is an authorization/integrity gap separate from cryptographic signature correctness: even with correct signature verification in the future, the server still needs to enforce participant membership and one-time submission rules to prevent non-selected validators (or network actors) from biasing quorum counting or sabotaging ceremonies.

Code Reference

- Session participant set is accepted from the start request and stored as `ParticipantAddresses`:

Copy Code

```
// Create DKG session in memory
var session = new DKGSession
{
    SessionId = request.SessionId,
    SmartContractUID = request.SmartContractUID,
    LeaderAddress = request.LeaderAddress,
    ParticipantAddresses = request.ParticipantAddresses,
    RequiredThreshold = request.RequiredThreshold,
    StartTimestamp = TimeUtil.GetTime()
};
```

- Round 1 stores commitments keyed by `ValidatorAddress` without checking membership or preventing overwrite:

Copy Code

```
// Get session
if (!FrostSessionStorage.DKGSessions.TryGetValue(commitment.SessionId, out var session))
{
    // ...
}
```

```
// Store commitment
session.Round1Commitments[commitment.ValidatorAddress] = commitment.CommitmentData;

var commitmentCount = session.Round1Commitments.Count;
var requiredCount = (int)Math.Ceiling(session.ParticipantAddresses.Count * (session.RequiredThreshold /
// ...
await context.Response.WriteAsync(JsonConvert.SerializeObject(new
{
    // ...
    CommitmentCount = commitmentCount,
    RequiredCount = requiredCount,
    ThresholdReached = commitmentCount >= requiredCount
}, Formatting.Indented));
```

- Round 3 stores verification results keyed by `ValidatorAddress` without checking membership or preventing overwrite, and completion is gated only on counts:

Copy Code

```
// Get session
if (!FrostSessionStorage.DKGSessions.TryGetValue(verification.SessionId, out var session))
{
    // ...
}

// Record verification result
session.Round3Verifications[verification.ValidatorAddress] = verification.Verified;

var verificationCount = session.Round3Verifications.Count;
var verifiedCount = session.Round3Verifications.Count(v => v.Value);
var requiredCount = (int)Math.Ceiling(session.ParticipantAddresses.Count * (session.RequiredThreshold /

// If threshold reached and all verified, compute final result
if (verifiedCount >= requiredCount && !session.IsCompleted)
{
    session.GroupPublicKey = GeneratePlaceholderGroupPublicKey(session.Round1Commitments);
    session.TaprootAddress = GeneratePlaceholderTaprootAddress(session.GroupPublicKey);
    session.DKGProof = GeneratePlaceholderDKGProof(session.SessionId, session.GroupPublicKey);
    session.IsCompleted = true;
}
```

Impact

- **Ceremony sabotage / liveness failure**: non-participants can inject arbitrary submissions that change quorum accounting, causing ceremonies to fail, stall, or “complete” with inconsistent data.
- **Quorum manipulation**: counting non-participants toward the threshold can effectively lower the security assumptions of the ceremony (the set of contributors diverges from the intended participant set).
- **Potentially incorrect MPC outputs**: if the final aggregation step consumes attacker-influenced inputs, the resulting group public key / deposit address (or future signing material) may be invalid or not controlled by the intended validator set, risking loss of funds or permanent withdrawal inability.

Attack Scenario

During a DKG ceremony, a malicious non-selected validator (or any network actor able to reach validator FROST servers) submits many `POST /frost/dkg/round1` messages using their own distinct `ValidatorAddress` strings (or repeatedly overwrites a target participant key). This inflates `Round1Commitments.Count` and can make `ThresholdReached` true without receiving genuine participant contributions. Later, the attacker submits `POST /frost/dkg/round3` messages with `Verified=true` to meet `verifiedCount >= requiredCount`, pushing the session into `IsCompleted` and producing MPC

outputs derived from untrusted/non-participant data. Even if other honest participants submit correct values, the attacker can overwrite or bias the stored dictionaries and prevent a valid ceremony completion.

BVSS

AO:A/AC:M/AX:L/R:N/S:U/C:N/A:H/I:N/D:H/Y:H (7.5)

Recommendation

It is recommended to enforce strict session authorization on all DKG message endpoints so only addresses in ParticipantAddresses can submit round messages and submissions cannot be overwritten once accepted. It is also recommended to ensure quorum and threshold counting only considers valid participants, and to bind the authenticated sender identity to the ValidatorAddress claimed in each payload so participants cannot be impersonated.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by preventing DKG Round 1 and Round 3 submission overwrites by switching to TryAdd with 409 conflict handling, ensuring each participant can submit only once per round and eliminating quorum manipulation via repeated overwrites.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/372c027213bc8993ec5da7494b3e303aef41c820

7.11 MISSING CONSENSUS SUPPORT FOR WITHDRAWAL CANCELLATION CAN LEAVE FUNDS STUCK AND PREVENT RECOVERY GOVERNANCE

// HIGH

Description

The **vBTC V2** specification explicitly includes a cancellation governance mechanism for failed withdrawals: the owner submits a cancellation request and validators vote (75% approval) to cancel and unlock funds. The transaction model includes two dedicated consensus transaction types for this flow: `VBTC_V2_WITHDRAWAL_CANCEL` and `VBTC_V2_WITHDRAWAL_VOTE`.

In the devnet implementation, these transaction types are present in `TransactionType`, but they are **not**:

- validated in `TransactionValidatorService.VerifyTX()` (no vBTC V2 rules exist for cancel/vote), nor
- handled in `StateData.UpdateTreis()` (no type-based state transition like the existing withdrawal request/complete handlers), nor
- processed/validated in `BlockTransactionValidatorService`'s vBTC V2 transaction processing (only transfer/request/complete are covered).

This means cancellation/vote transactions can be mined while **doing nothing** to the withdrawal state, burning fees and leaving the withdrawal lifecycle without a recovery path when Bitcoin settlement fails or becomes ambiguous. Operationally, this increases the likelihood of withdrawals becoming stuck and funds requiring manual intervention.

Code Reference

- the transaction types exist in consensus enum:

Copy Code

```
VBTC_V2_CONTRACT_CREATE,         // Create vBTC v2 contract
VBTC_V2_TRANSFER,                // Transfer vBTC v2 tokens
VBTC_V2_WITHDRAWAL_REQUEST,      // Request withdrawal to BTC
VBTC_V2_WITHDRAWAL_COMPLETE,     // Complete withdrawal
VBTC_V2_WITHDRAWAL_CANCEL,       // Request cancellation
VBTC_V2_WITHDRAWAL_VOTE          // Validator votes on cancellation
```

- spec defines the cancel + vote API and the intended governance semantics:

Copy Code

```
**4. Withdrawal Operations**
- `POST RequestWithdrawal` - Owner requests withdrawal to BTC
  - Validates balance, checks no active withdrawal
  - Payload: `VBTCWithdrawalPayload` (SmartContractUID, OwnerAddress, BTCAddress, Amount, FeeRate)
- `POST CompleteWithdrawal` - Complete withdrawal via FROST signing
  - **FROST Integration Point #2**: 2-round signing ceremony
  - Requires 51% of active validators
  - Broadcasts signed BTC transaction
  - Payload: `VBTCWithdrawalCompletePayload` (SmartContractUID, WithdrawalRequestHash)
```

```
- `POST CancelWithdrawal` - Request cancellation with failure proof
  - Payload: `VBTCCancellationPayload` (SmartContractUID, OwnerAddress, WithdrawalRequestHash, BTCTxHash
- `POST VoteOnCancellation` - Validators vote on cancellation (75% required)
  - **FROST Integration Point #3**: Validator signature verification
  - Payload: `VBTCCancellationVotePayload` (CancellationUID, ValidatorAddress, Approve, ValidatorSignatu
```

- `StateData.UpdateTreis()` only has explicit vBTC V2 handlers for withdrawal request/complete (no cancel/vote):

```
// vBTC V2 Withdrawal Request/Complete Handling
if (tx.TransactionType == TransactionType.VBTC_V2_WITHDRAWAL_REQUEST)
{
    RequestVBTCV2Withdrawal(tx);
}

if (tx.TransactionType == TransactionType.VBTC_V2_WITHDRAWAL_COMPLETE)
{
    CompleteVBTCV2Withdrawal(tx);
}
```

- the vBTC V2 transaction processing/validation in `BlockTransactionValidatorService` covers only transfer/request/complete and then falls through (no cancel/vote branches):

```
// vBTC V2 Transaction Processing
if (tx.TransactionType == TransactionType.VBTC_V2_TRANSFER)
{
    // ...
}

if (tx.TransactionType == TransactionType.VBTC_V2_WITHDRAWAL_REQUEST)
{
    // ...
}

if (tx.TransactionType == TransactionType.VBTC_V2_WITHDRAWAL_COMPLETE)
{
    // ...
}
```

Impact

- **Failed-withdrawal recovery is broken**: when a withdrawal cannot be completed (validator quorum, BTC broadcast issues, reorgs, mempool replacement, etc.), the intended cancel/vote governance flow cannot be applied at consensus level.
- **Funds can remain stuck in “Requested/Pending”** without a supported resolution path, increasing manual operations risk and degrading user trust.
- **Fee-burning no-ops**: if cancel/vote TX types are broadcast and mined, they can consume fees while making no state changes, confusing operators and users.

Attack Scenario

A user’s withdrawal request enters `Requested` state, but the BTC payout fails (insufficient validator participation or broadcast failure). The user attempts `VBTC_V2_WITHDRAWAL_CANCEL`, and validators attempt `VBTC_V2_WITHDRAWAL_VOTE` to reach the 75% cancellation threshold. The transactions can be

mined, but because no consensus handler applies them, the withdrawal remains stuck and the user must rely on manual intervention or out-of-band fixes.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:M/Y:M (7.5)

Recommendation

It is recommended to implement the withdrawal cancellation governance flow end-to-end at the consensus layer by adding deterministic state handling for VBTC_V2_WITHDRAWAL_CANCEL and VBTC_V2_WITHDRAWAL_VOTE and corresponding consensus validation rules. It is also recommended to enforce authorization, one-vote-per-validator, and the 75% approval threshold using deterministic on-chain data or consensus state structures rather than local-only databases.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by implementing consensus validation and deterministic state handlers for `VBTC_V2_WITHDRAWAL_CANCEL` and `VBTC_V2_WITHDRAWAL_VOTE`, including the `Cancellation_Requested` intermediate status and a 75% validator-approval unlock rule.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/f576bc19ccb8e88f870b2627ccd99ab97cca5c8f

7.12 INCORRECT REPLAY-PREVENTION KEYING CAN ENABLE CROSS-USER COLLISIONS AND DENIAL OF SERVICE

// MEDIUM

Description

The vBTC v2 raw withdrawal flow uses the `VBTCWithdrawalRequest` model to implement replay prevention and “only one active withdrawal” constraints. Retrieval of records is correctly scoped to `(RequestorAddress, OriginalUniqueId, SmartContractUID)`. However, persistence uses `OriginalUniqueId` as a global key by calling `FindOne(x => x.OriginalUniqueId == request.OriginalUniqueId)` without scoping to the requestor or contract.

This inconsistency means the system can incorrectly treat a request as “duplicate” even when it belongs to a different requestor and/or a different contract, potentially blocking legitimate requests and undermining the intended replay-prevention model.

Code Reference

- `ReserveBlockCore/Bitcoin/Models/VBTCWithdrawalRequest.cs`

Copy Code

```
public static VBTCWithdrawalRequest? GetByUniqueId(string address, string uniqueId, string scUID)
{
    var vwrDb = GetVBTCWithdrawalRequestDb();
    if (vwrDb == null)
    {
        ErrorLogUtility.LogError("GetVBTCWithdrawalRequestDb() returned a null value.", "VBTCWithdrawalR
        return null;
    }

    var request = vwrDb.Query()
        .Where(x => x.RequestorAddress == address &&
                    x.OriginalUniqueId == uniqueId &&
                    x.SmartContractUID == scUID)
        .FirstOrDefault();

    return request;
}
```

- `ReserveBlockCore/Bitcoin/Models/VBTCWithdrawalRequest.cs` (global uniqueness on `OriginalUniqueId`)

Copy Code

```
public static bool Save(VBTCWithdrawalRequest request, bool update = false)
{
    var vwrDb = GetVBTCWithdrawalRequestDb();
    if (vwrDb == null)
    {
        ErrorLogUtility.LogError("GetVBTCWithdrawalRequestDb() returned a null value.", "VBTCWithdrawalR
        return false;
    }

    var existingRequest = vwrDb.FindOne(x => x.OriginalUniqueId == request.OriginalUniqueId);
    if (existingRequest != null)
    {
        if (!update)
```

```
            return false;
        // ...
    }
    else
    {
        vwrDb.InsertSafe(request);
        return true;
    }
}
```

Impact

If `OriginalUniqueId` is not guaranteed to be globally unique across all users and all contracts (and especially if it is user-provided), collisions can cause the raw withdrawal path to reject legitimate withdrawal requests as duplicates. This can degrade availability and create operational friction in replay-prevention logic, potentially resulting in user-visible failures.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:L/D:L/Y:L (6.9)

Recommendation

It is recommended to align replay-prevention persistence with the intended request identity by enforcing uniqueness on a composite key rather than on OriginalUniqueId alone. It is also recommended to document the required uniqueness and entropy expectations for OriginalUniqueId so identifiers cannot be reused across users or contracts.

Remediation Comment

SOLVED: The **VerifiedX team f**ixed the finding by updating `VBTCWithdrawalRequest.Save` to enforce uniqueness using the composite key `RequestorAddress`, `OriginalUniqueId`, and `SmartContractUID`, eliminating cross-user and cross-contract `OriginalUniqueId` collisions.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/a6aff5ad459c76add831fe09208b279dcf671949

7.13 NON-CANONICAL ADDRESS ENCODINGS CAN ENABLE ADDRESS ALIASING AND USER-ASSISTED FUND LOSS

// MEDIUM

Description

`AddressValidateUtility.ValidateRBXAddress()` implements a Base58Check-style checksum verification (double SHA-256, compare 4-byte digest). However, its Base58 decoding routine `DecodeBase58`) intentionally uses a fixed 25-byte output buffer and **does not reject overflow** (the `p != 0` carry check is commented out). This creates **non-canonical encodings**: multiple distinct Base58 strings can decode to the same 25 bytes (payload + checksum) under the truncated decoder.

As a result, an attacker can craft an “alias address” string that:

- passes `ValidateRBXAddress()` (checksum OK),
- passes transaction validation (length checks OK),
- but is not the canonical string representation produced by `AccountData.GetHumanAddress()` / `ReserveAccount.GetHumanAddress()`.

Because account identity throughout the system is string-based `FromAddress`, `ToAddress`, state trei keys), sending funds to such an alias can create balances under an address string that no private key holder can spend from (since signature verification binds the address string to the canonical derived address). This is a realistic user-assisted loss/griefing vector (“valid-looking address” that cannot be spent).

Code Reference

- `ValidateRBXAddress` validates checksum over 25 decoded bytes:

Copy Code

```
public static bool ValidateRBXAddress(string address)
{
    if (address.Length < 26 || address.Length > 35) return false; // wrong length
    var decoded = DecodeBase58(address);
    var d1 = Hash(decoded.SubArray(0, 21));
    var d2 = Hash(d1);
    if (!decoded.SubArray(21, 4).SequenceEqual(d2.SubArray(0, 4))) return false; //bad digest
    return true;
}
```

- The Base58 decoder truncates to 25 bytes and does **not** reject overflow (carry), enabling non-canonical encodings:

Copy Code

```
private static byte[] DecodeBase58(string input)
{
    var output = new byte[Size];
    foreach (var t in input)
    {
        var p = Alphabet.IndexOf(t);
```

```
            if (p == -1) throw new Exception("invalid character found");
            var j = Size;
            while (--j >= 0)
            {
                p += 58 * output[j];
                output[j] = (byte)(p % 256);
                p /= 256;
            }
            //if (p != 0) throw new Exception("address too long");
        }
        return output;
    }
```

- Consensus/mempool validation relies on this function for `ToAddress`:

Copy Code

```
if (txRequest.ToAddress != "Adnr_Base" &&
    txRequest.ToAddress != "DecShop_Base" &&
    txRequest.ToAddress != "Topic_Base" &&
    txRequest.ToAddress != "Vote_Base" &&
    txRequest.ToAddress != "Reserve_Base" &&
    txRequest.ToAddress != "Token_Base" &&
    txRequest.ToAddress != "TW_Base")
{
    if (!AddressValidateUtility.ValidateAddress(txRequest.ToAddress))
        return (txResult, "To Address failed to validate");

    if(txRequest.ToAddress.Length < 32)
        return (txResult, "Address length is too short.");
}
```

- Signature verification binds `address` to a canonical derived address string, meaning an “alias” string cannot be used as a spendable `FromAddress`:

Copy Code

```
public static bool VerifySignature(string address, string message, string sigScript)
{
    try
    {
        // ...
        var _PublicKey = "04" + ByteToHex(publicKey.toString());
        var _Address = address.StartsWith("xRBX") ? ReserveAccount.GetHumanAddress(_PublicKey) : Account

        if (address != _Address)
        {
            return false;
        }

        return Ecdsa.verify(message, Signature.fromBase64(sigScriptArray[0]), publicKey);
    }
    catch(Exception ex)
    {
        // ...
    }
}
```

Impact

- **User-assisted fund loss / griefing**: a victim can be convinced to send funds to an attacker-provided destination string that passes validation but is not a canonical address; balances can be created under an unspendable string identity.
- **State/UX divergence**: wallets and UIs that pre-check with `ValidateAddress` may treat the alias as safe, but later the recipient cannot spend (signature binding requires canonical string).

- **Security policy bypass via string predicates (secondary)**: any logic that relies on string prefixes (`StartsWith("xRBX")`) can be bypassed for destination strings while still passing checksum validation.

Proof of Concept

Copy Code

```
import os, hashlib

ALPHABET = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'
ALPH_IDX = {c:i for i,c in enumerate(ALPHABET)}
SIZE = 25
MOD = 256**SIZE


def sha256(b: bytes) -> bytes:
    return hashlib.sha256(b).digest()

def base58_encode_int(n: int) -> str:
    if n == 0:
        return ALPHABET[0]
    s = ''
    while n > 0:
        n, rem = divmod(n, 58)
        s = ALPHABET[rem] + s
    return s

def base58_encode_bytes(b: bytes) -> str:
    # canonical, like ReserveAccount.Base58Encode
    n = 0
    for x in b:
        n = n*256 + x
    s = ''
    while n > 0:
        n, rem = divmod(n, 58)
        s = ALPHABET[rem] + s
    # leading zeros
    for x in b:
        if x == 0:
            s = ALPHABET[0] + s
        else:
            break
    return s

def decode_base58_trunc(s: str) -> bytes:
    out = [0]*SIZE
    for ch in s:
        p = ALPH_IDX.get(ch, -1)
        if p == -1:
            raise Exception('invalid character found')
        j = SIZE
        while True:
            j -= 1
            if j < 0:
                break
            p += 58 * out[j]
            out[j] = p % 256
            p //= 256
    return bytes(out)

def validate_rbx_address(s: str) -> bool:
    if len(s) < 26 or len(s) > 35:
        return False
    dec = decode_base58_trunc(s)
    d1 = sha256(dec[:21])
    d2 = sha256(d1)
    return dec[21:25] == d2[:4]
```

```
# generate a canonical address and an alias that decodes to same 25 bytes under truncation
for attempt in range(1, 200000):
    payload21 = os.urandom(21)
    chk = sha256(sha256(payload21))[:4]
    b25 = payload21 + chk
    canonical = base58_encode_bytes(b25)
    if not (26 <= len(canonical) <= 34):
        continue

    val = int.from_bytes(b25, 'big')
    alias_val = val + (1 << (8*SIZE))  # +2^200
    alias = base58_encode_int(alias_val)
    if not (26 <= len(alias) <= 35):
        continue

    if canonical == alias:
        continue

    # verify both pass checksum under truncated decode
    if not validate_rbx_address(canonical):
        continue
    if not validate_rbx_address(alias):
        continue

    d_can = decode_base58_trunc(canonical)
    d_alias = decode_base58_trunc(alias)
    if d_can != d_alias:
        continue

    print('FOUND')
    print('canonical', canonical, 'len', len(canonical))
    print('alias    ', alias, 'len', len(alias))
    print('same decoded bytes:', d_can == d_alias)
    print('starts canonical:', canonical[:4], 'alias:', alias[:4])
    break
else:
    print('not found within attempts')
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:M/D:L/Y:L (6.9)

Recommendation

It is recommended to ensure address validation is canonical and unambiguous by rejecting invalid Base58 decoding conditions and enforcing a single canonical Base58Check representation for valid inputs. It is also recommended to enforce strict address schema constraints and to ensure addresses are normalized to a canonical form before being persisted or used as state keys so the same payload cannot be represented by multiple strings.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by restoring Base58 overflow rejection in DecodeBase58() and enforcing canonical Base58Check encoding in ValidateRBXAddress() via decode -> re-encode -> compare, preventing non-canonical alias addresses from passing validation.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/d57ff372122637f1cff9985a4bc24b2aa8f08dba

7.14 MISSING STATE-TRANSITION HANDLING CAN CAUSE VBTC V2 TRANSFERS TO BURN FEES WITHOUT APPLYING ON-CHAIN BALANCE UPDATES

// MEDIUM

Description

The vBTC V2 implementation is designed to support token transfers via `TransactionType.VBTC_V2_TRANSFER` (per docs and `VBTCService.TransferVBTC()`), where the transaction carries JSON payload fields `ContractUID`, `FromAddress`, `ToAddress`, `Amount`).

However, the consensus state update path `StateData.UpdateTreis`) does **not** apply any state transition for `TransactionType.VBTC_V2_TRANSFER`. Instead, vBTC balance updates are routed through a **function-string switch** that can call `TransferVBTCV2()` only for certain non`TX` transaction families `SC_TX` / `TKNZ_TX`), and through explicit handling for vBTC V2 withdrawal request/complete transaction types.

As a result, a `VBTC_V2_TRANSFER` transaction can be accepted and propagated (and may be shown as successful in wallets), but it will not modify `SmartContractStateTreiTokenizationTXes`, meaning the intended token transfer does not occur on-chain. This creates a discrepancy between the expected vBTC V2 transfer behavior and the actual chain state, and can lead to repeated fee-burning transfers without token movement.

Code Reference

- `VBTCService.TransferVBTC()` constructs a transaction with `TransactionType.VBTC_V2_TRANSFER`:

Copy Code

```
// Create transaction data
var txData = JsonConvert.SerializeObject(new
{
    Function = "VBTCTransfer()",
    ContractUID = scUID,
    FromAddress = fromAddress,
    ToAddress = toAddress,
    Amount = amount
});

// Build transaction
var tokenTx = new Transaction
{
    Timestamp = TimeUtil.GetTime(),
    FromAddress = fromAddress,
    ToAddress = toAddress,
    Amount = 0.0M, // No VFX transferred, only vBTC
    Fee = 0.0M,
    Nonce = AccountStateTrei.GetNextNonce(fromAddress),
    TransactionType = TransactionType.VBTC_V2_TRANSFER,
    Data = txData
};
```

- `StateData.UpdateTreis()` routes balance updates through a function-based dispatcher for `SC_TXTKNZ_TX`/etc, including a `TransferVBTCV2()` case (but **not** `VBTC_V2_TRANSFER`):

```
if (tx.TransactionType != TransactionType.TX)
{
    if (tx.TransactionType == TransactionType.NFT_TX
        || tx.TransactionType == TransactionType.NFT_MINT
        || tx.TransactionType == TransactionType.NFT_BURN
        || tx.TransactionType == TransactionType.FTKN_MINT
        || tx.TransactionType == TransactionType.FTKN_TX
        || tx.TransactionType == TransactionType.FTKN_BURN
        || tx.TransactionType == TransactionType.TKNZ_MINT
        || tx.TransactionType == TransactionType.TKNZ_TX
        || tx.TransactionType == TransactionType.TKNZ_BURN
        || tx.TransactionType == TransactionType.SC_MINT
        || tx.TransactionType == TransactionType.SC_TX
        || tx.TransactionType == TransactionType.SC_BURN
        || tx.TransactionType == TransactionType.TKNZ_WD_ARB
        || tx.TransactionType == TransactionType.TKNZ_WD_OWNER)
    {
        // ... parse function ...
        if (!string.IsNullOrWhiteSpace(function))
        {
            switch (function)
            {
                // ...
                case "TransferVBTCV2()":
                    TransferVBTCV2(tx);
                    break;
                // ...
            }
        }
    }
```

- The same `UpdateTreis()` method applies withdrawal request/complete by vBTC V2 transaction type, but does not apply any transfer-by-type handling:

```
// vBTC V2 Withdrawal Request/Complete Handling
if (tx.TransactionType == TransactionType.VBTC_V2_WITHDRAWAL_REQUEST)
{
    RequestVBTCV2Withdrawal(tx);
}

if (tx.TransactionType == TransactionType.VBTC_V2_WITHDRAWAL_COMPLETE)
{
    CompleteVBTCV2Withdrawal(tx);
}
```

Impact

This mismatch can lead to:

- **Silent transfer failure**: users sign and broadcast `VBTC_V2_TRANSFER` transactions expecting token movement, but the canonical chain state does not reflect the transfer.
- **Fee-burning griefing (user-assisted)**: any workflow that induces users to sign repeated transfers can cause users to lose VFX fees without achieving token movement.
- **State/UX divergence**: local wallet processing may mark transactions successful while balances remain unchanged, undermining trust in vBTC V2 transfer functionality.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:L/A:L/I:M/D:N/Y:N (6.3)

Recommendation

It is recommended to ensure vBTC V2 transfers are applied deterministically at consensus by aligning the TransactionType used for transfers with the state-transition path that updates balances. It is also recommended to enforce consensus validation that binds the effective sender and receiver to `tx.FromAddress` and `tx.ToAddress` and enforces amount constraints so transfer state updates cannot be driven by untrusted `tx.Data`. If vBTC V2 transfers are intentionally unsupported, it is recommended to document this explicitly and to ensure APIs do not produce transactions that appear successful but have no state effect.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by adding explicit TransactionType.VBTC_V2_TRANSFER handling in StateData.UpdateTreis() by calling TransferVBTCV2(tx), ensuring transfers are applied on-chain, and by binding sender and receiver to tx.FromAddress and tx.ToAddress rather than untrusted tx.Data fields.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/a65957545d1cd7792f906dead38df2d3c2d6a521

7.15 MISSING CONSENSUS-STATE HANDLING CAN CAUSE VBTC V2 CONTRACT CREATION TO SUCCEED LOCALLY BUT HAVE NO ON-CHAIN EFFECT

// MEDIUM

Description

vBTC V2 contract creation uses `VBTCController.CreateVBTCContract()` to generate a smart contract and then broadcast a mint transaction using `TransactionType.VBTC_V2_CONTRACT_CREATE`. The mint transaction payload is the standard smart-contract mint format with `Function = "Mint()"` and a `ContractUID`.

However, `StateData.UpdateTreis()` only dispatches the `"Mint()"` function through its smart-contract function router for a fixed list of transaction types `SC_MINT`, `NFT_MINT`, `TKNZ_MINT`, etc.). It **does not include** `TransactionType.VBTC_V2_CONTRACT_CREATE` in that dispatcher list, and it also does not have an explicit type-based handler for `VBTC_V2_CONTRACT_CREATE`.

As a result, a `VBTC_V2_CONTRACT_CREATE` transaction can be accepted into a block (normal nonce/fee/signature checks) but **never applied to the Smart Contract State Trei**, meaning the contract is not minted at consensus state level. This creates a sharp divergence between local node DB state (contract saved locally) and on-chain consensus state (contract absent), breaking downstream vBTC V2 operations that depend on the contract existing in state.

Code Reference

- vBTC V2 contract creation broadcasts a mint transaction using `TransactionType.VBTC_V2_CONTRACT_CREATE`:

Copy Code

```
// Save smart contract to databases
SmartContractMain.SmartContractData.SaveSmartContract(result.Item2, result.Item1);
await VBTCContractV2.SaveSmartContract(result.Item2, result.Item1, payload.OwnerAddress);

// Create and broadcast mint transaction
var scTx = await SmartContractService.MintSmartContractTx(result.Item2, TransactionType.VBTC_V2_CONTRACT
```

- the mint transaction's `Data` uses `Function = "Mint()"` (standard SC mint payload) while `TransactionType` is set to the passed-in `txType`:

Copy Code

```
string function = scData.Item3 ? "TokenDeploy()" : "Mint()";
var newSCInfo = new[]
{
    new { Function = function, ContractUID = scMain.SmartContractUID, Data = scBase64, MD5List = md5List
};

txData = JsonConvert.SerializeObject(newSCInfo);

scTx = new Transaction
{
```

```
    Timestamp = TimeUtil.GetTime(),
    FromAddress = scMain.MinterAddress,
    ToAddress = scMain.MinterAddress,
    Amount = 0.0M,
    Fee = 0,
    Nonce = AccountStateTrei.GetNextNonce(scMain.MinterAddress),
    TransactionType = scData.Item3 ? TransactionType.TKNZ_MINT : txType,
    Data = txData
};
```

- `StateData.UpdateTreis()` only runs the `"Mint()"` dispatcher for a fixed set of SC/NFT/tokenization TX families, excluding `VBTC_V2_CONTRACT_CREATE`:

Copy Code

```
if (tx.TransactionType != TransactionType.TX)
{
    if (tx.TransactionType == TransactionType.NFT_TX
        || tx.TransactionType == TransactionType.NFT_MINT
        || tx.TransactionType == TransactionType.NFT_BURN
        || tx.TransactionType == TransactionType.FTKN_MINT
        || tx.TransactionType == TransactionType.FTKN_TX
        || tx.TransactionType == TransactionType.FTKN_BURN
        || tx.TransactionType == TransactionType.TKNZ_MINT
        || tx.TransactionType == TransactionType.TKNZ_TX
        || tx.TransactionType == TransactionType.TKNZ_BURN
        || tx.TransactionType == TransactionType.SC_MINT
        || tx.TransactionType == TransactionType.SC_TX
        || tx.TransactionType == TransactionType.SC_BURN
        || tx.TransactionType == TransactionType.TKNZ_WD_ARB
        || tx.TransactionType == TransactionType.TKNZ_WD_OWNER)
    {
        // ... switch(function) includes case "Mint()": AddNewlyMintedContract(tx);
    }
}
```

- block validation parses `FunctionContractUID` from `tx.Data` for non-base TX types, but this does not imply the transaction's state transition is applied:

Copy Code

```
if (blkTransaction.TransactionType != TransactionType.TX &&
    blkTransaction.TransactionType != TransactionType.ADNR &&
    blkTransaction.TransactionType != TransactionType.VOTE &&
    blkTransaction.TransactionType != TransactionType.VOTE_TOPIC &&
    blkTransaction.TransactionType != TransactionType.DSTR &&
    blkTransaction.TransactionType != TransactionType.RESERVE &&
    blkTransaction.TransactionType != TransactionType.NFT_SALE)
{
    if (blkTransaction.Data != null)
    {
        // ...
        var scInfo = TransactionUtility.GetSCTXFunctionAndUID(blkTransaction);
        if (!scInfo.Item1)
            return false;
        // ...
    }
}
```

Impact

- **Protocol liveness failure (contract creation):** vBTC V2 contract mint transactions may be mined but do not mint the contract in consensus state, so other nodes do not observe the contract as created.
- **State divergence vs UX:** nodes that create the contract will have local DB entries (`SmartContractData`, `VBTCContractV2`) but the canonical `SmartContractStateTrei` will not contain

the contract, causing broken reads and downstream operations.

- **Downstream breakage**: transfers/withdrawals that depend on contract existence at state level can fail or behave inconsistently across nodes.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:M/Y:M (6.3)

Recommendation

It is recommended to ensure vBTC V2 contract creation is applied deterministically at consensus by adding explicit state handling for `TransactionType.VBTC_V2_CONTRACT_CREATE` within `StateData.UpdateTreis()` so the intended mint action is executed rather than treated as a no-op. It is also recommended to enforce consensus validation so a `VBTC_V2_CONTRACT_CREATE` transaction cannot be mined without a well-formed mint payload that results in a deterministic state update.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by adding TransactionType.VBTC_V2_CONTRACT_CREATE to the StateData.UpdateTreis() smart-contract dispatcher
so Function="Mint()" routes to AddNewlyMintedContract(tx), making vBTC V2 contract creation apply at consensus.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/4222bd4a75b042d9fe49ebfb0771e1bacdb41a69

7.16 AMBIGUOUS LOCAL CONTRACT RECORDS CAN BYPASS WITHDRAWAL AUTHORIZATION AND ENABLE UNAUTHORIZED VBTC V2 WITHDRAWALS

// MEDIUM

Description

The vBTC V2 implementation persists multiple `VBTCContractV2` rows per `SmartContractUID` inside the same LiteDB collection: one row is the canonical “contract owner” record created at contract creation, while additional rows are created for token holders (watch/balance holder records) when tokens are transferred. However, `VBTCContractV2.GetContract(smartContractUID)` and several update helpers query the database using `FindOne(x => x.SmartContractUID == smartContractUID)` without distinguishing which record is authoritative.

Because vBTC business logic and local state updates rely on `GetContract()` for owner checks and withdrawal state, the record ambiguity can cause a node to treat a holder/watch record as the contract record. This can (a) bypass “only owner” checks in local flows that rely on the cached record and (b) corrupt the node’s local vBTC withdrawal state tracking (recording withdrawal status against the wrong row), leading to user-facing inconsistencies and denial of service for withdrawals on that node.

Code Reference

- `ReserveBlockCore/Bitcoin/Models/VBTCContractV2.cs` `GetContract()` selects by `SmartContractUID` only)

Copy Code

```
public static VBTCContractV2? GetContract(string smartContractUID)
{
    var contracts = GetDb();
    if (contracts != null)
    {
        var contract = contracts.FindOne(x => x.SmartContractUID == smartContractUID);
        if (contract != null)
        {
            return contract;
        }
    }

    return null;
}
```

- `ReserveBlockCore/Bitcoin/Models/VBTCContractV2.cs` `SaveSmartContractTransfer()` inserts holder/watch rows for the same `SmartContractUID`)

Copy Code

```
public static async Task SaveSmartContractTransfer(SmartContractMain scMain, string rbxAddress, string?
{
    var contracts = GetDb();

    // Check if contract already exists for this specific address
    var exist = contracts.FindOne(x => x.SmartContractUID == scMain.SmartContractUID && x.OwnerAddress =
```

```
    if (exist == null)
    {
        // ...
        VBTCContractV2 contract = new VBTCContractV2
        {
            SmartContractUID = scMain.SmartContractUID,
            OwnerAddress = rbxAddress, // This user is a holder, not the original owner
            // ...
        };
        contracts.InsertSafe(contract);
    }
}
```

- `ReserveBlockCore/Bitcoin/Models/VBTCContractV2.cs` (multiple update helpers select by `SmartContractUID` only)

Copy Code

```
public static void UpdateBalance(string smartContractUID, decimal newBalance)
{
    try
    {
        var contracts = GetDb();
        var contract = contracts.FindOne(x => x.SmartContractUID == smartContractUID);

        if (contract != null)
        {
            contract.Balance = newBalance;
            contracts.UpdateSafe(contract);
        }
    }
    catch (Exception ex)
    {
        ErrorLogUtility.LogError(ex.ToString(), "VBTCContractV2.UpdateBalance()");
    }
}

public static void UpdateWithdrawalStatus(string smartContractUID, VBTCWithdrawalStatus status,
    string? btcDestination = null, decimal? amount = null, string? requestHash = null, long? requestBloc
{
    try
    {
        var contracts = GetDb();
        var contract = contracts.FindOne(x => x.SmartContractUID == smartContractUID);
        // ...
    }
    catch (Exception ex)
    {
        ErrorLogUtility.LogError(ex.ToString(), "VBTCContractV2.UpdateWithdrawalStatus()");
    }
}
```

- `ReserveBlockCore/Services/BlockTransactionValidatorService.cs` (owner check depends on `GetContract(scUID)`)

Copy Code

```
// Validate contract exists
var contract = VBTCContractV2.GetContract(scUID);
if (contract == null)
{
    // ...
    return;
}

// Validate owner
if (contract.OwnerAddress != ownerAddress)
{
    // ...
```

```
    return;
}
```

- `ReserveBlockCore/Data/StateData.cs` (withdrawal state updates persist against the `GetContract(scUID)` record)

Copy Code

```
// Get the contract
var contract = VBTCContractV2.GetContract(scUID);
if (contract == null)
{
    ErrorLogUtility.LogError($"RequestVBTCV2Withdrawal failed: Contract not found - {scUID}", "StateData
    return;
}

// Update contract with withdrawal request details
contract.WithdrawalStatus = VBTCWithdrawalStatus.Requested;
contract.ActiveWithdrawalRequestHash = tx.Hash;
contract.ActiveWithdrawalAmount = amount.Value;
contract.ActiveWithdrawalBTCDestination = btcAddress;

// Save updated contract
VBTCContractV2.UpdateContract(contract);
```

Impact

If a holder/watch record is returned by `GetContract(scUID)` instead of the canonical contract owner record, owner checks and withdrawal state checks that rely on this cached record become unreliable. This can lead to local authorization bypasses (in flows that trust the cached record), incorrect withdrawal state tracking, and incorrect vBTC balance/accounting on that node (for example, when state updates use `contract.OwnerAddress` from the wrong cached row).

While this does not, by itself, prove a remote attacker can influence third-party validator state, it is still a security-relevant integrity issue because it can cause user-facing failures (withdrawals blocked) and inconsistent local accounting for vBTC.

Attack Scenario

This issue manifests when a node has multiple `VBTCContractV2` rows for the same `SmartContractUID` (canonical owner record vs. holder/watch record), and `GetContract(scUID)` returns the wrong row because it does not disambiguate which record is authoritative. This can happen naturally because `rsrv_vbtc.db` is a local database and holder/watch rows are created during local account restore and balance-holder import flows:

1. A node creates a holder/watch row for a given `SmartContractUID` where the row's `OwnerAddress` equals a non-owner address (created via `SaveSmartContractTransfer()` during local restore/import).
2. The node later performs a withdrawal-related flow (or processes withdrawal-related state updates) that relies on `VBTCContractV2.GetContract(scUID)`.
3. Because `GetContract(scUID)` may resolve to the holder/watch row, owner checks and withdrawal state updates are evaluated against the wrong `OwnerAddress` and the wrong withdrawal state.
4. The node may incorrectly block a legitimate owner, incorrectly attribute withdrawal state, or corrupt local vBTC accounting, resulting in denial of service and inconsistent behavior for users of that node.

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:H/I:H/D:M/Y:M (5.9)

Recommendation

It is recommended to enforce a single canonical vBTC V2 contract record per `SmartContractUID` and to ensure all owner checks and withdrawal-state checks reference only that canonical source of truth. It is also recommended to separate canonical contract records from any holder or watch-only records using distinct storage or an explicit discriminator with strict uniqueness guarantees, and to derive critical withdrawal authorization decisions from deterministic on-chain state where possible rather than from local caches.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by updating `SaveSmartContractTransfer()` to stop creating holder-specific VBTCContractV2 records and to enforce a single canonical contract record per `SmartContractUID`, eliminating ambiguous `GetContract()` lookups and aligning balances with `SmartContractStateTrei`.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/87c2fdef26cd2738997a3f9ca27f16715c5f45f8

7.17 PLACEHOLDER MPC ARTIFACTS CAN BREAK DEPOSIT ADDRESS GENERATION AND WITHDRAWAL SIGNING FOR VBTC V2

// MEDIUM

Description

The vBTC V2 MPC layer does not produce real cryptographic outputs for two critical artifacts:

1. **Taproot deposit addresses** returned from DKG are generated by placeholder helper functions and are not derived using actual BIP340/BIP341/BIP350 Taproot address derivation from the group public key.
2. **Schnorr signatures** returned from signing ceremonies are generated by placeholder logic and returned with `SignatureValid = true`, then injected into the Bitcoin transaction witness.

As a result, contract creation can yield deposit addresses that are not reliably usable as real Bitcoin Taproot addresses, and withdrawal signing can yield invalid signatures/witnesses that will be rejected by the Bitcoin network (or fail earlier when libraries validate inputs).

Code Reference

- The coordinator generates a mock Taproot address by concatenating `bc1ptb1p` with a random GUID substring (not a real Bech32m encoding with checksum):

Copy Code

```
private static string GeneratePlaceholderTaprootAddress(string groupPublicKey)
{
    // Taproot addresses start with bc1p (mainnet) or tb1p (testnet)
    var prefix = Globals.IsTestNet ? "tb1p" : "bc1p";
    var random = Guid.NewGuid().ToString("N").Substring(0, 58);
    return $"{prefix}{random}";
}
```

- The coordinator generates a mock Schnorr signature and returns it as if valid:

Copy Code

```
private static string GeneratePlaceholderSchnorrSignature(string messageHash, Dictionary<string, string>
{
    // Schnorr signatures are 64 bytes (128 hex characters)
    var combined = messageHash + string.Join("", shares.Values);
    var hash = System.Security.Cryptography.SHA256.HashData(Encoding.UTF8.GetBytes(combined));
    var signature = Convert.ToHexString(hash) + Convert.ToHexString(hash); // 128 hex chars
    return signature;
}
```

- The validator server also returns a “Taproot address” as a `bc1ptb1p` prefix plus hex substring, without Bech32m encoding:

Copy Code

```
private static string GeneratePlaceholderTaprootAddress(string groupPublicKey)
{
    try
    {
        var prefix = Globals.IsTestNet ? "tb1p" : "bc1p";
```

```
        var hash = System.Security.Cryptography.SHA256.HashData(
            System.Text.Encoding.UTF8.GetBytes($"TAPROOT_{groupPublicKey}")
        );
        var addressPayload = Convert.ToHexString(hash).Substring(0, 58).ToLower();
        return $"{prefix}{addressPayload}";
    }
    catch (Exception ex)
    {
        var prefix = Globals.IsTestNet ? "tb1p" : "bc1p";
        var random = Guid.NewGuid().ToString("N").Substring(0, 58);
        return $"{prefix}{random}";
    }
}
```

- The Bitcoin withdrawal signing path consumes the MPC signing output and injects it into the Taproot witness:

Copy Code

```
// Coordinate FROST signing ceremony
var signingResult = await FrostMPCService.CoordinateSigningCeremony(
    messageToSign,
    scUID,
    validators,
    threshold);

if (signingResult == null || !signingResult.SignatureValid)
{
    return (false, string.Empty, string.Empty, "FROST signing failed");
}

// Convert it to Bitcoin witness format for Taproot
var aggregateSignatureBytes = Convert.FromHexString(signingResult.SchnorrSignature);

var witness = new WitScript(Op.GetPushOp(aggregateSignatureBytes));

for (int i = 0; i < unsignedTx.Inputs.Count; i++)
{
    unsignedTx.Inputs[i].WitScript = witness;
}
```

Impact

- **Deposits/contract creation reliability**: DKG results can produce deposit addresses that are not correctly derived Taproot addresses, making it impossible or unsafe for users to deposit BTC to the intended threshold-controlled output.
- **Withdrawals**: signing outputs are not produced by a real threshold signing ceremony and are treated as valid, causing Bitcoin transactions to fail broadcast or be rejected by consensus rules (invalid Schnorr signature/witness), resulting in stuck withdrawals and degraded system liveness.
- **Operational risk**: operators may incorrectly believe MPC is functional based on progress/status reporting, while the cryptographic artifacts are placeholders.

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:H/I:N/D:H/Y:H (5.6)

Recommendation

It is recommended to ensure the MPC layer produces real cryptographic artifacts end-to-end for both DKG and signing so deposit address generation and withdrawal signing cannot rely on placeholder outputs. It is also recommended to implement deterministic DKG and signing flows that validate derived Taproot addresses and aggregated signatures and fail closed when artifacts are invalid.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by rewriting `CoordinateShareDistribution` to collect Round 2 GeneratedShares and redistribute them via the signed batch endpoint POST `/frost/dkg/shares/{sessionId}`, persisting incoming shares in `ReceivedSharesJson` (including fixing POST `/frost/dkg/share`) and auto-triggering deterministic DKG finalization via `TryFinalizeDKG` calling `FrostNative.DKGRound3Finalize` and deriving a real Taproot address.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/5a1a1695c0543f6ebdb3bad2a6cae02915db1644

7.18 HARDCODED DEPOSIT ADDRESS OUTPUTS CAN MISDIRECT VBTC DEPOSITS AND CAUSE IRREVERSIBLE FUND LOSS

// MEDIUM

Description

`VBTCController` exposes an API endpoint intended to return the MPC-generated Taproot deposit address and associated public key data for a given vBTC V2 contract `GET vbtcapi/VBTC/GetMPCDepositAddress/{scUID}`. In this snapshot, the endpoint does not load the contract or its TokenizationV2 feature data. Instead, it returns hardcoded placeholder strings for `DepositAddress` and `FrostGroupPublicKey`.

Any UI, operator tooling, or external integration that relies on this endpoint to display or distribute the deposit address may provide users an incorrect address. Users can then deposit BTC to an unrelated or invalid destination, resulting in lost funds and broken vBTC deposit flows.

Code Reference

- The endpoint returns placeholder values and does not read contract state:

Copy Code

```
[HttpGet("GetMPCDepositAddress/{scUID}")]
[ProducesResponseType(typeof(object), StatusCodes.Status200OK)]
public async Task<string> GetMPCDepositAddress(string scUID)
{
    try
    {
        // Get smart contract
        // var sc = SmartContractMain.SmartContractData.GetSmartContract(scUID);
        // var tokenizationV2 = sc.Features.FirstOrDefault(x => x.FeatureName == FeatureName.Tokenizatio

        // TODO: FROST INTEGRATION
        // Retrieve FROST group public key and Taproot deposit address
        // PLACEHOLDER: Mock data

        return JsonConvert.SerializeObject(new
        {
            Success = true,
            Message = "Deposit address retrieved",
            SmartContractUID = scUID,
            DepositAddress = "bc1pFROST_TAPROOT_PLACEHOLDER",
            FrostGroupPublicKey = "PLACEHOLDER_FROST_GROUP_PUBLIC_KEY",
            RequiredThreshold = 51
        });
    }
    catch (Exception ex)
    {
        return JsonConvert.SerializeObject(new { Success = false, Message = $"Error: {ex.Message}" });
    }
}
```

Impact

- **Misdirected deposits**: clients can be instructed to deposit BTC to a placeholder or incorrect address, causing irreversible loss if funds are sent to an address that is not actually controlled by the vBTC MPC group key.
- **Operational breakage**: even if contract creation succeeds and a correct deposit address exists in contract data, this endpoint can override or contradict that truth for API consumers, causing deposit UX failures and support incidents.

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:M/I:M/D:H/Y:M (5.6)

Recommendation

It is recommended to ensure `GetMPCDepositAddress` returns only persisted contract MPC data by loading the `VBTCContractV2` record for `scUID` and returning `DepositAddress`, `FrostGroupPublicKey`, `DKGProof` and `RequiredThreshold`, failing closed when the contract is missing or incomplete. It is also recommended to remove any placeholder return paths or gate them behind an explicit dev-only configuration so operator tooling and UIs cannot accidentally surface non-real deposit information.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by updating VBTCController.GetMPCDepositAddress to stop returning hardcoded placeholder values and instead load the VBTCContractV2 record by scUID, returning DepositAddress, FrostGroupPublicKey, RequiredThreshold, and DKGProof with explicit error responses when the contract is missing or the deposit address has not been generated.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/2bab0a929e37bad2759beb2d8a3939bb22c0428b

7.19 UNRESTRICTED FROST SESSION CREATION CAN ENABLE REMOTE RESOURCE EXHAUSTION AND DEGRADE MPC AVAILABILITY

// MEDIUM

Description

vBTC V2 validators run an HTTP server (`FrostServer` + `FrostStartup`) that coordinates FROST DKG/signing ceremonies. In the current implementation, core endpoints (notably `POST /frost/dkg/start`) allow any remote client to create in-memory DKG sessions by supplying an arbitrary `SessionId`, `ParticipantAddresses`, and `RequiredThreshold`.

Sessions are stored in global `ConcurrentDictionary` instances (`FrostSessionStorage.DKGSessions` / `SigningSessions`) and the cleanup routine (`CleanupOldSessions`) is defined but never invoked from request handlers. As a result, an unauthenticated remote party can repeatedly create unique sessions and grow in-memory state until resource exhaustion (memory/CPU/log churn), degrading or preventing legitimate MPC ceremonies.

This issue is independent of cryptographic signature correctness: even though signature verification is marked TODO, the server still accepts and stores sessions based on the presence of fields (`LeaderSignature` must be non-empty), which is sufficient for remote session spam.

Code Reference

- FROST server listens on all interfaces and starts on validator nodes:

Copy Code

```
public static async Task Start()
{
    try
    {
        if (Globals.IsFrostValidator)
        {
            var builder = Host.CreateDefaultBuilder()
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseKestrel(options =>
                {
                    options.ListenAnyIP(Globals.FrostValidatorPort + 1, listenOption => { listenOption.U
                    options.ListenAnyIP(Globals.FrostValidatorPort);
                })
                .UseStartup<FrostStartup>()
                .ConfigureLogging(loggingBuilder => loggingBuilder.ClearProviders());
            });

            _ = builder.RunConsoleAsync();

            Console.WriteLine($"FROST Validator Server started on port {Globals.FrostValidatorPort}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"FROST Server Error: {ex}");
```

```
    }
}
```

- Unauthenticated DKG session creation; only checks that `LeaderSignature` is non-empty:

Copy Code

```
endpoints.MapPost("/frost/dkg/start", async context =>
{
    try
    {
        using (var reader = new StreamReader(context.Request.Body))
        {
            var body = await reader.ReadToEndAsync();
            var request = JsonConvert.DeserializeObject<FrostDKGStartRequest>(body);

            // ...

            // Validate leader signature
            // TODO: When FROST native library integrated, verify signature
            if (string.IsNullOrEmpty(request.LeaderSignature))
            {
                context.Response.StatusCode = StatusCodes.Status400BadRequest;
                await context.Response.WriteAsync(JsonConvert.SerializeObject(new
                {
                    Success = false,
                    Message = "Leader signature required"
                }));
                return;
            }

            // Create DKG session in memory
            var session = new DKGSession
            {
                SessionId = request.SessionId,
                SmartContractUID = request.SmartContractUID,
                LeaderAddress = request.LeaderAddress,
                ParticipantAddresses = request.ParticipantAddresses,
                RequiredThreshold = request.RequiredThreshold,
                StartTimestamp = TimeUtil.GetTime()
            };

            if (!FrostSessionStorage.DKGSessions.TryAdd(request.SessionId, session))
            {
                // ...
            }
            // ...
        }
    }
    catch (Exception ex)
    {
        // ...
    }
});
```

- Global in-memory storage + cleanup function that is not called from request handlers:

Copy Code

```
public static class FrostSessionStorage
{
    public static ConcurrentDictionary<string, DKGSession> DKGSessions { get; } = new();
    public static ConcurrentDictionary<string, SigningSession> SigningSessions { get; } = new();

    /// <summary>
    /// Clean up old sessions (older than 1 hour)
    /// </summary>
    public static void CleanupOldSessions()
    {
        // ... removes entries older than 1 hour ...
```

```
    }
}
```

Impact

- **Remote DoS of validator nodes**: an attacker can create many unique `SessionId` values and force unbounded growth of `DKGSessions` / `SigningSessions`, plus extra CPU/log work per request.
- **MPC ceremony disruption**: if enough validators are degraded/unreachable due to resource exhaustion, DKG/signing ceremonies fail to reach required participation thresholds, blocking vBTC V2 operations that depend on MPC (contract creation, withdrawals).

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:H/I:L/D:M/Y:M (5.3)

Recommendation

It is recommended to enforce authentication and authorization on all FROST endpoints so only registered active validators and session participants can create sessions or submit protocol messages. It is also recommended to enforce strict bounds on session creation and input sizes and to ensure stale session data is cleaned up so remote callers cannot accumulate unbounded in-memory state and degrade MPC ceremony availability.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by adding ECDSA signature verification and active-validator and participant authorization checks on FROST endpoints, enforcing strict session and input bounds, and enabling periodic and opportunistic session cleanup to prevent unbounded in-memory growth and remote DoS.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/bd7393454110b5d9918d02c165afe129b707a3a8

7.20 VBTC V2 VALIDATOR REGISTRATION AND EXIT FLOW BLOCKED BY ZERO FEE REQUIREMENT CONFLICTING WITH GLOBAL TRANSACTION VALIDATION

// MEDIUM

Description

The vBTC v2 protocol introduces `VBTC_V2_VALIDATOR_REGISTER` and `VBTC_V2_VALIDATOR_EXIT` transaction types that are explicitly required to be **free transactions** `Fee = 0`). However, the global transaction validation routine rejects any transaction with `Fee <= 0` (except for a legacy `TKNZ_WD_ARB` special case). As a result, vBTC v2 validator registration and exit transactions are rejected at the consensus validation layer, which prevents the validator set from being formed/maintained on-chain.

This is not simply an API-level issue: the node's validator startup flow constructs vBTC v2 validator registration transactions with `Fee = 0` (as required), and then immediately runs them through `TransactionValidatorService.VerifyTX()`, which will reject them due to the global "fee must be > 0" rule. Consequently, downstream vBTC v2 operations that rely on the availability of registered active validators (DKG ceremonies and threshold signing coordination) can become unavailable or require manual/local workarounds.

Code Reference

- `ReserveBlockCore/Services/TransactionValidatorService.cs` (global fee rule rejects zero-fee TX)

Copy Code

```
if (txRequest.Fee <= 0 && txRequest.TransactionType != TransactionType.TKNZ_WD_ARB)
{
    return (txResult, "Fee cannot be less than or equal to zero.");
}

if(Globals.LastBlock.Height > Globals.TXHeightRule2) //around April 7, 2023 at 18:30 UTC
{
    if (txRequest.Fee <= 0.000003M && txRequest.TransactionType != TransactionType.TKNZ_WD_ARB)
    {
        return (txResult, "Fee cannot be less than 0.000003 VFX");
    }
}
```

- `ReserveBlockCore/Services/TransactionValidatorService.cs` (vBTC v2 validator registration explicitly requires `Fee = 0`)

Copy Code

```
// VBTC V2 Validator Registration
if (txRequest.TransactionType == TransactionType.VBTC_V2_VALIDATOR_REGISTER)
{
    var txData = txRequest.Data;
    if (txData != null)
    {
        try
        {
            var jobj = JObject.Parse(txData);
            var validatorAddress = jobj["ValidatorAddress"]?.ToObject<string>();
```

```
            var ipAddress = jobj["IPAddress"]?.ToObject<string>();

            if (string.IsNullOrEmpty(validatorAddress) || string.IsNullOrEmpty(ipAddress))
                return (txResult, "Validator address and IP address cannot be null.");

            if (txRequest.FromAddress != validatorAddress)
                return (txResult, "From address must match validator address.");

            if (txRequest.ToAddress != validatorAddress)
                return (txResult, "To address must match validator address (self-transaction).");

            if (txRequest.Fee != 0M)
                return (txResult, "Validator registration must be a free transaction (Fee = 0).")

            if (txRequest.Amount != 0M)
                return (txResult, "Validator registration cannot have an amount.");
```

- `ReserveBlockCore/Services/ValidatorService.cs` (startup path constructs `Fee = 0` registration TX and calls `VerifyTX`)

Copy Code

```
// Create VBTC_V2_VALIDATOR_REGISTER transaction
var registerTx = new Transaction
{
    Timestamp = TimeUtil.GetTime(),
    FromAddress = validator.Address,
    ToAddress = validator.Address,  // Self transaction
    Amount = 0M,
    Fee = 0M,  // FREE transaction
    Nonce = sTreiAcct.Nonce,
    TransactionType = TransactionType.VBTC_V2_VALIDATOR_REGISTER,
    Data = JsonConvert.SerializeObject(new
    {
        ValidatorAddress = validator.Address,
        IPAddress = ipAddress,
        FrostPublicKey = "PLACEHOLDER_FROST_PUBLIC_KEY",  // TODO: Replace with actual FROST key
        RegistrationBlockHeight = Globals.LastBlock.Height,
        Signature = signature
    })
};

...


var result = await TransactionValidatorService.VerifyTX(registerTx);
```

Impact

If vBTC v2 relies on on-chain registered active validators to perform DKG and withdrawal signing ceremonies, then preventing validator registration/exit at the consensus layer can lead to:

- Inability to reliably form and maintain the vBTC v2 validator set.
- Operational denial of service for features depending on validator participation (contract creation ceremonies and withdrawals), potentially resulting in delayed or blocked redemptions.

Attack Scenario

1. A legitimate validator node starts and attempts to create the vBTC v2 validator registration transaction `Fee = 0` as required).
2. The node validates the transaction locally using `TransactionValidatorService.VerifyTX()`.
3. The global fee rule rejects the transaction because `Fee <= 0`.

4. The validator cannot register/exit via the intended on-chain mechanism, and vBTC v2 flows depending on validator availability become unreliable or unavailable.

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:H/I:N/D:M/Y:M (5.0)

Recommendation

It is recommended to align fee requirements for vBTC V2 validator lifecycle transactions with global transaction validation so protocol-required validator operations cannot be rejected due to inconsistent fee rules. It is also recommended to ensure the chosen model (free lifecycle transactions or universally fee-bearing transactions) is enforced consistently across consensus validation, implementation, documentation, and tests.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by exempting `VBTC_V2_VALIDATOR_REGISTER` and `VBTC_V2_VALIDATOR_EXIT` from global fee validation in `VerifyTX()`, allowing required `Fee = 0` transactions to proceed to validator-specific checks.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/a621c463c5c4f385388ff7fb1696edb0b32931c3

7.21 INSUFFICIENT TRUST AND I/O HARDENING FOR ELECTRUMX CAN ALLOW CHAIN-DATA TAMPERING AND WITHDRAWAL DISRUPTION

// MEDIUM

Description

The ElectrumX client `ReserveBlockCore/Bitcoin/ElectrumX/Client.cs`) explicitly disables TLS certificate validation by providing an `SslStream` validation callback that always returns `true`. As a result, any attacker who can position themselves on-path (or influence DNS / routing) can transparently intercept and modify Electrum protocol responses.

This is security-relevant for vBTC V2 because withdrawals are executed by building a Bitcoin transaction using ElectrumX-provided UTXOs + previous raw transactions. A MITM can spoof UTXOs / prevouts (or simply stall / inflate responses), causing repeated withdrawal failures, incorrect local accounting, and validator process instability.

Additionally, the SSL receive loop reads unbounded data into a `StringBuilder` until `JToken.Parse()` succeeds, which provides a practical memory-exhaustion vector if the upstream is malicious or compromised.

Code Reference

- TLS certificate validation is fully bypassed:

Copy Code

```
protected async Task ConnectWithSsl()
{
    if (IsConnected)
        return;
    try
    {
        TcpClient = new TcpClient();
        await TcpClient.ConnectAsync(Host, Port);
        SslStream = new SslStream(TcpClient.GetStream(), true,
            (sender, certificate, chan, sslPolicy) => true);
        await SslStream.AuthenticateAsClientAsync(Host);
        IsConnected = true;
    }
```

- Unbounded response read until JSON parse succeeds (memory/CPU DoS surface):

Copy Code

```
private async Task<string> SendMessageWithSsl(byte[] requestData)
{
    const int bufferSize = 8192; // Use a smaller buffer size for manageable chunks
    var buffer = new byte[bufferSize];
    await SslStream.WriteAsync(requestData, 0, requestData.Length);

    var responseBuilder = new StringBuilder();

    while (true)
    {
```

```
        int read = await SslStream.ReadAsync(buffer, 0, buffer.Length);
        if (read == 0)
            break; // No more data to read

        responseBuilder.Append(Encoding.ASCII.GetString(buffer, 0, read));

        // Check if the current response forms a complete JSON
        if (IsCompleteJson(responseBuilder.ToString()))
            break;
    }

    Disconnect();
    return responseBuilder.ToString();
}
private bool IsCompleteJson(string response)
{
    response = response.Trim();
    if ((response.StartsWith("{") && response.EndsWith("}")) || // Object
        (response.StartsWith("[") && response.EndsWith("]")))   // Array
    {
        try
        {
            JToken.Parse(response);
            return true;
        }
        catch (JsonReaderException)
        {
            // Not a complete JSON
            return false;
        }
    }
    return false;
}
```

Impact

By disabling TLS verification, the node's Bitcoin-chain "source of truth" becomes trivially spoofable in realistic network threat models (public WiFi, hostile datacenter neighbor, malicious ISP, DNS poisoning, compromised router).

Attack Scenario

An attacker positions themselves as a MITM between a validator/operator and its configured ElectrumX host (or hijacks DNS for that host). Because the TLS validation callback accepts any certificate, the attacker can inject fabricated `blockchain.scripthash.listunspent` / `blockchain.transaction.get` responses or stream oversized/never-terminating payloads. The node then fails to reliably build the Taproot withdrawal transaction (or exhausts memory/CPU), causing vBTC V2 withdrawals to stall and degrading validator reliability.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

It is recommended to ensure transport authenticity for ElectrumX connections so Bitcoin chain data consumed by withdrawal logic cannot be silently manipulated by an on-path adversary. It is also recommended to harden ElectrumX response handling by enforcing bounded I/O, strict timeouts, and

protocol-aligned message parsing so malformed or oversized responses cannot degrade service or cause repeated withdrawal failures.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by hardening ElectrumX client I/O with a 10 MB maximum response size and a 30-second read timeout in `Client.SendMessageWithSsl()` and `Client.SendMessageNoSsl()`, with remediation focused on bounded I/O and failover given the intentional TLS certificate validation bypass for public ElectrumX compatibility.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/427604c42c526059cb12904d48228a19d276f2b2

7.22 WITHDRAWAL STATE MACHINE CAN ENTER A TERMINAL STATE THAT BLOCKS FUTURE WITHDRAWALS

// LOW

Description

vBTC V2 enforces “only one active withdrawal at a time” by requiring `WithdrawalStatus == None` before accepting a new `VBTC_V2_WITHDRAWAL_REQUEST`. However, the state transition on withdrawal completion sets the status to `Completed` and does not reset it back to `None`. As a result, after a successful withdrawal cycle, subsequent withdrawal requests will be rejected indefinitely, which can prevent further redemptions and cause funds to remain locked from the perspective of the vBTC V2 redemption flow.

This does not require an attacker: a legitimate withdrawal completion can put the contract into a terminal state that blocks future withdrawals, which is inconsistent with the presence of a dedicated `VBTCWithdrawalRequest` model meant to support replay protection and repeated withdrawal cycles.

Code Reference

- `ReserveBlockCore/Bitcoin/Services/VBTCService.cs` (client-side request gating)

Copy Code

```
// Check no active withdrawal exists
if (vbtcContract.WithdrawalStatus != VBTCWithdrawalStatus.None)
{
    SCLogUtility.Log($"Active withdrawal already exists. Status: {vbtcContract.WithdrawalStatus}", "VBTC
    return (false, $"Active withdrawal already exists. Status: {vbtcContract.WithdrawalStatus}");
}
```

- `ReserveBlockCore/Services/BlockTransactionValidatorService.cs` (consensus validation gating)

Copy Code

```
// Check no active withdrawal exists
if (contract.WithdrawalStatus != VBTCWithdrawalStatus.None)
{
    SCLogUtility.Log($"VBTC_V2_WITHDRAWAL_REQUEST validation failed: Active withdrawal already ex
        "BlockTransactionValidatorService.ProcessIncomingTransactions()");
    var txdata = TransactionData.GetAll();
    tx.TransactionStatus = TransactionStatus.Invalid;
    txdata.InsertSafe(tx);
    return;
}
```

- `ReserveBlockCore/Data/StateData.cs` (completion transition sets terminal status)

Copy Code

```
// Update contract status to Completed and clear active withdrawal fields
contract.WithdrawalStatus = VBTCWithdrawalStatus.Completed;
contract.ActiveWithdrawalRequestHash = null;
contract.ActiveWithdrawalAmount = 0;
contract.ActiveWithdrawalBTCDestination = null;
contract.ActiveWithdrawalFeeRate = 0;
contract.ActiveWithdrawalRequestTime = 0;

```

```
// Save updated contract
VBTCContractV2.UpdateContract(contract);
```

Impact

Once a contract reaches `WithdrawalStatus = Completed`, the vBTC V2 withdrawal request path becomes permanently unavailable for that contract, since both the client-side service and consensus validation require `WithdrawalStatus == None`. This can result in a systemic denial of service for vBTC redemptions on affected contracts, and may lock user funds within the tokenization system's intended redemption mechanism.

Attack Scenario

1. The legitimate contract owner requests a vBTC V2 withdrawal and completes it successfully (normal expected operation).
2. During completion, the contract is persisted with `WithdrawalStatus = Completed`.
3. Any later attempt to request another withdrawal (partial or subsequent redemption) is rejected because the system requires `WithdrawalStatus == None` before accepting a new request.
4. The contract's redemption flow is effectively disabled, impacting availability and potentially leaving funds locked relative to expected product behavior.

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:N/I:N/D:M/Y:M (3.1)

Recommendation

It is recommended to align the vBTC V2 withdrawal state machine with the intended redemption lifecycle by ensuring a completed withdrawal cannot permanently block future withdrawals unless that is an explicit protocol requirement. It is also recommended to document and enforce the intended model consistently across validation logic and user-facing behavior so withdrawal eligibility cannot diverge from the actual contract state.

Remediation Comment

SOLVED: The `VerifiedX` team fixed the finding by removing contract-level `WithdrawalStatus` gating in `VBTCService` and switching to per-user `VBTCWithdrawalRequest` tracking, so a terminal `Completed` state no longer blocks future withdrawals.

Client Note:
FIND-003 Fixed. The terminal withdrawal state issue has been resolved by completing the migration to per-user withdrawal tracking. The client-side VBTCService.cs now uses VBTCWithdrawalRequest.GetActiveRequest() and GetByTransactionHash() instead of checking contract.WithdrawalStatus. This was a partial fix to complete the work started in FIND-002. The contract-level WithdrawalStatus field is no longer used for gating new withdrawal requests. Each user can now have

one active withdrawal at a time tracked via VBTCWithdrawalRequest records, and after completion they can start new withdrawals immediately. The consensus-level validation was already updated in FIND-002, so both client-side and consensus validation now use the same per-user tracking model.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/d29a78a58852d427f3b3bb0ffbdc5457a471b13e

7.23 INACCURATE DEPLOYMENT DOCUMENTATION CAN CAUSE MISCONFIGURATION AND BREAK VBTC V2 MPC LIVENESS

// LOW

Description

The vBTC V2 “multi-node” documentation (`vBTC-MPC.md` and `VBTC_V2_IMPLEMENTATION_SPEC.md`) describes the FROST validator server as listening on port **19900** by default and suggests enabling it by setting `Globals.IsFrostValidator = true` in code. However, the actual default and network-specific values for `Globals.FrostValidatorPort` are **not 19900** (they are 7295 on mainnet defaults, and 17295/27295 on testnets), and `Globals.IsFrostValidator` is not exposed via `config.txt` or CLI arguments.

As a result, an operator following the documented runbook is likely to open the wrong port(s) and/or be unable to enable the FROST server without modifying code, leading to multi-node deployments where validators appear “up” but MPC ceremonies fail due to unreachable or non-started FROST endpoints.

Code Reference

- Documentation states FROST default port is 19900 and suggests enabling by setting a global flag:

Copy Code

```
### 3. Validator Configuration
**Port Requirements**:
- `Globals.FrostValidatorPort` (default: 19900) - HTTP/REST API
- `Globals.FrostValidatorPort + 1` - HTTPS (optional)

**Firewall Rules**:
- Allow incoming TCP on port 19900
- Allow outgoing HTTP to other validators

**Configuration**:

Globals.IsFrostValidator = true;  // Enable FROST server
Globals.ValidatorAddress = "RValidatorAddress";
```

- Implementation defaults differ: `Globals.FrostValidatorPort` is 7295 by default and is overridden to 17295/27295 based on network selection:

Copy Code

```
public static int ArbiterPort = 3342;
public static int FrostValidatorPort = 7295;
public static int APIPort = 7292;
public static int ValAPIPort = 7294;
public static int APIPortSSL = 7777;
```

Copy Code

```
if (config.TestNet == true || Globals.IsTestNet)
{
// ...
Globals.APIPort = 17292;
Globals.ValAPIPort = 17294;
```

```
Globals.FrostValidatorPort = 17295;
Globals.APIPortSSL = 17777;
// ...
}
```

- The flag described in docs is not operator-configurable and defaults to false:

Copy Code

```
public static bool IsBlockCaster = false;
public static bool IsWardenMonitoring = false;
public static bool IsFrostValidator = false;
public static bool IsValidatorPortOpen = false;
public static bool IsValidatorAPIPortOpen = false;
public static bool IsFROSTAPIPortOpen = false;
```

Impact

Multi-node vBTC V2 deployments are sensitive to correct port configuration and service activation. If operators open port 19900 as instructed but the implementation listens on 7295/17295/27295, coordinator and validators will not be able to reach required `/frost/*` endpoints. Similarly, if operators cannot enable the FROST server via config/CLI, validators may never run the service even when vBTC V2 flows depend on it. This manifests as “liveness failures” in DKG/signing ceremonies (contract creation and withdrawals failing), degraded operational readiness checks, and prolonged troubleshooting time.

BVSS

AO:A/AC:L/AX:L/R:F/S:U/C:N/A:H/I:N/D:M/Y:M (2.5)

Recommendation

It is recommended to ensure the multi-node runbook remains authoritative by aligning it with the implemented FROST port selection and activation behavior, including network-specific overrides. It is also recommended to ensure FrostValidatorPort configuration and precedence rules are clearly documented so operators can deploy validator nodes deterministically.

Remediation Comment

SOLVED: The **VerifiedX team** fixed the finding by updating the multi-node documentation to accurately reflect the existing runtime FROST port selection and validator-mode FROST server startup behavior, resolving the runbook mismatch without requiring code changes.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/549e8090b479d1f78c65db729b6a4a799435b3c8

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.