QuillAudits

AUDIT REPORT

June 2026

For

MST BLOCKCHAIN

YOUR PATH TO A DECENTRALIZED FUTURE

Table of Content

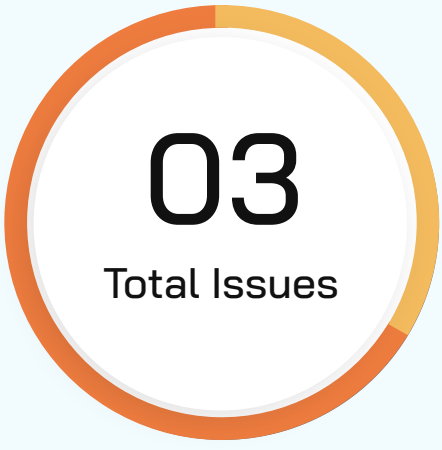
Executive Summary

Project Name	MST Blockchain
Protocol Type	L1 chain
Project URL	https://mstblockchain.com/
Overview	MST Chain is a Layer 1 blockchain presented as an EVM-compatible network with low fees and high throughput.It is as a fork of the BNB Smart Chain, using a familiar account model and developer stack. The project's native token, MSTC, is used for gas fees and ecosystem activity across the network.
Audit Scope	The scope of this Audit was to analyze the MST Blockchain Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/mst-chain/mst-chain
Branch	Main
Commit Hash	5c7d31a4251aac7cec24355505e3fd500bff3985
Language	Go
Blockchain	MST Fork of BNB Chain network
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	15th June 2026 - 19th June 2026
Updated Code Received	27th June 2026
Review 2	30th June 2026
Fixed In	f111bc5ea8ced713d8466b8f49c2f7f0cf15fc13

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0(0.0%)
High	0(0.0%)
Medium	3(75.0%)
Low	1 (25.0%)
Informational	0(0.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	0	**3**	**1**	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Hardcoded Multiplier in conf1g . go	Medium	Resolved
2	Missing Block Interval Configuration Parameter	Medium	Resolved
3	No Configuration Validation at Startup	Medium	Resolved
2	Missing Block Interval Configuration Parameter	Low	Resolved

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception Disorder
- Gasless Send
- Balance Equality
- Byte Array
- Transfer Forwards All Gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Medium Issues

Hardcoded Multiplier in config.go

Resolved

Path

cnd/geth/conf1g. go line 304

Description

The blob parameter MinTimeDurationForBlobRequests is recalculated using a hardcoded multiplier (3.0) instead of deriving it from the block interval. This is a critical maintenance bug because if a configuration file with incorrect values is loaded, the code will silently recalculate wrong parameters with no validation.

Root cause code

```
params.MinTimeDurationForBlobRequests =
uint64(float64(params.MinBlocksForBlobRequests) * 3.0)
```

If MinBlocksForBlobRequests is incorrect (e.g., Fermi value of 3,494,400 instead of MST's 524,160), this multiplier produces 10,483,200 seconds (121 days) instead of expected 1,572,480 seconds (18.2 days). It will result in a 6.67× blob retention miscalculation and the disk would fill unexpectedly in weeks, which may lead to data corruption from disk full conditions.

Example Scenario:

Load Fermi config with MinBlocksForBlobRequests = 3,494,400 on MST node. The hardcoded 3.0 multiplier calculates blob retention as 121 days instead of 18.2 days. It will cause disk space to fill up 6.67× faster, causing node failure in weeks.

Recommendation

Replace hardcoded 3.0 with dynamic block interval calculation:
blockIntervalSeconds := float64(params.BlockIntervalMillis) / 1000.0
params.MinTimeDurationForBlobRequests = uint64(float64(params.MinBlocksForBlobRequests) * blockIntervalSeconds)

Medium Issues

Missing Block Interval Configuration Parameter

Resolved

Path

params/protoco1_parans.go

Description

The block interval (3.0 seconds on MST) is hardcoded at multiple places, making it difficult to change consistently during upgrades. When the block interval must change (e.g., hard fork to 2.5 seconds), developers must edit multiple files. Forgetting even one location means blob retention calculations become 20-30% wrong in that location, causing silent data integrity issues.

Root cause

Some (not all) code references:

```
// params/protocol_params.go
MinBlocksForBlobRequests = uint64(float64(MinTimeDurationForBlobRequests) /
3.0)
DefaultExtraReserveForBlobRequests uint64 = uint64(24 * 3600 / 3.0)

// cmd/geth/config.go
params.MinTimeDurationForBlobRequests =
uint64(float64(params.MinBlocksForBlobRequests) * 3.0)

// core/rawdb/chain_freezer.go
expectedTime := uint64(numBlocks) * 3.0
```

Example Scenario:

Hard fork changes block interval from 3.0s to 2.5s. Engineer updates params/protocol_params.go but forgets cmd/geth/config.go, causing one file to calculate with 2.5s, another with 3.0s, which may cause blob retention to be inconsistent.

Recommendation

Create a single BlockIntervalMillis constant and helper function to be used everywhere:

```
// params/protocol_params.go - SINGLE SOURCE OF TRUTH
const BlockIntervalMillis uint64 = 3000  // milliseconds

func GetBlockIntervalSeconds() float64 {
    return float64(BlockIntervalMillis) / 1000.0
}

// params/protocol_params.go:
MinBlocksForBlobRequests = uint64(float64(MinTimeDurationForBlobRequests) /
GetBlockIntervalSeconds())
DefaultExtraReserveForBlobRequests uint64 = uint64(24 * 3600 /
GetBlockIntervalSeconds())
```

```
// cmd/geth/config.go:
params.MinTimeDurationForBlobRequests =
uint64(float64(params.MinBlocksForBlobRequests) * GetBlockIntervalSeconds())

// core/rawdb/chain_freezer.go:
expectedTime := uint64(numBlocks) * uint64(GetBlockIntervalSeconds())
```

Important

Use GetBlockIntervalSeconds() consistently everywhere. This ensures that if the block interval ever changes during a hard fork, all calculations automatically use the new value without requiring code changes in multiple locations. Never use hardcoded multipliers (3.0) or divide by BlockIntervalMillis directly, as that causes integer division truncation (e.g., 86400 / 3000 = 28 instead of 28800). Always cast to float first to ensure proper division: uint64(float64(24*3600) / GetBlockIntervalSeconds()).

Medium Issues

No Configuration Validation at Startup

Resolved

Path

cmd/geth/config.go

Description

The blob parameters have a required mathematical relationship: MinBlocksForBlobRequests × 3.0 = MinTimeDurationForBlobRequests (using MST's hardcoded 3.0 second block interval). If this breaks due to config errors, manual edits, or migration mistakes, the node silently continues with wrong blob retention calculations. This misconfiguration propagates to the chain freezer's pruning logic, causing blobs to be deleted prematurely. When validators later request those deleted blobs, data availability checks fail, breaking consensus and halting the chain. Additionally, no error message alerts operators until chain failure occurs.

Root cause

Some (not all) code references:

```
// cmd/geth/config.go (line 304)
params.MinTimeDurationForBlobRequests =
uint64(float64(params.MinBlocksForBlobRequests) * 3.0)

// No validation happens anywhere

// If params.MinBlocksForBlobRequests is wrong, the recalculation is silently
wrong too
```

Example Scenario:

Protocol Params file has MinBlocksForBlobRequests = 3,494,400 (Fermi value, wrong for MST). MinTimeDurationForBlobRequests will be recalculated as 3,494,400 × 3.0 = 10,483,200 seconds instead of expected 1,572,480. Node will start silently accepting 6.67× wrong blob retention, causing disk corruption in weeks.

Recommendation - Part 1: Startup Validation

Add validation at node startup that checks whether the three parameters maintain their required mathematical relationship.

Potential Suggestion:

```
func ValidateBlobConfiguration() error {
    const blockIntervalSeconds = 3.0  // MST block interval (hardcoded until
Bug #2 creates BlockIntervalMillis)
    expectedDuration := uint64(float64(params.MinBlocksForBlobRequests) *
blockIntervalSeconds)

    // Allow 0.5% tolerance for floating-point rounding errors
    // For MST (1,572,480 sec), this is ~7,862 seconds (2.2 hours tolerance)
    // Real config errors (like Fermi vs MST) are 6.67× difference, far beyond
this tolerance
    const tolerancePercent = 0.005  // 0.5% - adjust based on your precision
requirements
    tolerance := math.Abs(float64(params.MinTimeDurationForBlobRequests) *
tolerancePercent)
    actualDifference := math.Abs(float64(expectedDuration) -
float64(params.MinTimeDurationForBlobRequests))

    if actualDifference > tolerance {
        return fmt.Errorf(
            "CRITICAL: Blob configuration mismatch!\n"+
            "  Expected: %d seconds (from %d blocks × %.2f sec/block)\n"+
            "  Actual: %d seconds\n",
            expectedDuration, params.MinBlocksForBlobRequests,
blockIntervalSeconds,
            params.MinTimeDurationForBlobRequests,
        )
    }
    return nil
}

// Call the function at node startup
```

Note: This validation uses a hardcoded 3.0 seconds block interval. Once Bug #2 creates a BlockIntervalMillis constant and GetBlockIntervalSeconds() helper function, use the helper instead.

Recommendation - Part 2: Chain Freezer Validation

Chain freezer (core/rawdb/chain_freezer.go) also relies on the configuration being correct before making pruning decisions. While detailed implementation is beyond this scope, the chain freezer should also validate the blob configuration before proceeding with data pruning. Consider adding similar validation checks in the freezer's pruning logic to prevent cascading failures if the configuration is incorrect

Real-World Failure Scenario for Chain Freezer:

This scenario demonstrates why validation is critical. A misconfiguration in params/protocol_params.go silently propagates through core/rawdb/chain_freezer.go, causing blobs to be pruned prematurely. When validators later request those deleted blobs, data availability checks fail, breaking consensus.

A config mistake makes MinBlocksForBlobRequests = 100,000 (should be 524,160)

Step 1: Wrong Configuration Loaded into params/protocol_params.go

```
  EXPECTED: MinBlocksForBlobRequests = 524,160 (MST correct value)
  ACTUAL: MinBlocksForBlobRequests = 100,000 (loaded from older version or
manual mistake)
  EXPECTED: Validation check should be: 100,000 × 3.0 = 300,000 ≠ 1,572,480 →
FAIL LOUDLY
  ACTUAL: No validation occurs → node starts silently with wrong config
```

Step 2: Chain Freezer (core/rawdb/chain_freezer.go) uses wrong config for pruning

```
 Current Block: 1,000,000
  EXPECTED: Reserve threshold = 524,160 + 28,800 = 552,960 blocks
            Prune anything older than: 1,000,000 - 552,960 = 447,040
  ACTUAL: Reserve threshold = 100,000 + 28,800 = 128,800 blocks (wrong!)
            Prune anything older than: 1,000,000 - 128,800 = 871,200

  Result: Blocks 0-871,200 are deleted
          BUT blobs created near block 871,200 should still be alive
          Those blobs should be retained until: 871,200 + 524,160 = 1,395,360
          However, they were already deleted during pruning
```

Scenario B: Fermi Config Used on MST

```
DevOps loads Fermi backup config by mistake:
  MinBlocksForBlobRequests = 3,494,400 (Fermi value)
  MinTimeDurationForBlobRequests = 1,572,480 (Fermi value)
  Current block interval: 3.0 seconds (MST, hardcoded)

Expected calculation for MST:
  3,494,400 × 3.0 = 10,483,200 seconds ≠ 1,572,480

Current behavior: NODE STARTS with wrong blob retention (6.67× miscalculation)
Expected behavior: NODE FAILS LOUDLY with validation error at startup
```

Low Severity Issues

Missing Relationship Documentation and Code Organization

Resolved

Path

params/protocol_params.go

Description

The blob parameters have an interdependent mathematical relationship that is not properly documented. Additionally, Fermi configuration code is interleaved with active MST code, making it hard to distinguish which version is active and creating risk of accidentally uncommenting wrong values during maintenance.

Recommendation

Add clear documentation explaining the relationship between parameters and separate deprecated and active configurations with clear visual barriers and comments explaining what will happen if wrong values are used:

```
// FERMI Configuration (Deprecated - Only for Reference)
//
═══════════════════════════════════════════════════════════════════════════
══
// Block interval: 450 ms (0.45 seconds)
// DO NOT UNCOMMENT as it will cause 6.67× blob retention miscalculation on MST
//
// fermiBlockInterval = 0.45
// MinBlocksForBlobRequests = uint64(float64(MinTimeDurationForBlobRequests) /
0.45)       // 3,494,400 blocks
// DefaultExtraReserveForBlobRequests = uint64(24 * 3600 /
0.45)                          // 192,000 blocks

//
═══════════════════════════════════════════════════════════════════════════
══
// MST Configuration (Active)
//
═══════════════════════════════════════════════════════════════════════════
══
// Block interval: 3000 ms (3.0 seconds)
// MinTime = MinBlocks × 3
// If you change one, you MUST update others to maintain this relationship

// Time parameters: Keep blobs for 18.2 days (wall clock time)
MinTimeDurationForBlobRequests uint64 = uint64(float64(24*3600) *
18.2)                    // 1,572,480 seconds (18.2 days)

// Block parameters: Derived from time using Block interval = 3
// Calculated: MinTimeDurationForBlobRequests / 3 = MinBlocksForBlobRequests
```

```
// DO NOT change MinTimeDurationForBlobRequests without recalculating
MinBlocksForBlobRequests!
MinBlocksForBlobRequests uint64 =
uint64(float64(MinTimeDurationForBlobRequests) / 3.0)    // 524,160 blocks
// Extra safety margin: 1 day buffer in blocks for latency tolerance
// Calculated: 86400 / 3 = ExtraBlocks
DefaultExtraReserveForBlobRequests uint64 = uint64(24 * 3600 /
3.0)                        // 28,800 blocks (1 da
```

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of MST Blockchain. We performed our audit according to the procedure described above.

No critical issues in MST Blockchain, just 3 issues of medium severity and 1 Low severity were found, which were resolved by the team.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

June 2026

For

MST BLOCKCHAIN

YOUR PATH TO A DECENTRALIZED FUTURE

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com