Giant Mammoth

Smart Contract Security Audit

Prepared by ShellBoxes

April 3rd, 2023 - April 10th, 2023

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Giant Mammoth Chain
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/MammothDevMaster/giantmammoth/tree/main/genesis/contracts	aef463d43ee7a1026d50e44d85b7c628f42ea62c

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Giant Mammoth Chain engaged ShellBoxes to conduct a security assessment on the Giant Mammoth beginning on April 3rd, 2023 and ending April 10th, 2023. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Giant Mammoth Chain

Giant Mammoth Chain solves the problem of scalability and security and builds a high-level network. It is designed for applications that build their own chain, including higher speeds and lower network gas costs than before, EVM compatibility, and risk mitigation.

GMMT is a project that started with inspiration from the Layer 1 and Layer 2 solutions that are currently attracting a lot of attention. It is designed to go beyond the limitations of a Layer 2 chain belonging to one Layer 1 chain, and ultimately build a true multi-chain by belonging to multiple Layer 1 chains.

Issuer	Giant Mammoth Chain
Website	`https://www.mmtchain.io/`
Type	Solidity Smart Contract
Whitepaper	`https://gmmtchain.io/whitepaper/giant_mammoth_whitepaper_en.pdf`
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact				
	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low

Likelihood

2 Findings Overview

2.1 Disclaimer

During the audit, it was noted that the RuntimeUpgrade contract performs external calls to another contract called the RuntimeUpgradeEvmHook. It is important to note that the RuntimeUpgradeEvmHook contract is out of scope for this audit and was not reviewed as part of this assessment.

While the RuntimeUpgradeEvmHook contract is out of scope, it is assumed that the contract has been thoroughly tested and will always act as intended. However, it is important to keep in mind that any issues or vulnerabilities within the RuntimeUpgradeEvmHook contract could potentially affect the security and reliability of the RuntimeUpgrade contract.

2.2 Summary

The following is a synopsis of our conclusions from our analysis of the Giant Mammoth implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.3 Key Findings

In general, the genesis smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , 1 high-severity, 5 medium-severity, 6 low-severity, 1 informational-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Lost Shares In The advanceStakingRewards Modifier	HIGH	Acknowledged
SHB.2. Potential Desynchronization Between Staking and StakingPool Contracts	MEDIUM	Acknowledged

SHB.3. Division Before Multiplication Can Cause a Precision Loss in Reward Calculation	MEDIUM	Fixed
SHB.4. Mismatch Between Whitepaper and Code Implementation on Reward Allocation	MEDIUM	Fixed
SHB.5. Front run attack vector	MEDIUM	Acknowledged
SHB.6. Banned Deployer Can Still Deploy Contracts	MEDIUM	Fixed
SHB.7. Usage of .transfer() to Transfer Ether	LOW	Fixed
SHB.8. Mismatch Between Whitepaper and Code Implementation on Validator Selection	LOW	Acknowledged
SHB.9. Missing Value Verification	LOW	Fixed
SHB.10. Lack of Check for Contract Address	LOW	Fixed
SHB.11. Inaccurate Comparison in _claimSystemFee Function	LOW	Fixed
SHB.12. Floating Pragma	LOW	Acknowledged
SHB.13. Missing Setter For The _systemTreasury	INFORMATIONAL	Fixed

3 Finding Details

SHB.1 Lost Shares In The advanceStakingRewards Modifier

- Severity : HIGH
- Status : Acknowledged
- Likelihood : 3
- Impact : 2

Description:

The advanceStakingRewards modifier is responsible for re-delegating previous rewards, but it has been identified that the StakingPool contract does not provide the staker with shares as it does in the stake function implementation.

This issue can result in incorrect calculation of the staked amount and the rewards, leading to discrepancies between the actual rewards and the rewards distributed to the staker. Additionally, it can lead to confusion for users who will need to call the Staking contract directly to undelegate or claim their rewards.

Files Affected:

SHB.1.1: StakingPool.sol

```
71 modifier advanceStakingRewards(address validator) {
72     {
73         ValidatorPool memory validatorPool = _getValidatorPool(validator)
               ↪ ;
74         // claim rewards from staking contract
75         (uint256 stakedAmount, uint256 dustRewards) =
               ↪ _calcUnclaimedDelegatorFee(validatorPool);
76         _stakingContract.claimDelegatorFee(validator);
77         // re-delegate just arrived rewards
78         if (stakedAmount > 0) {
79             _stakingContract.delegate{value : stakedAmount}(validator);
80         }
```

```
81          // increase total accumulated rewards
82          validatorPool.totalStakedAmount += stakedAmount;
83          validatorPool.dustRewards = dustRewards;
84          // save validator pool changes
85          _validatorPools[validator] = validatorPool;
86      }
87      _;
88  }
```

SHB.1.2: StakingPool.sol

```
118 function stake(address validator) external payable advanceStakingRewards
        ↪ (validator) override {
119     ValidatorPool memory validatorPool = _getValidatorPool(validator);
120     uint256 shares = msg.value * _calcRatio(validatorPool) / 1e18;
121     // increase total accumulated shares for the staker
122     _stakerShares[validator][msg.sender] += shares;
123     // increase staking params for ratio calculation
124     validatorPool.totalStakedAmount += msg.value;
125     validatorPool.sharesSupply += shares;
126     // save validator pool
127     _validatorPools[validator] = validatorPool;
128     // delegate these tokens to the staking contract
129     _stakingContract.delegate{value : msg.value}(validator);
130     // emit event
131     emit Stake(validator, msg.sender, msg.value);
132 }
```

Recommendation:

To address this issue, we recommend modifying the code to include the calculation of shares when redeeming rewards in the advanceStakingRewards modifier. This can be achieved by including the same calculation of shares as in the stake function implementation, ensuring consistency across the contract.

Updates

The Giant Mammoth Chain team acknowledged the issue, stating that the contract is not being used, and there have been no transactions on the StakingPool contract for the last 6 months.

SHB.2 Potential Desynchronization Between Staking and StakingPool Contracts

- Severity : MEDIUM
- Status : Acknowledged
- Likelihood : 3
- Impact : 1

Description:

The StakingPool contract contains a stake function, which is designed to allow users to delegate their funds to a validator. This function calls the delegate function in the Staking contract to perform the staking. However, it has been identified that a user can directly call the delegate function, bypassing the stake function and causing a desynchronization between the StakingPool and Staking contracts.

This issue can lead to a desynchronization of staked funds, which can cause a confusion for both the stakers and the validators. The same issue applies to the unstake function with the undelegate function.

Files Affected:

SHB.2.1: StakingPool.sol

```
118 function stake(address validator) external payable advanceStakingRewards
    ↪ (validator) override {
119     ValidatorPool memory validatorPool = _getValidatorPool(validator);
120     uint256 shares = msg.value * _calcRatio(validatorPool) / 1e18;
121     // increase total accumulated shares for the staker
122     _stakerShares[validator][msg.sender] += shares;
```

```
123     // increase staking params for ratio calculation
124     validatorPool.totalStakedAmount += msg.value;
125     validatorPool.sharesSupply += shares;
126     // save validator pool
127     _validatorPools[validator] = validatorPool;
128     // delegate these tokens to the staking contract
129     _stakingContract.delegate{value : msg.value}(validator);
130     // emit event
131     emit Stake(validator, msg.sender, msg.value);
132 }
```

SHB.2.2: Staking.sol

```
184 function delegate(address validatorAddress) payable external override {
185     _delegateTo(msg.sender, validatorAddress, msg.value);
186 }
```

SHB.2.3: StakingPool.sol

```
134     function unstake(address validator, uint256 amount) external
            ↪ advanceStakingRewards(validator) override {
135         ValidatorPool memory validatorPool = _getValidatorPool(validator)
                ↪ ;
136         require(validatorPool.totalStakedAmount > 0, "StakingPool:
                ↪ nothing to unstake");
137         // make sure user doesn't have pending undelegates (we don't
                ↪ support it here)
138         require(_pendingUnstakes[validator][msg.sender].epoch == 0, "
                ↪ StakingPool: undelegate pending");
139         // calculate shares and make sure user have enough balance
140         uint256 shares = amount * _calcRatio(validatorPool) / 1e18;
141         require(shares <= _stakerShares[validator][msg.sender], "
                ↪ StakingPool: not enough shares");
142         // save new undelegate
143         IChainConfig chainConfig = IInjector(address(_stakingContract)).
                ↪ getChainConfig();
```

```
144         _pendingUnstakes[validator][msg.sender] = PendingUnstake({
145         amount : amount,
146         shares : shares,
147         epoch : _stakingContract.nextEpoch() + chainConfig.
                ↪ getUndelegatePeriod()
148         });
149         validatorPool.pendingUnstake += amount;
150         _validatorPools[validator] = validatorPool;
151         // undelegate
152         _stakingContract.undelegate(validator, amount);
153         // emit event
154         emit Unstake(validator, msg.sender, amount);
155     }
```

SHB.2.4: Staking.sol

```
188     function undelegate(address validatorAddress, uint256 amount)
            ↪ external override {
189         _undelegateFrom(msg.sender, validatorAddress, amount);
190     }
```

Recommendation:

To mitigate this issue, we recommend modifying the code so that the delegate function can only be invoked by the StakingPool contract. This can be achieved by adding a modifier to the delegate function that checks if the caller is the StakingPool contract. By doing so, the integrity and security of the staking process can be maintained. The same recommendation goes for the unstake function with the undelegate function.

Updates

The Giant Mammoth Chain team acknowledged the issue, stating that the contract is not being used, and there have been no transactions on the contract for the last 6 months. Furthermore, the team is unable to migrate existing staked users to StakingPool.

SHB.3 Division Before Multiplication Can Cause a Precision Loss in Reward Calculation

- Severity : MEDIUM
- Likelihood : 2
- Status : Fixed
- Impact : 2

Description:

It has been identified that there is a division before multiplication in the reward calculation in the Staking contract, which can cause a precision loss. This issue can result in inaccurate calculation of rewards, leading to discrepancies between the actual rewards and the rewards distributed to the staker. It can also lead to confusion for users who may expect the reward calculation to be accurate.

Files Affected:

SHB.3.1: Staking.sol

```
452 function _calcValidatorSnapshotEpochPayout(ValidatorSnapshot memory
        ↪ validatorSnapshot) internal view returns (uint256 delegatorFee,
        ↪ uint256 ownerFee, uint256 systemFee) {
453     uint256 totalDelegatedAmount = 0;
454     for(uint256 i=0; i <_activeValidatorsList.length; i++){
455     address validatorAddress = _activeValidatorsList[i];
456     Validator memory validator = _validatorsMap[validatorAddress];
457     ValidatorSnapshot memory snapshot = _validatorSnapshots[validator.
            ↪ validatorAddress][validator.changedAt];
458     totalDelegatedAmount += uint256(snapshot.totalDelegated) *
            ↪ BALANCE_COMPACT_PRECISION;
459     }
460     uint256 addEcoReward = validatorSnapshot.totalRewards +
            ↪ ECOSYSTEM_REWARD * validatorSnapshot.totalDelegated /
            ↪ totalDelegatedAmount * BALANCE_COMPACT_PRECISION ;
```

Recommendation:

To address this issue, we recommend modifying the code to perform the multiplication before the division, ensuring that the precision is maintained, resulting in a more accurate reward calculation.

Updates

The Giant Mammoth Chain team resolved the issue by performing the multiplications before the division to preserve more precision in the output.

SHB.4 Mismatch Between Whitepaper and Code Implementation on Reward Allocation

- Severity : MEDIUM
- Likelihood : 2
- Status : Fixed
- Impact : 2

Description:

The Giant Mammoth Whitepaper states that 1/16 of the compensation from each transaction goes to the system funds, which can be used for system needs, such as applying bridging costs. However, the code implementation only sends rewards to the system treasury if the validator is slashed.

This discrepancy can lead to confusion for users and investors who may expect the reward allocation to match the description in the whitepaper. Additionally, it can result in incorrect calculation of rewards and the misallocation of funds.

Files Affected:

SHB.4.1: Staking.sol

```
452 function _calcValidatorSnapshotEpochPayout(ValidatorSnapshot memory
      ↪ validatorSnapshot) internal view returns (uint256 delegatorFee,
      ↪ uint256 ownerFee, uint256 systemFee) {
```

```
453     uint256 totalDelegatedAmount = 0;
454     for(uint256 i=0; i <_activeValidatorsList.length; i++){
455     address validatorAddress = _activeValidatorsList[i];
456     Validator memory validator = _validatorsMap[validatorAddress];
457     ValidatorSnapshot memory snapshot = _validatorSnapshots[validator.
            ↪ validatorAddress][validator.changedAt];
458     totalDelegatedAmount += uint256(snapshot.totalDelegated) *
            ↪ BALANCE_COMPACT_PRECISION;
459     }
460     uint256 addEcoReward = validatorSnapshot.totalRewards +
            ↪ ECOSYSTEM_REWARD * validatorSnapshot.totalDelegated /
            ↪ totalDelegatedAmount * BALANCE_COMPACT_PRECISION ;
461     // detect validator slashing to transfer all rewards to treasury
462     if (validatorSnapshot.slashesCount >= _chainConfigContract.
            ↪ getMisdemeanorThreshold()) {
463         return (delegatorFee = 0, ownerFee = 0, systemFee =
                ↪ validatorSnapshot.totalRewards);
464     } else if (validatorSnapshot.totalDelegated == 0) {
465         return (delegatorFee = 0, ownerFee = addEcoReward, systemFee = 0)
                ↪ ;
466     }
467     // ownerFee_(18+4-4=18) = totalRewards_18 * commissionRate_4 / 1e4
468     ownerFee = addEcoReward * validatorSnapshot.commissionRate / 1e4 ;
469     // delegatorRewards = totalRewards - ownerFee
470     delegatorFee = addEcoReward - ownerFee;
471     // default system fee is zero for epoch
472     systemFee = 0;
473 }
```

Recommendation:

To address this issue, we recommend modifying the code to match the description in the whitepaper. This can be achieved by adding logic to the reward allocation function to allocate 1/16 of the compensation to the system funds for each transaction, as described in the

whitepaper. By doing so, the reward allocation will be consistent with the whitepaper, ensuring accuracy and transparency in the reward distribution process.

Updates

The Giant Mammoth Chain team resolved the issue, stating that the allocation is being performed by the mainnet core and not in the contracts.

SHB.5 Front run attack vector

- Severity : MEDIUM
- Status : Acknowledged
- Likelihood : 3
- Impact : 1

Description:

All the contracts use a function called ctor to initialize the state, which can be front-runned by an attacker.

This issue can result in the initialization of the contract state with malicious or incorrect values, leading to potential security vulnerabilities or incorrect functionality of the contract.

The Injector contract also has two functions, namely init and initManually, that are used to initialize the contract, and both functions are vulnerable to front-run attacks.

Exploit Scenario:

A front-run attack occurs when an attacker listens to the mempool and detects a transaction to the ctor or init function with a low gas price. The attacker then submits a transaction to the same function with a higher gas price, effectively replacing the original transaction in the mempool. This allows the attacker to control the initialization of the contract state with malicious or incorrect values.

The initManually function is particularly susceptible because it requires contract addresses to be passed as parameters, which can be intercepted and manipulated by an attacker. As a result, an attacker could potentially call the initManually function after contract

deployment and initialize all the contracts with incorrect or malicious values, leading to potential security vulnerabilities or incorrect functionality of the contract.

Files Affected:

SHB.5.1: ChainConfig.sol

```
function ctor(
    uint32 activeValidatorsLength,
    uint32 epochBlockInterval,
    uint32 misdemeanorThreshold,
    uint32 felonyThreshold,
    uint32 validatorJailEpochLength,
    uint32 undelegatePeriod,
    uint256 minValidatorStakeAmount,
    uint256 minStakingAmount
) external whenNotInitialized {
```

SHB.5.2: DeployerProxy.sol

```
function ctor(address[] memory deployers) external whenNotInitialized {
    for (uint256 i = 0; i < deployers.length; i++) {
        _addDeployer(deployers[i]);
    }
}
```

SHB.5.3: Governance.sol

```
function ctor(uint256 newVotingPeriod) external whenNotInitialized {
    _setVotingPeriod(newVotingPeriod);
}
```

SHB.5.4: RuntimeUpgrade.sol

```
function ctor(address evmHookAddress) external whenNotInitialized {
    _evmHookAddress = evmHookAddress;
}
```

SHB.5.5: Staking.sol

```
function ctor(address[] calldata validators, uint256[] calldata initialStakes, uint16 commissionRate) external whenNotInitialized {
```

SHB.5.6: SystemRewards.sol

```
function ctor(address[] calldata accounts, uint16[] calldata shares) external whenNotInitialized {
```

SHB.5.7: Injector.sol

```
function init() external initializer {
    // BSC compatible addresses
    _stakingContract = IStaking(0x0000000000000000000000000000000000001000);
    _slashingIndicatorContract = ISlashingIndicator(0x0000000000000000000000000000000000001001);
    _systemRewardContract = ISystemReward(0x0000000000000000000000000000000000001002);
    // BAS defined addresses
    _stakingPoolContract = IStakingPool(0x0000000000000000000000000000000000007001);
    _governanceContract = IGovernance(0x0000000000000000000000000000000000007002);
    _chainConfigContract = IChainConfig(0x0000000000000000000000000000000000007003);
    _runtimeUpgradeContract = IRuntimeUpgrade(0x0000000000000000000000000000000000007004);
    _deployerProxyContract = IDeployerProxy(0x0000000000000000000000000000000000007005);
    // invoke constructor
    _invokeContractConstructor();
}
```

SHB.5.8: Injector.sol

```
77 function initManually(
78     IStaking stakingContract,
79     ISlashingIndicator slashingIndicatorContract,
80     ISystemReward systemRewardContract,
81     IStakingPool stakingPoolContract,
82     IGovernance governanceContract,
83     IChainConfig chainConfigContract,
84     IRuntimeUpgrade runtimeUpgradeContract,
85     IDeployerProxy deployerProxyContract
86 ) public initializer {
87     // BSC-compatible
88     _stakingContract = stakingContract;
89     _slashingIndicatorContract = slashingIndicatorContract;
90     _systemRewardContract = systemRewardContract;
91     // BAS-defined
92     _stakingPoolContract = stakingPoolContract;
93     _governanceContract = governanceContract;
94     _chainConfigContract = chainConfigContract;
95     _runtimeUpgradeContract = runtimeUpgradeContract;
96     _deployerProxyContract = deployerProxyContract;
97     // invoke constructor
98     _invokeContractConstructor();
99 }
```

Recommendation:

To address this issue, we recommend deploying the contract and executing the ctor or the init/initManually function in the case of the Injector in the same transaction to prevent front-run attacks, or adding access control to the ctor and init/initManually functions, so it cannot be initialized by anyone.

Updates

The Giant Mammoth Chain team acknowledged the issue, stating that the ctor function was never called other than when the genesis block was created.

SHB.6 Banned Deployer Can Still Deploy Contracts

- Severity : MEDIUM
- Likelihood : 3
- Status : Fixed
- Impact : 1

Description:

The _registerDeployedContract function checks that the deployer is allowed by checking the isDeployer function. However, it doesn't check if the deployer is banned. A banned deployer might still deploy contracts.

Files Affected:

SHB.6.1: DeployerProxy.sol

```
108 function _registerDeployedContract(address deployer, address impl)
        ↪ internal {
109     // make sure this call is allowed
110     require(isDeployer(deployer), "Deployer: deployer is not allowed");
111     // remember who deployed contract
112     SmartContract memory dc = _smartContracts[impl];
113     require(dc.impl == address(0x00), "Deployer: contract is deployed
            ↪ already");
114     dc.state = ContractState.Enabled;
115     dc.impl = impl;
116     dc.deployer = deployer;
117     _smartContracts[impl] = dc;
118     // emit event
```

```
119         emit ContractDeployed(deployer, impl);
120     }
```

Recommendation:

We recommend adding a check for banned deployer before registering the deployed contract.

SHB.6.2: DeployerProxy.sol

```
    require(!isBanned(deployer), "Deployer: deployer is banned");
```

Updates

The Giant Mammoth Chain team resolved the issue by adding a require check that makes sure the deployer is not banned when the _registerDeployedContract is being called.

SHB.7 Usage of .transfer() to Transfer Ether

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

During the audit, it was noted that the project is using the .transfer() function to transfer ether between contracts. While .transfer() is a built-in function in Solidity and provides a quick and easy way to transfer ether, it is important to note that it is a dangerous function to use. Although transfer() and send() are recommended as a security best-practice to prevent reentrancy attacks because they only forward 2300 gas, the gas repricing of opcodes may break deployed contracts.

Files Affected:

SHB.7.1: StakingPool.sol

```
176 delete _pendingUnstakes[validator][msg.sender];
177 // its safe to use call here (state is clear)
178 require(address(this).balance >= amount, "StakingPool: not enough
        ↪ balance");
179 payable(address(msg.sender)).transfer(amount);
180 // emit event
181 emit Claim(validator, msg.sender, amount);
```

SHB.7.2: SystemReward.sol

```
102 if (_systemTreasury != address(0x00)) {
103     address payable payableTreasury = payable(_systemTreasury);
104     payableTreasury.transfer(amountToPay);
105     emit FeeClaimed(_systemTreasury, amountToPay);
106     return;
107 }
```

SHB.7.3: SystemReward.sol

```
110 for (uint256 i = 0; i < _distributionShares.length; i++) {
111     DistributionShare memory ds = _distributionShares[i];
112     uint256 accountFee = amountToPay * ds.share / SHARE_MAX_VALUE;
113     payable(ds.account).transfer(accountFee);
114     emit FeeClaimed(ds.account, accountFee);
115     totalPaid += accountFee;
116 }
```

Recommendation:

Consider using .call{ value: ... }("") instead, without hard-coded gas limits along with reentrancy guards for reentrancy protection.

Updates

The Giant Mammoth Chain team resolved the issue by implementing the use of .call{ value: ... }("") for transferring ETH from the contract.

SHB.8 Mismatch Between Whitepaper and Code Implementation on Validator Selection

- Severity : LOW
- Likelihood : 2
- Status : Acknowledged
- Impact : 1

Description:

The Giant Mammoth Whitepaper states that the node consists of 21 validators, and that new validators with the most GMMT staking are selected each day. However, the code does not verify the number of validators to be 21, and the number of validators is modifiable by the governance. This discrepancy can lead to confusion for users and investors, who may expect the number of validators to be fixed at 21, as described in the whitepaper. Additionally, it can result in incorrect calculation of rewards and the misallocation of funds.

Files Affected:

SHB.8.1: ChainConfig.sol

```
65 function setActiveValidatorsLength(uint32 newValue) external override
       ↪ onlyFromGovernance {
66     uint32 prevValue = _consensusParams.activeValidatorsLength;
67     _consensusParams.activeValidatorsLength = newValue;
68     emit ActiveValidatorsLengthChanged(prevValue, newValue);
69 }
```

Recommendation:

To address this issue, we recommend modifying the code to match the description in the whitepaper. This can be achieved by adding logic to the contract to ensure that the number of validators is fixed at 21, and that new validators with the most GMMT staking are selected each day. By doing so, the contract will be consistent with the description in the whitepaper, ensuring accuracy and transparency in the validator selection and reward distribution process.

Updates

The Giant Mammoth Chain team acknowledged the issue, stating that the whitepaper is being updated to match the code.

SHB.9 Missing Value Verification

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The ctor functions do not include value verification for the arguments, which can lead to the initialization of the contract state with invalid or unexpected values. This issue can cause potential security vulnerabilities or incorrect functionality of the contract.

Files Affected:

SHB.9.1: ChainConfig.sol

```
33 function ctor(
34     uint32 activeValidatorsLength,
35     uint32 epochBlockInterval,
36     uint32 misdemeanorThreshold,
37     uint32 felonyThreshold,
```

```
38      uint32 validatorJailEpochLength,
39      uint32 undelegatePeriod,
40      uint256 minValidatorStakeAmount,
41      uint256 minStakingAmount
42  ) external whenNotInitialized {
```

SHB.9.2: DeployerProxy.sol

```
41  function ctor(address[] memory deployers) external whenNotInitialized {
42      for (uint256 i = 0; i < deployers.length; i++) {
43          _addDeployer(deployers[i]);
44      }
45  }
```

SHB.9.3: Governance.sol

```
17  function ctor(uint256 newVotingPeriod) external whenNotInitialized {
18      _setVotingPeriod(newVotingPeriod);
19  }
```

SHB.9.4: RuntimeUpgrade.sol

```
18  function ctor(address evmHookAddress) external whenNotInitialized {
19      _evmHookAddress = evmHookAddress;
20  }
```

SHB.9.5: Staking.sol

```
86  function ctor(address[] calldata validators, uint256[] calldata
        ↪ initialStakes, uint16 commissionRate) external whenNotInitialized
        ↪  {
```

SHB.9.6: SystemRewards.sol

```
43  function ctor(address[] calldata accounts, uint16[] calldata shares)
        ↪ external whenNotInitialized {
```

Recommendation:

To address this issue, we recommend modifying the code to include input validation checks in the ctor functions for the arguments' values. These checks should ensure that the argument values fall within the expected range of values, preventing the initialization of the contract state with invalid values. By adding input validation checks, the contract will be better protected against unexpected input values, ensuring the integrity and security of the initialization process.

Updates

The Giant Mammoth Chain team resolved the issue by adding the input checks for the ctor functions.

SHB.10 Lack of Check for Contract Address

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The initManually function is responsible for manually initializing the contracts used by the Injector contract. This function takes several contract addresses as input parameters and assigns them to the appropriate variables.

However, there is an issue with this implementation as the function does not check whether the addresses passed as parameters are actually contract addresses. As a result, if an attacker passes a non-contract address as a parameter to the initManually function, it could result in the system behaving unexpectedly or a Denial of Service.

Files Affected:

SHB.10.1: Injector.sol

```
77  function initManually(
78      IStaking stakingContract,
79      ISlashingIndicator slashingIndicatorContract,
80      ISystemReward systemRewardContract,
81      IStakingPool stakingPoolContract,
82      IGovernance governanceContract,
83      IChainConfig chainConfigContract,
84      IRuntimeUpgrade runtimeUpgradeContract,
85      IDeployerProxy deployerProxyContract
86  ) public initializer {
87      // BSC-compatible
88      _stakingContract = stakingContract;
89      _slashingIndicatorContract = slashingIndicatorContract;
90      _systemRewardContract = systemRewardContract;
91      // BAS-defined
92      _stakingPoolContract = stakingPoolContract;
93      _governanceContract = governanceContract;
94      _chainConfigContract = chainConfigContract;
95      _runtimeUpgradeContract = runtimeUpgradeContract;
96      _deployerProxyContract = deployerProxyContract;
97      // invoke constructor
98      _invokeContractConstructor();
99  }
```

Recommendation:

We recommend adding a check to ensure that the addresses passed as parameters to the initManually function are valid contract addresses. This can be done by using the isContract function from the OpenZeppelin library to check if the address is a contract address before assigning it to a variable.

Updates

The Giant Mammoth Chain team resolved the issue by adding a check that makes sure the addresses provided in the arguments represent contracts.

SHB.11 Inaccurate Comparison in _claimSystemFee Function

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 1

Description:

The _claimSystemFee function is responsible for handling the received portion of the rewards. The function checks the amount to be paid, amountToPay, against the TREASURY_MIN_CLAIM_THRESHOLD constant before distributing the funds. If the amountToPay is less than or equal to the TREASURY_MIN_CLAIM_THRESHOLD, the function will not proceed with the distribution.

However, there is an issue with the comparison between amountToPay and TREASURY_MIN_CLAIM_THRESHOLD. If amountToPay is equal to the TREASURY_MIN_CLAIM_THRESHOLD, the function will not distribute the shares, even though the threshold has been met. This can potentially cause issues with the proper functioning of the treasury system.

Files Affected:

SHB.11.1: SystemReward.sol

```
97  function _claimSystemFee() internal {
98      uint256 amountToPay = _systemFee;
99      if (amountToPay <= TREASURY_MIN_CLAIM_THRESHOLD) {
100         return;
101     }
```

Recommendation:

We recommend changing the comparison in the _claimSystemFee function to a strict comparison.

SHB.11.2: SystemReward.sol

```
97  function _claimSystemFee() internal {
98      uint256 amountToPay = _systemFee;
99      if (amountToPay < TREASURY_MIN_CLAIM_THRESHOLD) {
100         return;
101     }
```

By making this change, the contract's functionality and reliability can be improved, ensuring that the system treasury operates as intended.

Updates

The Giant Mammoth Chain team resolved the issue by changing the comparison in the _claimSystemFee function to a strict comparison.

SHB.12 Floating Pragma

- Severity : LOW
- Status : Acknowledged
- Likelihood : 1
- Impact : 1

Description:

The contract makes use of the floating-point pragma 0.8.0. Contracts should be deployed using the same compiler version. Locking the pragma helps ensure that contracts will not unintentionally be deployed using another pragma, which in some cases may be an obsolete version, that may introduce issues to the contract system.

Files Affected:

All contracts

Recommendation:

Consider locking the pragma version. It is advised that the floating pragma should not be used in production.

Updates

The Giant Mammoth Chain team acknowledged the issue, stating that they will be fixing the pragma version to 0.8.11 in the next updates.

SHB.13 Missing Setter For The _systemTreasury

- Severity : INFORMATIONAL
- Status : Fixed
- Likelihood : 1
- Impact : 0

Description:

The _systemTreasury address is a critical component of the contract that receives a portion of the transaction fees for system needs. However, there is no function within the contract that allows for the _systemTreasury address to be updated.

This means that if, for any reason, the treasury address needs to be updated, such as a change in ownership or the need for a new address, it is currently impossible to do so. This could potentially cause problems in the future if the current treasury address becomes compromised or is no longer accessible.

Files Affected:

SHB.13.1: SystemReward.sol

```
29 address internal _systemTreasury;
```

Recommendation:

We recommend adding a function to the contract that allows for the _systemTreasury address to be updated. This function should only be accessible by the governance.

Updates

The Giant Mammoth Chain team resolved the issue by adding a setter (setNewSystemReward) to allow the _systemTreasury address to be updated.

4 Best Practices

BP.1 Remove Unused Contract GovernorVotes

Description:

When importing external contracts into a smart contract project, it is important to ensure that the imported contracts are actually used in the project. Importing unused contracts can unnecessarily increase the project's codebase and complexity, which can lead to potential security vulnerabilities or performance issues.

It has been identified that the GovernorVotes contract from OpenZeppelin is imported into the project but is not used anywhere in the code. To adhere to best practices and minimize potential security risks, we recommend removing the import statement and its associated contract.

By removing unused contracts, the project's codebase can be simplified and the risk of potential security vulnerabilities can be reduced.

Files Affected:

BP.1.1: Governance.sol

```
6 import "@openzeppelin/contracts/governance/extensions/GovernorVotes.sol
    ↪ ";
```

Status - Acknowledged

BP.2 Optimize Struct Storage

Description:

When declaring a struct in a smart contract project, the order in which the struct's attributes are declared can have an impact on the storage size and efficiency of the contract.

It has been identified that the ValidatorDelegation struct in the project can be optimized for storage by rearranging the order of its attributes. By declaring the delegateGap and un-

delegateGap attributes after the delegateQueue and undelegateQueue attributes, these attributes can be stored in the same slot, reducing the overall storage size of the struct.

Files Affected:

BP.2.1: Staking.sol

```
63 struct ValidatorDelegation {
64     DelegationOpDelegate[] delegateQueue;
65     uint64 delegateGap;
66     DelegationOpUndelegate[] undelegateQueue;
67     uint64 undelegateGap;
68 }
```

To adhere to best practices and optimize storage in the contract, we recommend rearranging the ValidatorDelegation struct's attributes as follows:

BP.2.2: Staking.sol

```
63 struct ValidatorDelegation {
64     DelegationOpDelegate[] delegateQueue;
65     DelegationOpUndelegate[] undelegateQueue;
66     uint64 delegateGap;
67     uint64 undelegateGap;
68 }
```

Status - Fixed

BP.3 Remove Unnecessary Initializations

Description:

In Solidity, variables are automatically initialized to their default values when they are declared. For example, the default value for a uint256 variable is 0, and the default value for a bool variable is false.

It has been identified that there are instances in the code where variables are unnecessarily initialized to their default values, which can result in unnecessary gas consumption and increased contract size.

To adhere to best practices and optimize the contract's performance, we recommend removing any unnecessary initialization with a variable's default value.

Files Affected:

BP.3.1: Staking.sol

```
88 uint256 totalStakes = 0;
```

BP.3.2: SystemReward.sol

```
55 uint16 totalShares = 0;
```

Status - Acknowledged

BP.4 Avoid Unnecessary Updates to Mappings

Description:

When working with mappings in Solidity, it is important to ensure that updates to mappings are only made when necessary. Unnecessary updates to mappings can result in increased gas consumption and longer contract execution times. It has been identified that there is an unnecessary update to the _validatorsMap mapping in the _delegateTo function. In this case, the mapping is updated even though the validator struct has not been modified. To adhere to best practices and optimize the contract's performance, we recommend removing the unnecessary update to the _validatorsMap mapping in the _delegateTo function. By doing so, the contract can reduce its gas consumption and improve its execution time.

Files Affected:

BP.4.1: Staking.sol

```
242 function _delegateTo(address fromDelegator, address toValidator, uint256
    ↪  amount) internal {
243     // check is minimum delegate amount
244     require(amount >= _chainConfigContract.getMinStakingAmount() &&
        ↪ amount != 0, "Staking: amount is too low");
```

```
245     require(amount % BALANCE_COMPACT_PRECISION == 0, "Staking: amount
            ↪ have a remainder");
246     // make sure amount is greater than min staking amount
247     // make sure validator exists at least
248     Validator memory validator = _validatorsMap[toValidator];
249     require(validator.status != ValidatorStatus.NotFound, "Staking:
            ↪ validator not found");
250     uint64 atEpoch = _nextEpoch();
251     // Lets upgrade next snapshot parameters:
252     // + find snapshot for the next epoch after current block
253     // + increase total delegated amount in the next epoch for this
            ↪ validator
254     // + re-save validator because last affected epoch might change
255     ValidatorSnapshot storage validatorSnapshot =
            ↪ _touchValidatorSnapshot(validator, atEpoch);
256     validatorSnapshot.totalDelegated += uint112(amount /
            ↪ BALANCE_COMPACT_PRECISION);
257     _validatorsMap[toValidator] = validator;
```

Status - Acknowledged

BP.5 Remove Tautologies

Description:

In programming, a tautology is a logical expression that is always true, regardless of its input values. In Solidity, it is important to avoid tautologies in code, as they can make the code more difficult to read and potentially introduce unnecessary security risks.

It has been identified that there are instances in the contract where tautologies are used in require statements to check that a value is greater than or equal to a minimum value of 0.

To adhere to best practices and simplify the code, we recommend removing the tautologies from the require statements and checking only the upper bound.

Files Affected:

BP.5.1: SystemReward.sol

```
59 require(share >= SHARE_MIN_VALUE && share <= SHARE_MAX_VALUE, "
       ↪ SystemReward: bad share distribution");
```

BP.5.2: Staking.sol

```
495 require(commissionRate >= COMMISSION_RATE_MIN_VALUE && commissionRate <=
       ↪  COMMISSION_RATE_MAX_VALUE, "Staking: bad commission rate");
```

Status - Acknowledged

5 Conclusion

In this audit, we examined the design and implementation of Giant Mammoth contract and discovered several issues of varying severity. Giant Mammoth Chain team addressed 8 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised Giant Mammoth Chain Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

6 Scope Files

6.1 Audit

Files	MD5 Hash
contracts/ChainConfig.sol	b9f47e453f71b08fd24f9efef5f118bb
contracts/DeployerProxy.sol	2e869a770fdc12614262ea14ffa0337e
contracts/Governance.sol	d56241249d744846bd70b23694e13475
contracts/Injector.sol	05ebb6b39b5e647ccfb033c6bf008986
contracts/RuntimeUpgrade.sol	f14f0d0547007397858e37f2f0fdb0cb
contracts/SlashingIndicator.sol	dd5081addaaa070750f430260646bd4d
contracts/Staking.sol	aee412b82043e6a8c87b41c8bb0025f3
contracts/StakingPool.sol	c690b0ed5306560a1eb119b12793c00b
contracts/SystemReward.sol	ceba86075f29f10c0cd05bbe4cd279cc

6.2 Re-Audit

Files	MD5 Hash
contracts/ChainConfig.sol	ad9550b9116e5a25247ce208e2851a77
contracts/DeployerProxy.sol	22d3f11527ec3359dc73639aa1c99057
contracts/Governance.sol	afd3126daeaa14f84055b8f5334fac86
contracts/Injector.sol	5a84fb21f4f4600a42686f7f86dc9494
contracts/RuntimeUpgrade.sol	9656028c89a800980f26ca18586ec673

contracts/SlashingIndicator.sol	3b782ab7bda0f17cc04b56adfbf97c1a
contracts/Staking.sol	5b3bbe8b24594011dbffd4213a03b808
contracts/StakingPool.sol	4d8f313b3ab44878fa6a9413d1b47412
contracts/SystemReward.sol	ee19126557135a2f9596f8343c508cd4

7 Disclaimer

Shellboxes reports should not be construed as "endorsements" or "disapprovals" of particular teams or projects. These reports do not reflect the economics or value of any "product" or "asset" produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

For a Contract Audit, contact us at contact@shellboxes.com