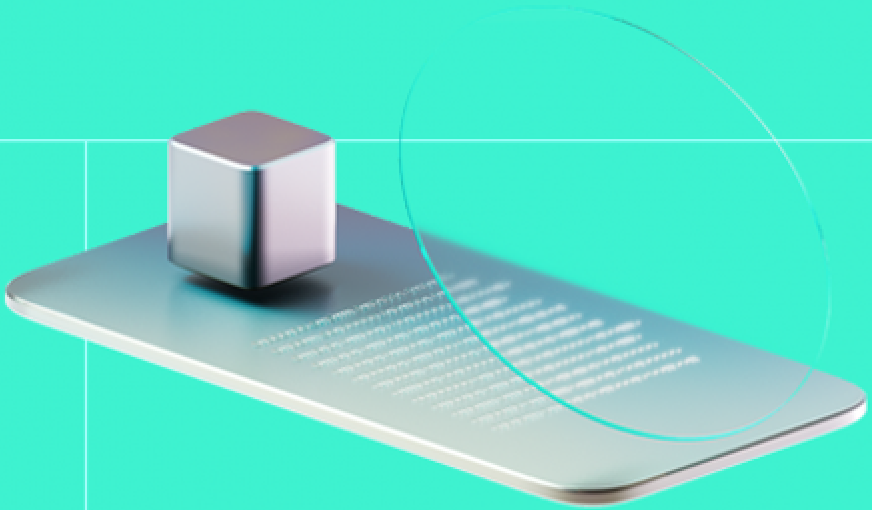
Smart Contract Code Review And Security Analysis Report



Customer: Bitlayer

Date: 01/04/2024

We express our gratitude to the Bitlayer team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Bitlayer is the layer 2 based on BitVM. It consists an EVM compatible chain/sequencer that can map BTC ecological assets and facilitate the entry of BTC users.

Platform: EVM

Language: Solidity

Tags: ERC20, Layer2, Staking, Token Factory, MultiSig

Timeline: 06/03/2024 - 01/04/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/bitlayer-org/bitlayer-contracts
Commit	751c34b1072b7539c1a063b38522cf7606c9268a

Audit Summary

10/10	8/10	62.5%	7/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 7.8/10

The system users should acknowledge all the risks summed up in the risks section of the report

4	3	1	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	1
Medium	0
Low	3

Vulnerability	Status
F-2024-1566 - Coarse grained access control in Vault contract leads to redundant whitelist feature	Accepted
F-2024-1335 - Missing validation of beneficiary address can lead to token locked	Fixed
F-2024-1549 - ReStaking() and reDelegation() lead to ERC20 token lock	Fixed
F-2024-1554 - Missing address validation for _foundationPool in Staking contract's initialise function	Fixed

Document

Name	Smart Contract Code Review and Security Analysis Report for Bitlayer
Audited By	Niccolò Pozzolini, Kornel Światłowski
Approved By	Przemyslaw Swiatowiec
Website	https://www.bitlayer.org/
Changelog	21/03/2024 - Preliminary Report
	01/04/2024 - Final Report

Table of Contents

System Overview

The project revolves around a blockchain framework, focusing on staking mechanisms and validator management to achieve consensus. Central components include a suite of smart contracts for staking (Staking.sol, Validator.sol), token management (BRC.sol, TokenFactory.sol), and security features (LockingContract.sol, MultiSigWallet.sol). The system allows for validator registration under a permission-based approach, transitioning to permission-less to enhance inclusivity. Validators compete for network consensus roles based on the amount of BRC tokens staked, with mechanisms for reward distribution, delegation, and penalties for non-compliance embedded within the protocol.

Privileged roles

- TokenFactory.AdminRole: can create and mint ERC20 tokens
- TokenFactory.OwnerRole: admin of the role AdminRole
- Vault.AdminRole: can add/remove to/from whitelist, release ERC20/native tokens
- Vault.OwnerRole: admin of the role AdminRole
- Validator.owner is the staking contract. Most user action on Validator need to pass by the Staking contract
- Staking.admin: can make validators registration permissionless, and change the foundation pool address

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **7** out of **10**.

- Functional requirements are present, but only at a high-level.
- Technical description is not complete.

Code quality

The total Code Quality score is **8** out of **10**.

- The code duplicates commonly known contracts instead of reusing them.
- Best practice violations.

Test coverage

Code coverage of the project is **62.5%** (branch coverage).

Security score

Upon auditing, the code was found to contain **0** critical, **0** high, **0** medium, and **1** low severity issues, leading to a security score of **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **7.8**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

No additional risks were identified.

Findings

Vulnerability Details

F-2024-1549 - ReStaking() and reDelegation() lead to ERC20 token lock - High

Description:

The `Staking` contract includes functionalities to transfer staked and delegated tokens from one validator to another using the `reStake()` and `reDelegate()` functions. Staked BRC tokens are held in the `Staking` contract, eliminating the need for additional token `ERC20.transfer()`. The amounts of staked and delegated tokens are tracked in variables within each `Validator` contract:

- selfStake: the amount of staked tokens by the validator owner
- totalStake: the sum of selfStake and delegated tokens

The sum of the `totalStake` amounts of each registered validator should match the BRC balance of the `Staking` contract. During the execution of `reStake()` or `reDelegate()`, the sender provides an `_amount` parameter corresponding to the amount of BRC tokens that will be moved from one validator to another. Firstly, depending on the context, `subStake()` or `subDelegation()` is executed, an `_amount` value is deducted from `selfStake` and `totalStake`, but BRC tokens are not returned to the caller. Later, these functions make use of the function `addStakeOrDelegation()` which requires the sender to provide additional tokens equal to the provided `_amount`, which are transferred to the `Staking` contract. Normally, approval of BRC token transfer is required to perform `reStake()` or `reDelegate()` execution, but an unaware sender or unlimited approval in the past will result in the sender's tokens being locked. The `Staking` contract lacks the functionality to withdraw these redundant/unassigned tokens leading to tokens being locked in the contract.

```
function doReStake(address _oldVal, address _newVal, uint256 _amount, boo
l _byValidator) private {
require(_amount > 0, "E23");

IValidator oldVal = valMaps[_oldVal];
RankingOp op = RankingOp.Noop;

if (_byValidator) {
doClaimAny(_oldVal, true);
op = oldVal.subStake(_amount, false);
} else {
doClaimAny(_oldVal, false);
op = oldVal.subDelegation(_amount, msg.sender, false);
}
afterLessStake(_oldVal, oldVal, _amount, op);
//@audits additionals funds are require here as it is new stake or delega
tion
addStakeOrDelegation(_newVal, msg.sender, _amount, false);
}
```

Scenario (same as in Evidences Poc section):

```
Validator1 stakes 100001 BRC tokens
> selfStake == 100001
> totalStake == 100001
```

```
Validator2 stakes 50002 BRC tokens
> selfStake == 50002
> totalStake == 50002

SUM OF STAKED TOKENS == 150003
BALANCE OF STAKING CONTRACT = 150003

Validator1 admin decides to reStake() 50001 BRC tokens to Validator2 cont
ract.
An additional 50001 BRC is taken from Validator1 admin, resulting in:

Validator1 stakes 50000 BRC tokens (100001 - 50001 = 50000)
> selfStake == 50000
> totalStake == 50000

Validator2 stakes 100003 BRC tokens (50002 + 50001 = 100003)
> selfStake == 100003
> totalStake == 100003

SUM OF STAKED TOKENS == 150003
BALANCE OF STAKING CONTRACT = 200004
```

Assets:

- contracts/builtin/Staking.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Fixed

Classification

Severity: High

Impact:

Likelihood [1-5]: 4
Impact [1-5]: 4
Exploitability [0-2]: 0
Complexity [0-2]: 1
Final Score: 3.8 (High)
Hacken Calculator Version: 0.6

Recommendations

Recommendation: It is recommended to refactor `reStake()` and `reDelegation()` functions flow so that no additional tokens are needed from the sender and re-staked/re-dalegated tokens are not locked in `Staking` contract.

Remediation (commit: 1080bfa): The function `addStakeOrDelegation` is now requiring tokens only when not restaking.

Evidences

PoC

Reproduce:

```
import { loadFixture } from "@nomicfoundation/hardhat-toolbox/network-hel
pers";
import { expect } from "chai";
import { ethers } from "hardhat";
```

```
describe("audit staking", function async() {

const epoch = 2;


async function deploySystem() {
const [
admin, fundation, brcHolder,
val1, valAdmin1,
val2, valAdmin2,
val3, valAdmin3,
delegator1, delegator2
] = await ethers.getSigners();

const ERC20 = await ethers.getContractFactory('BRC');
const erc20 = await ERC20.deploy(
[brcHolder.address],
[ethers.parseEther("1000000000")] //10_000_000_00
);

const Staking = await ethers.getContractFactory("Staking");
const staking = await Staking.deploy();

await staking.initialize(
admin.address,
erc20.getAddress(),
epoch,
fundation.address
);

return {
staking, erc20,
admin, fundation, brcHolder,
val1, valAdmin1,
val2, valAdmin2,
val3, valAdmin3,
delegator1, delegator2
};
}

describe("reStaking()", async function () {
it("PASS - staked needed", async function () {
const { admin, brcHolder, erc20, staking, val1, valAdmin1, val2, valAdmin
2} = await loadFixture(deploySystem);
const Validator = await ethers.getContractFactory("Validator");
const rate = 50;
const stakeAmount = ethers.parseEther('100001');
const stakeAmount2 = ethers.parseEther('50002');
const reStakeAmount = ethers.parseEther('50001');

//@audit val1Admin stakes 100001 BRC tokens
await staking.connect(admin).removePermission();
await erc20.connect(brcHolder).transfer(valAdmin1.address, stakeAmount);
await erc20.connect(valAdmin1).approve(await stak
```

See more

Results:

```
audit staking
reStaking()
BRC balance of Staking before restake(): 150003000000000000000000n
BRC balance of Staking after restake(): 200004000000000000000000n
✔ PASS - staked needed (5060ms)
reDelegation()
BRC balance of Staking before reDelegation(): 110222000000000000000000n
BRC balance of Staking after reDelegation(): 110444000000000000000000n
✔ PASS (375ms)


2 passing (5s)
```

F-2024-1335 - Missing validation of beneficiary address can lead to token locked - Low

Description: The `constructor()` and `changeBeneficiary()` functions of the `LockingContract` contract lack validation of beneficiary address parameter, permitting the assignment of a vesting schedule to a `0x0` address. This can lead to a situation where vested tokens assigned to the `0x0` address become locked within the `LockingContract` without the possibility of withdrawal.

Assets:

- contracts/basic/LockingContract.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Fixed

Classification

Severity: Low

Impact: Likelihood [1-5]: 2
Impact [1-5]: 3
Exploitability [0-2]: 0
Complexity [0-2]: 0
Final Score: 2.5 (Low)
Hacken Calculator Version: 0.6

Recommendations

Recommendation: It is recommended to implement input validation within the `constructor()` and `changeBeneficiary()` functions of the `LockingContract` to ensure that the vesting schedule can not be assigned to the `0x0` address.

Remediation (commit: e6f3d3d): The provided addresses are now validated against the zero address.

Evidences

POC

Reproduce:

```
import { time, loadFixture } from "@nomicfoundation/hardhat-toolbox/netwo
rk-helpers";
import { expect } from "chai";
import { ethers } from "hardhat";
import { getContractAddress } from '@ethersproject/address';

describe("audit locking", function () {

const lockAmount1 = ethers.parseEther("10");
const lockAmount2 = ethers.parseEther("100");
const cliffPeriod1 = 1;
const cliffPeriod2 = 2;
const vestingPeriod1 = 10;
const vestingPeriod2 = 100;
const periodTime = 100;

async function deployLockingContract() {
const [owner, acc1, acc2, acc3] = await ethers.getSigners();

const token = await ethers.deployContract('TestUSDT');
await token.waitForDeployment();
const transactionCount = await ethers.provider.getTransactionCount(owner.
address);
const futureAddress = getContractAddress({
from: owner.address,
nonce: transactionCount+1
});

await token.connect(owner).mint(futureAddress, ethers.parseEther("110"));

const locking = await ethers.deployContract('LockingContract', [
[acc1.address, acc2.address],
[lockAmount1, lockAmount2],
[cliffPeriod1, cliffPeriod2],
[vestingPeriod1, vestingPeriod2],
periodTime,
token.getAddress()
])
await locking.waitForDeployment();
return { locking, token, owner, acc1, acc2, acc3 };
}

describe("changeBeneficiary()", async function () {
it("changeBeneficiary() to zero address", async function () {
const { locking, token, owner, acc1} = await loadFixture(deployLockingCon
tract);
const zeroAddress = '0x0000000000000000000000000000000000000000';

const vestingInfo = await locking.vestingSchedules(acc1.address);
expect(vestingInfo['lockingAmount']).to.be.equals(lockAmount1);
expect(vestingInfo['releasedAmount']).to.be.equ
```

See more

F-2024-1554 - Missing address validation for _foundationPool in Staking contract's initialise function - Low

Description:	In the `initialize` function of the `Staking.sol` contract, the `_admin` and `_brcAddress` parameters are validated to ensure they are not zero addresses. However, the `_foundationPool` address is not validated in the same way. This could potentially lead to a significant issue if the `_foundationPool` address is set to the zero address. In such a case, 20% of all block fees would be sent to the zero address and effectively burnt, which is likely not the intended behavior.
Assets:	• contracts/builtin/Staking.sol [https://github.com/bitlayer-org/bitlayer-contracts]
Status:	Fixed

Classification

Severity:	Low
Impact:	Likelihood [1-5]: 3 Impact [1-5]: 4 Exploitability [0-2]: 2 Complexity [0-2]: 0 Final Score: 2.3 (Low)

Recommendations

Recommendation:	To mitigate this issue, it is suggested to add a validation check for the `_foundationPool` address in the `initialize` function. This check should ensure that `_foundationPool` is not the zero address. The same check should be performed in the setter function `changeFoundationPool`. This change would prevent the accidental burning of block fees and ensure that the `Staking` contract behaves as expected. **Remediation** (commit: 68eb9d9): The mentioned parameters are now checked against the zero address.

F-2024-1566 - Coarse grained access control in Vault contract leads to redundant whitelist feature - Low

Description:	The current implementation of the `Vault` contract uses the `AdminRole` for both adding addresses to the whitelist and releasing funds to whitelisted addresses. This could potentially pose a security risk if an admin's address is compromised. An attacker could add a malicious address to the whitelist and then release funds to it using the `releaseTreasure` function.
Assets:	• contracts/basic/Vault.sol [https://github.com/bitlayer-org/bitlayer-contracts]
Status:	Accepted

Classification

Severity:	Low
Impact:	Likelihood [1-5]: 2 Impact [1-5]: 5 Exploitability [0-2]: 2 Complexity [0-2]: 0 Final Score: 2.3 (Low) Hacken Calculator Version: 0.6

Recommendations

Recommendation:	To mitigate this risk, it is suggested to separate these two operations into different roles. For instance, a `WhitelistAdminRole` could be used for adding and removing addresses from the whitelist, and the `AdminRole` could be used for releasing funds. This would ensure that even if an admin's address is compromised, the attacker would not be able to add new addresses to the whitelist.

Observation Details

F-2024-1329 - Floating Pragma - Info

Description:	The project uses floating pragmas `^0.8.0` and `^0.8.9.` This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version which may include bugs that affect the system negatively.
Assets:	• contracts/basic/BRC.sol [https://github.com/bitlayer-org/bitlayer-contracts] • contracts/basic/LockingContract.sol [https://github.com/bitlayer-org/bitlayer-contracts] • contracts/basic/Vault.sol [https://github.com/bitlayer-org/bitlayer-contracts] • contracts/basic/TokenFactory.sol [https://github.com/bitlayer-org/bitlayer-contracts] • contracts/basic/MultiSigWallet.sol [https://github.com/bitlayer-org/bitlayer-contracts]
Status:	Accepted

Recommendations

Recommendation:	Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs for the compiler version that is chosen.

F-2024-1330 - State variables only set in the constructor should be declared immutable - Info

Description:

Compared to regular state variables, the Gas costs of constant and immutable variables are much lower. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed.

Variable`s `periodTime`, `LockingToken` and `startTimestamp` values is set in the constructor of `LockingContract`. These variables can be declared immutable.

The same applies for the variables `factory` and `_decimals` in `CustomERC20` contract.

This will lower the Gas fees during operations.

Assets:

- contracts/basic/LockingContract.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/basic/TokenFactory.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Fixed

Recommendations

Recommendation:

Consider marking state variables as an immutable that never changes on the contract.

Remediation (commit: 4d546b1): Mentioned variables are now immutable.

F-2024-1331 - Gas inefficiency due to missing usage of Solidity custom errors - Info

Description:

Starting from Solidity version 0.8.4, the language introduced a feature known as "custom errors". These custom errors provide a way for developers to define more descriptive and semantically meaningful error conditions without relying on string messages. Prior to this version, developers often used the `require` statement with string error messages to handle specific conditions or validations. However, every unique string used as a revert reason consumes gas, making transactions more expensive.

Custom errors, on the other hand, are identified by their name and the types of their parameters only, and they do not have the overhead of string storage. This means that, when using custom errors instead of `require` statements with string messages, the gas consumption can be significantly reduced, leading to more gas-efficient contracts.

Assets:

- contracts/builtin/Staking.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/builtin/Validator.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/basic/BRC.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/basic/LockingContract.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/basic/Vault.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/basic/TokenFactory.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/basic/MultiSigWallet.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/builtin/Params.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/builtin/WithAdmin.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/builtin/library/SafeSend.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/builtin/library/initializable.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/builtin/library/ReentrancyGuard.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Accepted

Recommendations

Recommendation: It is recommended to use custom errors instead of reverting strings to reduce increased Gas usage, especially during contract deployment. Custom errors can be defined using the `error` keyword and can include dynamic information.

F-2024-1332 - Unchecked transfers of ERC20 tokens - Info

Description: `SafeERC20` is a library allows to safely interact with different partially incompatible ERC20 tokens.

The analysis identified that there are omitted verifications for the return values of ERC20 transfer functions. This oversight can lead to vulnerabilities since certain tokens might deviate from the ERC20 standards, either by returning `false` upon a transfer failure or by not issuing any return value whatsoever.

Functions that transfer do not use `SafeERC20` and do not check return value of transfers:

- LockingContract: claim(),
- Vault: releaseERC20()

Assets:

- contracts/basic/LockingContract.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/basic/Vault.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Fixed

Recommendations

Recommendation: Consider using `SafeERC20` library in the aforementioned functions to interact with tokens safely.

Remediation (commit: f067d8f): The SafeERC20 library has been used in `claim()` and `releaseERC20()` functions.

[F-2024-1333](#) - Use of transfer() instead of call() to send native tokens - Info

Description:

The `Vault` contract uses built-in `transfer()` function for transferring native tokens.

The `transfer()` function was commonly used in earlier versions of Solidity for its simplicity and automatic reentrancy protection. However, it was identified as potentially problematic due to its fixed Gas limit of `2300`.

The usage of `transfer()` function can lead to unintended function call revert when the receiving contract's `receive()` or `fallback()` functions require more than `2300` Gas for processing.

Additionally, the Gas costs are subject to change, so it may happen in future release that `2300` Gas Limit is not enough to process transfer.

Assets:

- contracts/basic/Vault.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Fixed

Recommendations

Recommendation:

It is recommended to use built-in `call()` function instead of `transfer()` to transfer native assets. This method does not impose a gas limit, it provides greater flexibility and compatibility with contracts having more complex business logic upon receiving the native tokens. When working with then `call()` function ensure that its execution is successful by checking the returned boolean value. It is also recommended to fallow the Check-Effects-Interactions (CEI) pattern in every case to prevent reentrancy issues.

Remediation (commit: 0f7afff): The `transfer()` function has been replaced with the `call()` function.

F-2024-1334 - Copy of well known contract - Info

Description: The system employs replicated contracts sourced from established projects such as OpenZeppelin. It is advised that these contracts should be directly imported from their original source. Given that these projects are currently under active development, the contracts within them might undergo updates in the future.

Assets:

- contracts/basic/BRC.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/builtin/library/initializable.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/builtin/library/ReentrancyGuard.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/builtin/library/SafeERC20.sol [https://github.com/bitlayer-org/bitlayer-contracts]
- contracts/interfaces/IERC20.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Accepted

Recommendations

Recommendation: **Direct Source Import:** Rather than relying on these copies, it is suggested to directly import the contracts from their original source repositories. This approach ensures that the latest updates, bug fixes, and security enhancements introduced by the project maintainers are seamlessly incorporated into your system.

Version Pinning: In addition to importing contracts directly from source, it is prudent to pin the version of the imported contracts. This safeguards your system against potential compatibility issues that could arise from future updates to the source contracts. By specifying a precise version, you gain greater control over the integration process and mitigate the risks associated with the inadvertent adoption of incompatible changes.

Remediation: The finding is accepted by the Client's team.

F-2024-1339 - Readability improvement for totalSupply initialization - Info

Description:

In the `BRC.sol` contract, the `totalSupply` variable is initialized with a long literal value. However, there is a minor issue with this initialization: the long literal could be hard to read.

Here is the code reference:

```
totalSupply = 1000000000 ether;
```

To improve readability, the long literal could be split using the underscore (_) character. This would make the number easier to read and understand.

Assets:

- contracts/basic/BRC.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Fixed

Recommendations

Recommendation:

To fix this issue, the long literal should be split using the underscore character. The corrected code would look like this:

```
totalSupply = 1_000_000_000 ether;
```

Remediation (commit: 05cec62): The totalSupply variable value has been assigned with underscore characters.

F-2024-1340 - Redundant constructor visibility - Info

Description: Starting solidity compiler version 0.7 there is no need to declare the constructor visibility anymore because it is ignored. `constructor()` of `BRC` contract has declared `public` visibility.

```
constructor (
address[] memory accounts,
uint256[] memory amounts
) public {
// constructor logic
}
```

Assets:

- contracts/basic/BRC.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Fixed

Recommendations

Recommendation: It is recommended to remove `constructor()` visibility of `BRC` contract.

Remediation (commit: 3fa576d): Constructor visibility has been removed.

F-2024-1550 - Add supply cap feature to CustomERC20 contract for enhanced security - Info

Description: In the current implementation of the `CustomERC20` contract, the minting of tokens is controlled by the administrators of the `TokenFactory` contract. This could potentially pose a security risk if an administrator's address is compromised, as the attacker could arbitrarily mint a large number of tokens, potentially destabilizing the token's value and liquidity pools.

Assets:

- contracts/basic/TokenFactory.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Accepted

Recommendations

Recommendation: To mitigate this risk, it is suggested to add an optional supply cap feature to the `CustomERC20` contract. This would limit the total number of tokens that can be minted, providing an additional layer of security for users.

This feature could be implemented as an immutable variable set in the constructor of the `CustomERC20` contract, and a check could be added in the `mint` function to ensure that the total supply doesn't exceed this cap.

This enhancement would provide users with greater confidence in the stability of the token's value and the integrity of its liquidity pools.

Remediation: The finding is accepted by the Client's team with following reason: “We don’t know the max supply of tokens deployed by 3rd party.”.

F-2024-1552 - Missing decimal check for brcToken in the Staking contract - Info

Description:

In the `Staking.sol` contract, the `brcToken` is assumed to have **18** decimals as per the code comment. However, there is no validation in the code to ensure this.

This could potentially lead to issues if a token with a different number of decimals is used. For instance, calculations involving this token could yield incorrect results, leading to imbalances in the contract's state.

Assets:

- contracts/builtin/Staking.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Accepted

Recommendations

Recommendation:

To mitigate this risk, it is suggested to add a requirement statement in the `initialize` function where the `brcToken` variable is set. This requirement should check that the `decimals` function of the `brcToken` returns 18.

This change would ensure that only tokens with the correct number of decimals are used with the `Staking` contract, preventing potential issues and enhancing the contract's robustness.

Remediation: The finding is accepted by the Client's team with following reason: “The BRC-token contract would have not been deployed when the Staking contract initialize”.

F-2024-1556 - The updateActiveValidatorSet function in Staking.sol contract can be optimised - Info

Description: In the `updateActiveValidatorSet` function of the `Staking.sol` contract, the `for` loops use `activeValidators.length` in the iteration condition. `activeValidators` is a storage variable, and accessing its length in each iteration results in a storage read, which is more costly in terms of gas than a memory read.

Assets:

- contracts/builtin/Staking.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Fixed

Recommendations

Recommendation: To optimize this function and reduce gas costs, it is suggested to cache the length of the `activeValidators` array to a memory variable at the start of the function, and use this variable in the `for` loop conditions.

This change would involve adding a line of code at the start of the function to cache the length of `activeValidators`, and replacing `activeValidators.length` with this new variable in the `for` loop conditions.

This optimization would reduce the gas costs of the `updateActiveValidatorSet` function, making it more efficient.

Remediation (commit: 138eb9c)**:** The `activeValidatorsLen` variable has been introduced and stores the length of `activeValidators`. The `activeValidatorsLen` variable has been used in the `for` loop.

F-2024-1557 - The LazyPunishRecord struct in Staking.sol contract can be optimized - Info

Description: In the `Staking.sol` contract, the `lazyPunishRecords` mapping maps a validator address to a `LazyPunishRecord` struct. This struct contains a boolean field `exist` to check if a validator is in the set.

Assets:

- contracts/builtin/Staking.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Accepted

Recommendations

Recommendation: However, this `exist` field could be removed to optimize the struct. Instead of using `exist`, a +1 offset could be added to the `index` field, and `index > 0` could be used to check if a validator is in the set.

This change would involve removing the `exist` field from the `LazyPunishRecord` struct, adding 1 to the `index` field when it is set, and replacing checks for `!exist` with checks for `index == 0`. The function `decreaseMissedBlocksCounter` would also need to be revised.

This optimization would reduce the storage requirements of the `LazyPunishRecord` struct, making the contract more efficient.

Remediation: The finding is accepted by the Client's team with following reason: “this optimization involves a lot of changes, it is not considered to avoid introducing new problems.”

F-2024-1558 - Public function should be external - Info

Description: Functions that are meant to be exclusively invoked from external sources should be designated as `external` rather than `public`. Changing the visibility of `anyClaimable` function from public to external would optimize its gas consumption.

Assets:

- contracts/builtin/Staking.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Fixed

Recommendations

Recommendation: To optimize gas usage and improve code clarity, declare functions that are not called internally within the contract and are intended for external access as `external` rather than `public`. This ensures that these functions are only callable externally, reducing unnecessary gas consumption and potential security risks.

Remediation (commit: b878811): The visibility of `anyClaimable()` function has been changed to external.

F-2024-1562 - CEI violation in validatorClaimAny function in Validator.sol contract - Info

Description: In the `validatorClaimAny` and `delegatorClaimAny` functions of the `Validator.sol` contract, there is a Checks-Effects-Interactions (CEI) pattern violation. The `sendValue` function, which performs an external call, is invoked before state variables are updated.

This could potentially lead to reentrancy attacks if the external call is to a malicious contract that calls back into `validatorClaimAny` before the original call has finished.

Assets:

- contracts/builtin/Validator.sol [https://github.com/bitlayer-org/bitlayer-contracts]

Status: Fixed

Recommendations

Recommendation: To mitigate this risk, it is suggested to move the `sendValue` call to after all state changes have been made. This would involve moving the `sendValue` call and the associated `if` statement to after the state variables update.

This change would ensure that the `validatorClaimAny` and `delegatorClaimAny` functions follow the Checks-Effects-Interactions pattern, reducing the risk of reentrancy attacks and enhancing the contract's security.

Remediation (commit: 9862935): `sendValue()` function has been moved after all state changes, `validatorClaimAny()` and `delegatorClaimAny()` functions follow the Checks-Effects-Interactions pattern.

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.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
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.

Appendix 2. Scope

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

Scope Details

Repository	https://github.com/bitlayer-org/bitlayer-contracts
Commit	751c34b1072b7539c1a063b38522cf7606c9268a
Whitepaper	-
Requirements	File name: Audit content description.pdf; SHA3: 8b659ffa690b7fa2733a7d9ab557b8d41892fd82fb96fa514f771ba6a7e64b40
Technical Requirements	File name: Audit content description.pdf; SHA3: 8b659ffa690b7fa2733a7d9ab557b8d41892fd82fb96fa514f771ba6a7e64b40

Contracts in Scope

contracts/builtin/Staking.sol

contracts/builtin/Validator.sol

contracts/basic/BRC.sol

contracts/basic/LockingContract.sol

contracts/basic/Vault.sol

contracts/basic/TokenFactory.sol

contracts/basic/MultiSigWallet.sol

contracts/builtin/Params.sol

contracts/builtin/library/SortedList.sol

contracts/builtin/WithAdmin.sol

contracts/builtin/library/SafeSend.sol

contracts/builtin/library/initializable.sol

contracts/builtin/library/ReentrancyGuard.sol

contracts/builtin/library/SafeERC20.sol

contracts/builtin/interfaces/IValidator.sol

contracts/builtin/interfaces/types.sol

contracts/interfaces/IERC20.sol