Tetragaze

Smart contract audit

BitBook

Tetragaze

September 12, 2021 | v 3.0

Pass

Tetragaze Security Team has concluded that there are no issues that can have an impact on contract security. The contract is well written and is production-ready.

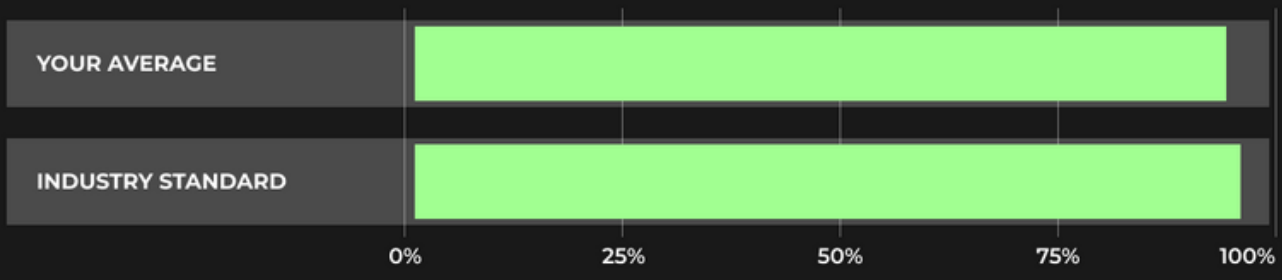
Technical summary

This document outlines the overall security of the BitBook Stacking smart contracts, evaluated by Tetragaze Blockchain Security team.
The scope of this audit was to analyze and document the BitBook Stacking smart contract codebase for quality, security, and correctness

Contract Status

There were no critical issues found during the audit.

Testable Code



Testable code is 92%, which is close to the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that's able to withstand the Ethereum network's fast-paced and rapidly changing environment, we at Tetragaze recommend that the BitBook team put in place a bug bounty program to encourage further and active analysis of the smart contract.

Table of content

Auditing Strategy and Techniques Applied

Requirements: FIP8 spec

During the review process, we made sure to carefully check that the token contract:

- Adheres to established Token standards and ensures effective implementation;
- Ensures that documentation and code comments accurately reflect the logic and behavior of the code;
- Ensures that token distribution is in line with calculations;
- Utilizes best practices to efficiently use gas, avoiding unnecessary waste;
- Employs methods that are safe from reentrance attacks and immune to the latest vulnerabilities;
- Follows best practices for code readability

Tetragaze team of expert pentesters and smart contract developers carefully evaluated the repository for security issues, code quality, and compliance with specifications and best practices. They conducted a line-by-line review, documenting any issues that were identified during the process. Special care was taken to ensure that the review was thorough and comprehensive.

1	Due diligence in assessing the overall code quality of the codebase.	2	Cross-comparison with other, similar smart contracts by industry leaders.
3	Testing contract logic against common and uncommon attack vectors.	4	Thorough, manual review of the codebase, line-by-line.

Summary

In summary, we examined the security of the BitBookStaking platform through the specified audit process. Our audit revealed a range of issues with varying severity levels. The majority of these issues were addressed and acknowledged by the Auditee.

Structure and Organization of Document

For ease of navigation, sections are arranged from most critical to least critical. Issues are tagged “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the ability of the contract to compile or operate in a significant way.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

Complete Analysis

It is possible that the massUpdatePools() function will be reversed.

High

Loops that do not have a fixed number of iterations, such as those that rely on stored values, must be used with caution in contracts. This is because the block gas limit limits the amount of gas that can be consumed by a transaction. If the number of iterations in a loop exceeds the block gas limit, it can cause the entire contract to stall. Furthermore, using unbounded loops can result in unnecessary gas costs. The massUpdatePools() function in this case may reach a maximum length of 1000 in the future. During the audit, it was found that each transaction in this function cost approximately 30869 gas. If the length of poolInfo reaches 1000 (as tested on the Ropsten network), the total gas cost would be approximately 30,869,000. However, the block gas limit is only 30 million, meaning that if the massUpdatePools() function consumes more gas than this limit, the transaction may fail due to insufficient gas.

Recomendation:

"To avoid the "out of gas" issue, we can carefully test the maxLength variable in the function to determine the maximum number of items that can be successfully passed. One solution is to pass in the number of iterations that a loop can execute.

```
   function massUpdatePools(uint256 from, uint256 to) public {
       uint256 length = poolInfo.length;
         require(from < length, “from should be less than the
length”);
      require(to <= length, “to should be less than the length”);
      require( to.sub(from) < maxLength, “the interval has reached
to the maximum length”)
        for (uint256 pid = from; pid < to; ++pid) {
           updatePool(pid);
        }
   }
```

Complete Analysis

The code mentioned before would assist the person using the contract in creating multiple batches to update the pool, which helps prevent transaction failure issues due to insufficient gas.

Complete Analysis

Insufficient check in the canHarvest() function

Medium

The function canHarvest() does not properly validate the variables _pid and _user, as it always returns true regardless of whether these variables exist. As a result, the block.timestamp is always greater than the user.nextHarvestUntil (which is set to 0 if the _user is invalid).

```
   function canHarvest(uint256 _pid, address _user) public view
returns (bool) {
        UserInfo storage user = userInfo[_pid][_user];
        return block.timestamp >= user.nextHarvestUntil;
    }
```

Recomendation:

We recommend validating and checking the existence of _pid and _user.

Complete Analysis

Centralization Risks

Medium

The role owner has the authority to update the critical settings:
- set()
- setLockDeposit()

Recomendation:

It is important for the client to be cautious when handling the governance account in order to prevent a potential hack. Some solutions that the client should consider include: implementing measures to ensure timely notification of privileged operations within the community, implementing a multisignature process with community-elected, independent co-signers, and implementing a DAO or governance module to increase transparency and community involvement.

Complete Analysis

Lack of event emissions

Low

Without the event being recorded, it is challenging to monitor any changes in off-chain liquidity fees. It is important to emit an event for significant transactions that utilize the following functions in order to accurately track these fees.

- set()
- depositRewardToken()
- emergencyAdminWithdraw()
- updatePaused()
- setLockDeposit()

Recomendation:

We suggest emitting an event to record the updates of the following variables when using the set() function: _tokenPerBlock, _depositFeeBP, _minDeposit, _harvestInterval

We also suggest emitting an event to log the updates of the following variables when using the depositRewardToken() function: msg.sender, poolId, amount

Additionally, we recommend emitting an event to log the updates of the following variables when using the emergencyAdminWithdraw() function: _pid, balanceOf(address(this)), pool.accTokenPerShare, pool.tokenPerBlock, pool.lastRewardBlock

Finally, we suggest emitting an event to record the updates of the following variables when using the updatePaused() function: paused

We also recommend emitting an event to log the updates of the following variables when using the setLockDeposit() function: pid, poolInfo[pid].lockDeposit

Complete Analysis

The use of loops within the Contract incurs significant costs.

Low

The for loops in the entire codebase includes state variables .length of a non-memory array, in the condition of the for loops. As a result, these state variables consume a lot more extra gas for every iteration of the for a loop.

Recomendation:

We recommend using a local variable instead of a state variable .length in a loop for the entire codebase. For example:

```
_withdrawFeeLength =_withdrawFee.length for (uint i=0; i <
_withdrawFeeLength; i++) { …….
}
```

Complete Analysis

Comparison to boolean constants

Low

In the contract, boolean variables are used in comparison to either true or false, for instance: require(paused == false, 'BITBOOK_STAKING: Paused!'); However, these boolean variables can be utilized directly without the need for comparison to true or false.

Recomendation:

We recommend removing the equality to the boolean constant in the entire codebase.

Complete Analysis

Missing zero address validation

Low

The following parameters have been identified as missing zero address validation:

- _owner and token in the constructor function
- _stakedToken and _rewardToken in the add function

Recomendation:

To avoid unexpected behaviors or wasted gas, it is recommended to consider using require statements to validate all user-controlled input, including in the constructor. This will help ensure that any erroneous values are detected and prevent them from causing problems.

Complete Analysis

Lack of Input Validation

Low

In both the emergencyWithdraw() and emergencyAdminWithdraw() functions, the balance of the msg.sender and address(this) are not checked before allowing a withdrawal. This means that it is possible for someone to withdraw more than their current balance.

Recomendation:

It is recommended to include a check for an empty balance in the emergencyAdminWithdraw() and emergencyAdminWithdraw() function to prevent incorrect values from causing unintended behaviors or wasting gas.

Complete Analysis

State Variable Default Visibility

Informational

"If the visibility of the mentioned variable is not specified, it can be difficult to identify incorrect assumptions about who is able to access it. Clearly labeling the visibility helps to prevent this issue.

mapping(address => mapping(address => bool)) poolExists;

Recomendation:

By default, state variables are set to internal, but it is advisable to specify this explicitly.

Complete Analysis

Not using delete to zero values

Informational

In the emergencyWithdraw() function within the contract, the following variables are manually replaced with Zero:

- user.amount = 0;
- user.rewardDebt = 0;
- user.rewardLockedUp = 0;
- user.nextHarvestUntil = 0;

And other places in emergencyAdminWithdraw() , such as:

- pool.accTokenPerShare = 0;
- pool.tokenPerBlock = 0;

Recomendation:

To simplify the code and clarify intent, consider using delete instead.

Complete Analysis

Use double quotes for string literals

Informational

The single quotes in the above string variables are being used, while the double quotes are being utilized for other string literals.

Recomendation:

It is recommended to use double quotes for string literals throughout the entire codebase.

Complete Analysis

Public function that could be declared external

Informational

The following public functions should be declared as external because they are never called within the contract to save on gas usage:

- initialize()
- set()
- massUpdatePools()
- deposit(), withdraw()
- emergencyWithdraw()
- emergencyAdminWithdraw()
- updatePaused()
- and setLockDeposit().

Recomendation:

Function that can be publicly declared as external

Complete Analysis

Inconsistent coding style

Informational

Throughout the entire codebase, deviations from the Solidity Style Guide were identified. To improve the readability of the project, it is recommended to enforce a standard coding style with the help of linter tools such as Solhint. Consistent coding style adds value to the project and can be achieved with the use of these tools.

Complete Analysis

block.timestamp may not be reliable

Informational

In the Time contract, the block timestamp is included in the calculations and time checks. However, it is possible for miners to slightly alter the timestamp in order to gain an advantage in contracts that rely heavily on it. Therefore, it is important to be aware of this issue and inform users that such manipulation could occur.

Complete Analysis

Missing docstrings

Informational

Docstrings make functions more readable and easier to maintain. Documentation of functions should include information on their purpose, potential failure situations, who can call them, returned values, and emitted events to help users understand and use them effectively.

We are thankful for the opportunity to collaborate with the BitBook team.

This document's statements should not be taken as investment or legal advice, and the authors cannot be held responsible for any decisions made based on them.

It is suggested by the Tetragaze Security Team that the BitBook team implement a bug bounty program in order to encourage external parties to scrutinize the smart contract further.