Security Audit Report for SecondLive Smart Contracts

Date: September 11, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.09.11 17:26:45 +08'00'

Contents

Report Manifest

Item	Description
Client	SecondLive
Target	SecondLive Smart Contracts

Version History

Version	Date	Description
1.0	September 11, 2023	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the token contract and the reward distribution contract of the SecondLive Smart Contracts project in `https://github.com/Secondlive23/Core`.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
SecondLive Core	`Version 1`	`89070b044dc1aa412336c9b74c4abbac74e7faef`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.

- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **one** potential issue. Besides, we also have **two** recommendations and **one** note.

- Low Risk: 1
- Recommendation: 2
- Note: 1

ID	Severity	Description	Category	Status
1	Low	Potential insufficient reward for users	Software Security	Acknowledged
2	-	Refactor inconsistent whitelist and blacklist mechanism	Recommendation	Acknowledged
3	-	Remove duplicated calculations	Recommendation	Acknowledged
4	-	Does not support deflationary tokens	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential insufficient reward for users

Severity Low

Status Acknowledged

Introduced by Version 1

Description In the safeSLTTransfer function of the SecondLiveBeanReward contract, if the contract does not have enough balance of the SLT token, the reward transferred to the users can be less than the actual reward, resulting in the loss of the user. The reward distribution process relies on that the project maintainers transfer sufficient amount of the SLT token to the SecondLiveBeanReward contract.

```
function safeSLTTransfer(address _to, uint256 _amount) internal {
    require(slt != IERC20(0x0), "No harvest began");
    uint256 sltBal = slt.balanceOf(address(this));
    if (_amount > sltBal) {
        slt.transfer(_to, sltBal);
    } else {
        slt.transfer(_to, _amount);
    }
}
```

Listing 2.1: SecondLiveBeanReward.full.sol

Impact Users can lose their rewards if the balance of the SLT token of the SecondLiveBeanReward contract is not enough.

Suggestion Ensure that the reward contract has sufficient SLT tokens for reward distribution.

Feedback from the Developers The project maintainers would ensure that there are sufficient SLT token balance in the reward contract to avoid user loss in the reward distribution process.

2.2 Additional Recommendation

2.2.1 Refactor inconsistent whitelist and blacklist mechanism

Status Acknowledged

Introduced by `Version 1`

Description The `BEAN.SecondLive` token has evolved three versions. In the original `SecondLiveBean` token contract, a blacklist mechanism exists that restricts addresses on the blacklist from conducting token transfers. In the `SecondLiveBeanV2` contract, a whitelist and a pausing mechanism were introduced, which only permits token transfers from whitelisted addresses when the contract is paused. However, there are several issues with these mechanisms:

1. There are no checks in place when setting the whitelist and blacklist. This means an address can be both whitelisted and blacklisted simultaneously. In such cases, the blacklist takes precedence.
2. The whitelist and blacklist mechanisms are not applied to the `burn` function.

Impact N/A

Suggestion Refactor the whitelist and blacklist mechanism.

2.2.2 Remove duplicated calculations

Status Acknowledged

Introduced by `Version 1`

Description In the `SecondLiveBeanReward` contract, there are multiple duplicated calculations. For example, in the `harvest` function, the field `user.reward` has been calculated and written by the `updateReward` modifier. However, the `harvest` function recalculated the reward amount using the `earned` function. The calculations are implemented correctly, but duplicated calculations can cause extra gas costs.

```
function harvest(
    uint256 _pid
) public updateReward(_pid, msg.sender) nonReentrant {
    uint256 reward = earned(_pid, msg.sender);
    require(reward > 0, "no reward");
    safeSLTTransfer(msg.sender, reward);
    UserInfo storage user = userInfo[_pid][msg.sender];
    user.reward = 0;

    emit Harvest(msg.sender, _pid, reward);
}
```

Listing 2.2: SecondLiveBeanReward.full.sol

```
modifier updateReward(uint256 _pid, address account) {
    PoolInfo storage pool = poolInfo[_pid];
    pool.rewardPerTokenStored = rewardPerToken(_pid);
    pool.lastUpdateTime = lastTimeRewardApplicable(_pid);
    if (account != address(0)) {
        UserInfo storage user = userInfo[_pid][account];
        user.reward = earned(_pid, account);
```

```
            user.rewardPerTokenPaid = pool.rewardPerTokenStored;
        }
        _;
    }
```

Listing 2.3: SecondLiveBeanReward.full.sol

Impact Duplicated calculations can cause extra gas costs.

Suggestion Remove duplicated calculations.

2.3 Note

2.3.1 Does not support deflationary tokens

Description The `SecondLiveBeanReward` contract is a staking contract that potentially allows users to deposit any token (the “LP token”) to the contract and earn rewards (if the corresponding pool is added by the project maintainers). However, according to the implementation of the `deposit` function, the deflationary tokens are not supported. As shown in the following code segment, the actual transfer amount is captured by the `stakeAmount` variable, but the weight increased for the pool and the user is the amount specified in the parameter. The amounts can be different if the LP token of the pool is a deflationary token.

```
function deposit(
    uint256 _pid,
    uint256 _amount
) public updateReward(_pid, msg.sender) checkStart(_pid) nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    uint256 balanceBefore = IERC20(pool.lpToken).balanceOf(address(this));
    IERC20(pool.lpToken).safeTransferFrom(
        msg.sender,
        address(this),
        _amount
    );
    uint256 balanceEnd = IERC20(pool.lpToken).balanceOf(address(this));

    uint256 stakeAmount = balanceEnd.sub(balanceBefore);
    require(stakeAmount > 0, "amount is error");
    ISecondLiveBean(pool.lpToken).burn(stakeAmount);

    pool.totalWeight = pool.totalWeight.add(_amount);

    UserInfo storage user = userInfo[_pid][msg.sender];
    user.amount = user.amount.add(_amount);

    emit Deposit(msg.sender, _pid, _amount);
}
```

Listing 2.4: SecondLiveBeanReward.full.sol

Feedback from the Developers The project maintainers ensures that the LP token for each pool created is the token represented by the `SecondLiveBeanV3` contract.