SALUS SECURITY

APR 2024

CODE SECURITY ASSESSMENT

SHUI LST

Overview

Project Summary

- Name: Shui LST - Farming contract
- Platform: Conflux Network
- Language: Solidity
- Repository:
    - https://github.com/Shui-LST/shui-lsd-core
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Shui LST - Farming contract
Version	v2
Type	Solidity
Dates	Apr 11 2024
Logs	Apr 09 2024; Apr 11 2024

Vulnerability Summary

Total High-Severity issues	0
Total Medium-Severity issues	0
Total Low-Severity issues	2
Total informational issues	5
Total	7

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Precision loss can lead to certain rewards getting locked in the contract	Low	Business Logic	Resolved
2	Unchecked return value	Low	Code Quality	Resolved
3	stakeToken is assumed with 18 decimals	Informational	Business Logic	Resolved
4	Should use upgradeable versions of dependencies	Informational	Code Quality	Acknowledged
5	Should not leave the implementation contract uninitialized	Informational	Business Logic	Resolved
6	Use of floating pragma	Informational	Configuration	Resolved
7	Missing zero address checks	Informational	Data Validation	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Precision loss can lead to certain rewards getting locked in the contract

Severity: Low	Category: Business Logic
Target: - contracts/espace/Farming.sol	

Description

contracts/espace/Farming.sol:L149

```
uint256 _rewardPerBlock = _totalReward / (_endBlock - _startBlock);
```

The addPool() function accepts a _totalReward input and uses the above formula to compute the _rewardPerBlock.

However, the resulting _rewardPerBlock might be less than expected due to the precision loss in division. In such cases, the total reward (i.e. _rewardPerBlock * (_endBlock - _startBlock)) to distribute is less than the _totalReward transferred to the contract, meaning some reward tokens are locked in the contract.

Recommendation

Consider inputting the _rewardPerBlock parameter and using it to compute the _totalReward in the addPool() function.

Status

The team has resolved this issue in commit f60463d.

2. Unchecked return value

Severity: Low	Category: Code Quality
Target: - contracts/espace/Farming.sol	

Description

contracts/espace/Farming.sol:L69,L84,L95,L149

```
pool.stakeToken.transferFrom(msg.sender, address(this), _amount);
pool.stakeToken.transfer(msg.sender, _amount);
poolInfo[_pid].rewardToken.transfer(msg.sender, _reward);
IERC20(_rewardToken).transferFrom(_msgSender(), address(this), _totalReward);
```

The Farming contract uses transfer/transferFrom functions to transfer ERC20 tokens without checking their return value.

Tokens not compliant with the ERC20 specification could return false from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

```
Callers MUST handle false from returns (bool success). Callers MUST NOT assume that
false is never returned!
```

Recommendation

Consider using the SafeERC20 library implementation from OpenZeppelin and call safeTransfer or safeTransferFrom when transferring ERC20 tokens.

Status

The team has resolved this issue in commit f60463d.

2.3 Informational Findings

3. stakeToken is assumed with 18 decimals

Severity: Informational	Category: Business Logic
Target: - contracts/espace/Farming.sol	

Description

Based on the computation in the code, the stakeToken is assumed to have 18 decimals. If a token with a different decimal value is added as the stakeToken in a pool, the distributed rewards may not match expectations.

Recommendation

If the stakeToken is expected to have 18 decimals, consider documenting it in the code. Otherwise, consider using the token's actual decimals in the computation.

Status

The team has resolved this issue in commit f60463d.

4. Should use upgradeable versions of dependencies

Severity: Informational	Category: Code Quality
Target: - contracts/espace/Farming.sol	

Description

Based on the context of the code, the Farming contract is designed to be deployed as an upgradeable proxy contract.

However, the current implementation inherits an non-upgradeable version of the Ownable contract.

Recommendation

It is recommended to use the OwnableUpgradeable contract instead of the Ownable contract.

Alternatively, you may use the Ownable2StepUpgradeable contract, which uses a two-step ownership transfer pattern.

Status

This issue has been acknowledged by the team.

5. Should not leave the implementation contract uninitialized

Severity: Informational	Category: Business Logic
Target: - contracts/espace/Farming.sol	

Description

According to OpenZeppelin, the implementation contract should not be left uninitialized. It's recommended that you invoke the _disableInitializers() function in the constructor to automatically lock it, so that the implementation contract can not be initialized by malicious users.

Recommendation

To prevent the implementation contract from being used, consider invoking the _disableInitializers function in the constructor of the Farming contract to automatically lock it when it is deployed.

Status

The team has resolved this issue in commit f60463d.

6. Use of floating pragma

Severity: Informational	Category: Configuration
Target: - contracts/espace/Farming.sol	

Description

```
pragma solidity ^0.8.18;
```

The Farming contract uses a floating compiler version 0.8.18.

Using a floating pragma 0.8.18 statement is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.

Recommendation

It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

Status

The team has resolved this issue in commit f60463d.

7. Missing zero address checks

Severity: Informational	Category: Data Validation
Target: - contracts/espace/Farming.sol	

Description

It is considered a security best practice to verify addresses against the zero address during initialization or setting. However, this precautionary step is absent for address variables FarmPool.stakeToken.

contracts/espace/Farming.sol:L156

```
poolInfo[_pid] = FarmPool({
    ...
    stakeToken: IERC20(_stakeToken), //@audit zero address check
    ...
});
```

Recommendation

Consider adding zero address checks for address variables FarmPool.stakeToken.

Status

The team has resolved this issue in commit f60463d. The zero address cannot be accepted by the function.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit 6b149d0:

File	SHA-1 hash
contracts/espace/Farming.sol	7833cad5443a9eb08adb476d3d639b369655c3da