SALUS SECURITY

NOV 2023

CODE SECURITY ASSESSMENT

B2B.MONEY

Overview

Project Summary

- Name: b2b.money
- Platform: EVM-compatible chains
- Language: Solidity
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	b2b.money
Version	v1
Type	Solidity
Dates	Nov 7 2023
Logs	Nov 7 2023

Vulnerability Summary

Total High-Severity issues	0
Total Medium-Severity issues	1
Total Low-Severity issues	1
Total informational issues	4
Total	6

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	Missing implementation of pause() and unpause()	Medium	Business logic	Pending
2	Missing zero-address validation for stakeToken	Low	Data Validation	Pending
3	Gas optimization suggestions	Informational	Gas Optimization	Pending
4	Floating pragma version	Informational	Configuration	Pending
5	Implementation contract should not be left uninitialized	Informational	Configuration	Pending
6	Could use Ownable2StepUpgradeable instead of OwnableUpgradeable	Informational	Access Control	Pending

2.2 Notable Findings

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

1. Missing implementation of pause() and unpause()

Severity: Medium	Category: Business logic
Target: - contract/StakingRewards.sol	

Description

The project lacks the ability to pause the StakingRewards contract.

The StakingRewards contract inherits from the PausableUpgradeable contract and utilizes the whenNotPaused modifier in some functions.

However, the PausableUpgradeable contract only provides internal functions _pause() and _unpause(), but there is no public or external function implemented in the StakingRewards contract to call them. Consequently, the contract remains in an unpaused state, as there's no way to pause or unpause it externally.

Recommendation

Consider implementing the pause() and unpause() external functions in the contract.

2. Missing zero-address validation for stakeToken

Severity: Low	Category: Data Validation
Target: - contract/StakingRewards.sol	

Description

contract/StakingRewards.sol:L35-L44

```
function initialize(address _rewardsToken, address _stakingToken, uint256
_rewardsDuration) public initializer {
    __ReentrancyGuard_init();
    __Ownable_init();
    __Pausable_init();

    require(_stakingToken != address(0), "stakingToken is zero address");
    rewardsToken = IERC20Upgradeable(_rewardsToken);
    stakingToken = IERC20Upgradeable(_stakingToken);
    rewardsDuration = _rewardsDuration;
}
```

The initialize() function doesn't check whether the stakeToken is set to address 0, and there are no other functions provided in the StakingRewards contract to update it. This means that once it's set to address 0 during initialization, it cannot be modified later.

contract/StakingRewards.sol:L130-L133

```
function notifyRewardAmount(uint256 reward) external onlyOwner updateReward(address(0))
{
    ...
    if (address(rewardsToken) != address(0)) {
        uint balance = rewardsToken.balanceOf(address(this));
        require(rewardRate <= balance.div(rewardsDuration), "Provided reward too high");
    }

    ...
}
```

Additionally, when stakeToken is set to 0, the notifyRewardAmount() function can execute successfully and will not check whether the contract has a sufficient balance to pay rewards. This means that in this scenario, users won't be able to claim rewards.

Recommendation

Consider adding a validation check in the initialize() function to ensure that stakeToken cannot be set to address 0. In addition, consider reverting the transaction in the notifyRewardAmount() function if stakeToken is address 0.

2.3 Informational Findings

3. Gas optimization suggestions

Severity: Informational	Category: Gas Optimization

Target:
- contract/StakingRewards.sol
- contract/LaunchpadStake.sol

Description

contract/StakingRewards.sol:L35

```
function initialize(address _rewardsToken, address _stakingToken, uint256
_rewardsDuration) public initializer {
    ...
}
```

contract/LaunchpadStake.sol:L32

```
function initialize(address owner) public initializer {
    ...
}
```

In the StakingRewards and LaunchpadStake contracts, the initializer functions can be marked as external to save gas.

contract/StakingRewards.sol:L21-L22

```
uint256 public periodFinish = 0;
uint256 public rewardRate = 0;
```

The uint256 variables are initialized to 0 by default. You can remove the assignments to save gas.

Recommendation

Consider making changes based on the above suggestions.

4. Floating pragma version

Severity: Informational	Category: Configuration
Target: - contract/StakingRewards.sol	

Description

```
pragma solidity ^0.8.0;
```

Using a floating pragma 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.

5. Implementation contract should not be left uninitialized

Severity: Informational	Category: Configuration
Target: - contract/StakingRewards.sol - contract/LaunchpadStake.sol	

Description

The LaunchpadStake and StakingRewards contracts use the upgradeable contract pattern. According to OpenZeppelin documentation, developers shouldn't leave an implementation contract uninitialized. Therefore, it's best to invoke the _disableInitializers function in the constructor of the upgradeable contracts. This step prevents the implementation contract from being initialized by malicious actors.

Recommendation

Consider making changes based on the above suggestion.

6. Could use Ownable2StepUpgradeable instead of OwnableUpgradeable

Severity: Informational	Category: Access Control
Target: - contract/LaunchpadStake.sol	

Description

The LaunchpadStake contract uses the transferOwnership() function from the OwnableUpgradeable contract. However, it's recommended to use another transferOwnership() function in the Ownable2StepUpgradeable contract. It provides a 2-stage ownership transfer mechanism that can prevent ownership from being transferred to an unintended or incorrect address.

contract/StakingRewards.sol:L35

```
function initialize(address owner) public initializer {
    __Ownable_init();
    __ReentrancyGuard_init();
    transferOwnership(owner);
}
```

Recommendation

Consider making changes based on the above suggestion.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files provided by the client:

File	SHA-1 hash
contract/LaunchpadStake.sol	e9cf8d485c34bfc0d4e5d63394e0efb7f35545f4
contract/StakingRewards.sol	a32fe2d7b0b1a8db235b203c979af66e038ecdaa