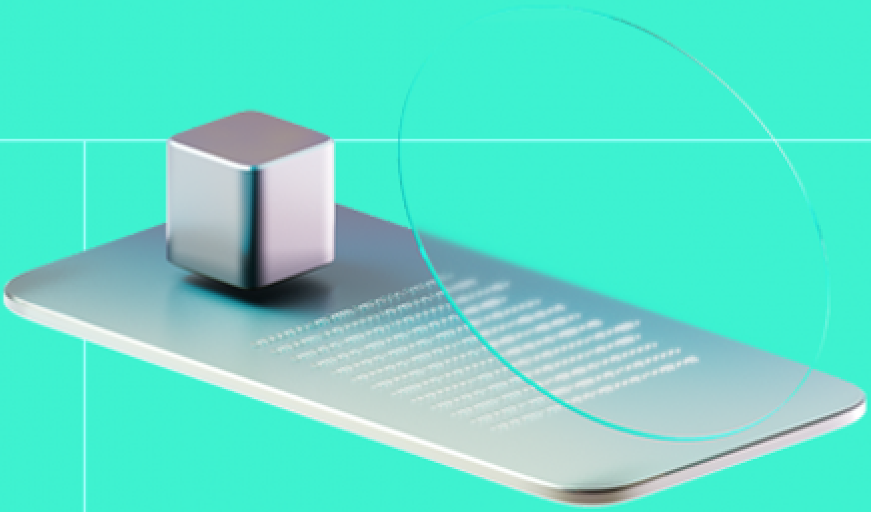
Smart Contract Code Review And Security Analysis Report



Customer: Lovefy Inc

Date: 16/02/2024

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

Lovefy Inc is a decentralized protocol for peer to peer value transfer connecting Web3, with traditional Social Media and E-Commerce platforms

Platform: EVM

Language: Solidity

Tags: Staking

Timeline: 30/01/2024 - 16/02/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/gotbitlabs/love-lending
Commit	b679cf2

Audit Summary

10/10	9/10	100%	10/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.8/10

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

3	3	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity	
Critical	1
High	0
Medium	1
Low	0

Vulnerability	Status
F-2024-0702 - Potential Locking of Unclaimed Rewards in Minting Contract	Fixed
F-2024-0713 - Stake and Reward Locking in Staking Contract for Consecutive Reward Periods	Fixed
F-2024-0716 - Checks-Effects-Interactions Pattern Violation in stake Function	Fixed

Document

Name	Smart Contract Code Review and Security Analysis Report for Lovefy Inc
Audited By	Ivan Bondar
Approved By	Przemyslaw Swiatowiec
Website	https://love.io/
Changelog	01/02/2024 - Preliminary Report; 16/02/2024 - Final Report

Table of Contents

System Overview

Lovefy Inc is a decentralized protocol for peer to peer value transfer connecting Web3, with traditional Social Media and E-Commerce platforms. The Minting Contract designed for staking tokens to earn rewards. It centralizes the process of token staking and reward distribution, offering a transparent and efficient mechanism for managing stakes and calculating rewards.

Files in the Scope:
Minting.sol - This contract is the backbone of Lovefy Inc's framework, offering a range of functionalities:

- Token Staking and Reward Distribution:
  - Central to the system, it manages the staking of tokens by users.
  - Responsible for tracking individual stakes, total staked tokens, and calculating rewards based on stake duration and amount.
- Withdrawal and Reward Claiming Mechanisms:
  - Enables users to withdraw their staked tokens along with the earned rewards.
  - Supports management of individual stakes and facilitates the claiming of rewards.

Privileged roles

- Contract Owner:
  - Possesses the authority to transfer ownership, and to pause or unpause the contract.
  - Responsible for updating reward rates and distribution periods, pivotal for the system's reward mechanism.

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 **10** out of **10**.

- Functional requirements are detailed:
  - Project overview is detailed.
  - All roles in the system are described.
  - Use cases described.
  - For each contract all futures are described.
  - All interactions are described.
- Technical description is robust:
  - Run instructions are provided.
  - Technical specification is provided.
  - NatSpec is sufficient.

Code quality

The total Code Quality score is **9** out of **10**.

- The development environment is configured.
- Solidity Style Guide violations.

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is present.
- Interactions by several users are tested.

Security score

Upon auditing, the code was found to contain **1** critical, **0** high, **1** medium, and **0** low severity issues. All issues were fixed in the remediation part of this audit, 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 **9.8**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- Importance of Setting Correct Reward Rates:
    - The contract's functionality relies on the accurate setting of reward rates to cover the designated reward period. If the reward rates are not configured correctly, it may lead to issues related to insufficient rewards, disrupting the expected distribution of rewards to users.

Findings

Vulnerability Details

F-2024-0713 - Stake and Reward Locking in Staking Contract for Consecutive Reward Periods - Critical

Description:

The staking contract exhibits a critical flaw in the `withdraw` function, specifically in the handling of `maxPotentialDebt`. This issue arises due to the contract's methodology for calculating and updating `maxPotentialDebt` during the seting of the new reward rate, staking and reward distribution processes.

The vulnerability is rooted in the contract's logic for adjusting `maxPotentialDebt`. When a user attempts to withdraw their stake and claim rewards, the contract reduces `maxPotentialDebt` by the amount of the reward associated with the withdrawing stake. However, this approach does not account for scenarios where additional stakes are made or new rewards are added during an active reward period. Consequently, `maxPotentialDebt` may not accurately reflect the total potential rewards claimable by all stakers, leading to a situation where `maxPotentialDebt` is less than the actual rewards claimable at the time of withdrawal.

The issue is particularly evident in the following snippet from the `withdraw` function:

```
if (totalSupply == 0) {
maxPotentialDebt = 0;
} else {
maxPotentialDebt -= reward;
}
```

This logic fails to consider the dynamic nature of staking and reward distribution. It assumes a linear and static relationship between the staked amount, rewards, and `maxPotentialDebt`, which is not always the case, especially when new stakes are made or additional rewards are added mid-period or post initial period.

If `maxPotentialDebt` is less than the rewards due at the time of withdrawal, the withdrawal transaction will revert, effectively locking both the user's stake and their earned rewards.

Assets:

- Minting.sol [https://github.com/gotbitlabs/love-lending]

Status: Fixed

Classification

Severity:	Critical
Impact:	Likelihood [1-5]: 5
	Impact [1-5]: 5
	Exploitability [1,2]: 1
	Complexity [0-2]: 0
	Final Score: 5.0 [Critical]

Recommendations

Recommendation:

To mitigate this vulnerability, the contract should implement a more robust mechanism for recalculating or adjusting `maxPotentialDebt` :

- Implementation of Emergency Withdraw Feature:
  - To enhance the robustness and trustworthiness of the Minting contract, it is recommended to implement an emergency withdrawal mechanism. This feature would allow stakers to unstake their tokens without claiming rewards in the event of issues with the reward system or other unforeseen problems.
- Dynamic Recalculation of `maxPotentialDebt`:
  - Implement a mechanism to dynamically recalculate `maxPotentialDebt` whenever there is a significant change in the staking pool. This includes not only when rewards are added or the reward rate is changed but also when new stakes are made or existing stakes are withdrawn.
  - This can be achieved by creating a function, which recalculates `maxPotentialDebt` based on the current `totalSupply`, `rewardRate`, and `rewardsDuration`.
  - Integration with Staking and Withdrawal Functions:
    - Integrate this function into the `stake` and `withdraw` functions to ensure `maxPotentialDebt` is updated in real-time with every staking activity.
    - This ensures that `maxPotentialDebt` always aligns with the total rewards claimable, preventing discrepancies that could lead to stake and reward locking.
  - Safeguard in the withdraw Function:
    - In the `withdraw` function, before reducing `maxPotentialDebt` by the reward amount, add a check to ensure that `maxPotentialDebt` does not become less than the total rewards claimable by all stakers.
    - If such a situation is detected, trigger a recalculation of `maxPotentialDebt` to align it with the actual reward liabilities.

Remediation (Revised commit: b679cf2) : The staking contract's `withdraw` function was updated to address the critical flaw related to `maxPotentialDebt`. The `maxPotentialDebt` variable was removed, and the reward tracking system was revised to ensure accurate and dynamic calculation of rewards. Although an emergency withdrawal feature was not implemented, the updated reward tracking system effectively negates potential issues related to stake and reward locking.

Evidences

Evaluating Stake and Reward Locking in Staking Contract for Consecutive Reward Periods

Reproduce:

PoC Steps:

- Deploy Contracts and Set Up Test Environment:
  - Deploy the staking contract and the ERC20 token contract.
  - Assign sufficient tokens to the deployer for minting rewards.
- Initial Reward Calculation and Minting:
  - Calculate the total reward based on the desired reward duration.
  - Mint the calculated reward amount to the staking contract's address.
- Activate Initial Reward Distribution:
  - Call the notifyRewardAmount function on the staking contract with the initial reward rate.
- First User Stakes Tokens:
  - Execute the stake function from user1 with a specified token amount.
  - Log the staking contract's balance and user1's balance post-stake.
- Simulate Time Passage:
  - Wait for half the duration of the reward period to simulate the passage of time.
- Second User Stakes Tokens:
  - Execute the stake function from user2 with the same token amount.
- Set Up and Activate Second Reward Period:
  - Calculate a new, smaller reward amount and mint these tokens to the staking contract.
  - Call notifyRewardAmount again with the new reward rate.
- Wait for Completion of Both Reward Periods:
  - Pause the test execution until both reward periods have fully elapsed.
- Attempt Withdrawal by First User:
  - User1 attempt to withdraw their stake and claim rewards.
  - Expect this transaction to be reverted due to insufficient funds related to maxPotentialDebt.
- Log Final Token Balances:
  - Log the final token balance of the staking contract and user1 to confirm the locking scenario.

```
it('Evaluating Stake and Reward Locking in Staking Contract for Cons
ecutive Reward Periods', async () => {
```

```
// Initial reward setup
let reward
```

See more

Results:

```
Minting contract
Staking Contract Balance (Post-User1 Stake): 1000499.999999999984096
ETH
User1 Balance (Post-Stake): 500.0 ETH
Staking Contract Final Balance: 1500999.999999999976144 ETH
User1 Final Balance (Expected to be unchanged): 500.0 ETH
√ Evaluating Stake and Reward Locking in Staking Contract for Consec
utive Reward Periods
```

Files: Minting.fundLock.test.ts

F-2024-0702 - Potential Locking of Unclaimed Rewards in Minting Contract - Medium

Description: The Minting contract's design for staking and reward distribution potentially leads to the locking of unclaimed rewards. This issue arises due to the mechanism of setting the reward rate and the `periodFinish` timestamp, which does not account for the possibility of delayed staking by users.

The `notifyRewardAmount` function sets the `rewardRate` and `periodFinish` based on the current timestamp and `rewardsDuration`. However, if there is a delay between setting these values and the first user staking tokens, the rewards corresponding to the elapsed time since `periodFinish` was set remain unclaimed and locked in the contract:

```
function notifyRewardAmount(uint128 newRewardRate) external onlyOwne
r {
// ...
periodFinish = uint80(block.timestamp) + rewardsDuration;
// ...
}
```

In scenarios where users stake after a delay, the rewards for the time elapsed prior to the first stake remain unclaimed and are effectively locked in the contract, as the reward calculation is based on the `periodFinish` timestamp.

Assets:

- Minting.sol [https://github.com/gotbitlabs/love-lending]

Status: Fixed

Classification

Severity: Medium

Impact: Likelihood [1-5]: 5

Impact [1-5]: 2

Exploitability [1,2]: 1

Complexity [0-2]: 0

Final Score: 3.5 [Medium]

Recommendations

Recommendation: To mitigate this issue, consider implementing a mechanism to adjust the `periodFinish` or the reward calculation to account for the time elapsed before the first stake is made (`globalId == 0`). This could involve recalculating the `periodFinish` based on the timestamp of the first stake or adjusting the reward distribution mechanism to distribute the unclaimed rewards proportionally to future stakers.

Remediation (Revised commit: b679cf2) : The Minting contract was updated to track unused rewards in case of delays in the first user stake or absence of stakes within a reward period. Unused rewards are now transferred to the owner upon creation of a new reward period, ensuring no rewards are locked in the contract.

Evidences

Reward Locking Scenario in Staking Contract

Reproduce:

PoC Steps:

- Mint and Transfer Rewards:
  - Calculate the total reward amount and reward rate.
  - Mint the calculated reward amount to the staking contract address.
- Initialize Reward Distribution:
  - Call notifyRewardAmount on the staking contract with the calculated reward rate.
- Simulate Delay Before Staking:
  - Wait for a specified period (e.g., 1 day) to simulate a delay before any user stakes tokens.
- Execute User Stake:
  - Have a user (e.g., user1) call the stake function with a specified token amount.
- Wait for Reward Period to End:
  - Wait for the entire reward duration plus an additional buffer time to pass.
- Perform Withdrawal:
  - Have the same user call the withdraw function to unstake tokens and claim rewards.
- Verify Contract State:
  - Check the token balance of the staking contract to confirm if any unclaimed rewards are left.
  - Compare the final token balance of the user to ensure rewards were correctly claimed.

PoC Test:

```
it('Reward Locking Scenario in Staking Contract', async () => {
// Calculate the reward based on the desired reward duration
reward = ethers.constants.WeiPerEther.mul(1_000_000)
.div(rewardDuration)
.mul(rewardDuration);

// Determine the reward rate per second
rewardRate = reward.div(rewardDuration);
```

```
// Mint the calculated reward tokens to the staking contract
// This is necessary to ensure the staking contract has enough token
s to pay out rewards
await token.connect(deployer).mint(staking.address, reward);

// Initialize the reward distribution with the calculated reward rat
e
await staking.notifyRewardAmount(rewardRate);

// Assertions to verify that the reward rate and remaining rewards a
re set correctly
expect(await
```

See more

Results:

```
Minting contract
Staking Contract Balance (Post-Reward Transfer): 999999.999999999984
096 ETH
User Initial Balance: 1000.0 ETH
Staking Contract Balance (Post-User Stake): 1000499.999999999984096
ETH
User Balance (Post-User Stake): 500.0 ETH
Staking Contract Balance (Post-Withdrawal): 2739.757737189243995 ETH
User Balance (Post-Withdrawal): 998260.242262810740101 ETH
√ Reward Locking Scenario in Staking Contract
```

Files: Minting.noRewardsPeriod.test.ts

F-2024-0716 - Checks-Effects-Interactions Pattern Violation in stake Function - Info

Description:

The `stake` function in the staking contract allows users to stake tokens by updating contract state variables and performing a token transfer from the user to the contract. The sequence of these operations is crucial for maintaining contract security and integrity.

```
function stake(uint128 amount) external whenNotPaused {
_updateReward(0);
require(amount > 0, 'Cannot stake 0');
require(periodFinish > block.timestamp, 'Reward period not activated
');

// ... state updates ...

stakes[++globalId] = Stake({
owner: msg.sender,
amount: amount,
earned: 0,
userRewardPerTokenPaid: rewardPerTokenStored,
timestamp: uint80(block.timestamp),
unstakedAtBlockNumber: 0,
unstakedAtBlockTimestamp: 0
});

// ... more state updates ...

stakingToken.safeTransferFrom(msg.sender, address(this), amount);
}
```

In the current implementation, the `safeTransferFrom` call is made after updating the contract's state. This call acts as a proof that the user possesses the required token amount and has granted the contract permission to access their tokens. It also confirms the successful receipt of these tokens by the contract.

This order of operations can be exploited in certain scenarios:

- Malicious or Flawed Staking Token Contract: If the staking token contract is maliciously designed or has implementation flaws, it could enable reentrancy attacks during the token transfer. An attacker could take advantage of the state changes that have already occurred before the transfer.
- Impact on Dependent Contracts: If other contracts rely on the proof of stake provided by this contract (e.g., for enabling certain actions for stakers), they could also be impacted by this vulnerability. The premature state change before the token transfer could lead to incorrect assumptions or validations in those dependent contracts.

Assets:

- Minting.sol [https://github.com/gotbitlabs/love-lending]

Status: Fixed

Classification

Severity:	Info
Impact:	Likelihood [1-5]: 2 Impact [1-5]: 2 Exploitability [1,2]: 2 Complexity [0-2]: 0 **Final Score:** 1.6 [Informational]

Recommendations

Recommendation:	Modify the `stake` function to perform the `safeTransferFrom` call immediately after the initial checks and reward update. **Remediation** (Revised commit: b679cf2) : The `stake` function in the staking contract was updated to perform the `safeTransferFrom` call immediately after initial checks.

Observation Details

F-2024-0715 - Unsafe `uint128` casting - Info

Description: The staking contract exhibits potential data type mismatches in two functions: `earned` and `_notifyRewardAmount`. In both cases, `uint128` is used for return values and intermediate calculations, while the underlying logic involves `uint256` computations. This inconsistency could lead to data truncation if the calculated values exceed the storage capacity of `uint128`.

Key Code Snippets:

`earned` function:

```
function earned(uint256 id) public view returns (uint128) {
Stake memory _stake = stakes[id];
if (_stake.unstakedAtBlockNumber == 0) {
return uint128(
(_stake.amount * (rewardPerToken() - _stake.userRewardPerTokenPaid))
/
ACCURACY + _stake.earned
);
}
return 0;
}
```

`_notifyRewardAmount` function:

```
function _notifyRewardAmount(uint128 newRewardRate) private {
uint128 reward = uint128(newRewardRate * rewardsDuration);
// ... rest of the function
}
```

The primary concern is the potential for inaccurate calculations due to data truncation. This could affect the fairness and reliability of the staking mechanism and reward rate settings.

Assets:

- Minting.sol [https://github.com/gotbitlabs/love-lending]

Status: Fixed

Recommendations

Recommendation: To mitigate these risks, consider the following adjustments:

- Uniform Data Types:
  - Update the return type of the `earned` function and the local variable `reward` in `_notifyRewardAmount` to `uint256`. This change aligns the data types with the internal calculation data types, preventing potential truncation issues.
- Implement Safe Casting Mechanisms:

- Utilize OpenZeppelin's safe-casting library to handle casting operations securely. This library includes checks to prevent type conversion errors, ensuring safe and accurate casting practices.

Remediation (Revised commit: b679cf2) : The staking contract was updated to address potential data type mismatches. The `earned` function now returns `uint256` instead of `uint128` to prevent data truncation. Additionally, checks have been introduced in the `_notifyRewardAmount` function to ensure values do not exceed the maximum rate, enhancing the accuracy and fairness of reward calculations and distributions.

F-2024-0717 - Redundant periodFinish Check in the Stake Function - Info

Description:

In the `stake` function of the contract, there is a redundant check related to the `periodFinish` variable.

The first `require` statement checks if the reward period is activated, ensuring that the function cannot proceed if the condition is not met. Subsequently, there is another check within the same function that verifies the same condition. Since the function already checks this condition in the initial `require` statement, the second check is redundant and can be safely removed.

```
function stake(uint128 amount) external whenNotPaused {

//..
require(periodFinish > block.timestamp, 'Reward period not activated
');
// Redundant check for periodFinish
if (totalSupply == 0 && uint80(block.timestamp) < periodFinish) {
maxPotentialDebt =
(periodFinish - uint80(block.timestamp)) *
uint256(rewardRate);
}
//..
})
```

This redundancy can be removed to streamline the function and make it more efficient.

Assets:

- Minting.sol [https://github.com/gotbitlabs/love-lending]

Status: Fixed

Recommendations

Recommendation:

Remove the second check for the `periodFinish` from the `stake` function to simplify the code.

Remediation (Revised commit: b679cf2) : The redundant check for `periodFinish` in the `stake` function of the contract was removed.

F-2024-0718 - Function Parameter "paused" Shadows Function paused() from Pausable Contract - Info

Description: In the Minting contract's `setPaused` function, the parameter `paused` shadows the `paused()` function inherited from the Pausable contract. Shadowing occurs when a local identifier (variable, parameter, etc.) in a scope has the same name as an identifier in an outer scope, potentially leading to confusion and errors in understanding the code.

This issue does not pose a direct security risk, but it can lead to readability and maintainability issues.

Assets:

- Minting.sol [https://github.com/gotbitlabs/love-lending]

Status: Fixed

Recommendations

Recommendation: Rename the `paused` parameter in the `setPaused` function to a distinct name that does not conflict with any inherited functions or state variables.

Remediation (Revised commit: b679cf2) : The Minting contract's `setPaused` function parameter was renamed from `paused` to `state` to avoid shadowing the `paused()` function inherited from the Pausable contract.

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/gotbitlabs/love-lending
Commit	c62c919b4e89cec629fe1478cec4bb578d903264
Whitepaper	NA
Requirements	NatSpec
Technical Requirements	NatSpec
Deployed Address ETH	0xBb39219BE50f2743353e23Db204997bA421275de
Deployed Address BSC	0x2CB5398C6dDa35636324c6050960E8722a7E2Dc1
Deployed Address PLS	0xb7F1198a651e8009052b3eBb59C53dADD9AF8D25

Contracts in Scope

./contracts/Minting.sol