Audit Report

Illiquid Labs

v1.0

January 10, 2023

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Illiquid Labs to perform a security audit of NFT non-custodial loan and raffle contracts.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following GitHub repository:

https://github.com/Illiquidly/illiquidlabs-contracts

Commit hash: `6b9209ebd8a26a129bdb1c784ef72c9783d90b5d`

The following directories have been in scope:

- `contracts/nft-loans-non-custodial`
- `contracts/raffles`
- `contracts/randomness_verifier`
- `relevant imports from packages/*`

Fixes have been verified on the commit with the following hash:
`7c68ee5d73c105d4bdfb7ff1cf9a0dd0c61f748b`

This audit has originally been started on the following GitHub repository. The code in the scope of this audit has later been applied to the repository above. The original repository was:

https://github.com/Illiquidly/illiquidly-contracts-private

Commit hash: `27b9ff6cc8133ea20502963520c96c4be85da098`

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line by line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The submitted code features Illiquid Lab's NFT collateralized loans and raffles contract. The NFT collateralized loans contract allows a user to borrow money against their NFTs as collateral and lend money to earn yield, while the raffle contract creates new options for the exchanging of liquid and illiquid assets through the buying or selling of raffle tickets.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending,** **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**Low-Medium**	-
Code readability and clarity	**Medium-High**	The codebase contains straightforward code comments.
Level of documentation	**Medium-High**	Documentation was available at https://illiquidlabs.gitbook.io/illiquid-labs/illiquid-labs-platform/nft-collateralised-loans.
Test coverage	**Medium-High**	Cargo tarpaulin reports a code coverage of 65.85% and 62.15%.

Summary of Findings

No	Description	Severity	Status
1	Attacker can drain funds by refusing completed offers	Critical	Resolved
2	Borrowers are forced to default on loans with zero fees or interest	Critical	Resolved
3	Attackers can steal NFTs with approvals on contract	Major	Resolved
4	Malicious raffle owner can replay `CancelRaffle` message to steal NFTs in contract	Major	Resolved
5	Incorrect implementation of `CW20 Receive` message	Minor	Resolved
6	Consider validating fee distribution address and fee rate	Minor	Resolved
7	Incorrect specification of `rand_fee` and `raffle_fee` may cause overflows	Minor	Resolved
8	Updated configurations are not enforced and validated	Minor	Resolved
9	Users can provide zero assets or tokens for raffles and loans	Minor	Resolved
10	Insufficient validation of new raffle configuration	Minor	Partially Resolved
11	Inconsistent public key input type	Minor	Resolved
12	Duplicate storage read when performing a withdrawal is inefficient	Informational	Resolved
13	Redundant code in raffles contract	Informational	Resolved
14	Use of magic numbers	Informational	Resolved
15	Inefficient validation of sent native tokens	Informational	Resolved
16	Inconsistent use of fixed point arithmetic	Informational	Resolved
17	Contracts should implement a two step ownership transfer	Informational	Resolved

18	Codebase contains outstanding TODOs	Informational	Resolved
19	Incorrect owner value error message	Informational	Resolved
20	Overflow checks not enabled for release profile	Informational	Resolved

Detailed Findings

1. Attacker can drain funds by refusing completed offers

Severity: Critical

In `contracts/nft-loans-non-custodial/src/execute.rs:336`, a borrower can refuse an offer even if the offer had been accepted. Suppose a borrower calls `RefuseOffer` for an accepted loan – that allows the lender to call `WithdrawRefusedOffer` in order to withdraw the deposited funds. This is problematic because the lender already had their principal and interest repaid back in `contracts/nft-loans-non-custodial/src/execute.rs:546-549`, resulting in the contract losing funds.

An attacker can exploit this issue by completing and refusing a loan to withdraw excess funds as the lender. This can be exploited repeatedly to drain all funds from the contract.

Please see the `test_steal_funds` test case to reproduce the issue.

Recommendation

We recommend only allowing the borrower to refuse an offer for the `Published` state.

Status: Resolved

2. Borrowers are forced to default on loans with zero fees or interest

Severity: Critical

In `contracts/nft-loans-non-custodial/src/execute.rs:557-567`, the contract tries to send fees to the treasury without validating that the amount is greater than zero. Suppose a lender accepted a loan with the loan term's interest as 0. The calculated fee amount would be zero (see line `531`), causing the contract to send 0 funds to the fee contract. Since Cosmos SDK does not allow 0 amount transfers, borrowers would not be able to repay in time, causing their loans to default.

A lender can exploit this issue by providing zero-interest loans, forcing the borrower to default on their loans so the lender can have the NFT in return.

Additionally, this issue will also occur if the admin sets the fee rate to the maximum value, preventing all borrowers from repaying their loans successfully.

Recommendation

We recommend only sending funds if the amount is greater than zero.

Status: Resolved

3. Attackers can steal NFTs with approvals on contract

Severity: Major

In `contracts/nft-loans-non-custodial/src/execute.rs:433-440` and `contracts/raffles/src/execute.rs:64-71`, the `CW721` NFT is transferred to the contract using the `TransferNft` message without verifying the caller is the owner of the NFT. Since the transferred NFT is stored under the caller's raffle or loan, the caller can withdraw the NFT after it ends, causing the real owner to lose their NFT.

The possibility of this could happen when the user approves their NFT in the first transaction but fails to create a successful raffle or loan in the second transaction (e. g. due to invalid arguments). An attacker can then exploit the vulnerability If the approval is not revoked and does not expire in the next block.

We classify this as a major issue because the attack requires pre-approval on the contract for successful exploitation.

Recommendation

We recommend validating that the caller of the `CW721` NFT is the correct owner using the `OwnerOf` query message.

Status: Resolved

4. Malicious raffle owner can replay `CancelRaffle` message to steal NFTs in contract

Severity: Major

In `contracts/raffles/src/execute.rs:172`, the `get_raffle_owner_messages` function is called to refund the owner their NFT when the owner cancels a raffle. The `execute_cancel_raffle` function does not prevent replay attacks, allowing the owner to cancel a raffle as long as there are no tickets bought.

An attacker can exploit this issue by creating a raffle and immediately canceling it, resulting in the contract storing a valid `RAFFLE_INFO` for the specific raffle identifier value. After that, the attacker sells the NFT in a marketplace to a victim. Once the victim creates a raffle and deposits the NFT into the contract, the attacker executes `CancelRaffle` and steals the NFT.

We classify this as a major issue due to the high exploit difficulty.

Recommendation

We recommend preventing replay attacks in `execute_cancel_raffle` by validating that the raffle has not ended.

Status: Resolved

5. Incorrect implementation of `CW20 Receive` message

Severity: Minor

In `contracts/raffles/src/contract.rs:113`, users can deposit CW20 tokens into the contract through a receive callback function. However, the callback message does not follow the CW20 specification. Consequently, users are unable to deposit CW20 into the raffle contract directly.

We consider this a minor issue as users can still deposit CW20 tokens using the allowance functionality.

Recommendation

We recommend implementing the CW20 receive callback message as shown in the example here.

Status: Resolved

6. Consider validating fee distribution address and fee rate

Severity: Minor

In `contracts/nft-loans-non-custodial/src/contract.rs:45` and `contracts/nft-loans-non-custodial/src/admin.rs:40`, the contract owner instantiates and updates the fee distributor address. However, in each case, the address is not validated, which could lead to the contract being unable to execute as transfers may be made to an invalid address.

Similarly, the contract owner instantiates and updates the fee rate in `contracts/nft-loans-non-custodial/src/contract.rs:46` and `contracts/nft-loans-non-custodial/src/admin.rs:61`. A misconfigured fee rate would prevent users from repaying borrowed funds due to overflows occurring in `contracts/nft-loans-non-custodial/src/execute.rs:527`.

We classify this as a minor issue since only the owner can cause it.

Recommendation

We recommend validating the fee_distributor to be a valid address and ensuring the fee_rate is less than or equal to 10_000 prior to storage during instantiation and update of the contract config.

Status: Resolved

7. Incorrect specification of rand_fee and raffle_fee prevents ending a raffle

Severity: Minor

In contracts/raffles/src/contract.rs:60-63 and contracts/raffles/src/contract.rs:244-251, the contract owner is able to define the raffle and randomness provider fee rates. Should the sum of rand_fee and raffle_fee be greater than 10_000, then this could cause an underflow to occur in contracts/raffles/src/state.rs:228, preventing a raffle from being ended.

We classify this as a minor issue since only the admin can cause it.

Recommendation:

We recommend validating that the total of rand_fee and raffle_fee is less than 10_000 during the instantiation and update of the raffle contracts.

Status: Resolved

8. Configuration updates are not enforced or validated

Severity: Minor

In contracts/raffles/src/contract.rs:236-254, the contract owner is able to update the configuration. However, unlike during instantiation, validation is not performed. For example, the minimum raffle duration can be updated to a value lower than the hardcoded MINIMUM_RAFFLE_DURATION constant.

This could lead to unexpected outcomes, for instance, if rand_fee is set to zero the randomness provider cannot be reimbursed.

We classify this as a minor issue since only the owner can cause it.

Recommendation

We recommend validating config variables during updates as done during instantiation in `contracts/raffles/src/contract.rs:43-70`.

Status: Resolved

9. Users can provide zero assets or tokens for raffles and loans

Severity: Minor

In `contracts/raffles/src/execute.rs:50` and `contracts/nft-loans-non-custodial/src/execute.rs:40`, a caller can provide empty `all_assets` and `tokens` vectors. The former would cause the winner of the raffle to receive no NFTs in return, while the latter would cause the borrower to receive an undercollateralized loan.

We classify this issue as minor because the raffle requires active users to participate, and a lender must be willing to accept the offer without any return assets.

Recommendation

We recommend ensuring the vector lengths are equal to or greater than one.

Status: Resolved

10. Insufficient validation of new raffle configuration

Severity: Minor

In `contracts/raffles/src/contract.rs:52-59`, the contract owner defines the minimum duration and timeout allowed for a raffle. Similarly, in `packages/raffles/src/state.rs:145-178`, a new raffle is defined, and the specification is validated to ensure it does not violate the minimum duration and timeout.

However, users are nonetheless able to set raffle timeouts and durations with a maximum value of `u64::max`. Additionally, no validation is performed to ensure the start time is not in the past. This could lead to unexpected behavior for raffle participants, including locking participant funds for extended periods of time.

Recommendation

We recommend performing additional sanity checks on the configuration of new raffles, including maximum raffle duration and timeouts, and ensuring that the raffle start time is not in the past.

Status: Partially Resolved

The client states that they do not commit to putting a maximum date or timeout on raffles. Those parameters should be chosen freely and they do not see a limit that could correspond to all users. On top of that, an extra maximum parameter does not really make sense.

11. Inconsistent public key input type

Severity: Minor

In `contracts/raffles/src/contract.rs:68`, the public key is stored without decoding as base64. As a reference, the `execute_change_parameter` function in line `260` sets the value after decoding it from base64. If the owner provided a base64 encoded public key during contract instantiation, the `random_pubkey` would not be decoded accordingly.

We classify this issue as minor since only the contract owner can cause it.

Recommendation

We recommend decoding `msg.random_pubkey` as base64 in line `68`.

Status: Resolved

12. Duplicate storage read when performing a withdrawal is inefficient

Severity: Informational

When executing the `_withdraw_offer_unsafe` function in `contracts/nft-loans-non-custodial/src/execute.rs:314`, the loan offer is retrieved from storage. The contract then reads the deposited funds from the offer and creates a `BankMsg` to return the funds to the original sender.

However, prior to the execution of the `_withdraw_offer_unsafe` function, the contract has already read storage and retrieved the offer. This duplicate read is unnecessary and inefficient.

Recommendation

We recommend removing the retrieval of the offer in the `_withdraw_offer_unsafe` function in `contracts/nft-loans-non-custodial/src/execute.rs:320` and instead passing the offer as an argument.

Status: Resolved

13. Redundant code in raffles contract

Severity: Informational

In `contracts/raffles/src/contract.rs:185-195`, the contract owner is able to set the current contract address as the admin. This action is also possible through the `execute_change_parameter` function in `contracts/raffles/src/contract.rs218-271`. Duplication of functionality increases both the code complexity and size.

Recommendation

We recommend removing the function `execute_renounce` to simplify the codebase.

Status: Resolved

14. Use of magic numbers

Severity: Informational

Throughout the codebase, numeric literals are used without description or context, so-called “magic numbers”, which reduces code-readability and increases complexity.

Instances of magic numbers can be found in:

- `contracts/nft-loans-non-custodial/src/execute.rs:527-528` and
- `contracts/raffles/src/state.rs:226-227`.

Recommendation

We recommend removing magic numbers throughout the codebase and replacing them with descriptive constants.

Status: Resolved

15. Inefficient validation of sent native tokens

Severity: Informational

In `contracts/raffles/src/execute.rs:245-252`, the contract ensures that when tickets are bought with native tokens, the assets defined as arguments match the tokens sent. The `if` statements in lines `245` and `248` duplicate part of the logic, which is inefficient and reduces code readability.

Recommendation

We recommend rewriting the `if` statement in lines `contracts/raffles/src/execute.rs:245-252` into a single logical statement.

Status: Resolved

16. Inconsistent use of fixed point arithmetic

Severity: Informational

Throughout the codebase, integers are used to perform fixed point arithmetic, e.g. validate fractions and perform multiplication. However, between contracts, the number of fixed point decimals differs. This increases the complexity of the codebase and decreases maintainability.

Recommendation

We recommend using fixed point arithmetic consistently throughout the codebase.

Status: Resolved

17. Contracts should implement a two-step ownership transfer

Severity: Informational

The contracts within the scope of this audit allow the current owner to execute a one-step ownership transfer. While this is common practice, it presents a risk for the ownership of the contract to become lost if the owner transfers ownership to the incorrect address. A two-step ownership transfer will allow the current owner to propose a new owner, and then the account that is proposed as the new owner may call a function that will allow them to claim ownership and actually execute the config update.

Recommendation

We recommend implementing a two-step ownership transfer. The flow can be as follows:

1. The current owner proposes a new owner address that is validated and lowercase.
2. The new owner account claims ownership, which applies the configuration changes.

Status: Resolved

18. Codebase contains outstanding TODOs

Severity: Informational

The codebase contains outstanding TODO items in the following locations:

- `contracts/nft-loans-non-custodial/src/query.rs:24` and
- `packages/nft-loans/src/msg.rs:31`.

Recommendation

We recommend completion of all outstanding TODO items.

Status: Resolved

19. Incorrect owner value error message

Severity: Informational

In `contracts/raffles/src/contract.rs:341`, the error evaluates as “raffle owner not found in context” if the `owner` value is invalid. This is misleading as the caller is the randomness provider and not the raffle owner as seen in `contracts/raffles/src/execute.rs:417`.

Recommendation

We recommend modifying the error to refer to the randomness provider to prevent confusion.

Status: Resolved

20. Overflow checks not enabled for release profile

Severity: Informational

The following packages and contracts do not enable `overflow-checks` for the release profile:

- `contracts/nft-loans-non-custodial/cargo.toml`

- `contracts/raffles/cargo.toml`
- `contracts/randomness_verifier/cargo.toml`

While enabled implicitly through the workspace manifest, a future refactoring might break this assumption.

Recommendation

We recommend enabling overflow checks in all packages, including those that do not currently perform calculations, to prevent unintended consequences if changes are added in future releases or during refactoring. Note that enabling overflow checks in packages other than the workspace manifest will lead to compiler warnings.

Status: Resolved

Appendix A: Test Cases

1. Test case for “Attackers can steal funds by refusing completed offers”

The test case should fail if the vulnerability is patched.

```rust
#[test]
fn test_steal_funds() {
    // modification of test_normal_flow() test case
    // reproduced in contracts/nft-loans-non-custodial/src/testing/tests.rs

    // note: attacker is both the lender and borrower
    let mut deps = mock_dependencies();
    let env = mock_env();
    init_helper(deps.as_mut());

    // malicious terms, interest set to 0 to prevent fee distribution
    let terms = LoanTerms {
        principle: coin(1000, "luna"),
        interest: Uint128::new(0),
        duration_in_blocks: 1,
    };

    // attacker deposit nft collateral
    add_collateral_helper(
        deps.as_mut(),
        "attacker",
        "nft",
        "58",
        Some(Uint128::new(1000_u128)),
        Some(terms.clone()),
    )
    .unwrap();

    // attacker accepts their own offer
    // 1. attacker send 1000 LUNA to contract
    // 2. contract takes attacker's NFT
    // 3. contract sends 1000 LUNA to attacker
    accept_loan_helper(deps.as_mut(), "attacker", "attacker", 0, coins(1000,
"luna")).unwrap();

    // attacker repay the funds
    // 1. attacker send 1000 LUNA to contract
    // 2. contract sends back attacker's NFT
    repay_borrowed_funds_helper(deps.as_mut(), "attacker", 0, coins(1000,
"luna"), env).unwrap();
```

```
    // attacker calls `RefuseOffer` to mutate offer state to `Refused`
    refuse_offer_helper(deps.as_mut(), "attacker", "1").unwrap();

    // attacker calls `WithdrawRefusedOffer` to get their refund
    // contract sends 1000 LUNA to attacker
    let res = withdraw_refused_offer_helper(deps.as_mut(), "attacker",
"1").unwrap();

    // total profit by attacker: 1000 LUNA
    assert_eq!(
        res.messages,
        vec![SubMsg::new(BankMsg::Send {
            to_address: "attacker".to_string(),
            amount: coins(1000, "luna"),
        }),]
    );
}
```