Audit Report

Risk Harbor

v1.0

March 22, 2022

Table of Contents

License

Disclaimer

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 Terraform Labs to perform a security audit of the Risk Harbor smart 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/Risk-Harbor/riskharbor-terra

Commit hash: `62a5f41236f8be531af4b91e70cf9ce17d89c07a`

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

Risk Harbor is a risk management marketplace for decentralized finance (DeFi) that utilizes an impartial claims process to protect liquidity providers and stakers against smart contract risks, hacks, and attacks. Risk Harbor's automated claims assessment process checks the redeemability of claim tokens with the protocol that issued them by examining key invariants that differ from protocol to protocol.

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 improve 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 a security audit and vice versa.

Summary of Findings

No	Description	Severity	Status
1	Lack of pool status validation in claim function leads to a race between underwriters to withdraw and insurees to claim funds	Major	Resolved
2	Wrong payout calculation may lead to last claiming insurees not being able to claim	Major	Resolved
3	Owner may update default ratio to prevent claims from being made	Major	Resolved
4	Lack of validation of state parameters	Minor	Resolved
5	Pool may be configured with an incorrect credit token	Minor	Resolved
6	Owner may update parameters that can negatively impact users	Minor	Acknowledged
7	Claim payout attribute value is missing tax deduction	Minor	Resolved
8	Lack of validation of `Cw20ReceiveMsg msg` can surprise users	Minor	Acknowledged
9	Default state assessment of protocols added in the future may be subject to manipulation	Minor	Acknowledged
10	Hard-coded Anchor market address decreases flexibility	Informational	Resolved
11	Selective usage of CW20 features may lead to wrong user expectations	Informational	Resolved
12	Lack of validation of pool name length may have adverse consequences	Informational	Resolved
13	Unused validation can be removed to increase efficiency	Informational	Resolved
14	Inefficient loading of state	Informational	Resolved
15	Error messages could include more details	Informational	Resolved

Code Quality Criteria

Criteria	Status	Comment
Code complexity	Low-Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	Low	No documentation besides the whitepaper was provided. The code contains minimal documentation, but the low complexity and small size of the codebase makes most behaviour self-explanatory.
Test coverage	Medium-High	-

Detailed Findings

1. Lack of pool status validation in claim function leads to a race between underwriters to withdraw and insurees to claim funds

Severity: Major

The `claim` function in `src/contract.rs:251` does not perform a check to ensure that the pool is not in a closed state. If the pool status is closed and a hack/default event occurs, swaps can still be performed. Insurees will only claim if the payout amount is bigger than the value of the covered token. Rational underwriters will anticipate that, and rush to withdraw liquidity to maximize their return. Rational insurees will anticipate that rush, and try to claim as fast as possible. This race can lower trust in the protocol, but will eventually put users to a disadvantage that act slower than others.

Recommendation

We recommend ensuring that the pool has not been closed in the `claim` function. This has a negative implication though – if a hack occurs before the pool has expired, but then the pool expires, a quick underwriter could enforce a status of closed. That would mean the pool will never get into the hacked state, despite a hack event having occurred. To solve this, we recommend extending the logic in the `check_if_at_status` function in `512-521` to check if the pool is in a default state and setting the status to hacked or closed accordingly.

Status: Resolved

2. Wrong payout calculation may lead to last claiming insurees not being able to claim

Severity: Major

The calculation of `payout` in the `claim` function in `src/contract.rs:302` uses the `default_ratio` rather than the `payout_ratio`. This is different from the calculation of the capacity in line `476` and may result in the last insuree trying to swap running into out-of-funds errors.

Recommendation

We recommend replacing `default_ratio` with `payout_ratio` in line `302`.

Status: Resolved

3. Owner may update default ratio to prevent claims from being made

Severity: Major

`default_ratio`, `payout_ratio`, and `expiration` are updatable variables that are used to determine whether or not the default event occurred in `src/contract.rs:288`, how much to pay out in `288`, and whether the pool is closed in line `512-514`. If the owner account was compromised or the owner simply wanted to control whether claims should be possible and what height they should have, they could update these values, affecting any users that have not claimed yet.

Recommendation

We recommend removing the ability to update values through the `update_config` function in `src/contract.rs:408` once a pool is no longer in the open state.

Status: Resolved

4. Lack of validation of state parameters

Severity: Minor

The `instantiate` and `update_config` functions in `src/contract.rs:28` and `408` lack validation of the following parameters:

- `expiration`: If set to a time in the past, this will cause the next invocation of `check_if_at_status` to transition the pool status to closed.
- `payout_ratio`: If set to zero, the next invocation of the purchase would attempt to divide by zero in line `476` while attempting to calculate the `total_capicity`, which would cause a panic.
- `price` (validated in the `instantiate`, but not in the `update_config function`): If set to zero, the `purchase` function would panic in line `220`. A price of more than one would economically not make much sense.

Recommendation

We recommend performing validation to ensure that `expiration` is set to a time in the future, `payout_ratio` is greater than `0`, and `price` is set to a value between `0` and `100`.

Status: Resolved

5. Pool may be configured with an incorrect credit token

Severity: Minor

The `InstantiateMsg` struct in `src/msg.rs:11` defines `credit_token` with a type of `Addr`, which is then stored without validation in `src/contract.rs:70`. This allows instantiation of a pool with an invalid credit token address, which may only be detected when claims happen after a hack/default.

Recommendation

We recommend defining `credit_token` as a `String` and then performing validation on that string in the `instantiate` function in `src/contract.rs:28`.

Status: Resolved

6. Owner may update parameters that can negatively impact users

Severity: Minor

The `update_config` function in `src/contract.rs:408-439` allows the owner to update the following values:

- `price`
- `expiration`
- `default_ratio`
- `payout_ratio`

Any underwriters that provide liquidity to the pool at the time of such a change would be subject to these changes, but they have no way to withdraw the pool if the values change to their disadvantage. This disincentivizes underwriters from entering the pool in the first place, and also adds to centralization of the protocol.

We classify this finding as minor because only the owner can make these changes.

Recommendation

We recommend allowing underwriters to leave the pool after changes have been made, applying updates only to new underwriters, or removing the ability to update the config altogether.

Status: Acknowledged

The Risk Harbor team states that in this version, only TFL and the Terra Community Fund are expected to be underwriters.

7. Claim payout attribute value is missing tax deduction

Severity: Minor

The payout attribute in the `claim` function in `src/msg.rs:324` contains the pre-tax payout value. This value will be different from the amount that is actually sent after deducting the tax.

Recommendation

We recommend changing the `payout` attribute to reflect the post-tax amount.

Status: Resolved

8. Lack of validation of `Cw20ReceiveMsg` `msg` can surprise users

Severity: Minor

The `claim` function in `src/contract.rs:255` receives a `Cw20ReceiveMsg` but does not match/use the contained `msg`. This means that the contract will execute a claim regardless of the received message. This goes against user expectations.

Recommendation

We recommend returning an error if the contained `msg` is not a `Claim` message.

Status: Acknowledged

9. Default state assessment of protocols added in the future may be subject to manipulation

Severity: Minor

The audited smart contract only supports protection against drops in the Anchor exchange rate. That exchange rate can only be manipulated by an attacker that is a validator and deliberately gets slashed. Future extensions of the protocol might support insurance of tokens from other protocols though, with default conditions that may be subject to manipulation. In the past, many exploits of protocols were based on manipulation of spot prices, which is a concern for the Risk Harbor protocol.

Even though this issue does not affect the current iteration of Risk Harbor, we still classify it as minor since it poses a potential risk of a future iteration.

Recommendation

We recommend refraining from using spot prices and other inputs that can be manipulated by an attacker to determine the default state of a pool. Time-weighted average prices (TWAPS)

are emerging as the standard way to impede such manipulation attempts, but they can also not offer full protection.

Status: Acknowledged

10. Hard-coded Anchor market address decreases flexibility

Severity: Informational

The calculation of the `redemption_ratio` in `src/contract.rs:285` uses a hard-coded (currently placeholder) address for the Anchor market contract. This decreases the flexibility of the contract.

Recommendation

We recommend adding the Anchor market address to the config rather than hard-coding the value.

Status: Resolved

11. Selective usage of CW20 features may lead to wrong user expectations

Severity: Informational

The contract selectively imports and uses code from the CW20 reference implementation. At the same time, the contract does not expose all of the CW20 execution and query message entry points. Depending on the frontend users are using, bought coverage might show up as a CW20 token to the user, but functionality such as transferring, sending, or burning those tokens is not available. Moreover, this selective usage of CW20 code increases the complexity of the contract.

Recommendation

We recommend either exposing all CW20 entry points or removing the dependency on CW20.

Status: Resolved

12. Lack of validation of pool name length may have adverse consequences

Severity: Informational

The contract stores the token info using the CW20 base format in `src/contract.rs:38-49`. The `CW20_base` crate performs a validation check to ensure that the token name adheres to the expected format and returns an error if this condition is not met.

Using too long names might have adverse consequences, for example for user interfaces that expect limited token names.

Recommendation

We recommend validating that the `msg.pool_name` contains the appropriate characters before storing the token info.

Status: Resolved

13. Unused validation can be removed to increase efficiency

Severity: Informational

The contract performs a validation to ensure that `underwriting_token` is a native token in `src/contract.rs:58`. This condition will never return an error because in `src/contract.rs:51-56` `underwriting_token` is defined as an `AssetInfo::NativeToken`.

Recommendation

We recommend removing the condition in `src/contract.rs:58-60`.

Status: Resolved

14. Inefficient loading of state

Severity: Informational

The contract loads `STATE` on every invocation of the `check_if_at_status` function in `src/contract.rs:505`, while `STATE` has already been loaded in the calling context. It would be more efficient to pass `STATE` as an argument.

Recommendation

We recommend passing `STATE` as an argument and removing `src/contract.rs:505`.

Status: Resolved

15. Error messages could include more details

Severity: Informational

Multiple error messages are lacking details that could improve usability:

- The `Insufficient shares` error in `src/error.rs:19` would be more useful to the end-user if it provided the `available` and `requested` values in the error message.
- The `Invalid pool status` error in `src/error.rs:31` would be more useful to the end-user if it provided the `expected` and `actual` values in the error message.
- The `Not enough capacity` error in `src/error.rs:37` would be more useful to the end-user if it provided the `available_capacity` value in the error message.
- The `Not enough policy tokens for claim amount` error in `src/error.rs:43` would be more useful to the end-user if it provided the `user_balance` and `claim_amount` values in the error message.

Recommendation

We recommend adding the values mentioned above to their corresponding error messages to improve overall usability.

Status: Resolved