Audit Report

Alice Protocol Smart Contracts

December 7, 2021

Version 1.0

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 Alice Software, Inc to perform a security audit of the Alice Protocol 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 behaviour.
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/alice-ltd/smart-contracts

Commit hash: `42515a3579cde770c0216f76967dd93836bf3e3a`

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 implements the smart contracts of Alice, a non-custodial wallet featuring integration with the Anchor money market and offering meta-transactions.

The functionality is split into an extended CW20 contract, AliceUST, and an Overseer that manages contract upgrades.

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**. Informational notes do not have a status, since we consider them optional recommendations.

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	Meta transactions can be replayed across contracts and networks	Critical	Resolved
2	Execute deposit stable function uses relayer's funds instead of user's funds	Critical	Resolved
3	Anchor exchange rate used might be out of date due to missing block height argument	Major	Resolved
4	Missing validation during instantiate phase allows `fee_ratio` to be over 100%	Minor	Resolved
5	Sent tokens other than in stable denom are lost	Minor	Resolved
6	Querying registered contracts in `Overseer` is unbounded	Minor	Resolved
7	`execute_redeem_stable` returned amount might be different from actual amount	Minor	Resolved
8	aTerra exchange rate is queried twice during `execute_redeem_stable`	Informational	Resolved

Code Quality Criteria

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium-High	-
Level of Documentation	Medium-High	-
Test Coverage	Medium-High	-

Detailed Findings

1. Meta transactions can be replayed across contracts and networks

Severity: Critical

Currently, the implementation of meta transactions does not verify the contract and network of the transaction. As a result of this, it is possible to replay a meta transaction in different contracts such as Alice Terra tokens for UST and EUT and networks such as testnet or mainnet. That could lead to double-spending.

Recommendation

We recommend adding a contract and a network field to the `MetaTx` struct to make sure the meta transaction cannot be replayed across contracts and blockchain networks.

Status: Resolved

2. Execute deposit stable function uses relayer's funds instead of user's funds

Severity: Critical

When a user attempts to deposit funds into Anchor protocol, the request originates from `execute_relay` into `execute_deposit_stable` due to meta transactions. In `relay.rs:63-65`, the relayer attempts to execute the request as the user via mutating `info.sender` which then executes the message in line `81`.

When the operation continues to `execute_deposit_stable`, the request will see `info.sender` as user while `info.funds` are the relayer's funds. Since `execute_deposit_stable` determines the fund amount via checking `info.funds` (`execute.rs:94-99`), this would cause the relayer's funds to be deposited into Anchor instead of the user's funds.

Recommendation

We recommend sending the funds from the user to the relayer in `execute_relay`.

Status: Resolved

3. Anchor exchange rate used might be out of date due to missing block height argument

Severity: Major

The Anchor exchange rate is queried from the Anchor contracts, through the utility function `compute_exchange_rate` defined in `contracts/alice_terra_token/src/utils.rs`. However, no block height argument is supplied. Without the block height argument, Anchor returns a raw exchange rate from stored values without accruing interest since the last update, which means an outdated exchange rate is returned. Using this value could lead to slightly incorrect calculations.

Recommendation

We recommend passing the current block height as an argument in `contracts/alice_terra_token/src/utils.rs:28`:

```
let aterra_exchange_rate = query_aterra_exchange_rate(deps, block_height, block_height)?;
```

Status: Resolved

4. Missing validation during instantiate phase allows `fee_ratio` to be over 100%

Severity: Minor

`fee_ratio` is the percentage of fees Alice protocol would receive from user yields. During the instantiation phase (`alice_terra_token/src/contract.rs:50`) and migration phase (`contract.rs:141`), there is no validation to ensure the ratio should be 0-100%.

Recommendation

Consider adding validation checks to make sure the `fee_ratio` is less than 100%.

Status: Resolved

5. Sent tokens other than in stable denom are lost

Severity: Minor

During `execute_deposit_stable` and `execute_deposit_stable_authorized`, the user's `uusd` funds are processed and deposited to Anchor protocol. If the user deposited additional tokens (other than `uusd`), the funds would get "stuck" in the contract after the operation succeeds.

Recommendation

Consider reverting an Error if the user deposited more than one type of funds (eg. checking `info.funds.len`).

Status: Resolved

6. Querying registered contracts in `Overseer` is unbounded

Severity: Minor

The `RegisteredContracts` query message is unbounded in `alice_overseer/src/contract.rs`, which could cause calling transactions to run out of gas. Even if it is unlikely that there would be enough registered contracts to make the query out of gas, since registered contracts cannot be removed an out-of-gas situation could be irreversible.

Recommendation

We recommend adding `start_after`, `limit` and `order_by` parameters to the query.

Status: Resolved

7. `execute_redeem_stable` returned amount might be different from actual amount

Severity: Minor

The `execute_redeem_stable` function requests a stable denom amount and then returns a calculated amount to the user. The returned value may be different from the actual returned amount due to possible slashing penalties, fees, taxes etc.

Recommendation

We recommend storing the contract's balance before executing `execute_redeem_stable`, and re-query the contract's balance in a sub-message reply to retrieve the actual amount for the user to redeem.

Status: Resolved

8. aTerra exchange rate is queried twice during `execute_redeem_stable`

Severity: Informational

The functions `compute_and_update_exchange_rate` and `query_aterra_exchange_rate` both query aTerra exchange rate individually. During `execute_redeem_stable`, both functions are used to retrieve the aTerra exchange rate in `alice_terra_token/src/execute.rs:163` and `165` which is inefficient.

Recommendation

We recommend refactoring the code to only retrieve the aTerra exchange rate once.

Status: Resolved