CERTIK

CertiK Audit Report for Akropolis

Contents

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and Akropolis (the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites, static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 6.2B in assets. For more information: https://certik.org.

Executive Summary

This report has been prepared for Akropolis to discover issues and vulnerabilities in the source code of their AkropolisOS project. A comprehensive examination has been performed, utilizing Dynamic Analysis, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Scope

The files audited, and the files that all line references refer to, were all the Solidity files in:
https://github.com/akropolisio/akropolisOS/tree/497cd29bd7e2f005bd8381fa5c7cf0ab79dac95c

The post-changes, final files, are located at:

https://github.com/akropolisio/akropolisOS/tree/056ba5e5bdf96a7fe8de0b8876b5cad7790c9459

Testing Summary

SECURITY LEVEL

VULNERABILITY OVERVIEW

Smart Contract Audit

This report has been prepared as a product of the Smart Contract Audit request by Akropolis.

This audit was conducted to discover issues and vulnerabilities in the source code of Akropolis Smart Contracts.

TYPE	Smart Contracts
SOURCE CODE	https://github.com/akropolisio/akropolisOS/
PLATFORM	EVM
LANGUAGE	Solidity
REQUEST DATE	May 7, 2020
REVISION DATE	June 15, 2020
METHODS	Dynamic Analysis, Static Analysis, and Manual Review, a comprehensive examination has been performed.

Dynamic & Static Analysis Review Notes

The CertiK team applied state-of-the-art dynamic and static analysis tools on the source code of the smart contracts to pinpoint any vulnerabilities and issues that can be detected using analytical techniques applied on the source code directly.

The tools applied for the codebase were unable to identify any potential issues or vulnerabilities, inferring that the codebase is secure against common attack vectors and vulnerabilities.

An additional thing to note is that the team already enforces a strict Solidity linter and specifically solhint which is a capable tool and points to the fact that the team that has undertaken the development of the smart contracts is capable of maintaining a secure codebase and practically applies the best coding practices.

Manual Review Notes

Introduction

The CertiK team was invited by the Akropolis team to audit the design and implementations of the AkropolisOS smart contracts.

The goal of this audit was to review the source code of the protocol implementation for its business model, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

Methodology

The work was conducted and analyzed under different perspectives and with different tools such as static analysis tools as well as manual reviews by smart contract experts. In general, we make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

Summary

The manual line-by-line review of the source code revealed **no vulnerabilities or issues with the codebase** itself, confirming our initial interactions with Akropolis that their team enforces **high security standards** in developing their codebase. **The team has demonstrated consistency and reliability with regards to their code ethics and practices**.

Notes

Items are labeled CRITICAL, MAJOR, MINOR, INFO, DISCUSSION (in decreasing significance).

Findings

AddressList

1. [INFO] Swap require statement

The first require statement in `swap` (L166) should be `_data.isContain[_a] && _data.isContain[_b]`.

Fixed: Addressed in PR39

2. [MINOR] Swap function edge case

The `swap` function fails if `_a` immediately follows `_b`.

Fixed: Addressed in PR39

Pool

1. [MINOR] Public setMetadata function

`setMetadata` function should be `onlyOwner`.

Fixed: Addressed in PR39

FundsModule

1. [MINOR] Double use of pToken balances

We understand the intention to perform state changes in the various `withdraw` and `deposit` methods *before* performing the interaction with `PToken`. However, we feel this causes a vulnerability. Imagine a typical flow:

- A user deposits LTokens using the `deposit` functionality. They are minted PTokens.
- They create a debt proposal, and FundModule's `depositPTokens` function is called. The crucial part is it *first* increases the balances (L71) and then calls `transferFrom`.
- This calls the internal `_transfer`, which first updates the user balances due to distributions, and only then due to the transfer.
- While executing the first update, `distributionBalanceOf` takes into account both the user's currently held coins (which still hasn't been decreased), and also their balances in fundsModule (which has already been increased).
- In effect they are able to get up to double their token share, breaking the double-spend invariant.
- The solution is to have all distribution claims happen before any balances are changed. The easiest way would be to change pToken balances *after* the pToken interaction in FundsModule. (Since there already is a distribution claim as part of every pToken transfer) No re-entrancy is possible because pToken is a pre-approved token.

Fixed: Addressed in PR40

2. [INFO] Allowance not necessary

We believe the comment on L106, as well as lines 100, 116 and 136 in README.md is incorrect. We believe setting an allowance in pToken for fundsModule is not necessary. This is because both the `burnFrom` function and the `transferFrom` functions in pToken allow the fundsModule to make these calls without requiring allowance (a sufficient balance is still required). We don't think this is a vulnerability. The function is used:

- `withdraw` - the amount burned is strictly equal to the amount submitted by the user, so this is is expected behavior
- `withdrawForRepay` - this function is used:
  - `repayPTK` - the amount is strictly equal to amount submitted by the user, except when that overflows the amount necessary to repay outstanding interest and debt. So this is expected behavior.
  - `repayAllInterest` - this is notably called also in `withdraw` (in liquidity module). The user won't be able to claim their LTokens until they repay all their interest (expected). They can also repay loans partially using `repayPTK`.
  - `withdrawDebtDefaultPayment` - the `pAmount` here is at most that of the `pToken` balances of the user. However, it doesn't check the `pToken` balances inside `LoanProposalsModule`, so could one *hide* their balances in this way?

Fixed: No

LoanModule

1. [MINOR] Change payment date function

The function `__changeDebtLastPaymentDate` allows the owner to arbitrarily increase the interest, and hence the overall payment burder, for any user (as well as decreasing it to some degree). There is a natspec comment that this function should be deleted before release on mainnet.

Fixed: Addressed in PR39

2. [INFO] Incorrect line order

The `withdrawDebtDefaultPayment_calculatePInterest` function has lines 582 and 585 in the wrong order, we believe. `pInterest` is initialized to 0 at the enter of line 582.

Fixed: Addressed in PR39