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").

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's mission of every audit is to apply different approaches and detection methods, ranging from manual, static, and dynamic analysis, to ensure that the project is checked against known attacks and potential vulnerabilities. CertiK leverages a team of seasoned engineers and security auditors to apply testing methodologies and verifications on the project, in turn creating a more secure and robust software system.

CertiK has served more than 100 clients with high quality auditing and consulting services, ranging from stablecoins such as Binance's BGBP and Paxos Gold to decentralized oracles such as Band Protocol and Tellor.

Executive Summary

Akropolis is a project that enables decentralized and autonomous communities. The original protocol (which we have previously audited) is responsible for two things:

- Minting pool tokens by depositing liquid tokens
- Allowing people to pledge pool tokens to collateralize loans, and others to take out loans and pay them back

In this addition to the base protocol, the team has added the feature of reinvesting stored liquid tokens in various defi projects. The expectation is that they will accrue interest, which will be redistributed to the pool token holders. 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.

After conducting our initial review, we identified one a Major and a few Minor holes. All five points have been fixed by the Client, and are no longer present in the post-remediation commit referenced below.

Testing Summary

SECURITY LEVEL

TBD

TYPE	Smart Contract Audit
SOURCE CODE	https://github.com/akropolisio/akropolisOS
PLATFORM	Ethereum Virtual Machine
LANGUAGE	Solidity
REQUEST DATE	July 2, 2020
DELIVERY DATE	July 24, 2020
METHODS	A comprehensive examination has been performed using Whitebox Analysis. In detail, Dynamic Analysis, Static Analysis, and Manual Review were utilized.

Review Notes

Introduction

AkropolisOS is a protocol to enable autonomous and decentralized communities. We previously (June 2020) conducted a security audit of the entire smart contract codebase of AkropolisOS. This audit concerned itself with two pull requests to the base protocol. The audit was conducted from July 17, 2020 to July 24, 2020. The goal of this audit was to ensure the new code does not have bugs, and that changes to the original code do not introduce new security vulnerabilities.

Scope of Work

The exact scope was the following PR's:

https://github.com/akropolisio/akropolisOS/tree/feature/defi-ray, in particular commit:

80693e2d9563e163939363fac2cfe6a60e3642c0, and

https://github.com/akropolisio/akropolisOS/tree/feature/multi-coin-withdraw, in particular commit:

Da2d3b6be8bdd7a2ee352cbebda97b9ada126228

Post-remediation

After the Client's changes, the commit is:

343398c46f849471ea0e681c42d02e1638491989

Analysis

What follows is a selection of our high-level notes & analysis. The notes are organized by contract, by function. For the client that serves as an outline for a specification, and also as a reference for future work.

DefiModuleBase (abstract contract)

- Initialize
    - Adds _pool
    - Sets msg sender as DefiOperator
    - _createInitialDistribution()
        - We verified this is the only place where this function is called
- _createInitialDistribution()
    - Sets distributions[0] = Distribution (0)
- handleDeposit
    - Increments depositsSinceLastDistr
    - Calls abstract function handleDepositInternal
- Withdraw
    - Analogous to above, except with additional functionality:
        - Token can be set to 0, in which case the first token with sufficient liquidity is selected.
        - If none is found, the first token is returned. The execution will depend on withdrawInternal, we can imagine both a revert and withdrawing only partially. We'll go back to this later.
- withdrawAll
    - Low complexity, withdraws full balance from every token to funds module

    - Calls withdrawInternal
- calculateDistrAmount
    - Performs current_balance - (previous_balance + new_deposits - new_withdrawals) to calculate diff
    - Underflow is handled (returns 0)
- totalSupplyOfPTK -> pToken().distributionTotalSupply()
    - totalSupply - balances_lock_as_collateral
- _createDistribution
    - If no tokens has distribution, returns
    - Otherwise push Distribution(totalPTK, distribution_amounts, current_balances)
    - Null new_deposits and new_withdrawals
    - Set nextDistrTimestamp as the timestamp of the next day
- _calculateDistributEDAmount
    - For each distribution, for each token, add to accumulator my share of distributed_amount
- _updateUserBal
    - Every user has an investment_balance
    - This function gets all distributed amounts and adds that to availableBalances
    - nextDistr set to toDistr
- withdrawInterest
    - Calls withdrawInternal for each available_balance
    - Creates distribution if now is sufficient & fully updates user's balances, so always operates on latest state
- updatePTKBal
    - Hook for PToken (PToken.userBalanceChanged just calls this one)
    - Ensures that all other functions always operate on latest PTK balances
- availableInterest

    - Doesn't take into account just the current investment_balance, but also unclaimed tokens

APYBalanceDefiModule

- Initialize overrides super
    - Same as super
- normalizeAmount -> funds.normalizeLTokenValue
- denormalizeAmount -> funds.denormalizeLTokenValue
- tokenIndex
    - Loops through _registeredTokens
- isTokenRegistered
    - Analogous to above
- converterToTokenProtocolIdx
    - Loops through registeredProtocols
    - Returns index for which pr.canSwapToken & pr.supportedTokensCount > 1
- worstAPYProtocolIdx
    - Loops through all registeredProtocols, finds idx of lowest last APY
- _createDistribution overrides super
    - For each registered token, for each registered protocol:
    - Require now > periodStartTimestamp
    - Profit calculated analogously to distributionAmount in super
    - previousPeriodAPY is (profit / currentBalance), scaled to 1 year and up by 10^18, or 0 if profit = 0
    - Calculates deposits and withdrawals since last distribution for each token by looping over all registered protocols and adding up deposits since period start. This is the only place in the contract where these are updated
    - Calls super

    - Essentially, apart from mutating ProtocolInfo, it sets deposits and withdrawals SinceLastDistr to the sum across all protocols
    - This is the only fn together with updateSaldo (and registerProtocol) that mutates ProtocolInformation
- updateSaldo
    - Accepts a particular protocol and two arrays of balances_before and balances_after
    - For each supported token, updates deposits and withdrawals since period start
    - This is the only fn together with _createDistribution (and registerProtocol) that mutates ProtocolInformation
- splitByProtocolsForDeposit
    - There are two possible behaviors
        - Deposit all to one protocol
        - Deposit equally to all protocols by looping and adding amount / protocols.length
    - There are many cases
        - If all protocols have last apy 0, it distributes equally
        - If one protocol has last apy & balance 0, it distributes equally
        - If at least one protocol has a positive apy & each protocol has either apy or balance non-zero (basically the logical negation of the disjunction of the above cases), then it distributes it to that which has highest apy
- handleDepositInternal
    - splitByProtocolsForDeposit(token, amount)
    - For each protocol, it deposits the amount from above line
    - updateSaldo with balance diff
- handleDeposit overrides super -> handeDepositInternal

    - No longer increments depositsSinceLastDistr as this is now incremented (only) in _createDistribution
- withdrawInternalFromAllProtocols
    - We loop through all protocols, withdrawing all amounts until parameterized amount
    - If there is still amount remaining, we find the first protocol which can swap parametrized token and supports > 1 tokens
    - We withdraw liquid tokens from all protocols except the “converter” above, and deposit them into converter
    - We withdraw remaining amount from converter
- withdrawInternal
    - Withdraw as much as possible from worst-apy protocol
    - withdrawInternalFromAllProtocols for rest
    - updateSaldo with balances diffs
- withdrawFromAllProtocolsByTokens
    - This is one of the most complex functions in the entire system. The idea is to take as much as possible from “inflexible” protocols - those where withdrawals lead to receival of multiple tokens
    - The rest we withdraw from flexible protocols - effectively those that are one-token protocols
    - More specifically:
        - For each token, we find the protocol with the highest balance and the balance, and save that in flexProtocols and flexBalances, respectively
        - For each inflexible protocol, we get the balances of all tokens and save that in protocolBalance
        - For each inflexible protocol, we find “maxAmountToBalance” - a ratio (in 10^18 format) of how much of the balances to use.

- withdrawFromAllProtocols
    - First calculates total normalized balances of all registered tokens
    - The idea is: for each token find the share of that token's total balances, and withdraw that proportion of the amount requested to be withdrawn
- withdrawInternalMultiToken
    - Allows to withdraw a sum without specifying which tokens
    - Calls updateSaldo
- Withdraw
    - Analogous to handleDeposit()
    - One additional feature:
        - If token = 0, calls withdrawInternalMultiToken()
- registerProtocol
    - Check that protocol is not registered
    - For each supported token, make sure it is registered and create ProtocolInfo

CurveFiYProtocol

- Initialize
    - Adds _pool
    - Adds msg sender as DefiOperator
- Deposit
    - Deposits balances of all registered tokens (of this) to curveDeposit
    - Allows to specify minimum amount deposited (i.e. - minimum balance of this) of a particular token
- withdrawAll
    - Limit-order-style withdraw of all curve tokens from depositContract
    - Transfers full balances of all registered tokens to funds
- balanceOfAll

    - For each of the curve's coins we get our share
    - Then find value in underlying coins
- normalizeAmount -> funds.normalizeLTokenValue
- denormalizeAmount -> funds.normalizeLTokenValue
- getTokenIndex
    - Loops through registered tokens
- normalizedBalance
    - Calls balanceOfAll, normalizeAmount all, adds all up
- balanceOf
    - Same as balanceOfAll but for one token
- Withdraw
    - Calls CurveFiDeposit.remove_liquidity_imbalance
    - Then for each registered token sends that amount to param. beneficiary
- _unregisterToken
    - withdraw full balance to funds module
- _registerToken
    - If have balance, then deposit all
- setCurveFi
    - Make sure _registeredTokens is empty
    - Set deposit, swap and token contract adds
    - Register each underlying coin
- supportedTokens -> _registerTokens
- canSwapToToken
    - True if token in _registeredTokens

RayProtocol

- Initialize

    - Set _pool
    - Set msg sender as DefiOperator
    - Set baseToken and portfolioId
- rayStorage -> MODULE_RAY (external module)
- rayNAVCalculator -> rayStorage.getContractAddress(NAV_CALCULATOR_CONTRACT)
- rayPortfolioMan -> rayStorage.getContractAddress(PORTFOLIO_MANAGER_CONTRACT)
- normalizeAmount -> funds.normalizeLTokenValue
- denormalizeAmount -> funds.denormalizeLTokenValue
- supportedTokens
    - Return [baseToken]
- canSwapToToken
    - Only to baseToken
- balanceOf
    - rayNavCalculator.getTokenValue(portfolioId, rayTokenId)
- normalizedBalance
    - balanceOf(baseToken) ~> normalizeAmount
- balanceOfAll
    - [balanceOf(baseToken)]
- onERC721Received
    - Semantics is that contract can receive ERC721 tokens only from rayStorage.getContractAddress(RAY_TOKEN_CONTRACT) (will revert for all other tokens)
- Deposit
    - Correct handling of Ray-specific logic
- Withdraw
    - Transfer arbitrary amount to arbitrary beneficiary

Audit Findings

Exhibit 1

TITLE	TYPE	SEVERITY	LOCATION
lProposals can be > lBalance	Revert	Major	Cross-module

Description:

We think we found a vulnerability in the core protocol. We think the system assumes that lBalance will always >= lProposals. But since no such check is made in the proposal mechanism, we think lProposals could be > lBalance. This will mean that whenever lBalance.sub(lProposals) is called (four times in BaseFundsModule), it could revert.

Recommendation:

It is recommended that when new debt proposals are created, that it is checked that lProposals <= lBalance.

Client's response:

Fixed.

Check:

This issue has been fixed.

Exhibit 2

TITLE	TYPE	SEVERITY	LOCATION
MAX_UINT Approvals	Informational	Informational	Cross-module

Description:

There are multiple places where MAX_UINT is approved. While it is a trade-off between user experience and is up to the developers, we have to acknowledge that it does increase the potential consequences of a possible vulnerability.

Client's response:

In order to save gas, we decided to leave the preliminary approval, but added the ability to change (recall) it, if a vulnerability is detected. The operation of changing the approval is carried out by addresses with the DefiOperator role.

Check:

A new function resetProtocolApproval has been added that allows to set a new approval for arbitrary token and arbitrary protocol.

Exhibit 3

TITLE	TYPE	SEVERITY	LOCATION
Incorrect line order	Execution logic	Minor	APYBalanceDefiModule

Description:

In APYBalanceDefiModule, in _createDistribution (lines 347-351). totalDeposits and totalWithdraws are an accumulator, as we loop through all protocols to determine deposits and withdraws since last distribution. The issue is that we first reset those state values (347 & 348), and then add to the accumulator vars (350, 351). Consequently, the accumulators will be 0.

Recommendation:

Add to totalDeposits and totalWithdraws before resetting the state.

Client's response:

Fixed.

Check:

This issue has been fixed.

Exhibit 4

TITLE	TYPE	SEVERITY	LOCATION
Ternary operator	Syntax	Minor	APYBalancedDefiModule

Description:

The ternary operator is used incorrectly on lines 112, 142 and 161. In particular, in all three cases, the effective execution is max function, when in fact we want the min of the two values.

Recommendation:

Take the negation of the predicate, or flip the return values.

Client's response:

Fixed.

Check:

This issue has been fixed.

Exhibit 5

TITLE	TYPE	SEVERITY	LOCATION
Visibility Specifiers Missing	Syntax	Informational	DefiModuleBase: L33, L34

Description:

The contract variables “depositsSinceLastDistribution” and “withdrawalsSinceLastDistribution” do not have a visibility specifier set.

Recommendation:

It is a good security practice to always explicitly set function and contract variable visibility specifiers to prevent other contracts from being able to read from and execute arbitrary data / functions on a particular contract and to increase the legibility of the codebase.

We advise that the visibility specifiers for those two variables are explicitly set to “internal” or “private” depending on the expected inheritance of the contract’s code.

Client’s response:

Fixed.

Check:

This issue has been fixed.