SMART CONTRACT AUDIT PUBLIC REPORT

Audit of Folks Finance Defi Protocol - Phase 1
VPQ-20210652
BlockchainItalia.io - Folks Finance
24th February 2022

VANTAGEPOINT
Security at the Speed of Development

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

OVERVIEW

Vantage Point Security Pte Ltd was engaged by BlockchainItalia.io to conduct an Algorand smart contract security review of Folks.Finance Defi Protocol to identify security vulnerabilities, weaknesses and any instances of non-compliance to best practices within the smart contract. Testing commenced on the 17th January 2022 and was completed on the 16th February 2022.

Algorand smart contract security review was conducted based on the following materials provided.

- Documents
  - Folks.Finance Economic Model
  - Folks.Finance Protocol Overview
  - Folks.Finance Technical Design
- PyTeal Code
  - Private Repo
    - https://github.com/blockchain-italia/ff-vp-contracts
      Commit ID 73343e868ad433e50b1d6aaaf39625bbe1d495b3

Vantage Point performed this review by first understanding high-level business logic of both the Folks.Finance DeFi protocol and interaction between different smart contracts stated within the provided documents. We sought clarifications on potential issues, discrepancies, and flaws within the smart contract's logic through discussions with the Folks.Finance team.

Subsequently, an audit on the provided PyTeal code was completed to identify weaknesses, vulnerabilities, and non-compliance to Algorand best practices. Test cases included in this review have been amended in the appendix of this document.

The following issues were identified from this review.

1. **Lack of ASA Asset ID Validation during Asset Transfer Transaction**
   Due to lack of validation for Algorand Standard Asset asset identifier during asset transfer transactions identified in dispenser_approval_program.py's *on_provide_liquidity()* and *on_redeem()* functions, dispenser contracts can potentially accept an Algorand Standard Asset of a lower value if opted-in, when the user tries to redeem their fAssets such as fAlgo and fUSDC or deposit ASA such as goBTC or USDC when the user tries to provide liquidity.
2. **Lack of Sender Address Validation for Privileged Transactions**
   Privileged application calls which can only be called by privileged addresses such as *Admin Address* and *Platform Owner Address* should enforce checks against *Txn.Sender()* and the privileged addresses to ensure such application calls are approved only for the intended addresses. Affected functions do not sufficiently check the *Txn.Sender()* and thus could potentially allow any addresses to make privileged application call.
3. **Lack of hasValue() Check for localGetEx/globalGetEx External States**
   Affected functions make use of referenced local and global states from other external applications without validation to check if such state exists for the supplied *AppId* and *key*. Lack of validity checks could cause the smart contract to perform validation against wrong values as

when the state does not exist for supplied *AppId* and *Key*, referenced value would be equal to 0.

The outcome of this Algorand Smart Contract Security Review engagement is provided as a detailed technical report that provides the Smart Contract owners a full description of the vulnerabilities identified, the associated risk rating for each vulnerability, and detailed recommendations that will resolve the identified technical issue.

VULNERABILITY OVERVIEW

Severity	Count	Open	Closed
Critical	**0**	**0**	**0**
High	**0**	**0**	**0**
Medium	**2**	**0**	**2**
Low	**1**	**0**	**1**
Observational	**2**	**0**	**2**
Summary	**5**	**0**	**5**

Vulnerability Risk Score

All vulnerabilities found by Vantage Point will receive and individual risk rating based on the following four categories.

CRITICAL COMPONENT RISK SCORE

Critical severity findings relate to an issue, which requires immediate attention and should be given the highest priority by the business as it will critically impact business interest critically.

HIGH COMPONENT RISK SCORE

HIGH severity findings relate to an issue, which requires immediate attention and should be given the highest priority by the business.

MEDIUM COMPONENT RISK SCORE

A MEDIUM severity finding relates to an issue, which has the potential to present a serious risk to the business.

LOW COMPONENT RISK SCORE

LOW severity findings contradict security best practice and have minimal impact on the project or business.

OBSERVATIONAL

Observational findings relate primarily to non-compliance issues, security best practices or are considered an additional security feature that would increase the security stance of the environment which could be considered in the future versions of smart contract.

2. Project Details

Scope

Contact Name	Gidon Katten
Contact Email	gidonkatten@blockchainitalia.io
Application Name	Folks.Finance Defi Protocol
Testing Period	17th January 2022 ~ 15th February 2022
GIT Commit ID	73343e868ad433e50b1d6aaaf39625bbe1d495b3
Items Completed	Vantage Point completed the Smart Contract Security Review for below files • token_pair_approval_program.py • staking_clear_program.py • staking_approval_program.py • oracle_approval_program.py • oracle_adapter_approval_program.py • liquidity_approval_program.py • dispenser_approval_program.py • dispenser_approval_program.py • clear_program.py • oracle_adapter/shared.py • oracle_adapter/state.py • dispenser/shared.py • dispenser/state.py • common/formulae.py • common/inner_txn.py • common/math.py • common/transactions.py

Component	Review Type	Status
Algorand Smart Contract	Smart Contract Security Review	Completed
Algorand Smart Contract	Smart Contract Security Review Retest	Completed

VERSION HISTORY

Date	Version	Release Name
16th February 2022	v0.1	Draft
17th February 2022	v0.2	QA Release
21st February 2022	v0.3	Retest Update for 1.4 and 1.5
23rd February 2022	v0.5	Retest Update for 1.3
24th February 2022	v1.0	Final

3. Risk Assessment

This chapter contains an overview of the vulnerabilities discovered during the project. The vulnerabilities are sorted based on the risk categories of CRITICAL, HIGH, MEDIUM and LOW. The category OBSERVATIONAL refers to vulnerabilities that have no risk score and therefore have no immediate impact on the system.

Overview of Components and Their Vulnerabilities

1. Audit of Folks Finance Defi Protocol - Phase 1		MEDIUM RISK	⚠
1.1. Lack of ASA Asset ID Validation during Asset Transfer Transaction	Closed	MEDIUM RISK	⚠
1.2. Lack of Sender Address Validation for Privileged Transactions	Closed	MEDIUM RISK	⚠
1.3. Lack of hasValue() Check for localGetEx/globalGetEx	Closed	LOW RISK	⚠
1.4. Difference in Design Specification and Actual Implementation	Closed	OBSERVATIONAL	ⓘ
1.5. Insufficient Argument Range Validation	Closed	OBSERVATIONAL	ⓘ

4. Detailed Description of Vulnerabilities

1. Audit of Folks Finance Defi Protocol - Phase 1

MEDIUM RISK

1.1. Lack of ASA Asset ID Validation during Asset Transfer Transaction

MEDIUM RISK

VULNERABILITY TRACKING

STATUS: **Closed**

BACKGROUND

During Asset Transfer Transactions where "type" header is set to "axfer", Algorand Standard Assets (ASA) are transferred from one account to another and the type of specific asset being transferred is determined by the value of "xaid", the Asset Identifier. During transactions which involve transfer of ASAs, it is crucial to validate the asset identifier of the ASA being received and sent to avoid receiving and sending the wrong type of ASA to and from the smart contract.

DESCRIPTION

Instance 1

Instance 1 was identified independently from a separate audit and the commit made to address this instance of issue was verified in the current report.

Affected Code

- assets/dispenser_approval_program.py
    - on_redeem()
    - Line 499-506

It was noted that when a user tries to redeem their fAsset (Example, fAlgo, fgoBTC and etc) by sending the fAsset to dispenser smart contract, the dispenser smart contract does not validate the ID of the asset being transferred.

The `return_algo_or_asset` function is used to create an inner transaction which transfers either Algo or ASA based on the dispenser's `is_algo` and `asset_id` global state. The `Gtxn[1].xfer_asset()` is not being validated against the dispenser's `f_asset_id`. If the dispenser smart contract is opted-in to

ASA of a lower value and can receive the ASA of a lower value then the legitimate ASA(fAsset) expected, this could allow a malicious user to submit a transaction that redeems more valuable Assets such as Algo or ASA (non-fAsset) from the dispenser.

Instance 2

Affected File/Code

- dispenser_approval_program.py
    - Line 1515-1522 - deposit_asset_tx

It was noted that within function *on_provide_liquidity()*, no validation was observed for the ASA ID being deposited as part of *Provide Liquidity* action.

Although successful exploitation would require the dispenser to be opted-in to specific ASA being transferred, lack of ASA ID check potentially allows incorrect ASA to be considered as a deposit during *provide liquidity* action.

RECOMMENDATION

Instance 1

Validate `Gtxn[1].xfer_asset()` against the expected value, such as `f_asset_id`.

Instance 2

Validate `Gtxn[4].xfer_asset()` against the expected value, such as `asset_id`.

REGRESSION TESTING COMMENTS

11th February 2021 - This issue is closed.

Instance 1 – Closed

It was noted that additional commit 5b0f7ca9f26ee9293c39c6fcc464a91915a6a05f adds a validation for `Gtxn[1].xfer_asset()` against the dispenser's fAsset ID global state, `f_asset_id` per recommendation.

Instance 2 – Closed

It was noted that additional commit 3976bfb64b2a4a8e012e9db45564d958cb1e8e03 adds a validation for Gtxn[4].xfer_asset() against asset_id per recommendation.

REFERENCES

Algorand Developer Docs – Asset Transfer Transaction:
https://developer.algorand.org/docs/get-details/asa/#receiving-an-asset

Algorand Developer Docs – Transaction Reference – Asset Transfer Transaction:
https://developer.algorand.org/docs/get-details/transactions/transactions/#closeassetto

1.2. Lack of Sender Address Validation for Privileged Transactions

MEDIUM RISK

VULNERABILITY TRACKING

STATUS: **Closed**

BACKGROUND

Algorand Smart Contracts verify the sender's address of transactions if transactions are privileged operations which are only allowed from privileged addresses, such as platform owners or designated administrator addresses. Failing to verify the sender of the transaction for privileged transactions may allow unprivileged accounts to submit privileged transactions which could result in loss of assets, integrity or availability of the smart contract.

DESCRIPTION

Instance 1

Affected Code

- dispenser_approval_program.py
    - on_add_liquidity_need() Line 1425-1440
        - Instance 1-1 Gtxn[2].sender()
        - Instance 1-2 Gtxn[1].sender()

It was noted that when a *liquidity need* is being added, insufficient validation for transaction sender was observed in the affected instances.

Instance 1-1

Based on the `on_add_liquidity_need()` function which is supposed to validate Gtxn[2], it was observed that there is no validation for the sender address, even though Gtxn[2] is a privileged transaction which can only be sent from `platformOwnerAddr`.

Instance 1-2

Based on the *Add Liquidity Need* operation, `Gtxn[1].Sender()` must be equal to `Gtxn[2].accounts[1]` but such validation was not observed in both `on_add_liquidity_need()` and `on_add_liquidity_need_opt_in()` functions.

RECOMMENDATION

Instance 1-1

It is recommended to add appropriate validation for below condition.

```
Gtxn[2].sender() == platformOwnerAddr
```

Instance 1-2

It is recommended to add appropriate validation for below condition.

```
Gtxn[1].sender() == Gtxn[2].accounts[1]
```

REGRESSION TESTING COMMENT

10th February 2021 - This issue is closed.

Instance 1-1 and 1-2

Both Instance 1-1 and 1-2 have been remediated appropriately with commit id 579c3ec4591e1a7566d28cc5f10b861ef1ff9221 per recommendation.

VULNERABILITY REFERENCES

Algorand Developer Portal – Transaction Reference – Common Fields:
https://developer.algorand.org/docs/get-details/transactions/transactions/

1.3. Lack of hasValue() Check for localGetEx/globalGetEx

LOW RISK

VULNERABILITY TRACKING

STATUS: **Closed**

BACKGROUND

For Algorand smart contracts, `localGetEx` and `globalGetEx` methods are used to read global and local states of other applications. Unlike other state access methods, `App.globalGetEx` and `App.localGetEx` return a `MaybeValue`. Although `MaybeValue` cannot be used directly, `MaybeValue.hasValue()` returns `1` and `MaybeValue.value()` returns its value if the value exists. If such value does not exist, both `.hasValue()` and `.value()` return `0`. Therefore, `MaybeValue.hasValue()` should be checked before using the `MaybeValue.value()` for any further operations within the `Seq()` to avoid unintentional behaviors with `MaybeValue.value()` being equal to `0`, not because the actual value is `0` but is `0` because such key does not exist in the queried application's state, causing an unexpected behaviour for any logical operations.

DESCRIPTION

Following instances with use of `localGetEx` and `globalGetEx` without checks for `MaybeValue.hasValue()` were identified.

Instance 1 - token_pair_approval_program.py

Affected Code

- token_pair_approval_program.py
  - Instance 1-1 collateral_app_f_asset_id
    - Line 80-81
  - Instance 1-2 borrow_app_asset_id
    - Line 82-83

It was noted that the affected code uses `MaybeValue.value()` method to use the retrieved value from `localGetEx` and `globalGetEx` methods without checking if `maybeValue.hasValue()` returned a non-zero value, indicating that a value exists for `appId` and `key` supplied.

Especially for `borrow_app_asset_id`, if the asset being borrowed in this token pair is Algo, asset_id would be set to 0 as Algo is not an Algorand Standard Asset (ASA). Since `borrow_app_asset_id` can be set to 0 if the asset being borrowed is Algo, without checking if `borrow_app_asset_id.hasValue()==1`, using `collateral_app_f_asset_id.value()` could have two possible outcomes listed below.

Case 1

- `borrow_app_asset_id.value()` and `borrow_app_asset_id.hasValue()` both return `0` as no such key exists for `appId borrow_app_id`.

Case 2

- `borrow_app_asset_id.value()` returns `0` while `borrow_app_asset_id.hasValue()` returns `1` as such key exists for `appId borrow_app_id` and the value is equal to `0` as it is set to `0`, not because there is no such key.

For both cases, `borrow_asset_id == borrow_app_asset_id.value()` could be true, regardless of availability of such state for the `appId` and `key`. This unintended behaviour could allow `Assert(borrow_asset_id == borrow_app_asset_id.value())` to pass even when it is not supposed to and eventually approve the transaction. Aside from the above highlighted code, other parts of the code which uses `maybeValue.value()` without checking `maybeValue.hasValue()` were highlighted under affected code.

Instance 2 - liquidity_approval_program.py

Affected Code

- liquidity_approval_program.py
  - Instance 2 - on_creation()
    - f_asset_id Line 47-66

Identical to Instance 1, use of maybeValue.value() without validation of maybeValue.hasValue() was identified.

RECOMMENDATION

For all identified instances, it is recommended to consider verifying the value of `maybeValue.hasValue()` before using the `maybeValue.value()` for any further logical operations especially if valid range of values include 0.

REGRESSION TESTING COMMENT

16th February 2021 - This issue is closed.

Instance 1-1 & 1-2 - Closed

Additional validations per recommendation were added as validations to check if `collateral_app_f_asset_id.value()` and `borrow_app_asset_id.value()` returned a valid value, in commit 97039993440cafb32ca4cd86646e941af5ce89df. Instances 1-1 and 1-2 are closed.

Instance 2 – Closed

Commit ID 97039993440cafb32ca4cd86646e941af5ce89df added `Assert(f_asset_id.hasValue())` as a validation to check if f_asset_id.value() returned a valid value. Instance 2 is closed.

VULNERABILITY REFERENCES

PyTeal Documentation – pyteal.MaybeValue:
https://pyteal.readthedocs.io/en/v0.7.0/api.html#pyteal.MaybeValue

PyTeal Documentation – globalGetEx:
https://pyteal.readthedocs.io/en/stable/api.html#pyteal.App.globalGetEx

PyTeal Documentation – localGetEx:
https://pyteal.readthedocs.io/en/stable/api.html#pyteal.App.localGetEx

1.4. Difference in Design Specification and Actual Implementation

OBSERVATIONAL ⓘ

VULNERABILITY TRACKING

STATUS: **Closed**

BACKGROUND

Often, functional design or specification documents are drafted prior to the actual development of smart contract in order to capture necessary requirements before so that the smart contract code can accurately reflect the logic built in a form of code. Once available to larger group of audiences or public members, there is a trust on whitepapers or documentations being in sync with the actual code running on Smart Contract and this forms a basis for any logical decision making for every participants. However, if there is a difference between what was documented and what the smart contract actually does, it may create a negative impact on the reliability of the protocol, losing confidence from the community and other stakeholders in blockchain.

DESCRIPTION

Instances noted below do not pose direct threat to the core availability and integrity of the smart contract but more of a difference in implementation specifics. Regardless, it is recommended to synchronize both document and smart contract to avoid any confusion.

Instance 1

Affected File

- assets/common/math.py - line 5

According to *Economic Model* document page 6, below equation is used for converting interest rate i_{d_t} from annual percentage rate (APR) to second percentage rate a_{d_t}(Equation 10).

$$\alpha_{d_t} = \frac{i_{d_t}}{(365{,}25 * 24 * 60 * 60)}$$

However, in math.py, `SECONDS_IN_YEAR` was defined differently from the documentation above.

Code Snippet – math.py

```
SECONDS_IN_YEAR = Int(365 * 24 * 60 * 60)
```

As observed, seconds in year is calculated without a consideration for a leap year, while the documentation takes leap year into account and evenly distributes it throughout the 4 years by adding 0.25 to 365(days). Although there is no direct impact or financial loss from the use of `SECONDS_IN_YEAR` parameter, discrepancy between the document and the code may cause differences in expected interests calculated and cause unnecessary confusion for the users.

Instance 2

Affected File

- assets/oracle_approval_program.py - line 25~28

According to *Technical Design* document page 4, below paragraph describes the behavior and logic of *Centralised Oracle* smart contract, specific to the acceptable threshold range for price change.

Paragraph from Folks Finance - Technical Design - Page 4

"When the price of an asset is updated, the smart contract verifies that the change in price does not exceed a given threshold. E.g. if the price is 1000 and the threshold is 0.5% then only prices in the range [995, 1005] will be accepted."

However, within oracle_approval_program.py, the actual implementation in pyTeal uses `Lt` (Less than) operator instead of `Le` (Less than or equal to).

The `Lt` operator does not return `True` if the value of `price_change * Int(1000) / old.price_load()` is equal to `price_threshold`.

If the intended business logic is supposed to allow price updates when the price change ratio is equal to or less than, `Le` operator should be used instead of `Lt`.

RECOMMENDATION

Synchronization between the actual implementation (smart contract code) and business logic (Functional Specifications/Documentation) is important as any discrepancy may result to a difference between the expected and the actual outcome. Decisions are made based on expected outcomes and if there is such a discrepancy, it risks confidence from community participants and other stakeholders who may heavily depend on the documented logic instead of the low-level code, which may not necessarily be available publicly.

REGRESSION TESTING COMMENT

23rd February 2021 – This issue is closed

Instance 1 - Closed

Economic Model Page 6 & 16 have been updated and are now in sync with the implementation.

Instance 2 – Closed

Technical Design Page 4 has been updated and is now in sync with the implementation.

VULNERABILITY REFERENCES

PyTeal Documentation – Lt:
https://pyteal.readthedocs.io/en/latest/api.html?highlight=Lt#pyteal.Lt

1.5. Insufficient Argument Range Validation

OBSERVATIONAL

VULNERABILITY TRACKING

STATUS: **Closed**

BACKGROUND

Application Calls to Smart Contracts often take in application arguments supplied by users and process them with smart contract's logic. As the use of user-supplied inputs within the smart contract may have an acceptable range as certain operations may return error if the value of the argument supplied is out of acceptable range. Validation for such application argument values prevent a need for interventions by administrators or privileged accounts which may disrupt normal operation of dApps and ensure availability of the Smart Contract.

DESCRIPTION

Instances noted below can affect the availability of smart contract but such issues either can be resolved through application calls from `admin_addr` or can only occur if incorrect argument values have been submitted from the privileged address and thus the risk rating of this issue is set to observational.

Instance 1

Affected File/Code

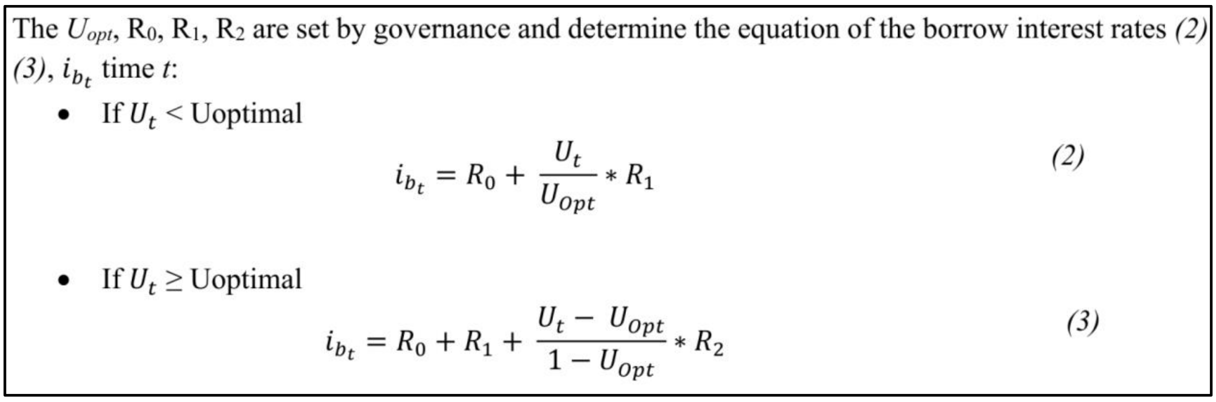
- assets/common/formulae.py
  - Line 44-56
- assets/dispenser/state.py
  - Line 33-34
- Assets/dispener_approval_program.py
  - Line 379-406

Following functions were noted within the formulae.py.

- Line 44-56 calc_borrow_interest_rate
  - Takes $R_0, R_1, R_2, U_t, U_{opt}$ and `rewards` as arguments
  - Return borrow interest rate i_{b_t}

It was noted that the function `calc_borrow_interest_rate` is used to take R_0, R_1, R_2, U_t, U_{opt} and `rewards` and return borrow interest rate ibt. It was noted that function `calc_deposit_interest_rate` (formulae.py Line 67-71) takes in U_t, i_{b_t}, rr (reserve rate) and `rewards` and return deposit interest rate, while function calc_retention_rate (formulae.py Line 30-31) takes in rf (reserve factor) and srr (staking rewards rate) and return the rr (reserve rate).

Equation for borrow interest rate i_{b_t}



The U_{opt}, R_0, R_1, R_2 are set by governance and determine the equation of the borrow interest rates *(2) (3)*, i_{b_t} time *t*:

- If U_t < Uoptimal

$$i_{b_t} = R_0 + \frac{U_t}{U_{Opt}} * R_1 \qquad (2)$$

- If $U_t \geq$ Uoptimal

$$i_{b_t} = R_0 + R_1 + \frac{U_t - U_{Opt}}{1 - U_{Opt}} * R_2 \qquad (3)$$

As noted in the code for calculating the borrow interest rate, following logical conditions are required to ensure i_{b_t} does not underflow, overflow or return an error, which eventually stops smart contract from updating interest rates, indexes and all other transactions dependent on it.

- As U_{opt} is a denominator when U_t is less than U_{opt}, U_{opt} cannot be equal to 0
- As $1 - U_{opt}$ is a denominator when U_t is greater than or equal to U_{opt}, U_{opt} cannot be equal to 0 and also cannot be greater than 1e14.

In addition, as i_{b_t} is a calculated value based on R_0, R_1, R_2, U_t and U_{opt}, the value of R_0, R_1 and R_2 could also have an upper bound set to a practical limit so that when the privileged admin account sets or updates the value of mentioned variables through an application call, updated values do not break the Smart Contract. This also prevents a need for manual interventions in case of as transactions would all fail as subroutines such as update_interest_indexes() would not succeed.

There are currently no range validation set for R_0, R_1, R_2 and U_{opt} values. This also impacts other calculation logics which use i_{b_t}, as it is not possible to identify the upper limit of i_{b_t}, other than $2^{64} - 1$.

Instance 2

Affected File

- assets/dispenser/state.py
  - Line 49-54
- assets/common/formulae.py
  - Line 30-31
  - Line 67-71

It was noted that function `calc_deposit_interest_rate` (formulae.py Line 67-71) takes in U_t, i_{b_t}, rr (retention rate) and rewards and return deposit interest rate, while function `calc_retention_rate` (formulae.py Line 30-31) takes in rf (reserve factor) and srr (staking rewards rate) and return the rr (retention rate).

Equation for RR and deposit interest rate idt

The deposit interest rate i_{d_t} at time t is directly dependent on i_{b_t}:

$$i_{d_t} = U_t * i_{b_t} * (1 - RR) \qquad (4)$$

Where *retention rate RR* represents the percentage of the revenue kept by the protocol from the interest paid by the borrowers . The retention rate is composed as follows:

$$RR = RF + SRR \qquad (5)$$

As noted in the logic for calculating the deposit interest rate, if RR is equal to 0, i_{d_t} becomes 0, which may not be a valid scenario as deposit interest rate will be equal to 0 even when Ut is non-zero, providing no incentive for the community to participate and deposit fAsset, regardless of U_t andi_{b_t}.

Furthermore, currently, there are no checks for potential overflows of values.

- rf and srr can be any value from 0 to $2^{64} - 1$ but the rr (Retention Ratio) which is sum of rf and srr also has to be equal to or below $2^{64} - 1$. There are no existing checks within the code which checks for the value of rr, rf and srr to make sure rr does not overflow.
- $\mathrm{DECIMALS} - \mathrm{RR}$ cannot be a negative number (formulae.py Line 67-71) and therefore RR should be less than or equal to 1e14 to make sure $1 - rr$ does not return an invalid value.

Instance 3

Affected File

- dispenser_approval_program.py
  - Line 213
- dispenser/state.py
  - Line 57-58

According to Folks.Finance economic model document page 6, ε or `EPS` is set to a value slightly higher than or equal to 1.

However, it was noted that there were no validations on the range of ε or `EPS`.

RECOMMENDATION

Consider lower and upper range of argument values and implement validation for specific input ranges, especially manually triggered application calls, to avoid breaking smart contract functions which may require manual intervention by a privileged account.

Instance 1-1

Validate `0 <` U_{opt} `< DECIMALS`

Instance 1-2

Consider a practical range of R_0, R_1 and R_2 based on the business logic and enforce the range so that other upper ranges can be identified for all dependent variables.

Instance 2

Validate `RF + SRR < DECIMALS` during creation or update actions.

Instance 3

Validate `EPS ≥ DECIMALS` during creation or update actions.

REGRESSION TESTING COMMENT

23rd February 2022 - This issue is closed.

Instance 1-1, Instance 2 and Instance 3 - Closed

Additional validations were added to the `verify_prarams()` subroutine with following commits.

- 579c3ec4591e1a7566d28cc5f10b861ef1ff9221
- 502c9eff3a68835ba3f71cc762400fa7a2336864

Added validations verify the following conditions

- U_{opt}
  - $0 < U_{opt} < 1e14$
- $RR + SRR \leq DECIMALS$
- $EPS \geq DECIMALS$

Instance 1-2 – Closed

Validations of range for R_0, R_1 and R_2 and `REWARDS` were added in the `verify_params()` function as part of commit c3739cc88fc31d8d9c8e16eaa60e62a4615ffed3.

VULNERABILITY REFERENCES

PyTeal – Arithmetic Operations:
https://pyteal.readthedocs.io/en/stable/arithmetic_expression.html

5. Appendix

Disclaimer

The material contained in this document is confidential and only for use by the company receiving this information from Vantage Point Security Pte. Ltd. (Vantage Point). The material will be held in the strictest confidence by the recipients and will not be used, in whole or in part, for any purpose other than the purpose for which it is provided without prior written consent by Vantage Point. The recipient assumes responsibility for further distribution of this document. In no event shall Vantage Point be liable to anyone for direct, special, incidental, collateral or consequential damages arising out of the use of this material, to the maximum extent permitted under law.

The security testing team made every effort to cover the systems in the test scope as effectively and completely as possible given the time budget available. There is however no guarantee that all existing vulnerabilities have been discovered due to the nature of manual code review. Furthermore, the security assessment applies to a snapshot of the current state at the examination time.

Scope of Audit

Vantage Point reviewed the smart contracts underlying codebase to identify any security or economic flaws, or non-compliance to Algorand best practices. The scope of this review included the following test-cases and audit points.

- Insufficient Sender Address Validation for Privileged Operations
- Lack of Validation for Validity of Referenced States from External Applications
- Insufficient Validation of Transaction Fields and Types
- Validation of RekeyTo address for non-rekeying transactions
- Validation of CloseRemainderTo and AssetCloseTo for non-closing transactions
- Validation of Asset Identifier for Asset Transfer Transactions
- Validation of GroupIndex and GroupSize for Transaction Groups
- Incorrect Order of Operations
- Smart Contract Versions
- Incorrect Use of ScratchVar, Local and Global States
- Flawed/Inaccurate Logical/Mathematical Operations
- Overflow or Underflow Possibilities based on Valid Argument Ranges
- Validation of user-supplied Application Arguments
- Use of Multisignatures for Privileged Accounts
- Other known Algorand Best Practices and Guidelines