

Security Audit Report

Hone Smart Contracts

Delivered: May 30th, 2022



Prepared for Hone by



Table of Contents

Executive Summary

Goal

Scope

Platform and Contract Features Description

Methodology

Disclaimer

Findings

A01: Precision loss in the calculation of the exchange rate can result in accumulation of assets by the contract

A02: Possibility for dAlgo price inflation and algos theft by sending dAlgos to the contract

A03: Multiple application calls can use the same payment to stake algos

A04: Loss of dAlgos sent to the user if the exchange rate multiplied by the amount staked is greater than 2^{64}

A05: Calculation of amounts of algo to be sent can overflow

A06: No check on the *fees_redeemed* variable when redeeming fees

A07: Malformed transaction groups are accepted for fee redemption

A08: User can call the unstake function sending 0 dAlgos in an asset transfer


A09: Staking can return 0 dAlgos

A10: The fee collector can receive all algos in the contract

A11: Migrations can be requested and performed in between the end of registration and redemption of fees

A12: No checks on specific Algorand fields and properties

Informative Findings

- 
- B01: Reformulation of the exchange rates calculation
 - B02: Enforcement of transaction group flexibility
 - B03: Transparency of administrative operations
 - B04: Unnecessary validations
 - B05: Unnecessary scaling of the fee multiplier
 - B06: Feasibility of enabling the contract to be updated
 - B07: Possibility for performing operations seconds after the time limit

Executive Summary

Hone engaged Runtime Verification Inc. to conduct a security audit of their smart contracts. The objective was to review the contracts' business logic and implementations in TEAL and identify any issues that could potentially cause the system to malfunction or be exploited.

The audit was conducted over the course of 5 calendar weeks (April 25, 2022 through May 30, 2022) focused on analyzing the security of the code related to the staking of algos and unstaking dAlgos, as well as all the administrative operations used by the contracts to handle its business logic.

The audit led to identifying potentially critical issues, which have been identified as follows:

- Implementation flaws: [A04](#), [A07](#), [A08](#), [A09](#), [B04](#)
- Potential asset security vulnerabilities: [A01](#), [A02](#), [A03](#), [A04](#), [A08](#), [A09](#), [A10](#), [A11](#), [B01](#)

In addition, several informative findings and general recommendations also have been made, including:

- Input validations: [A03](#), [A04](#), [A10](#)
- Best practices: [A05](#), [A06](#), [A07](#), [B02](#), [B03](#), [B05](#), [A12](#)
- Optimization related particularities: [B01](#)
- Blockchain related particularities: [A03](#), [A07](#), [A12](#), [B06](#), [B07](#)

All the potentially critical issues have been addressed, and the great majority of the informative findings and general recommendations have been incorporated as well. All of the remaining findings have been acknowledged by the client and deemed non-threatening to the integrity of the platform, when not intended by design.

Goal

The goal of the audit is twofold:

1. Review the high-level business logic (protocol design) of Hone's system based on the provided documentation;
2. Review the low-level implementation of the system given as smart contracts in TEAL.

The audit focuses on trying to identify issues in the system's logic and its implementation that could potentially render the system vulnerable to attacks or cause it to malfunction. Furthermore, the audit highlights informative findings that could be used to improve safety and efficiency of the implementation.

Platform and Contract Features Description

Hone is a platform built over the Algorand blockchain that allows users to stake their algos to obtain an asset named dAlgo. The algos staked in the platform are committed to the Algorand Governance project, which provides rewards to users based on the amount of algos committed during the governance period. The users can unstake their dAlgos to obtain their algos back, and in case the governance rewards have been issued, the user will obtain the amount of algos that have been staked plus a percentage of the rewards issued to the platform.

Hone aims to add liquidity to the Algorand blockchain, as the algos that have been committed during the governance period must remain in the accounts of the users. If the balance of the user goes below the committed amount in the governance period, no rewards will be issued for that user. By using Hone, the user can stake his algos through Hone's contracts while receiving assets that can be used in other DeFi protocols. The staked algos will be committed by Hone, which will hold them to receive the rewards once the governance period ends, taking a fee over the amount of dAlgos that have been committed. If the user still holds his dAlgos, the original amount of algos plus the rewards of the user can be claimed back by unstaking the users dAlgo.

There are two actors that interact with Hone's contracts. The first one is the normal user, which can stake algos and unstake dAlgos, while the second user is the contract

creator, which is capable of setting the rate for the fees, claiming the fees and handling other administrative operations.

Scope

The scope of the audit for the two first weeks is limited to the artifacts available at git-commit-id **703f8f86f595c3422548e8e0672f5d157bfbad99** for the branch **main** of a private repository provided by the client. For the remaining three weeks of the engagement, the git-commit-id **2fb5acf5f210999f8bdaf5a5006e59354dc405bc** of the branch **fix/staking-contract** of the same repository.


The comments provided in the code, and a general description of the project, including samples of intended transaction groups to be used for interacting with the platform, provided by the client were used as reference material.

The analyzed functionalities consider the handling and exchange of assets between user and contract accounts, which, in the git commit id **703f8f86f595c3422548e8e0672f5d157bfbad99**, is implemented using two contracts:

1. **staking_approval.teal**: implements the core functionalities of the platform, handling the application calls to set the contract's configuration variables and controlling the assets according to the specified logic;
2. **staking_clear.teal**: implements the actions of the staking contract when handling a clear state transaction;
3. **beneficiary_approval.teal**: implements the functionalities of the platform regarding the claiming of fees over the rewards and transferring the remaining assets to the staking contract account;
4. **beneficiary_clear.teal**: implements the actions of the beneficiary contract when handling a clear state transaction.

For the git-commit-id **2fb5acf5f210999f8bdaf5a5006e59354dc405bc**, the analyzed functionalities are implemented using a single contract, implemented using two files:

5. **staking_approval.teal**: implements the core functionalities of the platform, handling the application calls to set the contract's configuration variables and controlling the assets according to the specified logic. It also includes the functionalities regarding the handling of the fees applied over the rewards of the Algorand Governance program;

- 
6. **staking_clear.teal**: implements the actions of the contract when handling a clear state transaction;

The audit is limited in scope to the artifacts listed above. Off-chain and client-side portions of the codebase as well as deployment and upgrade scripts are not in the scope of this engagement.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in our [Disclaimer](#), we have followed the approaches described below to make our audit as thorough as possible.

First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic. To this end, we carefully analyzed all the proposed features of the platform and actors involved in the lifetime of a deployed contract.

Second, we thoroughly reviewed the contract source code to detect any unexpected (and possibly exploitable) behaviors. As TEAL is a relatively low-level language, we constructed different higher-level representations of the TEAL codebase, including:

1. Automatically generated control-flow graphs using an extended version of the [Tealer Static Analyzer](#) tool, coupled with internally developed tools to assist the code navigation and readability,
2. Created a high-level model in Python to simulate the operations performed by regular users in the Hone platform, maintaining the fidelity with regards to the limitations of size and types of variables in the *Algorand Virtual Machine* (AVM),
3. Modeled sequences of mathematical operations as equations and, considering the limitations of size and types of variables in the AVM, checking if all desired properties hold for any possible input value.

This approach enabled us to systematically check consistency between the logic and the low-level TEAL implementation.

Fourthly, we performed rounds of internal discussion with Algorand experts over the code and platform design, aiming to verify possible exploitation vectors and to identify improvements for the analyzed contracts.

Finally, we reviewed the TEAL guidelines published by Algorand to check for known issues.

Additionally, given the nascent TEAL development and auditing community, we reviewed [this list](#) of known Ethereum security vulnerabilities and attack vectors and checked whether they apply to TEAL smart contracts; if they apply, we checked whether the code is vulnerable to them.



Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Findings

Findings presented in this section are issues that can cause the system to fail, malfunction, and/or be exploited, and should be properly addressed.

All findings have a severity level ranging and an execution difficulty level, ranging from low to high, as well as categories in which it fits.

A01: Precision loss in the calculation of the exchange rate can result in accumulation of assets by the contract

[Severity: Low | Difficulty: Low | Category: Asset Security]

Description

On the second version of the staking contract, provided in the git commit id **2fb5acf5f210999f8bdaf5a5006e59354dc405bc**, after modifying the equation used for calculating the exchange rate on both staking and unstaking operations, which has been a result of the informational finding [B01](#), we identify the possibility of precision loss in the exchange rate calculation. This loss will directly affect the amount of algos and dAlgos received by users in the operations performed by the contract, also possibly accumulating small amounts of assets per staking/unstaking transaction, which is referred to as '*dust accumulation*', will affect the amounts of dAlgo in circulation and algos in the balance of the staking contract, progressively heightening the impact on exchange rates.

Let's assume the following variables:

- **AB**: The staking contract's algo balance minus its minimum allowed balance;
- **dAC**: The amount of dAlgo in circulation;
- **A_s**: Amount of algos sent by a user to the staking contract in a staking operation. Value in microAlgos;
- **dA_s**: Amount of dAlgos sent from the staking contract to a user in a staking operation. Value in microdAlgos, assuming a dAlgo to have 6 decimal places in its value;
- **A_{us}**: Amount of algos received by a user from the staking contract in a unstaking operation;

- **dA_{us}** : Amount of dAlgos sent by a user the staking contract in a unstaking operation;

For the analysis of the staking output calculation, the amount of dAlgos calculated by the contract to be sent to the staker is obtained as follows:

$$dA_s = \frac{\frac{dAC \cdot 10^6}{AB} \cdot A_s}{10^6}$$

Where $\frac{dAC \cdot 10^6}{AB}$ represents the staking exchange rate value scaled by 10^6 .

For the analysis of the unstaking output calculation, the amount of algos calculated by the contract to be sent to the unstaker is obtained as follows:

$$A_{us} = \frac{\frac{AB \cdot 10^6}{dAC} \cdot dA_{us}}{10^6}$$

Where $\frac{AB \cdot 10^6}{dAC}$ represents the unstaking exchange rate value scaled by 10^6 .

Notice that these equations are calculated abiding by the AVM rules, which implies that the division of two integers will result in an integer where the floating part of the division value is truncated and ignored.

As such, the presence of a division in the calculation of the exchange rates makes the outcome of this equation subject to precision loss.

Another consequence of this loss is that the amount of algos and dAlgos sent to the contract for the staking and unstaking operations, respectively, are slightly higher than they should be if value is loss in the calculation of the amount of assets returned to the user, resulting in the accumulation of assets in the contract.

Scenarios

To exemplify this finding and its consequences, We've elaborated a scenario where the contract is deployed and its balances are:

- dAlgos in circulation (**dAC**): 1000 (10000000000 microdAlgos);
- Algo balance (**AB**): 1100 (11000000000 microdAlgos), already excluding the minimum required balance amount.

Under such circumstances, when staking an amount of Algos (**A_s**), the amount of dAlgos obtained when staking (**dA_s**) on `_stake` function is:

$$dA_s = \frac{\frac{dAC \cdot 10^6}{AB} \cdot A_s}{10^6}$$

Now, if a user Alice is attempting to stake 100 algos, the above equation could be rewritten as, rounding at the second decimal place:

$$dA_s = \frac{\frac{1000 \cdot 10^6}{1100} \cdot (100 \cdot 10^6)}{10^6} = \frac{909090.91 \cdot (100 \cdot 10^6)}{10^6}$$

dAC and of AB were not scaled according to their 6 decimal places, as the result would remain the same. Notice the floating points in this calculation. Due to the precision issue faced when developing in TEAL, all the numbers handled in this equation must be integers, so the number 909090.91 is truncated, **losing all numbers after the floating point**, to 909090. And so **dA_s** is calculated as follows:

$$dA_s = \frac{909090 \cdot (100 \cdot 10^6)}{10^6} = 90909000$$

Or in other words, Alice received 90.909 dAlgos.

If immediately after staking, Alice attempts to unstake 90.909 dAlgos (dA_{us}), the following amount of Algos will be received, rounded to two decimal places:

$$A_{us} = \frac{\frac{AB \cdot 10^6}{dAC} \cdot dA_{us}}{10^6} = \frac{\frac{1200 \cdot 10^6}{1090.909} \cdot (90.909 \cdot 10^6)}{10^6}$$

$$A_{us} = \frac{1100000.09 \cdot 90909000}{10^6}$$

Once again, dAC and of AB were not scaled according to their 6 decimal places, as the result would remain the same. Notice the floating point once more present in the calculation. To truncate this value to an integer, we discard the .09, and so we have:

$$A_{us} = \frac{1100000 \cdot 90909000}{10^6}$$

For unstaking all her dAlgos, Alice receives 99.9999 Algos. 100 microAlgos are left in the contract account. It is important to highlight that for every other user that stakes algos in the contract, if the exchange rate has one of its variables truncated, the user will be issued a slightly smaller amount of dAlgos than they should, and more Algos will be accumulated in the contract.

To further investigate the remaining values of the pools and the effects of the operations described above, let's assume that the remaining 100 dAlgos in circulation are going to be unstaked by one single user Bob. Bob then proceeds to receive the following amounts of Algo:

$$A_{us} = \frac{\frac{1100.0001 \cdot 10^6}{1000} \cdot (1000 \cdot 10^6)}{10^6} = \frac{1100000.1 \cdot (1000 \cdot 10^6)}{10^6}$$

Again, dAC and of AB were not scaled according to their 6 decimal places, as the result would remain the same. 1100000.1, truncated to an integer, becomes 1100000. We then have:

$$A_{us} = \frac{1100000 \cdot (1000 \cdot 10^6)}{10^6} = 1100000000$$

Bob receives 1100 algos. The pool is left with 100 microAlgos.

Recommendation

The conclusions and recommendations explored below were based in the following assertions:

1. The exchange rate (E_{ad}) of algos to dAlgos stays the same or decreases through means of staking operations, as the calculated value E_{da} is truncated after its 6th decimal place, which multiplies the amount of algos to calculate the amount of dAlgos to be sent, thus possibly reducing the amount of circulating dAlgo; A consequence of this is the rise in the exchange rate (E_{da}) of dAlgos to algos;
2. The exchange rate (E_{da}) of dAlgos to algos stays the same or increases through means of unstaking operations, as the calculated value E_{ad} is truncated after its 6th decimal place, which multiplies the amount of dAlgos to calculate the amount of algos to be sent, thus possibly reducing the amount of algos that have been unstaked from the contract account; A consequence of this is a drop in the exchange rate (E_{ad}) of algos to dAlgos;
3. The exchange rate (E_{da}) of dAlgos to algos can also increase if a user sends algos or dAlgos to the staking contract in a transaction group without an application call, only increasing its balance; Any of these operations decreases the exchange rate of algos to dAlgos (E_{ad}).
4. If the amount of algos in the contract account is drastically greater than the amount of dAlgos in circulation, precision loss will affect the staking process. You might stake and receive no dAlgo, or less than you should. This is advantageous to the holder(s) of dAlgos;
5. If the amount of dAlgos in circulation is drastically greater than the amount of algos in the contract. You might unstake and receive no Algo, or less than you should. This would be advantageous to no one. Also, given that the only way to obtain algos from the staking contract is through unstaking while circulating dAlgos increase only if users stake, reaching this situation is impossible.

Taking these assertions as a foundation for more elaborate thoughts on the collection of small amounts of assets in the contract account as well as the precision loss for

calculating amounts sent to users, the conclusion that has been reached is that the no feasible threats will exist to Hones platform during the life cycle of the staking contract if it can maintain a healthy exchange rate, that is an exchange of algos to dAlgos E_{ad} greater than 10^{-3} . Keeping the exchange rate above this value prevents exploits based on the assertion (4) and a possibly rapid deterioration of the exchange rate based on the assertions (1) and (2), which can be aggravated by assertion (3).

One of the reasons this exchange rate has been chosen is because if this value is 10^{-3} , then users could lose at most 0.1 percent of the amount of dAlgos that they should receive. This happens due to the exchange rate being truncated and thus losing the decimal places that could express up to 0.09999...% of the value which is being multiplied. Even so, to reach an exchange rate like this, if there are 100 dAlgos in circulation, there must be at least 100000 algos in the staking contract account, and this scenario would only be financially feasible if the holder of dAlgos is the first individual to stake after the deployment of the contract (scenario further explored in finding [A02](#)).

Still, this percentage which has not been converted to dAlgos is distributed among all participants of the staking pool prior to the arrival of the current staker, thus older holders of dAlgo have an advantage here. This advantage does not have to be considered a substantial benefit, as the amounts of algos which will be part of their profits are negligible, as long as the exchange rate from algos to dAlgos is kept above the defined healthy threshold.

Finally, an implementation that could prevent dust collection (given the current protocol design and AVM limitations) would not be trivial as there would always be a degree of precision loss and all the assertions mentioned above would likely still hold, and the effort for developing it might not be the most viable option considering the platform's time to market. A simpler solution to this would be to deploy the contract and stake a small value of algos immediately after its deployment, preventing other users from exploiting the manipulation of the exchange rate. Once again, by having an honest user staking 100 algos right after deploying the contract, to distort the exchange rate of algos to dAlgos beyond 10^{-3} , the amount of dAlgos in circulation should remain 100 and the balance of algos should be of at least 100000. After this initial staking operation, any substantial modification in the exchange rates of the contract would come at a large financial setback to any user that attempts to try and modify it.

Status

This issue has been acknowledged by the client.

A02: Possibility for dAlgo price inflation and algos theft by sending dAlgos to the contract

[Severity: High | Difficulty: High | Category: Asset security]

Description

Considering the distribution of dAlgos in the ecosystem, it is important to guarantee the fairness in distribution of tokens to all individuals that want to participate in the protocol. If there is a way for two different users staking the same amount of algos and receiving drastically different amounts of dAlgos, it is reasonable to assume that the receiver of the lesser amount of assets is being wronged by the protocol as not only by being able to claim a lesser amount of algos back, but having a lesser possible profit margin in other DeFi applications.

Considering the exchange rate calculation implemented in `git-commit-id` **2fb5acf5f210999f8bdaf5a5006e59354dc405bc**, it is possible to tamper with the exchange rate of the staking and unstaking operations by sending algos or dAlgos to the contract's account without an accompanying application call for appropriately handling the funds sent. Intuitively, while sending algos or dAlgos to the staking contract would only benefit the individuals that have dAlgos for unstaking, which would be considered a donation, sending any of those assets to the contract can be used to effectively manipulate the exchange rate and reduce the amount of dAlgos being issued per algo staked. This means that early users of the platform will claim more dAlgos than more recent users, if the exchange rate is drastically modified by malicious users.

It is important to notice that, given that the number of dAlgos in circulation is divided by the amount of algos deposited in the contract during the staking operation, the amount of dAlgos issued for the amount of algos staked is going to be reduced if there are more algos in the contract than dAlgos circulating.

Joining the facts discussed above, if a malicious user can make the exchange rate of algos to dAlgos be smaller than 10^{-6} , any value staked smaller than 10^2 will return 0 dAlgos.

Scenario

An example of how this exploit can be achieved uses the following variable values:

1. Bob stakes 100 Algos, 100 dAlgos are transferred from the staking contract to Bob;
2. Bob sends 99.999999 dAlgos back to the contract account;
3. Alice wants to stake 10 algos. $(1 / (100 * 10^6)) * 10 = 10^{-7} \Rightarrow 0$ because of precision loss;
4. Pool has 110 algos, Bob has all dAlgos in circulation.

This scenario allows for Bob to “steal” Alice algos.

Recommendation

As in finding [A01](#), the best way to avoid a distorted exchange rate, allowing malicious users to perform such an exploit, is to have a trusted user stake an amount of algos greater than 100 right after the deployment of the contract.

Status

This issue has been acknowledged by the client.

A03: Multiple application calls can use the same payment to stake algos

[Severity: High | Difficulty: Low | Category: Asset security, Input validation, Blockchain related particularities]

Description

By the business logic designed for this platform, the “on_stake” function should be triggered by sending a payment followed by an application call under the correct configuration to execute the staking operation logic. The payment is verified by the index of the transaction in the group, which is expected to be 0. The application call, in turn, has no fixed position in the group as no verification over it is performed. Not only can other transactions be placed in between the payment and the application call, but other application calls calling the staking operation can be placed in the same transaction group.

In other words, there are no validations with regards to the transaction group size or relative position of the application call in the group, meaning that the first transaction in the group can be a payment followed by several equal application calls to the staking contract, where all of them call the staking operation with the same parameters. For every application call, dAlgos will be minted, even though all of them are using the same single payment.


The surplus dAlgo received by malicious users can be used for unstaking algos that do not belong to them, thus funding them for performing the operation repeatedly, until the exploit is not deemed profitable anymore.

An analogous version of this exploit does not exist on the unstaking operation, since there is an explicit verification of the transaction group in the “on_unstake” function.

Scenario

This scenario can be exemplified by the following sequence of actions:

1. Bob creates an empty list of transactions;
2. Bob adds a payment to the staking contract valued at an arbitrary amount of algos;
3. Following the payment, Bob adds 14 application calls to the staking contract. All application calls have “stake” as the single parameter for the contract execution;

- 
4. dAlgos will be minted for every application call in the group;
 5. Bob unstakes all the received dAlgo, receiving the originally staked algos plus a small percentage of the algos in the staking contract, originated by the dAlgos received from the last 13 application calls in the group.
 6. Bob now has more algos than the amount that he started with.

Recommendation

This exploit can be prevented by validating if the application call immediately follows the payment, or also by checking if the transaction group has size 2, as it is being done in the “on_unstake” function.

Status

This issue has been fully addressed.

A04: Loss of dAlgos sent to the user if the exchange rate multiplied by the amount staked is greater than 2^{64}

[Severity: High | Difficulty: Low | Category: Asset security, Input Validation]

Description


It has been noted that in the block of operations right after the return of the muldiv procedure call, inside the “on_stake” function, perform a pop operation without checking the value on top of the stack. At the top of the stack there is the calculated exchange rate for the staking operation and the amount of the payment provided by the user, which are then multiplied using the “mulw” opcode leaving a 128 bits value on top of the stack. Using the git commit id **2fb5acf5f210999f8bdaf5a5006e59354dc405bc** as reference for the contract’s code, the executed instructions are as follows:

Calculation of dAlgo amount to be returned on staking

```
260  callsub muldiv
261
262  gtxn 0 Amount
263  mulw
264  swap
265  pop
266  int 1000000
267  /
```

The upper 64 bits of the value returned by the “mulw” operation are removed from the stack without any validations.

Considering the numbers that represent amounts of algos, 6 digits are already in use to represent the decimals of the values, and the “mulw” operation multiplies the algo amount provided in the payment by the staking exchange rate, which initially is represented by 1 scaled to have 6 decimal places, or simply 10^6 . This means that the amount sent in the payment, in Algos, for staking must be smaller than $2^{64} / 10^{12}$, or 18446744 algos. If not, the top 64 bits of the result of multiplication will be discarded, meaning that a negligible amount of dAlgos will be minted in this operation compared to the amount of algos that have been staked. Of course, taking in consideration the circulation of algos in the ecosystem at the moment, staking such a high amount of



algos in a single operation would be uncommon, but if it ever happens, the staker would suffer a severe loss of algos.

The greater majority of the algos deposited in this operation would be distributed among all users in the pool, greatly increasing the price of dAlgos and decreasing the exchange rate from algos to dAlgos.

Recommendation

This issue can be prevented by checking if the top 64 bits returned by the “mulw” operation represent 0. If not, the transaction can be rejected. One option is to modify the “pop” operation in line 265 for an “assert” operation. Furthermore, to have a better usage of the range of values provided by 128 bit integers, the division opcode can be replaced by a “divmodw” operation, where the modulo can be discarded and the upper 64 bits of the output must be verified.

Status

This issue has been fully addressed.

A05: Calculation of amounts of algo to be sent can overflow

[Severity: Low | Difficulty: Medium | Category: Best practices]

Description

When calculating the amount of algos to be sent for the user when unstaking, the exchange rate, which is scaled by 10^6 , is multiplied by the amount of dAlgos sent by the user. Using the git commit id **2fb5acf5f210999f8bdaf5a5006e59354dc405bc** as reference for the contract's code, the executed instructions are as follows:

```
Calculation of algo amount to be returned on unstaking
418  callsub muldiv
419
420
421  gtxn 0 AssetAmount
422  *
423  int 1000000
424  /
```

The multiplication opcode executed in line 422 can overflow the 64 bit integer range, given the right conditions. Such conditions can be met if the exchange rate from dAlgos to algos has a high value and the user attempts to unstake a large amount of dAlgos.

This issue does not affect the normal operation of the contract, since unstaking operation of smaller values still can be executed successfully, and in case it is triggered, the immediate negative consequence is a rejection of the transaction by the AVM.

Scenario

This issue can be triggered on a scenario with a configuration similar to the described below:

1. The exchange rate of dAlgos to algos is 10^2 , which, when scaled to have 6 decimals, becomes 10^8 ;
2. The output of the muldiv operation is already scaled to 10^6 , which means that the amount of dAlgos transferred by the user is now being multiplied by 10^{14} ;

3. A user named Bob attempts to unstake 10^7 dAlgos, which results in a 64 bits multiplication of 10^7 by 10^{14} ;
4. 10^{21} , which is the outcome of the multiplication, is greater than the maximum value supported in a 64 bits integer, or $2^{64} - 1$, which triggers an exception in the AVM, rejecting the transaction.

Recommendation

It would be recommended to handle these operations while supporting a wider range of values. The unstaking value calculation can be performed using 128 bits integers similarly to how it is being done in the staking operation, assuming that the recommendations provided in the finding [A04](#) are being followed.

Status

This issue has been fully addressed.

A06: No check on the fees_redeemed variable when redeeming fees

[Severity: Low | Difficulty: Low | Category: Best practices]

Description

Using the git commit id **703f8f86f595c3422548e8e0672f5d157bfbad99** as reference for this finding, the handling of the staking operations and handling of the rewards and fees was performed by different contracts. The beneficiary contract was responsible for receiving the rewards and taking a fee from them, sending all remaining algos afterwards to the staking contract.

For this operation, there are no checks on the “fees_redeemed” variable at the beginning of the “on_redeem_fees” function. This function can be called repeatedly, although no fees will be transferred to the fee receiver, as the algos have been sent to the staking contract.

There are no advantageous use cases to a hypothetically compromised contract creator other than, exploited together with finding [A10](#), the possibility for claiming any algos received by the beneficiary contract unrelated to the algorand governance rewards. Still, the absence of a validity from a global variable used to keep track of this operation allows for performing unnecessary actions that, specially if this contract is updated in the future, can lead to the execution of undesired operations.

Recommendation

Verify if the fees have been redeemed in the beneficiary contract prior to the collection of the fees. Reject the transaction in case the fees have already been claimed. At the end of the next governance period prior to opening the redemption window to the users, reset the “fees_redeemed” variable.

Status

This issue has been fully addressed.

A07: Malformed transaction groups are accepted for fee redemption

[Severity: Medium | Difficulty: Medium | Category: Best practices, Blockchain related particularities]

Description

Using the git commit id **703f8f86f595c3422548e8e0672f5d157bfbad99** as reference for this finding, in the fee redemption operation performed by the creator account, two application call transactions must be sent to the beneficiary and staking contracts.

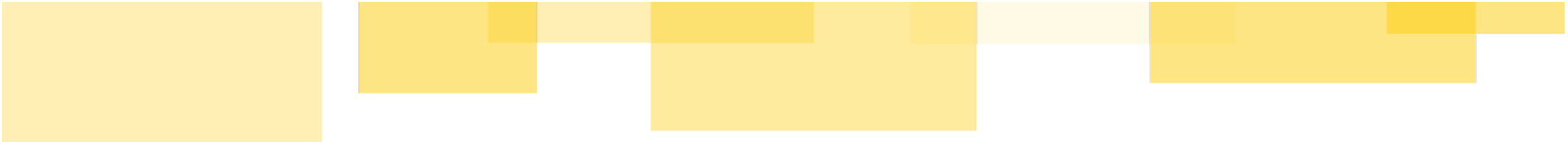
The execution of the application call directed to the staking contract is responsible for, aside from updating values and validating its own transaction, validating the following transaction in its group, which is the application call sent to the beneficiary contract. The application call directed to the beneficiary contract is responsible for updating statistical values stored in the contract, sending a percentage of the total amount of algos in the contract to the fee collector's address and sending the remaining algo balance to the staking contract.

The function which handles the fee redemption in the beneficiary contract has no validations to the prior application call that should precede it, and even though it is still protected by a validation of the sender, who should be the creator of the contract, given how critical the fee redemption operation is, validations to guarantee that the transaction group is configured as originally designed to be should be present in the beneficiary contract.

In its current state, in case the creator account becomes compromised, any of the validations performed in the staking contract can be bypassed by sending a transaction group containing two transactions. One of the transactions must be an application call to the beneficiary contract requesting the fee redemption, and the other can be an arbitrary transaction.

Recommendation

There should be validations in the fee redemption operation in the beneficiary contract guaranteeing that the transaction group is configured as expected.



Status

This issue has been fully addressed.

A08: User can call the unstake function sending 0 dAlgos in an asset transfer

[Severity: Low | Difficulty: Low | Category: Implementation flaw, Asset security]

Description

For this finding, considering the staking contract implementation in the git commit id **2fb5acf5f210999f8bdaf5a5006e59354dc405bc**, there are no checks that prevent the “on_unstake” function from being successfully executed when the user sends 0 dAlgos to the contract. This can be dangerous especially in a scenario where there are no circulating dAlgos in the ecosystem, but there are algos in the contract, as such occurrences would trigger the “return_all_algos function”.

The specific code piece that would generate this exploit is seen below:

Return of all algos in case there are no dAlgos in circulation

```
398    callsub get_dAlgo_circ_supply
399
400
401    gtxn 0 AssetAmount
402    +
403
404    dup
405    int 0
406    ==
407    bnz return_all_algos
```

If the return of “get_dAlgo_circ_supply” is 0 and the amount of dAlgos sent in the unstaking operation is 0, the top of the stack will be 0, thus triggering “return_all_algos” function, where all available algos in the staking contract are sent to the sender of the unstaking operation.

Recommendation

When executing the unstaking operation, validate if the sender sent more than 0 dAlgos to the staking contract.



Status

This issue has been fully addressed.

A09: Staking can return 0 dAlgos

[Severity: Medium | Difficulty: Medium | Category: Implementation flaw]

Description

Using the exchange rate calculation achieved through the finding [B01](#), the multiplication of the exchange rate by the payment amount, which is subsequently divided by 10^6 must be greater than 10^{-6} in order to not become zero due to the precision loss in divisions.

The staking contract currently has a constraint which enforces all deposits in staking operations to have its amount greater than 1000 microAlgos. Having this amount as a reference for the minimum staked value, if the exchange rate is in its healthy state (previously defined in finding [A01](#)), there still will be an amount returned to a user that stakes 1000 microAlgos, but no less.

If, for instance, the user were allowed to stake 1 microAlgo, the exchange rate could not be smaller than 10^6 , which will not be the case after the contract receives the governance rewards or a user makes a payment/asset transfer of dAlgos to the contract. This enforcement of minimal value is not present in the unstaking function, but it is not needed, given that the rate of algos to dAlgos in circulations only grows with time (arguments for this claim were provided in finding [A01](#)).

Recommendation

Maintain the exchange rate of algos to dAlgos above 10^{-3} , which can be done by having a trusted user being the first to stake at least 100 algos. These algos must remain staked in the contract to prevent the exchange rate issues from happening.

Status

This issue has been acknowledged by the client.

A10: The fee collector can receive all algos in the contract

[Severity: Low | Difficulty: High | Category: Input Validation, Asset security]

Description

Taking git commit id **777c5b21862b8bcf5dbca3a47098a7ecfe74f8cb** as reference for this finding, the logic of the contracts allow for the contract creator to set the fee value by modifying the “fee_basis_points” variable. This variable is used for calculating the total amount sent to a fee collector address, which receives a percentage of the rewards received in the beneficiary contract.

At a high level, the presented design for the logic of Hone’s contracts defines a 5% fee over the total amount of rewards received in the beneficiary contract, also providing means for modifying this percentage amount.

What can cause trust issues in users, as well as it can be a considerably dangerous scenario in case of a compromised creator account, is that there is no enforcement of a safe range for “fee_basis_points” in the fee redemption function, which implies that the contract creator account can set the fees to be 100% of the algos held by the contract. Also, there are no checks to ensure that the “fee_basis_points” is being set inside the zero to ten thousand range, which, if the contract is misconfigured, opens a possibility for overflow when trying to redeem the fees or even failure to transfer a higher amount of assets than what the contract holds.

Recommendation

When assigning a value for the “fee_basis_points” variable, validate if this value is within a safe range of, for instance, 0 to 30 percent.

Status

This issue has been fully addressed.

A11: Migrations can be requested and performed in between the end of registration and redemption of fees

[Severity: High | Difficulty: Low | Category: Asset security]

Description

By the initial business logic of Hone's contracts, the staking contract should be allowed to send all assets in the contract to an address defined by the contract creator. This could be done, for instance, in case of a significant modification in the Algorand Governance events, where the current protocol logic would not make sense, and an updated version of the contracts could receive all assets that the current staking contract holds, finally being able to proceed with its new business logic.

Taking git commit id **777c5b21862b8bcf5dbca3a47098a7ecfe74f8cb** as reference for this finding, with the implementation of the suggestions provided in finding [B03](#), to perform a migration, the creator account would have to send a transaction requesting the migration operation to be enabled, and will be able to trigger the migration of assets by sending a transaction to the contract after a certain amount of time has passed. This approach allows users to see that the administrator of the contracts will perform the asset migration, and can use this protection window in order to withdraw their assets from the platform in case they do not trust the address to which the staking contract assets are being transferred to.

Still, migrations can be performed after the time window for staking and unstaking has closed and before the redemption window opens. Even with the implementation of protection windows for administrative operations, if a malicious user compromises the creator account and requests migration of assets, users could only watch as the time goes by, since they would not be able to withdraw their assets until opening of the redemption window.

Recommendation

If requesting the migration operation to be enabled also allows users to unstake their dAlgos, trust issues can be mitigated.

Status

This issue has been fully addressed.

A12: No checks on specific Algorand fields and properties

[Severity: Low | Difficulty: High | Category: Blockchain related particularities, Best practices]

Description

With regards to the available transaction fields and their functionalities, it is recommended to verify possible misconfigured fields, either by malformation of transactions or interference by a malicious third party, that may have a significant impact on the assets held by an account.

Also, ASAs have, by default, mutable properties that are capable of interfering with the proper execution of a smart contract. For the case of the smart contract in question, the functionalities of asset freezing and reclaiming of assets, also referred to as clawing back assets, might impose a problem for the successful execution of the approval program of the analyzed approval programs.

Considering the aforementioned, the following findings were identified:

- No verification in the RekeyTo field: There are no checks on rekey fields for the payments and asset transfers. These fields are being validated only in application calls;
- No verification in the AssetCloseTo/CloseRemainderTo fields: When verifying the payments/asset transfers performed by the users, there are no checks on the AssetCloseTo/CloseRemainderTo field;
- No checks for frozen assets: There are no checks to verify if the assets that are being held in the contract are frozen when trying to transfer them to their respective owners;
- No checks for clawed back assets: There are no checks to verify if the assets that are being held in the contract have been clawed back when attempting to transfer them to their respective owners;
- The contract can be deleted: The logic of the contract allows for the creator of the contract to delete it. This is extremely dangerous in case of a compromised creator account, as it would result in the loss of all assets in the contract.



Recommendation

For the transaction fields mentioned above, it should be ensured that they have not been populated before being handled by the contract. For the mutable properties of the assets, it is recommended that the availability of assets should be verified before performing operations over them, preventing assets that will be handled from being freezed in the contract as well. Also, given that the dAlgo asset will be managed by the Hone team, the freeze account and clawback account field can be left empty.

Status

With regards to the verification of transaction fields and deletion of the contract, the issues have been addressed by the client, while the verification of the availability of the assets has been acknowledged.

Informative Findings

Findings presented in this section do not necessarily represent any flaw in the code itself. However, they indicate areas where the code may need some external support or where the code deviates from best practices. We have also included information on potential code size reductions and remarks on the operational perspective of the contract.

B01: Reformulation of the exchange rates calculation

[Severity: Low | Difficulty: - | Category: Optimization related particularities]

Description

Taking the git commit id **777c5b21862b8bcf5dbca3a47098a7ecfe74f8cb** as reference for this finding, in the staking contract, the current implementation of the calculation for the exchange rate in the “on_stake” function is considerably different than the proposed in the documentation.

Let's assume the following variables:

- **AB**: The staking contract's algo balance minus its minimum allowed balance;
- **dAC**: The amount of dAlgo in circulation;
- **A_s**: Amount of algos sent by a user to the staking contract in a staking operation. Value in microAlgos;
- **dA_s**: Amount of dAlgos sent from the staking contract to a user in a staking operation. Value in microdAlgos, assuming a dAlgo to have 6 decimal places in its value;
- **A_{us}**: Amount of algos received by a user from the staking contract in a unstaking operation;
- **dA_{us}**: Amount of dAlgos sent by a user the staking contract in a ustaking operation;
- **IdA**: Global variable that stores the amount of issued dAlgos. Updated by subtracting the amount of dAlgo sent in staking operations.

While the asset amounts obtained by the users on staking and unstaking operations on the proposed documentation are defined, respectively, as:

$$dA_s = \frac{dAC}{AB} \cdot A_s \quad A_{us} = \frac{AB}{dAC} \cdot dA_s$$

The implemented calculations are:

$$dA_s = \frac{E \cdot 10^6 \cdot A_s}{10^6} \quad A_{us} = \frac{\frac{AB \cdot 10^6}{IdA} \cdot dA_{us}}{10^6}$$

Where E is a global variable that stores $\frac{dAC}{AB}$, initially valued at 10^6 , assigned after sending the dAlgos to the last user that performed a staking operation.


The reason for such difference lies in the order of the transactions in the group, where the payment of algos to be staked comes before the application call, and thus the amount of the payment is already added to the application balance prior during the execution of the application call.

To have such different calculations implies the need for proving the equivalence of the staking business logic equation and its implemented version as to avoid different behaviors from what has been intended. Not only that, but by storing E and IdA in global variables, being modified only after performing the staking and unstaking operations, asset transfers and payments made to the contract will not be reflected in the calculations of the subsequent transactions.

Still, there is a simpler form of implementation to the exchange rate calculation, which is summarized by the following equations:

$$dA_s = \frac{\frac{dAC \cdot 10^6}{AB} \cdot A_s}{10^6} \quad A_{us} = \frac{\frac{AB \cdot 10^6}{dAC} \cdot dA_{us}}{10^6}$$

Where, for the staking operation, fAB is the current account balance minus the prior transaction payment amount, while, for the staking operation dAC is the total supply of dAlgos minus the current application account dAlgo balance plus the previous transaction asset transfer amount.



Given the equivalence of the theoretical and implemented dAC and AB , we can assume that the implementation of this calculation is equivalent to the mathematical equation proposed in the documentation of the staking function.

Status

This issue has been fully addressed by the client.

B02: Enforcement of transaction group flexibility

[Severity: - | Difficulty: - | Category: Best practices]

Description

It is recommended to reevaluate the flexibility of the transaction groups in the contract at the moment. Having more strict validations regarding the transaction groups can prevent front-running, injection of undesired transactions by a compromised front-end, and, among other possible exploitation points, general unpredictable behavior.

By having no validations with regards to the order and amount of transactions in a group, issues such as the ones mentioned in findings [A03](#) and [A07](#) can be exploited by malicious users.

Recommendation

After the addressment of the findings [A03](#) and [A07](#), two functions in the contract still remain without validations to the groups of transactions in which they should trigger them. The names of these functions are “on_redeem_fees” and “on_set_redemption_open”. If their operations are designed to be executed atomically, then there should be explicit validation of the transaction groups in the contract.

Status

This issue has been fully addressed by the client.

B03: Transparency of administrative operations

[Severity: Low | Difficulty: - | Category: Best practices]

Description

Given the nature of DeFi applications, the lack of a trusted, centralized entity that guarantees the security of an application, it is very common to have the community of the ecosystem validating the applications themselves. The depth of this validation can vary according to the technical background of the users, ranging from high-level checks, such as verifying the amount of assets that users already trusted to the application, to more convoluted analysis, such as the validation of the code itself.

In many cases, what might be considered a sign of danger in a contract's code is the presence of administrative operations capable of affecting the decisions of the contract or the handling of the assets deposited by users.

Some operations performed by Hone's contracts fall into this category, one example being the migration of assets functionality, or the governance voting functionality, where the values used to perform the operations are given as parameters for their execution. For these operations, the user is only able to know the details of the operation after it has been performed, which might be a source of trust issues for members of the Algorand community.

Recommendation

If designed to be more transparent, values that are currently given as application parameters by the creator address on protected operations can be stored in global variables. Protection windows can be added to prevent modifying or using the values before a stipulated timestamp, as to give the opportunity for more skeptical users to see the values which will be used in administrative operations ahead of time.

Status

The issue has been addressed by the client.

B04: Unnecessary validations and duplicated variable assignment

[Severity: Low | Difficulty: - | Category: Implementation flaw]

Description

In the staking contract, on the “on_create_app” function there are checks to ensure that the number of arguments of the application call is 0 and that the accounts and applications array is of size 0 as well. For the execution path of the creation of the contract, there is no need to perform such checks as there are no operations performed by this function that interact with the mentioned fields.

Also, previously to the modifications rooted into the informative finding [B07](#), when branching the actions based on the OnCompletion field of the application call, there is an explicit check for validating if this field is filled with the UpdateApplication value. Although it might be meant for aiding readability of the contract, it is an unnecessary check, as application calls with the OnCompletion field populated by the UpdateApplication value would fail regardless of this check.

On the new, unified version of the staking contract, the variable "skip_date_checks" is populated twice in the "on_create_app" function.

Status

The issue has been addressed by the client

B05: Unnecessary scaling of the fee multiplier

[Severity: - | Difficulty: - | Category: Best practices]

Description

One of the functionalities of the beneficiary contract is the capability to take a fraction of the total amount of rewards received and send it to a fee receiver address. This fraction is defined by the “fee_basis_points” variable, and was designed to express percentages starting from 0.01%. To do so, “fee_basis_points” should range in between 1 and, at most, 10000. Still right before being used for calculating the fraction of the total rewards received, it is being multiplied by 100 for adding decimal places in an attempt to reduce the loss by precision, as seen below:

```

                Multiplication of “fee_basis_points” by 100
159    byte "fee_basis_points"
160    app_global_get
161    int 100
162    *
```

This operation is not necessary, as scaling values are not needed for calculating percentages of an amount. The precision issues will still exist and any value that remains as the fractional part of the fee will be erased.

Since there are no validations to check if the amount should be rounded up or not, to calculate the "per-ten-thousand", the multiplication of “fee_basis_points” by 10000 should be removed, and the division later on for scaling purposes should be by 10000, instead of 1000000 as it is currently.

Status

The issue has been fully addressed.

B06: Feasibility of enabling the contract to be updated

[Severity: - | Difficulty: - | Category: Blockchain related particularities]

Description

Given the changing nature of the Algorand governance program, it is comprehensible to allow the contract to be updated. Protection windows can be used for "announcing" that a modification in the code is coming and storing the newly proposed version of the contract in a sequence of global variables, or a hash of it, which can be used to validate the code submitted in the update application transaction.

Users that may not trust the new version of the contract can unstake their assets and be more comfortable in providing their assets to be held by the application.

Status

The issue has been fully addressed.

B07: Possibility for performing operations seconds after the time limit

[Severity: Low | Difficulty: - | Category: Blockchain related particularities]

Description

Since it takes roughly 4.5 seconds to generate a block in the Algorand blockchain, depending on the timing when the transaction is submitted, it is possible to stake assets up to 4.5 seconds after the end of the time window reserved for staking and unstaking operations.

It is also important to note that there must be a protection interval in between the time window reserved for the staking and unstaking operations and the submission of the transaction sent for committing a certain amount of algos to the governance event. This protection interval should be, at the very least, greater than the time taken to generate a block in the Algorand blockchain in order to prevent cases of a last-second staked amount not being committed by the staking contract.

Status

The issue has been fully addressed.