COINFABRIK AUDIT REVIEW

Table of Contents

This document contains responses to the CoinFabrik audit report received in December of 2021.

Sections marked **Severity**, **Description** and **Recommendation** are from CoinFabrik.

Sections marked **Response** are from the CityCoins developers.

Last updated: 2021-02-16

CR-01: Lost Miners Funds after Shut Down

Severity: Critical

Description:

In the core contract, users can commit to mine as many as the next 200 blocks. The city wallet or the approvers can upgrade the core contract through the auth contract (auth.clar), stopping the mining in the previous core. Therefore, those miners who committed for the blocks after the upgrade will have lost their committed tokens.

Recommendation:

Implement a function to refund miners which is called by the shutdown function (shutdown-contract()). This function should store the committed value in a mapping. Finally, miners will be able to claim their tokens calling a withdrawal function.

Response:

This is acknowledged as a known issue, with the premise that any upgrades to a core contract should:

- only be performed when absolutely necessary (e.g. emergency bug fix, large protocol revision)
- be scheduled in advance to allow for interface/tooling/documentation updates
- be well-communicated to the community before launched
- include coordination with disabling mining from current user interfaces

The upgrade can only be initiated by the city's 2-of-3 multisig wallet or by a 3-of-5 job approval from the auth contract. This scenario can only happen if someone mines for an extended period right before an upgrade occurs.

The challenge with the recommendation that when a miner sends a mine-many() transaction and stacking is active, 30% of the STX spent by the miner are automatically sent to the city's wallet, and only 70% remains in the contract for them to claim.

Additional research will be done on this issue as part of the V2 discussion around the protocol, and could be possible now given the Stacks network performance upgrade in Stacks 2.0.5.

ME-01: Insecure Authentication through Tx-sender

Severity: Medium

Description:

Global variable tx-sender returns the original sender of the current transaction, or if as-contract was called to modify the sending context, it returns that contract principal. Using this variable for authentication is not secure. Actors in the system could be targets of phishing. This is analogous to what happens to tx.origin and msg.sender in Solidity. There, the convention is to use msg.sender, which works like contract-caller.

For instance, an approver can be tricked to call a malicious contract which calls auth.approve-job() against his will. Also, the malicious contract could call the token transfer function (token.transfer()) to send the user's tokens to itself.

Recommendation:

Prefer contract-caller to tx-sender for authentication. Contract-caller returns the caller of the current contract context.

Response:

This is acknowledged however additional safeguards are in place.

Regarding the job approval flow:

- a 2-of-3 multisig transaction from the city or 3-of-5 approvals are required to execute
- approvals are submitted as direct contract calls which require inputting the correct deployer address and contract name to access the functions
- tx-sender is used for creating a job, activating a job, and approving/disapproving a job, but:
    - contract-caller is used for adding arguments to the job via guard-add-argument, which verifies the creator of the job set by tx-sender
    - contract-caller is used for executing a job, which verifies the transaction was sent by an address on the approver list
- an approver has the option to disapprove a job after approving which reverses the vote

Regarding the unwanted transfer of tokens:

- tx-sender was chosen over contract-caller to match the logic for a STX transfer
  (err u4) -- the sender principal is not the current tx-sender
- this method is preferred because it allows for other contracts to manage funds, e.g. the Syvita Mining Pool contract
- post conditions provide an extra layer of protection since they are required for transferring any assets and help ensure the user is taking the action they intend to

Another advantage to Clarity smart contracts is that they are open source, available on the blockchain, and should be inspected to see what they will do before a contract function is called.

MI-01: Rounding Issue when Calculating Stacking Rewards

Severity: Minor

Description:

In the core contract, stacking rewards are calculated through get-entitled-stacking-reward(). The amount of city coins locked by the user is multiplied by the total amount of micro-stacks staked for a cycle, and then divided by the total amount of city coins stacked for that cycle. Because of the data type imprecision, remainders will be accumulated in the contract and will not be used.

Recommendation:

If there were a considerable amount of micro-stacks over the time, the contract could calculate this value and use it. When all the stackers have claimed their rewards, the difference between the claimed amount and totalUstxThisCycle is the sum of the remainders.

Response:

This is acknowledged as a known issue. For context, Stacking rewards involve calculating the amount a user stacked against the total stacked in a given cycle using the equation:

```
(totalUstxThisCycle * userStackedThisCycle) / totalStackedThisCycle
```

Given the remainder for this function is denoted in uSTX, where 1 STX = 1,000,000 uSTX, the remainder should be significantly smaller than fractions of a penny being rounded as part of normal banking transactions.

Example calculations for MIA based on the numbers from Cycle 7 and a STX value of $2.50:

- Total MIA Stacked in Contract: 2,355,579,154 MIA
- Total STX Rewards in Contract: 1,064,700 STX (1,064,700,000,000 uSTX)

Total MIA Stacked	Total uSTX Reward	Remainder	Value of Remainder
5,000	2,259,953	.774408381	$0.0000019360209525
100,000	45,199,075	.48816761	$0.000001220419025
1,000,000	451,990,754	.8816761	$0.00000220419025

Trying to calculate this value for later use adds to the runtime and write cost dimensions of the contract for each claim transaction, and requires designating an approved address to make the withdrawals.

This could be implemented using the city wallet’s address, but additional research would need to be done to make sure the amount claimed/recovered would be worth the transaction fee on the blockchain.

MI-02: Inactive Core Contract Can Activate Itself

Severity: Minor

Description:

All core contracts are stored in the auth contract with different states: STATE_DEPLOYED, STATE_ACTIVE and STATE_INACTIVE. The deployed state is the initial state. When it receives the required signals, it transits to the active state. Finally, if it is upgraded, the previous contract will be inactive. auth.activate-core-contract() only checks if the caller is in the core contract mapping. If it were a malicious contract which was replaced through an upgrade, it would call activate-core-contract() and activate itself.

Recommendation:

Verify the contract caller is a core registered with state STATE_DEPLOYED.

Response:

This is acknowledged as a bug in the auth contract, however in order to exploit this it would require a 2-of-3 multisig transaction from the city or 3-of-5 approvals that adds a malicious contract to the core contract map to start with.

The suggested mitigation was applied in commit 8c30dde29f597bd66eb80120bc31f223635595dd and unit tests were expanded to cover this scenario.

MI-03: Malicious Inactive Core Contract Can Mint and Burn Tokens

Severity: Minor

Description:
The token contract (token.clar) only validates if the contract caller is registered in the auth contract. However, contracts cannot be removed from the auth contract. Therefore, if a malicious core were added and then replaced by the approvers, it could still mint new tokens and burn someone else’s tokens.

Recommendation:

Instead of validating if the core contract is registered, the token contract should check if the core is active through get-active-core-contract().

Response:

This is not acknowledged as a bug but was instead a deliberate design choice. Users should be able to claim their mining rewards at any time, and the core contract contains the logic to calculate the winner for a given block height while the contract was active. This means an inactive, older core contract would still need the ability to mint new CityCoins based on the issuance schedule at the block heights in which it was active.

MI-04: Denial of Service in the Upgrade of a Malicious Core

Severity: Minor

Description:

upgrade-core-contract() and execute-upgrade-core-contract-job() are the functions that set a new active core contract. These functions call shutdown-contract() from the current active core to disable it. However, if the current core is a malicious contract, it might revert to stop the upgrade, making it impossible to change the core.

Recommendation:

Instead of calling the core contract to modify the variables, those variables could be placed in the auth contract and be checked by the core through external calls. Therefore, the upgrade does not depend on unknown external logic.

Response:

This is acknowledged as a bug in the auth contract, however in order to exploit this it would require a 2-of-3 multisig transaction from the city or 3-of-5 approvals that adds a malicious contract to the core contract map to start with.

The challenge with this recommendation is that calling external contracts can significantly increase the run-time and read cost dimensions, and based on the protocol, the auth contract status in the CoreContracts map of STATE_INACTIVE should be in alignment with the core contract variable isShutdown.

EN-01: Approvers Need to Approve his Own Jobs

Severity: Enhancement

Description:

In the auth contract (auth.clar), approvers can create a job. For a job to be executed, it requires a minimum amount of approvals. Now, if an approver creates a job through create-job(), he can call approve-job() to approve his own job

Recommendation:

The job creation function should also add the creator approval. Then, the job creator does not need to make another transaction to approve his own job.

Response:

This is not acknowledged as a bug but was instead a deliberate design choice. Any one of the approvers can create a job even if they disagree with it, followed by casting a disapproval vote and/or abstaining and allowing others to vote on the proposal.

EN-02: Inactive Core Contract Can be Target of an Upgrade

Severity: Enhancement

Description:

In the auth contract (auth.clar), upgrade-core-contract() and execute-upgrade-core-contract-job() does not check if the new core contract was already registered before. Therefore, an inactive core can be passed as an argument and then be activated. This would result in a poor experience because that core was shut down when it was replaced and most of the functions are disabled.

Recommendation:

Validate the new contract is not already registered in the auth contract.

Response:

This is acknowledged as a bug in the auth contract, however a valid core contract does not contain a function to reverse the variable isShutdown set by the shutdown-contract() function and would fail in calling it a second time.

The suggested mitigation was applied in commit b25b3eeba14a6a1588708ecf909ac237676893c2 and unit tests were expanded to cover this scenario.

Other Considerations

Description:

The burn function (token.burn()) allows a core contract to burn users' tokens without their consent. The audited core contract does not implement any function which calls token.burn(), but a new core contract can be included and call that function.

Response:

The burn function token.burn() was updated in version 1.0.1 to use the same authentication method as the transfer function token.transfer().

For any version 1.0.0 contracts deployed with this configuration, please note that:

- the original thought was to protect the burn function with the same guards as minting, which is only available to a core contract
- it is acknowledged that an upgraded core contract could implement this function, however the correct post conditions would have to be set by the user in order for it to succeed
- future deployments are unaffected, and future upgraded versions deployed per the protocol will follow the same standard as version 1.0.1 for burning tokens