



COINSPeCT

You build, we defend.



Flare
rNat Contracts
Smart Contract Audit



rNat Contracts Smart Contract Security Review

Version: v240701

Prepared for: Flare

July, 2024

Security Assessment

1. Executive Summary
2. Summary of Findings
 - 2.3 Solved issues & recommendations
3. Scope
4. Assessment
 - 4.1 Security assumptions
 - 4.2 Decentralization
 - 4.3 Testing and Code quality
5. Detailed Findings
 - FLRN-001 - Incorrect withdrawal calculations when rNat and WNat decimals don't match

FLRN-003 - Adversaries can take-over the RNatAccount implementation

FLRN-005 - Withdraw function allows the owner to pull all native token balance when zero amount is specified

6. Disclaimer

1. Executive Summary

In **June 2024**, **Flare** engaged **Coinspect** to perform a source code review of the reward token feature. The objective of the project was to evaluate the security of the Solidity smart contracts that add a new rewarding mechanism based on a yearly vesting schedule.

The **RNat** contract enables the claiming process and also serves as a non-transferrable soulbound token. This new mechanism works in tandem with the **RNatAccount**, a contract deployed for each reward recipient, in charge of storing each recipient's rewards and performing vesting calculations.

 Solved	 Caution Advised	 Resolution Pending
High 0	High 0	High 0
Medium 0	Medium 0	Medium 0
Low 1	Low 0	Low 0
No Risk 2	No Risk 0	No Risk 0
Total 3	Total 0	Total 0

2. Summary of Findings

2.3 Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
FLRN-001	Incorrect withdrawal calculations when rNat and WNat decimals don't match	Low
FLRN-003	Adversaries can take-over the RNatAccount implementation	None
FLRN-005	Withdraw function allows the owner to pull all native token balance when zero amount is specified	None

3. Scope

The scope was set to be the repository at <https://gitlab.com/flarenetwork/flare-smart-contracts-v2/-/tree/rNat/contracts/rNat>, branch `origin/rNat` at commit `62d3bcc8bfa47c7a492fa01285677170a24cead2`.

4 Assessment

The **RNat** contracts provide a new rewarding flow on top of Flare existing ones (e.g. via its main protocol through delegation rewards). This functionality is provided by a set of two contracts, **RNat** and **RNatAccount** and allow users to receive rewards released by a vesting mechanism.

This feature is implemented in a way that enables multiple whitelisted projects to distribute rewards. Rewards are expressed in **rNat**, a soulbound non-transferrable token. This token is backed by the same amount of **wNat**, which is only released through the yearly vesting mechanism. Each project assigns the reward amount in **rNat** on a monthly basis, as a consequence, each reward time-frame is unlocked in a 1/12 fraction after each month. When a year passes all the rewards received on that month are unlocked. Users can release the locked **wNat** after each month, if there are unlocked tokens after that period. This claiming process burns the **rNat** and transfers its backing **wNat** to the reward owner. As an alternative claiming flow, reward owners can decide to get all their **rNat** tokens burned, at the expense of losing half of their locked balance (penalty). All the tokens and vesting logic are located in the **RNatAccount** contract, which is a clone of a library deployed by the **RNat** for each reward recipient. External actors cannot modify its implementation as **RNat** creates the clones from an existing contract set by the manager, and then storing each contact's address.

One possible execution flow of this new feature works as follows. The **RNat** manager specifies and adds one or many allowed projects into the **RNat** contract and then sets the allowed amount of **rNat** to distribute by each project. Once rewards are allocated at a project level, the privileged administrator of each project (called *distributor*) specifies the amount of rewards for each user. Later, reward recipients are allowed to claim their **rNat** only past or current months (they not allowed to claim for future months). The **rNat** balance is allocated to each recipient's **RNatAccount**. Also, the claiming process generates **wNat** balance into the **RNatAccount**. However, this balance cannot be transferred out regularly, as it is backing the **rNat**. After each month, recipients are allowed to get their backing token (**wNat**) in a 1/12 fraction completing the vesting process after a year.

4.1 Security assumptions

The project assumes that neither the manager nor any of the project's distributors will act rogue. Also, malicious upgrades made to other protocol's contracts are assumed to be prevented at a different access control layer.

Additionally, it is assumed that since **WNat** tokens are going to be vested, no delegation rewards could be received by the balance locked into each **RNatAccount**. Reward recipients will have to withdraw the **WNat** tokens from each account and then perform the delegation by themselves, from outside the account contract.

4.2 Decentralization

This project includes different layers of centralization. The manager is the most over-powered role and is able to: update, create, enable/disable projects and assign/unassign rewards. Then, each project's distributor is in charge of specifying the **rNat** amount each recipient will receive for each month. Lastly, each recipient is able to withdraw ERC20 tokens from their **RNatAccount** (besides **WNat**), claim rewards and withdraw them.

Coinspect identified that actions performed by the manager that are non time-sensitive could be put behind a timelock. In that case, time-locking those methods that could be used as a countermeasure in the event of an adversarial scenario (e.g. disable a claiming schedule for projects) must be avoided.



4.3 Testing and Code quality

Several tests for the new contracts were added to the repository, this allowed Coinspect to quickly test new cases and scenarios. The code quality has high, easy to read and understand. It includes NatSpec on its core functions easing the process of understanding the functionality.

5. Detailed Findings

FLRN-001

Incorrect withdrawal calculations when rNat and WNat decimals don't match

Status Solved	Risk Low
	
Resolution Fixed	Impact Low Likelihood Medium

Location

```
./contracts/rNat/implementation/RNatAccount.sol:104  
./contracts/rNat/implementation/RNat.sol:123
```

Description

Each **RNatAccount** relies on the assumption that both the **rNat** and **wNat** (backing, valuable token) have the same decimals. As a consequence, withdrawal calculations can be disrupted if **rNat** and **WNat** tokens have different decimal places.

This happens because many calculations assume both tokens have the same number of decimals, such as **WNat**'s 18 decimal places. When **rNat** has less decimals than **sWNat**, the resulting calculations will be inaccurate, leading to errors in the withdrawal process:

```
uint128 rNatRewardsBalance = receivedRewards - withdrawnRewards;
uint128 balance = uint128(_wNat.balanceOf(address(this)));
assert(balance >= rNatRewardsBalance);
uint256 locked = _lockedRewards(_firstMonthStartTs);
require(balance - locked >= _amount, "insufficient balance");
// withdraw RNat rewards last, only if needed
_withdrawnRewards = balance - rNatRewardsBalance >= _amount ? 0 :
_amount - (balance - rNatRewardsBalance);
withdrawnRewards += _withdrawnRewards;
```

Coinspect observed that the testing suite uses a random number (5) for the **rNat** decimals, and the token's deployment has no way to ensure that its decimals will match **WNat**'s:

```
constructor(
    IGovernanceSettings _governanceSettings,
    address _initialGovernance,
    address _addressUpdater,
    string memory _name,
    string memory _symbol,
    uint8 _decimals,
    address _manager,
    uint256 _firstMonthStartTs
)
    Governed(_governanceSettings, _initialGovernance)
    IncentivePoolReceiver(_addressUpdater)
{
    require(_firstMonthStartTs <= block.timestamp, "first month
start in the future");
    _checkNonzeroAddress(_manager);
    name = _name;
    symbol = _symbol;
    decimals = _decimals;
    manager = _manager;
    firstMonthStartTs = _firstMonthStartTs;
}
```

```
rNat = new RNat(
    IGovernanceSettings(makeAddr("governanceSettings")),
    governance,
    addressUpdater,
    "rTest",
    "rT",
    5,
    manager,
    500
);
```

As the expected flow of **WNat** into each account's contract is controlled by the claiming process meaning that a 1:1 relationship is established, this issue is considered to have low impact.

Recommendation

Since the decimals of the **WNat** token are already known, set the **rNat** decimals as a constant and equal to **WNat**'s.


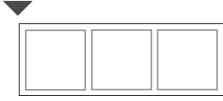
Status

Fixed on commit [115a99beaf885ab2c03734656efd083f3ee0cc8e](#).

A deployment script for **RNat** was provided and its decimals are set directly using `wNat.decimals()`. Additionally, Flare added a check to ensure that this relationship matches when updating the contract addresses.

FLRN-003

Adversaries can take-over the RNatAccount implementation

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -

Location

`./contracts/rNat/implementation/RNatAccount.sol:56`

Description

Each reward recipient receives a clone of the **RNatAccount**, which is a minimal proxy with an isolated state that forwards calls to the account's implementation. The proxy deployment atomically initializes every user's account contract, preventing adversaries from taking-over each user's account:

```
// create RNat account
_rNatAccount = IIRNatAccount(payable(createClone(libraryAddress)));
require(!_isContract(address(_rNatAccount)), "clone not created
successfully");
_rNatAccount.initialize(msg.sender, this);
```

However, the deployment process of the implementation does not perform its initialization, leaving the contract uninitialized thus allowing adversaries to take this contract over:

```
function initialize(  
    address _owner,  
    IRNat _rNat  
)  
    external  
{  
    require(address(owner) == address(0), "owner already set");  
    require(address(_owner) != address(0), "owner address zero");  
    require(address(_rNat) != address(0), "rNat address zero");  
    owner = _owner;  
    rNat = _rNat;  
    emit Initialized(owner, _rNat);  
}
```

Since this contract has no features that allow the owner to either destroy or upgrade this implementation, this issue has no risk.

Recommendation

Disable the implementation's initialize function directly through its constructor.



Status

Fixed on commit [115a99beaf885ab2c03734656efd083f3ee0cc8e](#).

A deployment script for `rNatAccount` was added where `initialize()` is called atomically after its deployment, preventing the mentioned issue.

FLRN-005

Withdraw function allows the owner to pull all native token balance when zero amount is specified

Status Solved	Risk None
	
Resolution Acknowledged	Impact Recommendation
	Likelihood -

Location

`./contracts/rNat/implementation/RNatAccount.sol:112`

Description

Reward recipient owners can abuse from the withdraw method from their **RNatAccounts** to drain all the **Nat** token balance. This happens because native token transfers send all the contract's balance to the owner and zero amount can be passed as a parameter when withdrawing **rNat**:

```
if (!_wrap) {
    disableAutoWrapping = true;
    _wNat.withdraw(_amount);
    disableAutoWrapping = false;
    _transferCurrentBalanceToOwner();
}
```

Then, `_transferCurrentBalanceToOwner()`:

```
function _transferCurrentBalanceToOwner() internal {
    uint256 balance = address(this).balance;
    if (balance > 0) {
        /* solhint-disable avoid-low-level-calls */
        //slither-disable-next-line arbitrary-send-eth
        (bool success, ) = owner.call{value: balance}("");
        /* solhint-enable avoid-low-level-calls */
        require(success, ERR_TRANSFER_FAILURE);
    }
}
```

Although **RNat** accounts are expected to have **WNat** balance instead, this alternative path allows owners to withdraw the contract's native balance and also make arbitrary calls. The latter, could be abused to trigger reentrant calls at a zero cost to any contract if the owner is a contract that implements malicious logic into their receive/fallback functions.

Recommendation

Revert withdrawals that set zero as the amount. Alternatively, document this potential adversarial scenario if this path is intended.

Status

Acknowledged.

The Flare Team added a comment to relevant functions clarifying the usage of the withdraw function and this alternative path.

6. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover out of scope systems, nor the general operational security of the organization that developed the code.