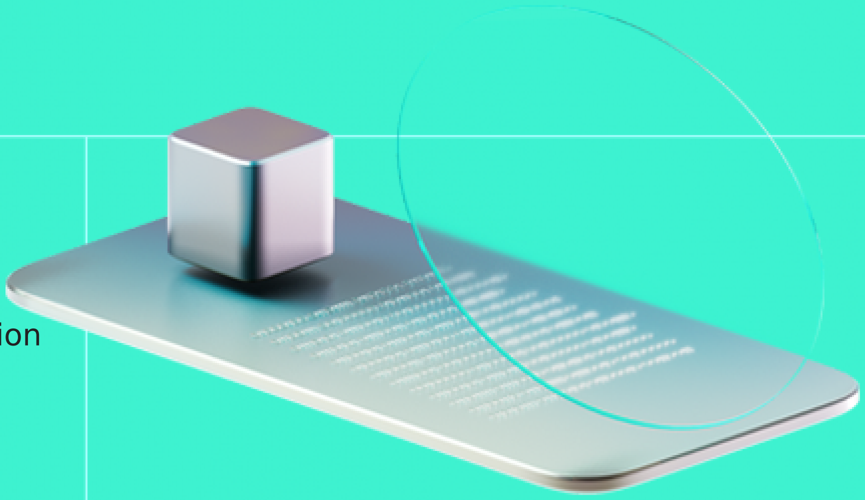
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Vechain Foundation

Date: 26/06/2025

We express our gratitude to the Vechain Foundation team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

The VeChain Stargate smart-contracts suite is a key pillar of the upcoming Hayabusa hard fork in VeChain network, supplying the on-chain framework for delegating stake to validator nodes and for managing exist and future rewards.

Document

Name	Smart Contract Code Review and Security Analysis Report for Vechain Foundation
Audited By	Stepan Chekhovskoi, Bohdan Hrytsak
Approved By	Ataberk Yavuzer, Seher Saylik
Website	https://www.vechain.org
Changelog	09/06/2025 - Preliminary Report
	26/06/2025 - Final Report
Platform	VeChain
Language	Solidity
Tags	Liquid Staking, Non-Fungible Token, Staking
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/vechain/stargate
Initial Commit	c49487aedc30830feca8f208bd7bdbfa70246c4d
Remediation Commit	86296662f1c7f10f7844fc7bb90e59a84d615fcd
Final Commit	f14509e12d407f725a7a57b1e6f071cc072d714b

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

13	11	2	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	1
Medium	2
Low	5

Vulnerability	Severity
F-2025-10841 - Claim Process DoS due to Reentrancy Protection	High
F-2025-10652 - Delegation on Behalf of Another User due to Ownership Check Bypass	Medium
F-2025-10805 - Estimated VTHO Reward Stop Timestamp May Miss Actual VeChain Network Fork Time	Medium
F-2025-10523 - Token URI ERC-721 Metadata Incompliance	Low
F-2025-10643 - Initializers Could Be Front-Run	Low
F-2025-10644 - Unexpected Return Values Caused by EIP-150 63/64 Gas-Rule in try/catch	Low
F-2025-10757 - Inconsistent and Always-True Boolean Return Pattern	Low
F-2025-10818 - Possible disabling the `isActive` flag for migrate tiers may cause false expectations	Low
F-2025-10633 - Revert Messages Optimization	Info
F-2025-10634 - Lack of Upgradeable Contracts Initialization	Info
F-2025-10815 - Lack of ERC20 Operation Success Validation	Info
F-2025-10820 - Useless Repeated Request Exit Calls	Info
F-2025-10840 - Unimplemented Scaled Rewards Functionality	Info

Documentation quality

- Functional requirements are shared internally.
    - Project overview is detailed.
    - Use-cases are provided.
- Technical description is provided.
    - Run instructions are provided.
    - Technical specification is provided.
    - The NatSpec documentation is partially inaccurate.

Code quality

- The code mostly follows best practices.
- The development environment is configured.
- Several redundant code patterns were found.
- Few TODO comments are identified.
- Missing events emit for certain configuration functions (`setVthoRewardPerBlockForAllLevels`, `setVthoRewardPerBlockForLevel`).
- Partial missing some input validation in critical functions.

Test coverage

Code coverage of the project is **92.9%** (branch coverage, without `NodeManagementV1` and `NodeManagementV2`).

- Deployment and user interactions are covered with tests.
- Some negative and edge cases coverage is missed.

Table of Contents

System Overview

The VeChain Stargate is an upgradeable system implementing special NFT Staking contains following contracts

`StargateNFT` - upgradeable ERC721 with pauseable functionality and governed by role-based access control. It is a continuation of the legacy Token Auction (X-Node and Eco Nodes) collection. This contract allows users to stake VET and receive an NFT in exchange, which represents a betting position. It also supports the claim of VTHO rewards for user stakes generated by the balance of the VET contract by the VeChain energy growth mechanism and the issuance of this reward to users.

- Uses `AccessControlUpgradeable` to manage roles.
- Inherit contracts from OpenZeppelin library:
  - `AccessControlUpgradeable`: Access control roles base system.
  - `ERC721Upgradeable`: Core ERC-721 implementation.
  - `ERC721EnumerableUpgradeable`: ERC-721 enumeration helpers.
  - `ERC721PausableUpgradeable`: Allows NFT transfers to be disabled while paused.
  - `ReentrancyGuardUpgradeable`: Helper for prevent reentrant calls to the stake / unstake / migrate / claim functionality.
  - `UUPSUpgradeable`: Support UUPS proxy mechanism.
- Uses the following protocol libraries to implement the main functionality:
  - `Clock`: Current block/timestamp/clock mode helper.
  - `DataTypes`: Data structures.
  - `Errors`: Custom errors definitions.
  - `Levels`: Levels management.
  - `MintingLogic`: Core stake/unstake & migration flow.
  - `Settings`: Contract-parameter management.
  - `Token`: Get tokens/users tokens data helper.
  - `VetGeneratedVtho`: Support dual-token VeChain economic model. Distribution generated VTH0 rewards.

`StargateDelegation` - upgradeable contract allows user that owns a StargateNFT start delegating, accumulate VTHO rewards with governed by role-based access control.

- Uses `AccessControlUpgradeable` to manage roles.
- Inherit contracts from OpenZeppelin library:
  - `AccessControlUpgradeable`: Access control roles base system.
  - `ReentrancyGuardUpgradeable`: Helper for prevent reentrant calls to the delegate / exit delegation / claim functionalities.
  - `UUPSUpgradeable`: Support UUPS proxy mechanism.

`NodeManagementV1` - upgradeable contract to manage node ownership and delegation within the VeBetter DAO ecosystem.

- Uses `AccessControlUpgradeable` to manage roles.

- Inherit contracts from OpenZeppelin library:
    - `AccessControlUpgradeable`: Provides access control roles base system.
    - `UUPSUpgradeable`: Provides functionality for support UUPS proxy mechanism.

`NodeManagementV2.sol` - Update for `NodeManagementV1` version, which includes new helper functions to review delegator and node state.

`NodeManagementV3.sol` - Update for `NodeManagementV2` version, which also includes support for `StargateNFT` node holders.

Privileged roles

`StargateNFT`

- `DEFAULT_ADMIN_ROLE`:
    - Super admin role that can grant / revoke other roles and manage the overall access control system.
    - Can configure the addresses of other contracts that are used, such as: `stargateDelegation` (Stargate Delegation), `legacyNodes` (Token Auction), `vthoToken` (VTHO token).
    - Can set VTHO rewards generation end timestamp (estimated Hayabusa hard fork in VeChain timestamp).
- `UPGRADER_ROLE`:
    - Can change proxy implementation.
- `PAUSER_ROLE`:
    - Can change the pause state of the contract, to disallow next actions from user during pause: `stake`, `stakeAndDelegate`, `migrate`, `migrateAndDelegate`, `unstake`, `claimVetGeneratedVtho`.
- `LEVEL_OPERATOR_ROLE`:
    - Can add / update / deactivate / activate the NFT levels (tiers) and their parameters.
- `MANAGER_ROLE`:
    - Can change base URI for token collection metadata.

`StargateDelegation`

- `DEFAULT_ADMIN_ROLE`:
    - Super admin role that can grant / revoke other roles and manage the overall access control system.
    - Inherit the permissions of other roles:
        - Can change current proxy implementation.
        - Can change the value of the block reward for NFT levels.
        - Can set the rewards accumulation end block.
- `UPGRADER_ROLE`:
    - Can change proxy implementation.
- `OPERATOR_ROLE`:
    - Can change the value of the block reward for NFT levels.
    - Can set the rewards accumulation end block.

`NodeMaangementV1`

- `DEFAULT_ADMIN_ROLE`:
  - Super admin role that can grant / revoke other roles and manage the overall access control system.
  - Can configure the address of VeChain Nodes contract.
- `UPGRADER_ROLE`:
  - Can change proxy implementation.

`NodeMaangementV2`

- `DEFAULT_ADMIN_ROLE`:
  - Super admin role that can grant / revoke other roles and manage the overall access control system.
  - Can configure the address of VeChain Nodes contract.
- `UPGRADER_ROLE`:
  - Can change proxy implementation.

`NodeMaangementV3`

- `DEFAULT_ADMIN_ROLE`:
  - Super admin role that can grant / revoke other roles and manage the overall access control system.
  - Can configure the address of VeChain Nodes contract.
  - Can configure the address of `StargateNFT` contract.
- `UPGRADER_ROLE`:
  - Can change proxy implementation.
  - Can initialize v3 parameters (`StargateNFT` contract address).

Potential Risks

- **Single Points of Failure and Control**: The project is fully or partially centralized, introducing single points of failure and control. Owner is able to change various parameters including critical configuration which may cause denial of the system. This centralization can lead to vulnerabilities in decision-making and operational processes, making the system more susceptible to targeted attacks or manipulation.
- **Flexibility and Risk in Contract Upgrades**: The project contracts are upgradeable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.
- **Dynamic Array Iteration Gas Limit Risks**: The project iterates over large dynamic arrays, which leads to excessive gas costs, risking denial of service due to out-of-gas errors, directly impacting contract usability and reliability.
- **Absence of Emergency Withdraw Functionality**: The contract is designed in a centralized way leaving all the controls to the system management team. In case of emergency situation, the team is expected to pause the contract and manage the situation. The users are not provided with emergency mechanism to act independently in critical situations.
- **System Reliance on External Contracts**: The functioning of the system relies on Token Auction contract external contract. The contract is responsible for Legacy NFT generation and further creation of Legacy NFTs may cause various system inconsistencies including system DoS requiring contract upgrade. Any flaws or vulnerabilities in these contracts adversely affect the audited project, potentially leading to security breaches or loss of funds.

Findings

Vulnerability Details

F-2025-10841 - Claim Process DoS due to Reentrancy Protection - High

Description: The `claimRewards` function of the `StargateDelegation` contract is implemented under the `nonReentrant` modifier. The function is invoked in the `delegate` function which is also implemented under the modifier causing the transaction revert.

This may lead to temporary denial of service situation for the delegate process confusing the end users.

```
function delegate(
    uint256 _tokenId,
    bool _delegateForever
) external nonReentrant returns (bool success) {
    ...

    if (claimableRewards(_tokenId) > 0) {
        claimRewards(_tokenId);
    }

    ...
}

function claimRewards(uint256 _tokenId) public nonReentrant returns (bool suc
cess) { ... }
```

Assets:

- StargateDelegation/StargateDelegation.sol [https://github.com/vechain/stargate-staking]

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation: Consider implementing `internal` function without the `nonReentrant` modifier containing the `claimRewards` logic and using it within the `delegate` and `claimRewards` functions.

```
function claimRewards(uint256 _tokenId) public nonReentrant returns (bool suc
cess) {
    return _claimRewards(_tokenId);
}

function delegate(
    uint256 _tokenId,
    bool _delegateForever
) external nonReentrant returns (bool success) {
    ...

    if (claimableRewards(_tokenId) > 0) {
        _claimRewards(_tokenId);
    }

    ...
}

function _claimRewards(uint256 _tokenId) internal returns (bool success) {
    ...
}
```

Resolution: The finding is fixed in the `beefe3b` commit. The `internal` function wrapping the `claimRewards` logic is implemented.

Evidences

PoC

Reproduce:

1. Stake VET and delegate minted NFT using the `StargateDelegation` contract
2. Exit delegation with `requestDelegationExit` function

3. Delegate the NFT second time without manual `claimRewards` invocation
4. Face the `ReentrancyGuardReentrantCall` error

```
    function test_failedDelegateIfRestRewardsNonReentrancyError() public {
        vm.startPrank(admin);
        stargateDelegation.setRewardsAccumulationEndBlock(block.number + 1000
000);
        vm.stopPrank();

        uint8 level = 1;
        uint256 amount = stargateNFT.getLevel(level).vetAmountRequiredToStake
;
        vm.deal(user1, amount);
        vm.startPrank(user1);
        (, uint256 tokenId) = stargateNFT.stakeAndDelegate{value: amount}(lev
el, true);

        vm.roll(stargateNFT.maturityPeriodEndBlock(tokenId) + 100);

        stargateDelegation.requestDelegationExit(tokenId);

        vm.roll(stargateDelegation.getDelegationEndBlock(tokenId));

        vm.startPrank(user1);
        vm.expectRevert(ReentrancyGuardUpgradeable.ReentrancyGuardReentrantCa
ll.selector);
        stargateDelegation.delegate(tokenId, false);
    }
```

F-2025-10652 - Delegation on Behalf of Another User due to Ownership Check Bypass - Medium

Description:

The `stakeAndDelegate` and `migrateAndDelegate` functions in `StargateNFT` mint a token and delegate it in the same transaction. Minting flows through `_safeMintCallback() → ERC721._safeMint()`, which triggers `onERC721Received`. Within that hook the recipient can execute arbitrary logic — e.g. transfer the freshly minted token to someone else — before `delegate()` is called.

Because `delegate()` is invoked by the StargateNFT contract itself, the usual ownership check is bypassed:

```
function delegate(...
    address owner = $.stargateNFT.ownerOf(_tokenId);
    if (owner != msg.sender && msg.sender != address($.stargateNFT)) {
        revert UnauthorizedUser(msg.sender);
    }
```

Even if `ownerOf` has already changed, the call still succeeds since `msg.sender` equals `StargateNFT`.

The original staker/migrator can delegate (and thus lock) a token they no longer own, preventing the unsuspecting new holder from transferring or unstaking it until the delegation ends.

Although the practical impact is currently limited — the staker/migrator pays the mint fee and is allowed to act during minting — the behavior breaks the intuitive assumption that, once an NFT is transferred, no further actions can be taken without the new owner's consent.

This case may occur within the scope of an NFT sale in certain third-party protocol implementations, which would also violate the description under the `Inheriting Staked VET` and `Transferring NFTs` documentation section.

Call sequence:

```
User A ──▶ StargateNFT.stakeAndDelegate
        ├─ _stake
        │   └─ _safeMintCallback
        │       └─ _safeMint
        │           └─ onERC721Received → transfer token to User B
        └─ StargateDelegation.delegate()  // passes check: msg.sender == Star
```

```
gateNFT
Outcome: User B now owns the NFT, but User A has delegated (locked) it.
```

Assets:

- StargateNFT/StargateNFT.sol [https://github.com/vechain/stargate-staking]
- StargateDelegation/StargateDelegation.sol [https://github.com/vechain/stargate-staking]
- StargateNFT/libraries/MintingLogic.sol [https://github.com/vechain/stargate-staking]

Status: Fixed

Classification

Impact Rate:	3/5
Likelihood Rate:	3/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Medium

Recommendations

Remediation: One possible solution is add a `require` for check change owner before calling `delegate()` within the corresponding `StargateNFT` methods, which in turn will not allow change ownership during `stakeAndDelegate` and `migrateAndDelegate` calls

```
 function migrateAndDelegate(...
     success = _migrate($, _tokenId);
     ...

+    address owner = IStargateNFT(address(this)).ownerOf(_tokenId);
+    if (owner != msg.sender) { // will revert if token owner changed during m
igrate proccess
+        revert Errors.NotOwner(_tokenId, msg.sender, owner);
+    }

     ...
     // Delegate the token
     success = $.stargateDelegation.delegate(_tokenId, _autorenew);
     ...
}
```

```
function stakeAndDelegate(...
    (success, tokenId) = _stake($, _levelId);
    ...

+   address owner = IStargateNFT(address(this)).ownerOf(_tokenId);
+   if (owner != msg.sender) { // will revert if token owner changed during s
take proccess
+       revert Errors.NotOwner(_tokenId, msg.sender, owner);
+   }

    ...
    // Delegate the token
    success = $.stargateDelegation.delegate(tokenId, _autorenew);
    ...
}
```

Resolution:

The Finding is resolved according the commit `8629662`. The `require` checks for additional checking for change of ownership after stake/migrate have been added, which makes it impossible to transfer nft in the operations of calling `stakeAndDelegate`, and `migrateAndDelegate` without affecting calls without delegation.

Evidences

PoC

Reproduce:

The tests below are intended to show that tokens are transferred to the delegated status even if the token owner has changed

```
  describe("Delegate for not own token during mint", async () => {
    it("stakeAndDelegate", async () => {
      let stakeUtility = await ethers.deployContract("StakeUtilityMock", [
        user2,
        stargateNFT,
        stargateDelegation,
      ]);
      const tokenId = (expectedTokenId = config.LEGACY_LAST_TOKEN_ID + 1);
      const levelId = TokenLevelId.Thunder;
      const levelSpec = await stargateNFT.getLevel(levelId);
      console.log(`Caller/Staker: ${user1.address}`);
      await stakeUtility
        .connect(user1)
        .stakeAndDelagate(levelId, false, { value: levelSpec.vetAmountRequire
dToStake });
```

```
      console.log(`Finally token owner(${tokenId}):`, await stargateNFT.owner
Of(tokenId));
      console.log(
        `isDelegationActive(${tokenId}):`,
        await stargateDelegation.isDelegationActive(expectedTokenId)
      );
    });
    it("migrateAndDelegate", async () => {
      const levelSpec = await stargateNFT.getLevel(StrengthLevel.MjolnirX);
      console.log(`Caller/Migrator: ${user1.address}`);

      let expectUtilityContractAddress = ethers.getCreateAddress({
        from: user1.address,
        nonce: (await user1.getNonce()) + 1,
      });
      await legacyNodes
        .connect(user1)
        .transferFrom(user1, expectUtilityContractAddress, legacyTokenId);
      let stakeUtility = await ethers.deployContract(
        "StakeUtilityMock",
        [user2, stargateNFT, stargateDelegation],
        user1
      );
      await stakeUtility
        .connect(user1)
        .migrateAndDelegate(legacyTokenId, false, { value: levelSpec.vetAmoun
tRequiredToStake });
      console.log(
        `Finally token owner(${legacyTokenId}):`,
        await stargateNFT.ownerOf(legacyTokenId)
      );
      console.log(
        `isDelegationActive(${legacyTokenId}):`,
```

See more

Results:

```
 @repo/contracts:test:hardhat:     Delegate for not own token during mint
@repo/contracts:test:hardhat: Caller/Staker: 0x70997970C51812dc3A010C7d01b50e
0d17dc79C8
@repo/contracts:test:hardhat: Finally token owner(6): 0x3C44CdDdB6a900fa2b585
dd299e03d12FA4293BC
@repo/contracts:test:hardhat: isDelegationActive(6): true
@repo/contracts:test:hardhat:       ✔ stakeAndDelegate
@repo/contracts:test:hardhat: Caller/Migrator: 0x70997970C51812dc3A010C7d01b5
0e0d17dc79C8
@repo/contracts:test:hardhat: Finally token owner(1): 0x3C44CdDdB6a900fa2b585
dd299e03d12FA4293BC
```

```
@repo/contracts:test:hardhat: isDelegationActive(1): true
@repo/contracts:test:hardhat:       ✔ migrateAndDelegate
```

F-2025-10805 - Estimated VTHO Reward Stop Timestamp May Miss Actual VeChain Network Fork Time - Medium

Description:

During the Hayabusa fork VeChain will stop rewards VTHO for VET holders. The **fork is triggered by block number**, so the exact UNIX-timestamp of the fork block cannot be known beforehand; it depends on block-time variance, validator behaviour, network latency, etc.

The `StargateNFT` contract exposes a variable `vthoGenerationEndTimestamp` (UNIX-timestamp of the time stop VHTO accumulate for VET holders) that must setup with exact fork timestamp. Because the timestamp can be set either before or after the fork actually happens, several failure modes arise:

Scenario	Condition	Impact
Pre-set, too late	`vthoGenerationEndTimestamp` > real fork timestamp	Calculate rewards continue after VTHO minting ends → phantom VTHO accrues, `StakingNFT` become insolvent when the last stakers call `claimRewards`/`unstake`.
Pre-set, too early	`vthoGenerationEndTimestamp` < real fork timestamp	Rewards stop prematurely → users are under-paid; a portion of genuine VTHO remains stranded in the contract.
Set after fork	Some users `unstake`/`claimRewards` after forking and before setting `vthoGenerationEndTimestamp`	Early exits receive excess VTHO; contract balance turns not enough, blocking last `unstake`/`claimRewards` calls for the last stakers.

As a result, the impact may be different: a surplus/deficit in the reward balance, the last users fail to `unstake` (which is unlikely), and failed `claimRewards` calls (very likely).

If `vthoGenerationEndTimestamp` is set ahead of time, the potential imbalance can be approximated by:

```
deficit_or_surplus ≈ totalVetBalance × (vthoGenerationEndTimestamp − actualHayabusaForkTimestamp) × 5e9
```

where `5 × 10⁹ wei` is the VTHO-generation rate per VET per second.

```
/// @notice Set the timestamp when the protocol will stop generating VTHO by holding VET
/// @dev By default it will be 0, meaning the protocol will keep generating V
THO indefinitely,
/// once the VeChain foundation will know the exact timestamp when the VTHO g
eneration will stop
/// we will set this value. It can be set back to 0, but then rewards will be
recalculated based on the
/// current timestamp. So set it back to 0 with caution.
/// @param _vthoGenerationEndTimestamp The timestamp when the protocol will s
top generating VTHO by holding VET
/// @dev Emits a {IStargateNFT.VthoGenerationEndTimestampUpdated} event
function setVthoGenerationEndTimestamp(
    uint48 _vthoGenerationEndTimestamp
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    DataTypes.StargateNFTStorage storage $ = _getStargateNFTStorage();
    Settings.setVthoGenerationEndTimestamp($, _vthoGenerationEndTimestamp);
}
```

Assets:

- StargateNFT/StargateNFT.sol [https://github.com/vechain/stargate-staking]
- StargateNFT/libraries/VetGeneratedVtho.sol [https://github.com/vechain/stargate-staking]
- StargateNFT/libraries/Settings.sol [https://github.com/vechain/stargate-staking]

Status: Fixed

Classification

Impact Rate:	3/5
Likelihood Rate:	4/5
Exploitability:	Semi-Dependent
Complexity:	Simple
Severity:	Medium

Recommendations

Remediation: A simple fix is to change the reward-suspension approach: pause contract activity a short while before the fork and resume it after the fork finishes. During this pause, the protocol admin can safely update `vthoGenerationEndTimestamp` to the exact value, avoiding any imbalance.

The pause can be triggered automatically, using block numbers (e.g., “N blocks before the fork” and “M blocks after”), or manually a set time ahead of the fork.

Consider documenting the strategy for staking finalization on Hayabusa hard fork.

To prevent mistakes, allow `setVthoGenerationEndTimestamp` to be called only while the contract is paused — ensuring the value is adjusted in a controlled state:

```
  function setVthoGenerationEndTimestamp(
      uint48 _vthoGenerationEndTimestamp
-) external onlyRole(DEFAULT_ADMIN_ROLE) {
+) external whenPaused onlyRole(DEFAULT_ADMIN_ROLE) {
```

Resolution: The Finding is resolved in the commit `8629662`. The documentation now explicitly states that the contract will be paused before the Hayabusa hard fork, and the `setVthoGenerationEndTimestamp` function is restricted to execution only while the contract is paused.

Evidences

Current Hayabuse Fork implementation

Reproduce:

Branch: `release/hayabusa`

Network Core: https://github.com/vechain/thor/

The moment of determining the end of energy rewards (VHTO) occurs at the moment when `PoS` is synchronized:

```
 posActive, activated, activeGroup, err := c.syncPOS(staker, header.Number())
...
if activated {
    builtin.Energy.Native(state, parent.Timestamp()).StopEnergyGrowth()
}
```

The corresponding timestamp will be determined only when the block is validated, without knowing in advance what this timestamp will be.

F-2025-10523 - Token URI ERC-721 Metadata Incompliance - Low

Description:

The `StargateNFT` contract is an upgradeable ERC-721 implementation. The contract declares `tokenURI` function meant to return the metadata URI for an NFT by the token id.

According to EIP-721 standard, the function should throw an error, in case specified token id is invalid. However, instead of reverting, the function returns an empty string violating the standard.

This may lead to various external services incorrectly process the token existence checks.

```
function tokenURI(
    uint256 _tokenId
) public view override(ERC721Upgradeable) returns (string memory) {
    DataTypes.StargateNFTStorage storage $ = _getStargateNFTStorage();

    // Get token level ID
    uint8 levelId = $.tokens[_tokenId].levelId;

    // If token level ID is 0, return empty string
    if (levelId == 0) {
        return "";
    }

    return Token.tokenURI($, _tokenId, Strings.toString(levelId));
}
```

Assets:

- StargateNFT/StargateNFT.sol [https://github.com/vechain/stargate-staking]

Status: Fixed

Classification

Impact Rate:	2/5
Likelihood Rate:	3/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: Consider implementing the token id validation.

For example, the standard implementation of ERC-721 from OZ provides a check for the presence of the owner available in the contract by inheritance.

```
function tokenURI(
    uint256 _tokenId
) public view override(ERC721Upgradeable) returns (string memory) {
    _requireOwned(_tokenId);

    DataTypes.StargateNFTStorage storage $ = _getStargateNFTStorage();
    ...
```

Resolution: The finding is **resolved** in the commit `bb9e149`. The `_requireOwned` helper is invoked within the `tokenURI` function.

F-2025-10643 - Initializers Could Be Front-Run - Low

Description:

In Solidity, initializer functions in upgradeable contracts configure state and run critical setup during deployment. If left unprotected, they can be front-run — called by an attacker before the legitimate deployer — which creates a risk of the contract takeover and lead to need of redeployment.

Modern proxy patterns allow to initialize the contract in same transaction with deployment. However, the deployment scripts in the project deploy and initialize the contracts in different transactions. Additionally, in the `StargateDelegation`, `StargateNFT`, `NodeManagementV1-V3` contracts, the `initialize` functions are public and are not protected with access control, which keeps the chance for front-run initialization to occur during setup process.

```
 function initialize(
    DataTypes.StargateNFTInitParams memory _initParams
) external initializer { ... }

function initialize(
    StargateDelegationInitParams memory _initParams
) external initializer { ... }

function initialize(
    address _vechainNodesContract,
    address _admin,
    address _upgrader
) external initializer { ... }
```

Assets:

- StargateNFT/StargateNFT.sol [https://github.com/vechain/stargate-staking]
- StargateDelegation/StargateDelegation.sol [https://github.com/vechain/stargate-staking]
- NodeManagement/NodeManagementV2.sol [https://github.com/vechain/stargate-staking]
- NodeManagement/NodeManagementV3.sol [https://github.com/vechain/stargate-staking]
- NodeManagement/NodeManagementV1.sol [https://github.com/vechain/stargate-staking]

Status: Accepted

Classification

Impact Rate:	2/5
Likelihood Rate:	2/5
Exploitability:	Independent
Complexity:	Medium
Severity:	Low

Recommendations

Remediation: Consider passing the initializer calldata to the proxy constructor. It delegates the call to the implementation within the same deployment transaction, leaving no opportunity for front-running.

Resolution: The client has acknowledged the finding with the following statement:

> If the initialization is frontrun the transaction will be reverted and we will redeploy the contract.
> And we cannot protect the initialize with access control since of course the contract was not yet
> initialized though it does not have any role assigned.
> We will keep this as is.

F-2025-10644 - Unexpected Return Values Caused by EIP-150 63/64 Gas-Rule in try/catch - Low

Description: In the `NodeManagementV3` contract, some functions derive data from both the `VeChainNodes` contract and `StargateNFT`. These functions wrap the `StargateNFT` or `VeChainNodes` call inside a `try/catch` block: if the call reverts, the catch branch returns default or empty values or determines the data in another way.

Because of EIP-150’s 63⁄64 gas-forwarding rule, there are cases where manipulation of gas will cause the `gasLimit` value of a transaction to be so low that the internal call will receive too little gas and will be returned even when it would otherwise be successful. The outer function interprets this forced revert as “NFT/node does not exist” and returns a false-negative or different result and creates a dependence of the final result on the value of gas, which is not the right approach.

Affected methods are

- `getNodeManager(uint256 nodeId)`
- `isNodeManager(address user, uint256 nodeId)`
- `getNodeLevel(uint256 nodeId)`
- `isLegacyNode(uint256 nodeId)`
- `getUserNodes(address user)`
- `isDirectNodeOwner(address user, uint256 nodeId)`

Impact of pushing the functions return false-negative result is considered to be limited within the system. However, this may potentially have an impact on the integration side, or future updates that use the functions for authorization or other validations.

Assets:

- NodeManagement/NodeManagementV3.sol [https://github.com/vechain/stargate-staking]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 3/5

Exploitability: Semi-Dependent

Complexity: Medium

Severity: Low

Recommendations

Remediation:

Replacing the `try/catch` wrappers with additional checks that do not lead to a failure, or explicitly expecting a failure instead `return` of `false/address(0)/etc` may be a suitable alternative.

Consider implementing `exists(uint256 tokenId) returns (bool)` function in `StargateNFT` validating if token exists and ensuring `ownerOf` function execute without revert.

```
function exists(uint256 tokenId) external view returns (bool) {
    address owner = _ownerOf(tokenId);
    if (owner == address(0)) {
        return false;
    }
    return true;
}
```

Call the function prior to `stargateNft.ownerOf()` in the `NodeManagementV3` contract avoiding need of `try/catch` blocks.

Resolution:

The Finding is fixed according commit `8629666`. Implemented additional methods (`NodeManagementV3.exists,` `StargateNFT.tokenExists`) to simplify the token existence check, which allowed to remove try/catch calls and avoid of abuse cases.

Evidences

PoC

Reproduce:

This test is to show how the amount of gas changes the return value to a completely different value than expected.

In the case of `isNodeManager`, which should return true indicating that it is the owner of the node, return false even though the passed address is the owner of this node

```
 const levelId = TokenLevelId.Thunder;
const levelSpec = await stargateNFT.getLevel(levelId);
await stargateNFT.connect(user1).stake(levelId, { value: levelSpec.vetAmountR
equiredToStake });
console.log("without gas limit:", await nodeManagement.isNodeManager(user1, e
xpectedTokenId));
console.log(
    "estimateGas:",
    await nodeManagement.isNodeManager.estimateGas(user1, expectedTokenId)
);
console.log(
    "with gas limit (49129):",
    await nodeManagement.isNodeManager(user1, expectedTokenId, { gasLimit: 49
129 })
);
```

Results:

```
 @repo/contracts:test:hardhat: without gas limit: true
@repo/contracts:test:hardhat: estimateGas: 51933n
@repo/contracts:test:hardhat: with gas limit (49129): false
```

F-2025-10757 - Inconsistent and Always-True Boolean Return Pattern - Low

Description:

Some methods are designed with a soft failure pattern - they return a `bool success` value.

However, all failure paths immediately `revert`, which means:

- `bool success` is always true on any reachable return path.
- the value `false` is unreachable.
- NatSpec comments contradict runtime behavior (some promise `false`, others promise a revert).

That is, in the current implementation, the returned `bool` does not carry any additional information: success is already confirmed by the absence of a revert, and failure will never be signaled by false. This makes the return type misleading and creates side effects:

- False expectations of integrators.
- Reduced readability: it's harder for developers to understand at a glance whether to catch a false or handle an exception.
- Possible future updates that start returning false may be a silent breaking change.

The same pattern in early ERC-20 tokens prompted the ecosystem to create wrappers like `SafeERC20`, showing the real-world friction such ambiguity causes.

Affected Methods:

- `StargateNFT`: `stake()`, `stakeAndDelegate()`, `migrate()`, `migrateAndDelegate()`, `unstake()`, `claimVetGeneratedVtho()`
- `StargateDelegation`: `delegate()`, `requestDelegationExit()`, `claimRewards()`

One of the important cases is where the NatSpec explicitly specifies returning `false` on failure, but this never happens because the only approach with a revert is used:

```
/// @return success - true if the delegation was started successfully, false otherwise
function delegate(
    uint256 _tokenId,
    bool _delegateForever
) external nonReentrant returns (bool success) { ... }

/// @return success - true if the delegation was exited successfully, false otherwise
```

```
function requestDelegationExit(
    uint256 _tokenId
) external nonReentrant returns (bool success) { ... }
```

Assets:

- StargateNFT/StargateNFT.sol [https://github.com/vechain/stargate-staking]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Consider removing the excessive return value or eliminate the NatSpec contradiction.

1. Use a single unified approach to public methods, the recommendation is to use reverts on fail case, without `bool success` return value where it is clearly not required.
2. Align NatSpec with implementation – update comments so they no longer promise contradictory behavior (where they mention return `false` when not successful, but in fact always revert).

Resolution: The Finding is fixed according commit `8629666`. The corresponding methods no longer return a success status but always reverted tx on a fail flow.

Evidences

Affected Methods

Reproduce:

`StargateDelegation.sol`:

```
 /// @return success - true if the delegation was started successfully, false
otherwise
function delegate(
        uint256 _tokenId,
        bool _delegateForever
) external nonReentrant returns (bool success) {...

/// @return success - true if the delegation was exited successfully, false o
therwise
function requestDelegationExit(uint256 _tokenId) external nonReentrant return
s (bool success) {...

function claimRewards(uint256 _tokenId) public nonReentrant returns (bool suc
cess) {...
```

`StargateNFT.sol`:

```
 /// @return success True if the minting was successful
function stake(
   uint8 _levelId
) external payable whenNotPaused nonReentrant returns (bool success, uint256
tokenId) {...

/// @return success True if the minting and delegation were successful
function stakeAndDelegate(
   uint8 _levelId,
   bool _autorenew
) external payable whenNotPaused nonReentrant returns (bool success, uint256
tokenId) {...

/// @return success - true if the migration was successful, will revert other
wise
function migrate(
   uint256 _tokenId
) external payable whenNotPaused nonReentrant returns (bool success) {...



/// @return success True if the migration and delegation were successful
/// @dev Emits a {IStargateNFT.TokenMinted} event
function migrateAndDelegate(
   uint256 _tokenId,
   bool _autorenew
) external payable whenNotPaused nonReentrant returns (bool success) {...

/// @return success - true if the unstake was successful, will revert otherwi
se
function unstake(uint256 _tokenId) external whenNotPaused nonReentrant return
s (bool success) {...
```

```
/// @return success - true if the rewards were claimed successfully, will rev
ert otherwise
/// @dev Emits a {IStargateNFT.BaseVTHORewardsClaimed} event
function claimVetGeneratedVtho(
   uint256 _tokenId
) external whenNotPaused nonReentrant returns (bool success) {...
```

F-2025-10818 - Possible disabling the `isActive` flag for migrate tiers may cause false expectations - Low

Description: The `StargateNFT` contract defines two categories of tiers:

- New tiers created directly in the current contract.
- Migrated tiers that originate from the legacy nodes NFT contract.

For migrated tiers, the migration flow is intentionally not halted when the `isActive` flag is set to `false`; only staking flows respect this flag. However, administrators can still set `isActive = false` for any tier through `updateLevel` or `toggleLevelIsActive`.

This creates a misleading expectation: administrators / observers may interpret the flag as disabling all actions for that tier, even though migration transactions will continue to succeed. A mismatch between the apparent state of a flag and its actual effect can lead to incorrect assumptions.

Assets:

- StargateNFT/StargateNFT.sol [https://github.com/vechain/stargate-staking]

Status: Fixed

Classification

Impact Rate:	2/5
Likelihood Rate:	3/5
Exploitability:	Semi-Dependent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: Consider applying the `isActive` flag also to the migration flow, which will make the behavior clear and also provide the ability to stop actions with tiers for migrations if necessary.

Alternatively, document the behavior providing additional information about the migration process.

Resolution: The related functionality was removed according commit `8629662`, which also fixed the issue.

F-2025-10633 - Revert Messages Optimization - Info

Description:	Using revert string messages increases the size of a smart contract and its deployment costs. Since Solidity version `0.8.4` custom errors are available. Custom Errors provide a way for developers to define descriptive and semantically meaningful error conditions without relying on string messages. The revert strings are used in the `NodeManagementV1`, `NodeManagementV2`, `NodeManagementV3` contracts.
Assets:	• NodeManagement/NodeManagementV2.sol [https://github.com/vechain/stargate-staking] • NodeManagement/NodeManagementV3.sol [https://github.com/vechain/stargate-staking] • NodeManagement/NodeManagementV1.sol [https://github.com/vechain/stargate-staking]
Status:	Fixed

Classification

Impact Rate:	1/5
Likelihood Rate:	5/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation:	To reduce gas costs during contract deployment, consider using custom errors instead of revert strings. Solidity compiler supports the use of Custom Errors within require statements since version `0.8.26` for the IR pipeline and since version `0.8.27` for the legacy pipeline. For the older Solidity versions, the pattern `if (!condition) revert CustomError()` should be used.

Resolution: The finding is fixed in the commit `9e17d64`. **Custom Errors** usage is implemented.

F-2025-10634 - Lack of Upgradeable Contracts Initialization - Info

Description:

The `StargateNFT` contract inherits from the `AccessControlUpgradeable`, `ERC721Upgradeable`, `ERC721EnumerableUpgradeable`, `ERC721PausableUpgradeable`, `ReentrancyGuardUpgradeable`, `UUPSUpgradeable` upgradeable contracts which require additional initialization for proper functioning.

However, the `initialize` function does not invoke initializers of the `ERC721EnumerableUpgradeable` and `ReentrancyGuardUpgradeable` contracts. While this omission does not currently break the contract logic, it is considered a bad practice as it may lead to potential issues in case of the dependency initializers logic change.

Assets:

- StargateNFT/StargateNFT.sol [https://github.com/vechain/stargate-staking]

Status: Fixed

Classification

Impact Rate:	1/5
Likelihood Rate:	5/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation: Consider adding the initializers to the `initialize` function.

```
 __ReentrancyGuard_init();
__ERC721Enumerable_init();
```

Resolution: The Finding is fixed in the commit `659edd5`. The missing initializers are invoked.

F-2025-10815 - Lack of ERC20 Operation Success Validation - Info

Description: Due to various fungible token implementations exist, it is recommended to utilize `SafeERC20` library for token transfers and approvals.

The `claimRewards` function of the `StargateDelegation` contract and `_claimRewards` function of the `VetGeneratedVtho` library invoke `transfer` method on the `$.vthoToken` token and manually validate the return value.

This causes risk of the system DoS in case the `$.vthoToken` token is not fully ERC-20 compatible (`transfer` method does not return any value).

Assets:

- StargateDelegation/StargateDelegation.sol [https://github.com/vechain/stargate-staking]
- StargateNFT/libraries/VetGeneratedVtho.sol [https://github.com/vechain/stargate-staking]

Status: Fixed

Classification

Impact Rate:	3/5
Likelihood Rate:	1/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation: Consider implementing `SafeERC20` library usage in the functions.

```
 using SafeERC20 for IERC20;


IERC20($.vthoToken).safeTransfer(to, amount);
```

Resolution: The Finding is fixed in the commit `f14509e`. Implementation is updated with using the OZ SafeERC20 library for ERC20 token transfers, replacing the simple `transfer` call.

```
- bool transferSuccess = $.vthoToken.transfer(recipient, amountToClaim);
- if (!transferSuccess) {
-     revert VthoTransferFailed(recipient, amountToClaim);
- }
+ $.vthoToken.safeTransfer(recipient, amountToClaim);

- // Transfer rewards to the caller
- bool success = IERC20($.vthoToken).transfer(owner, rewardsToClaim);Add comm
entMore actions
- if (!success) {
-   revert Errors.VthoTransferFailed();
- }
+ // Transfer rewards to the caller and revert if it fails
+ $.vthoToken.safeTransfer(owner, rewardsToClaim);Add comment
```

F-2025-10820 - Useless Repeated Request Exit Calls - Info

Description: The `requestDelegationExit` function is designed to be a one-way switch: once a user submits the request, the NFT's delegation is scheduled to end and the only way to resume delegating is to call `delegate` again after the exit has been processed.

However, the current implementation lets the same caller invoke `requestDelegationExit` repeatedly — even after `delegationEndBlock[tokenId]` has already been set. Each extra call has no effect; returns `true` instead of reverting,

While this behavior only creates minor practical issues — wasted gas and potential user confusion — it undermines consistency with approach in other similar functions.

Assets:

- StargateDelegation/StargateDelegation.sol [https://github.com/vechain/stargate-staking]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Consider implementing a check to disable the ability to re-calls of `requestDelegationExit` if it has already been called and does not lead to change `delegationEndBlock`.

Resolution: The finding is **fixed** in the `0044d2d` commit. The implementation of the `requestDelegationExit` method has been updated to prohibit futile calls for those who already have a delegation end date, but support cases of early exit in case with the end of the reward.

F-2025-10840 - Unimplemented Scaled Rewards Functionality - Info

Description:	The `Level` data structure implements the `scaledRewardFactor` field which is assumed to increase the rewards user with certain level obtains. However, the rewards scaling functionality is not implemented and the field is never used. The redundant field pose storage inefficiency and may be a sign of unfinalized code.
Assets:	• StargateNFT/libraries/Levels.sol [https://github.com/vechain/stargate-staking]
Status:	Accepted

Classification

Impact Rate:	1/5
Likelihood Rate:	5/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation: Consider removing the redundant field.

Resolution: The client has acknowledged the finding with the following statement:

> It is true that scaledRewardFactor is not used in v1 of contracts, but we are already working on v2 where this field is needed. We will keep this as is.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project’s performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/vechain/stargate
Initial Commit	c49487aedc30830feca8f208bd7bdbfa70246c4d
Retest Commit	86296662f1c7f10f7844fc7bb90e59a84d615fcd
Final Commit	f14509e12d407f725a7a57b1e6f071cc072d714b
Whitepaper	https://www.vechain.org/assets/whitepaper/whitepaper-3-0.pdf
Requirements	GitBook space shares internally
Technical Requirements	README.md

Asset	Type
NodeManagement/libraries/VechainNodesDataTypes.sol [https://github.com/vechain/stargate-staking]	Smart Contract
NodeManagement/NodeManagementV1.sol [https://github.com/vechain/stargate-staking]	Smart Contract
NodeManagement/NodeManagementV2.sol [https://github.com/vechain/stargate-staking]	Smart Contract
NodeManagement/NodeManagementV3.sol [https://github.com/vechain/stargate-staking]	Smart Contract
StargateDelegation/StargateDelegation.sol [https://github.com/vechain/stargate-staking]	Smart Contract
StargateNFT/libraries/Clock.sol [https://github.com/vechain/stargate-staking]	Smart Contract
StargateNFT/libraries/DataTypes.sol [https://github.com/vechain/stargate-staking]	Smart Contract
StargateNFT/libraries/Errors.sol [https://github.com/vechain/stargate-staking]	Smart Contract
StargateNFT/libraries/Levels.sol [https://github.com/vechain/stargate-staking]	Smart Contract
StargateNFT/libraries/MintingLogic.sol [https://github.com/vechain/stargate-staking]	Smart Contract
StargateNFT/libraries/Settings.sol [https://github.com/vechain/stargate-staking]	Smart Contract
StargateNFT/libraries/Token.sol [https://github.com/vechain/stargate-staking]	Smart Contract
StargateNFT/libraries/VetGeneratedVtho.sol [https://github.com/vechain/stargate-staking]	Smart Contract
StargateNFT/StargateNFT.sol [https://github.com/vechain/stargate-staking]	Smart Contract

Asset	Type
StargateProxy.sol [https://github.com/vechain/stargate-staking]	Smart Contract

Appendix 3. Additional Valuables

Verification of System Invariants

During the audit of VeChain / Stargate, Hacken followed its methodology by performing fuzz-testing on the project's main functions. Foundry, a tool used in the Solidity testing framework, was employed to check how the protocol behaves under various input conditions. Due to the complex and dynamic interactions within the protocol, unexpected edge cases might arise. Therefore, it was important to use invariant testing to ensure that several system invariants hold true in all situations.

Fuzz-testing allows the input of many random data points into the system, helping to identify issues that regular testing might miss. A specific Foundry fuzzing suite was prepared for this task, and throughout the assessment, 16 invariants were tested over >16M runs. This thorough testing ensured that the system works correctly even with unexpected or unusual inputs.

Invariant	Test Result	Run Count
Total NFT token supply must never exceed the sum of all level caps	Passed	1M
X-Level tokens can never be deactivated	Failed	1M
Recorded level (isX & maturityBlocks) parameters must remain immutable	Failed	1M
String contract parameters must be not empty (BaseURI, Name, Symbol)	Failed	1M
Core contract parameters addresses must never be zero	Passed	1M
Circulating supply for each level must stay within its cap	Passed	1M
NFT Total supply must equal the sum of all NFT holder balances	Passed	1M
Every minted token must belong to a valid level	Passed	1M
Total staked VET always eq VET deposited via stake	Passed	1M
NFT Token state match level requirement after mint, during nft live	Failed	1M
Level identifiers must never include zero	Passed	1M
Level identifiers must be in right order without duplicate	Passed	1M
NFT never in both can trasnfer and in delegated state	Passed	1M
Normal NFT count never exceed total supply	Failed	1M
xToken NFT count never exceed total supply	Failed	1M
Sum Normal and xToken equal total supply	Failed	1M

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.