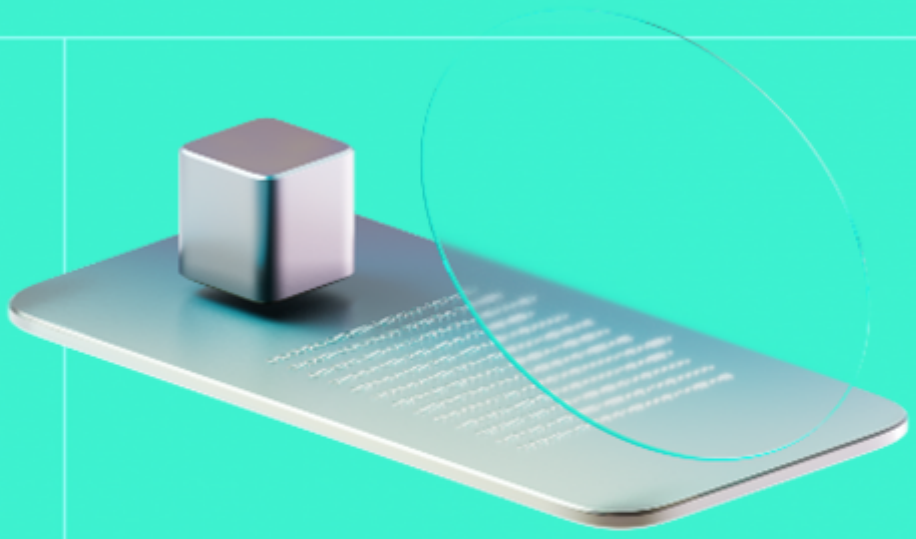
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: DragonX

Date: 17/01/2024

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

DragonX is a derivative of TitanX adoption, facilitates the creation of maximum-length TitanX stakes through publicly callable functions, utilizing Ethereum rewards to continuously purchase and burn TitanX off the market, while DragonX tokens serve as incentives for users max-staking their TitanX tokens during the launch phase.

Platform: DragonX

Language: Solidity

Tags: Staking, ERC20

Timeline: 11/01/2024 - 17/01/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/DragonX2024888/DragonX
Commit	95e66c5

Audit Summary

10/10	10/10	95.52%	10/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.8/10

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

0	0	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	0
Medium	0
Low	0

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for DragonX
Audited By	Kaan Caglan, Seher Saylik
Approved By	Ataberk Yavuzer
Website	https://dragonx.win/
Changelog	15/01/2024 - Preliminary Report
	17/01/2024 - Final Report

Table of Contents

System Overview

DragonX is a derivative of TitanX adoption, facilitates the creation of maximum-length TitanX stakes through publicly callable functions, utilizing Ethereum rewards to continuously purchase and burn TitanX off the market, while DragonX tokens serve as incentives for users max-staking their TitanX tokens during the launch phase.

DragonBuyAndBurn — a contract that swaps the cumulated ETH in the contract to DragonX and burns them to regulate token value. %3 of the operated amount gets sent to the caller as an incentive.

DragonX — is an ERC20 token contract that allows minting DragonX tokens for users exchanging TitanX tokens. The contract also manages the allocation of rewards, handles the deployment of new DragonStake instances, and enables the contract owner to claim accumulated assets from a Genesis Vault. DragonX token price differs by time and the price ratio once the mint phase is started is as followed:

- **Week 1** 1 TitanX : 1 DragonX
- **Week 2** 1 TitanX : 1 DragonX
- **Week 3** 1 TitanX : 0.95 DragonX
- **Week 4** 1 TitanX : 0.90 DragonX
- **Week 5** 1 TitanX : 0.85 DragonX
- **Week 6** 1 TitanX : 0.80 DragonX
- **Week 7** 1 TitanX : 0.75 DragonX
- **Week 8** 1 TitanX : 0.70 DragonX
- **Week 9** 1 TitanX : 0.65 DragonX
- **Week 10** 1 TitanX : 0.60 DragonX
- **Week 11** 1 TitanX : 0.55 DragonX
- **Week 12 - ∞** 1 TitanX : 0.50 DragonX

DragonX token has the following attributes:

- Name: DragonX
- Symbol: DRAGONX
- Decimals: 18
- Total supply: N/A - no max cap

TitanBuy — a contract that receives ETH rewards based on TitanX stakes from DragonX contract. 44.5% of the earned rewards from staking is sent to TitanBuy contract. TitanBuy swaps the received Eth for TitanX and sends it to the DragonX's vault. Caller of this swap operation in the TitanBuy contract is eligible to get %3 of the amount as an incentive to cover Gas fees.

DragonStake — a staking management contract that is created by DragonX contract. A new DragonStake instance is deployed whenever the number of open stakes in the current active instance exceeds the maximum allowed per wallet.

OracleLibrary — a library that helps to calculate the time-weighted means of tick and liquidity for a specified Uniswap V3 pool, based on observations made in the past.

PoolAddress — a library that provides functionality to compute the address of a Uniswap V3 pool based on the factory, tokens, and fee.

TickMath — a library that facilitates the precise calculation of square root prices based on tick values, and reciprocally, to determine the corresponding tick values for given square root prices.

Constants — a Solidity file that establishes various constants and parameters for the TitanX protocol and its associated functionalities. It defines numerical values, such as scaling factors and time-related constants, as well as specific parameters related to TitanX staking, bonus structures, and Uniswap-related addresses and liquidity pools.

Types — A simple enumerator to indicate the initial liquidity for the DragonX / TitanX pool has been minted

Privileged roles

- The owner of the DragonBuyAndBurn contract can:
  - create the initial liquidity for DragonX/TitanX
  - set the address of the DragonX contract
  - set the cap per swap
  - set the slippage percentage
  - set the buy and burn interval
  - set the average time to use in TWAP calculations of TitanX and DragonX
- The owner of DragonX contract can:
  - set the DRAGONX_BUY_AND_BURN address
  - set the TitanBuy address
  - withdraws the assets that are collected for the genesis vault
- The owner of the TitanBuy contract can:
  - set the DragonX address
  - set the cap per swap
  - set the slippage percentage
  - set the slippage percentage
  - set the buy and burn interval
  - set the average time to use in TWAP calculations of TitanX

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are provided
- Technical description is provided.
- NatSpec is sufficient.

Code quality

The total Code Quality score is **10** out of **10**.

- The code does follow the Solidity best practices.

Test coverage

Code coverage of the project is **95.52%** (branch coverage),

- Not all branches are covered with tests.

Security score

Upon auditing, the code was found to contain **0** critical, **0** high, **0** medium, and **0** low severity issues, leading to a security score of **10** out of **10**.

All identified suggestions to improve in-scope contracts are detailed in the “Observations” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **9.8**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- The implementation of `if (msg.sender != tx.origin)` in the `buyAndBurnDragonX`, `claim` and `buyTitanX` functions introduces a significant limitation, preventing these functions from being executed through any multisig wallet. This restriction could lead to reduced flexibility and potential operational inefficiencies for users relying on multisig wallets for enhanced security.

Findings

Observation Details

F-2024-0475 - Missing checks for `address(0)` - Info

Description:

In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the **zero address**. This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address.
This issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.
For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

Code Location

```
Path: ./contracts/DragonX.sol

TITANX_BUY = titanBuy;

DRAGONX_BUY_AND_BURN = dragonBuyAndBurn;
```

Assets:

- ./contracts/DragonX.sol

Status: Fixed

Recommendations

Recommendation:

It is strongly recommended to implement checks to prevent the zero address from being set during the initialization of contracts. This can be achieved by adding require statements that ensure address parameters are not the zero address.

Remediation (Revised commit: 9dda717): The DragonX team introduced zero address checks for the constructor function.

F-2024-0476 - Use `Ownable2Step` rather than `Ownable` - Info

Description:

Ownable2Step and Ownable2StepUpgradeable prevent the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner permissions actively accept via a contract call of its own.

Code Location

```
contract DragonX is ERC20, Ownable, ReentrancyGuard {
contract DragonBuyAndBurn is Ownable, ReentrancyGuard {
contract TitanBuy is Ownable, ReentrancyGuard {
contract DragonStake is Ownable {
```

Assets:

- ./contracts/DragonBuyAndBurn.sol
- ./contracts/DragonX.sol
- ./contracts/TitanBuy.sol
- ./contracts/lib/DragonStake.sol

Status: Fixed

Recommendations

Recommendation:

Consider using `Ownable2Step` or `Ownable2StepUpgradeable` instead of `Ownable` or `OwnableUpgradeable` from OpenZeppelin Contracts to enhance the security of your contract ownership management. These contracts prevent the accidental transfer of ownership to an address that cannot handle it, such as due to a typo, by requiring the recipient of owner permissions to actively accept ownership via a contract call. This two-step ownership transfer process adds an additional layer of security to your contract's ownership management.

Remediation (Revised commit: d0b4f3b): The DragonX team started using `Ownable2Step` contract of OpenZeppelin to prevent common mistakes, such as transfers of ownership to incorrect accounts. As addition, the DragonX team stated that the `DragonStake` contract will always be owned and deployed by `DragonX` contract. Therefore, a 2-step owner functionality for renouncing will not be necessary.

F-2024-0477 - Solidity version 0.8.20 might not work on all chains due to `PUSH0` - Info

Description: Solidity version 0.8.20 employs the recently introduced `PUSH0` opcode in the Shanghai EVM, this opcode might not be universally supported across all blockchain networks and Layer 2 solutions. It is advisable to use an earlier version of Solidity to ensure compatibility.

Assets:

- ./contracts/DragonBuyAndBurn.sol
- ./contracts/DragonX.sol
- ./contracts/TitanBuy.sol
- ./contracts/lib/Constants.sol
- ./contracts/lib/DragonStake.sol
- ./contracts/lib/Types.sol

Status: Accepted

Recommendations

Recommendation: To ensure compatibility with a wide range of blockchain networks and Layer 2 solutions, consider using an earlier version of Solidity that does not rely on the `PUSH0` opcode introduced in Solidity version 0.8.20. Using a more widely supported Solidity version can help avoid potential compatibility issues and ensure the smooth deployment and execution of your smart contracts.

Remediation: The DragonX team informed that the contracts will be deployed on only Ethereum mainnet.

F-2024-0478 - Unneeded initializations of uint256 and bool variable to 0/false - Info

Description:

In Solidity, it is common practice to initialize variables with default values when declaring them. However, initializing `uint256` variables to `0` and `bool` variables to `false` when they are not subsequently used in the code can lead to unnecessary gas consumption and code clutter. This issue points out instances where such initializations are present but serve no functional purpose..

Code Location

```
Path: ./contracts/DragonX.sol
for (uint256 idx = 0; idx < numDragonStakeContracts; idx++) {
```

Number of instances: 4

```
Path: ./contracts/lib/DragonStake.sol
for (uint256 idx = 0; idx < stakes.length; idx++) {
```

Assets:

- ./contracts/DragonX.sol
- ./contracts/lib/DragonStake.sol

Status: Fixed

Recommendations

Recommendation:

It is recommended not to initialize integer variables to `0` to and boolean variables to `false` to save some Gas.

Remediation (Revised commit: 953c80d): The DragonX team removed unnecessary initializations to optimize gas usage..

F-2024-0479 - Custom Errors in Solidity for Gas Efficiency - Info

Description:

Starting from Solidity version 0.8.4, the language introduced a feature known as "custom errors". These custom errors provide a way for developers to define more descriptive and semantically meaningful error conditions without relying on string messages. Prior to this version, developers often used the `require` statement with string error messages to handle specific conditions or validations. However, every unique string used as a revert reason consumes gas, making transactions more expensive.

Custom errors, on the other hand, are identified by their name and the types of their parameters only, and they do not have the overhead of string storage. This means that, when using custom errors instead of `require` statements with string messages, the gas consumption can be significantly reduced, leading to more gas-efficient contracts.

Number Of Instances Found
21

Assets:

- ./contracts/DragonBuyAndBurn.sol
- ./contracts/DragonX.sol
- ./contracts/TitanBuy.sol
- ./contracts/lib/DragonStake.sol

Status: Accepted

Recommendations

Recommendation:

It is recommended to use custom errors instead of revert strings to reduce gas costs, especially during contract deployment. Custom errors can be defined using the error keyword and can include dynamic information.

Remediation: The DragonX team acknowledged this observation.

F-2024-0480 - Unused Error Definition - Info

Description: The errors `error NoStakesToEnd();`, `error TitanTransferFailed();` and `error NoTitanPriceTwaData();` are declared, but never used.

This leaves redundant logic in code.

Assets:

- ./contracts/DragonX.sol
- ./contracts/TitanBuy.sol

Status: Fixed

Recommendations

Recommendation: Unused `error` definitions should be removed from the contract, and if needed, consolidated into a separate file to avoid duplication.

Remediation (Revised commit: 88a2f50): The DragonX team removed unused errors from contracts.

F-2024-0481 - Owner Can Renounce Ownership - Info

Description: The smart contract under inspection inherits from the `Ownable` library, which provides basic authorization control functions, simplifying the implementation of user permissions. While the contract allows for the transfer of ownership to a different address or account, it also retains the default `renounceOwnership` function from `Ownable`. Once the owner uses this function to renounce ownership, the contract becomes ownerless. Evidence in the transaction logs shows that, following the activation of the `renounceOwnership` function, any attempts to invoke functions requiring owner permissions fail, with the error message: `"Ownable: caller is not the owner."` This condition makes the contract's adjustable parameters immutable, potentially rendering the contract ineffective for any future administrative modifications that might be needed.

Code Location

```
contract DragonX is ERC20, Ownable, ReentrancyGuard {
contract DragonBuyAndBurn is Ownable, ReentrancyGuard {
contract TitanBuy is Ownable, ReentrancyGuard {
contract DragonStake is Ownable {
```

Assets:

- ./contracts/DragonBuyAndBurn.sol
- ./contracts/DragonX.sol
- ./contracts/TitanBuy.sol
- ./contracts/lib/DragonStake.sol

Status: Accepted

Recommendations

Recommendation: To mitigate this vulnerability:

- Override the `renounceOwnership` function to revert transactions: By overriding this function to simply revert any transaction, it will become impossible for the contract owner to unintentionally (or intentionally) render the contract ownerless and thus immutable.

Remediation: The DragonX team acknowledged this observation.

F-2024-0482 - Public Functions That Should Be External - Info

Description:

Functions that are meant to be exclusively invoked from external sources should be designated as `external` rather than `public`. This is essential to enhance both the gas efficiency and the overall security of the contract.

Code Location

```
Path: ./contracts/DragonBuyAndBurn.sol
function collectFees() public nonReentrant {
```

Assets:

- ./contracts/DragonBuyAndBurn.sol

Status: Fixed

Recommendations

Recommendation:

To optimize gas usage and improve code clarity, declare functions that are not called internally within the contract and are intended for external access as `external` rather than `public`. This ensures that these functions are only callable externally, reducing unnecessary gas consumption and potential security risks.

Remediation (Revised commit: 7852b66): The DragonX team fixed this issue by changing specified functions' visibility to `external` from `public`.

F-2024-0483 - Remove `hardhat` import - Info

Description: In Solidity development, `hardhat` is a commonly used development framework that provides tools for testing, debugging, and deploying smart contracts. However, imports from `hardhat` in your Solidity code, such as those used for testing or local development, should not make their way into the final production version of the contract. These imports can bloat the contract, lead to unnecessary complexity, and potentially introduce security risks or unexpected behavior. Ensuring that production code is clean, efficient, and free of development-only dependencies is crucial for maintaining security and performance.

Code Location

```
Path: ./contracts/DragonBuyAndBurn.sol
import "hardhat/console.sol";
```

Assets:

- ./contracts/DragonBuyAndBurn.sol

Status: Fixed

Recommendations

Recommendation: Before finalizing and deploying your Solidity contracts, thoroughly review the code to ensure all `hardhat` imports and related development-only code are removed.

Remediation (Revised commit: 48f85dd): The DragonX team resolved this observation by removing that import.

F-2024-0484 - Event is not properly `indexed - Info

Description:

Index event fields make the field more quickly accessible to off-chain tools that parse events. This is especially useful when it comes to filtering based on an address. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Where applicable, each `event` should use three `indexed` fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three applicable fields, all of the applicable fields should be `indexed`.

Code Location

```
Path: ./contracts/DragonX.sol

event DragonStakeInstanceCreated
    uint256 stakeContractId,
    address stakeContractAddress
);
```

Assets:

- ./contracts/DragonX.sol

Status: Fixed

Recommendations

Recommendation:

Enhance smart contract efficiency post-deployment by utilizing indexed events. This approach aids in efficiently tracking contract activities, significantly contributing to the reduction of gas costs.

Remediation (Revised commit: ab8d131): The DragonX team resolved this observation after introducing `indexed` keywords for the specific event.

F-2024-0485 - Use `private` Rather Than `public` For Constants - Info

Description:

In Solidity, constants represent immutable values that cannot be changed after they are set at compile-time. By default, constants have internal visibility, meaning they can be accessed within the contract they are declared in and in derived contracts. If a constant is explicitly declared as `public`, Solidity automatically generates a getter function for it. While this might seem harmless, it actually incurs a gas overhead, especially when the contract is deployed, as the EVM needs to generate bytecode for that getter. Conversely, declaring constants as `private` ensures that no additional getter is generated, optimizing gas usage.

```
uint256 public constant mintRatioWeekOne = BASIS;
uint256 public constant mintRatioWeekTwo = BASIS;
uint256 public constant mintRatioWeekThree = 9500;
uint256 public constant mintRatioWeekFour = 9000;
uint256 public constant mintRatioWeekFive = 8500;
uint256 public constant mintRatioWeekSix = 8000;
uint256 public constant mintRatioWeekSeven = 7500;
uint256 public constant mintRatioWeekEight = 7000;
uint256 public constant mintRatioWeekNine = 6500;
uint256 public constant mintRatioWeekTen = 6000;
uint256 public constant mintRatioWeekEleven = 5500;
uint256 public constant mintRatioWeekTwelve = 5000;
```

Assets:

- ./contracts/DragonX.sol

Status: Accepted

Recommendations

Recommendation:

To optimize gas usage in your Solidity contracts, declare constants with `private` visibility rather than `public` when possible. Using `private` prevents the automatic generation of a getter function, reducing gas overhead, especially during contract deployment.

Remediation: The DragonX team acknowledged the finding.

F-2024-0486 - Avoid Using State Variables Directly in `emit` for Gas Efficiency - Info

Description:

In Solidity, emitting events is a common way to log contract activity and changes, especially for off-chain monitoring and interfacing. However, using state variables directly in `emit` statements can lead to increased gas costs. Each access to a state variable incurs gas due to storage reading operations. When these variables are used directly in `emit` statements, especially within functions that perform multiple operations, the cumulative gas cost can become significant. Instead, caching state variables in memory and using these local copies in `emit` statements can optimize gas usage.

```
emit DragonStakeInstanceCreated(
    stakeContractId,
    activeDragonStakeContract
);

emit TitanStakeStarted(activeDragonStakeContract, amountToStake);
```

`activeDragonStakeContract` and `activeDragonStakeContract` are state variables and they can be cached.

Assets:

- ./contracts/DragonX.sol

Status: Fixed

Recommendations

Recommendation:

To optimize gas efficiency, cache state variables in memory when they are used multiple times within a function, including in `emit` statements.
Remediation (Revised commit: d0b4f3b): The DragonX team resolved this observation by caching state variables in order to consume less gas.

F-2024-0487 - Cache State Variables - Info

Description:

Cache state variables issues in Solidity refer to situations where developers fail to efficiently manage and update state variables in smart contracts. These issues can lead to suboptimal gas usage, decreased contract performance, and even vulnerabilities that can be exploited by malicious actors. Properly handling and caching state variables is crucial for maintaining efficient and secure smart contracts.

Code Location

```
Path: ./contracts/DragonX.sol

404:        } else if (block.timestamp < mintPhaseBegin + 35 days) {    // @audi

475:        if (vault >= TITANX_BPB_MAX_TITAN) {    // @audit-issue: State varial

566:        Address.sendValue(payable(DRAGONX_BUY_AND_BURN), buyAndBurnDragonX);

569:        Address.sendValue(payable(TITANX_BUY), buyTitanX);    // @audit-issu

630:        _mint(DRAGONX_BUY_AND_BURN, amount);    // @audit-issue: State varial

645:        nextStakeTs = mintPhaseBegin + 7 days;    // @audit-issue: State var

883:        _receiveEthAllowlist[activeDragonStakeContract] = true;    // @audit

914:        titanX.safeTransfer(activeDragonStakeContract, amountToStake);    //
```

```
Path: ./contracts/DragonBuyAndBurn.sol

295:            DRAGONX_ADDRESS    // @audit-issue: State variable `DRAGONX_ADDR

362:        DragonX dragonX = DragonX(payable(DRAGONX_ADDRESS));    // @audit-iss

386:        DragonX dragonX = DragonX(payable(DRAGONX_ADDRESS));    // @audit-iss

631:                DRAGONX_ADDRESS    // @audit-issue: State variable `DRAGONX_

647:        if (forBuy > capPerSwap) {    // @audit-issue: State variable `capPe

684:        uint256 amountOutMinimum = (expectedDragonAmount * (100 - slippage))

719:        token1 = DRAGONX_ADDRESS;    // @audit-issue: State variable `DRAGONX
```

```
Path: ./contracts/TitanBuy.sol

208:        if (DRAGONX_ADDRESS == address(0)) {    // @audit-issue: State varial

456:        if (forBuy > capPerSwap) {    // @audit-issue: State variable `capPe
```

Assets:

- ./contracts/DragonBuyAndBurn.sol
- ./contracts/DragonX.sol
- ./contracts/TitanBuy.sol

Status: Fixed

Recommendations

Recommendation: Enhance contract efficiency and security by caching state variables in memory when used multiple times in a function. This approach reduces gas consumption and potential vulnerabilities from frequent state variable updates.

Remediation (Revised commit: d0b4f3b): The DragonX team fixed this issue by caching state variables.

F-2024-0488 - UPPER_CASE names should be reserved for `constant`/`immutable` variables - Info

Description:

In Solidity, and in programming in general, naming conventions are essential for readability and understanding of the code. It's a common practice to use UPPER_CASE naming for `constant` or `immutable` variables. These are variables whose values are set at contract deployment and do not change thereafter. Adhering to this convention helps distinguish `constant` or `immutable` variables from other variables whose values may change. This practice enhances code clarity and makes it easier to understand the contract's behavior, especially for new developers or auditors reviewing the code. Using UPPER_CASE for non-constant variables can mislead and cause confusion about the nature of these variables.

Code Location

```
Path: ./contracts/DragonX.sol
address public TITANX_BUY;
address public DRAGONX_BUY_AND_BURN;
```

```
Path: ./contracts/TitanBuy.sol
address public DRAGONX_ADDRESS;
```

```
Path: ./contracts/DragonBuyAndBurn.sol
address public DRAGONX_ADDRESS;
```

Assets:

- ./contracts/DragonBuyAndBurn.sol
- ./contracts/DragonX.sol
- ./contracts/TitanBuy.sol

Status: Fixed

Recommendations

Recommendation:

Adopt a clear and consistent naming convention in your Solidity contracts by using UPPER_CASE names exclusively for `constant` or `immutable` variables. Review your codebase to ensure that this convention is consistently applied, and refactor any non-constant variables named in UPPER_CASE to a different naming style. This will enhance the readability and clarity of your code, making it more maintainable and easier to understand at a glance.

Remediation (Revised commit: 8636453): The DragonX team fixed the issue by changing the variable names.

External References:

- Constants

F-2024-0489 - Style Guide Violation - Info

Description:

Contract readability and code quality are influenced significantly by adherence to established style guidelines. In Solidity programming, there exist certain norms for code arrangement and ordering. These guidelines help to maintain a consistent structure across different contracts, libraries, or interfaces, making it easier for developers and auditors to understand and interact with the code.

The suggested order of elements within each contract, library, or interface is as follows:

1. Type declarations
2. State variables
3. Events
4. Errors
5. Modifiers
6. Functions

Functions should be ordered and grouped by their visibility as follows:

1. Constructor
2. Receive function (if exists)
3. Fallback function (if exists)
4. External functions
5. Public functions
6. Internal functions
7. Private functions

Within each grouping, `view` and `pure` functions should be placed at the end.

Furthermore, following the Solidity naming convention and adding NatSpec annotations for all functions are strongly recommended. These measures aid in the comprehension of code and enhance overall code quality.

Function `stakeReachedMaturity` should come before than function `updateVault` in `DragonX.sol` and function `createInitialLiquidity` should come before than function `collectFees`. And also `Events` should come before than `Errors`. This is not followed in `DragonX`, `DragonBuyAndBurn` and `TitanBuy`.

Assets:

- ./contracts/DragonBuyAndBurn.sol
- ./contracts/DragonX.sol

Status: Fixed

Recommendations

Recommendation: Consistent adherence to the official Solidity style guide is recommended. This enhances readability and maintainability of the code, facilitating seamless interaction with the contracts. Providing comprehensive NatSpec annotations for functions and following Solidity's naming conventions further enrich the quality of the code.

Remediation (Revised commit: 067b9c2): The DragonX team fixed the issue by following the described guideline above.

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. Severity Definitions

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, do not affect security score but can affect code quality score.

Appendix 2. Scope

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

Scope Details

Repository	https://github.com/DragonX2024888/DragonX
Commit	95e66c52a3cac7991f4d4cf51a9897d131eb95bb
Whitepaper	-
Requirements	https://github.com/DragonX2024888/DragonX/docs
Technical Requirements	https://github.com/DragonX2024888/DragonX/docs

Contracts in Scope

./contracts/DragonX.sol

./contracts/DragonBuyAndBurn.sol

./contracts/TitanBuy.sol

./contracts/lib/DragonStake.sol

./contracts/lib/Types.sol

./contracts/lib/Constants.sol

./contracts/lib/uniswap/PoolAddress.sol

./contracts/lib/interfaces/IWETH.sol

./contracts/lib/interfaces/ITitanX.sol

./contracts/lib/interfaces/INonfungiblePositionManager.sol