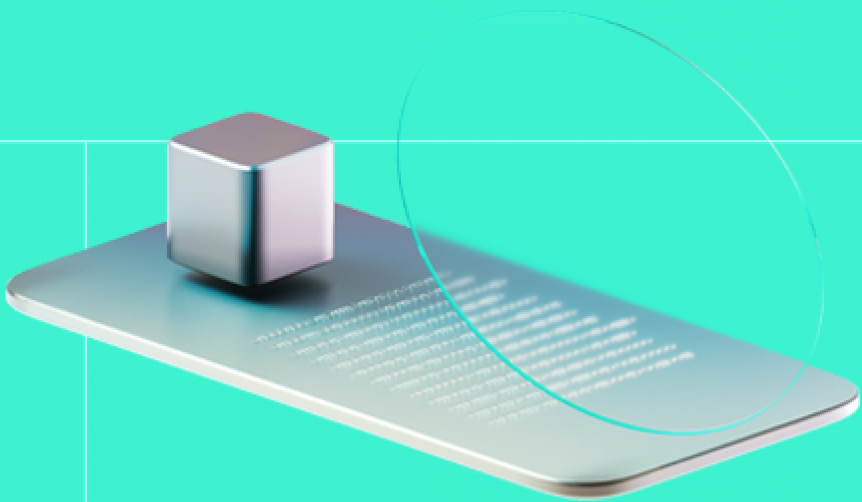
Smart Contract Code Review And Security Analysis Report



Customer: Dinari SA

Date: 30/12/2024

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

Dinari Securities Backed Tokens (dShares) provide direct exposure to the world's most trusted assets such as Google and Apple shares.

Document

Name	Smart Contract Code Review and Security Analysis Report for Dinari SA
Audited By	Stepan Chekhovskoi
Approved By	Przemyslaw Swiatowiec
Website	https://dinari.com
Changelog	12/12/2024 - Preliminary Report
	30/12/2024 - Second Report
Platform	Plume
Language	Solidity
Tags	Vault, DeFi
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/dinaricrypto/sbt-contracts
Initial Commit	1fa1a298373cefc252694984640db62d8f3a6b33
Second Commit	d327ea9a6fc5f86caf73769dd1983b5333e774b8

Audit Summary

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

1	0	1	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	1
Low	0

Vulnerability	Severity
F-2024-7639 - Possibly Invalid Swap Rate due to Using Unreliable Market Price	Medium

Documentation quality

- Functional Requirements are partially missed.
- Technical Description is provided.

Code quality

- The code architecture is clean and easy to read.
- The development environment is configured.
- While the NatSpec comments mention `OrderProcessor` provides price with 18 decimals, the calculations imply `36 - dShare.decimals()`.
- The `OracleLib` functionality does not revert transaction in case of overflow and returns zero.

Test coverage

Code coverage of the project is **74%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is partially missed.

Table of Contents

System Overview

`DinariAdapterToken` is a vault contract developed to use ETF dShare as one of liquid assets in Plume Nest money market.

The vault is built on top of out-of-scope `ComponentToken` contract - ERC-4626 Vault with ERC-7540 and ERC-7575 extensions.

The contract integrates external `OrderProcessor` used to exchange vault asset token to dShare and back.

Implemented functionality:

- `convertToShares`, `convertToAssets` - `ComponentToken` overrides provide info on the vault rate based on conversion rate from `OrderProcessor`,
- `requestDeposit`, `requestRedeem` - asynchronous vault deposit and redeem processing funds through the `OrderProcessor`,
- `processSubmittedOrders`, `processNextSubmittedOrder` - functions to validate if the `OrderProcessor` have executed the submitted requests.

Privileged roles

- The system is designed to work exclusively with Nest Staking.
- The system owner can upgrade the contract.

Potential Risks

- **System Reliance on External Contracts**: The functioning of the system significantly relies on Order Processor and Nest Staking external contracts. Any flaws or vulnerabilities in these contracts adversely affect the audited project, potentially leading to security breaches or loss of funds.
- **Dependency on External Logic for Implemented Logic**: The `DinariAdapterToken` logic highly depends on `ComponentToken` external contract not covered by the audit. This reliance introduces risks if these external contracts are compromised or contain vulnerabilities, affecting the audited project's integrity.
- **Dynamic Array Iteration Gas Limit Risks**: The `processSubmittedOrders` iterates over large dynamic arrays, which leads to excessive Gas costs, risking denial of service due to out-of-gas errors, alternative `processNextSubmittedOrder` function can be used to process requests one-by-one.
- **Single Points of Failure and Control**: The project is mostly centralized, introducing single points of failure and control. 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's 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.
- **Token Decimals Reliance**: The system relies on the dShare token has 18 decimals. Missing the invariant may cause inconsistencies in price decimals calculation leading to the price interpreted incorrectly breaking the conversion rate.
- **Zero Output in Case of Overflow**: The system returns zero shares in exchange for assets or zero assets in exchange for shares in case overflow happens in internal price calculations. Consider simulate transaction before executing to avoid the edge cases.

Findings

Vulnerability Details

F-2024-7639 - Possibly Invalid Swap Rate due to Using Unreliable Market Price - Medium

Description:

The `latestFillPrice` function over external `OrderProcessor` contract is used for calculations in the `convertToShares` and `convertToAssets` functions.

The `latestFillPrice` function is documented with `get the latest fill price for a token pair` what means it returns latest market data. Latest market data is supposed to on-chain price manipulations as it is a temporary value. To guarantee price consistency time-weighted price should be used.

The function returns `PricePoint` structure of `uint256 price` and `uint64 blocktime`. The `blocktime` parameter is not validated causing outdated price might be used.

This may lead to unexpectedly high amount of shares minted in the vault inflating the vault shares value.

```
 function convertToShares(uint256 assets) public view override(ComponentToken)
returns (uint256 shares) {
    ...
    IOrderProcessor.PricePoint memory price = orderContract.latestFillPrice($
.dshareToken, paymentToken);
    return wrappedDshareToken.convertToShares(((orderAmount + fees) * price.p
rice) / 1 ether);
}

function convertToAssets(uint256 shares) public view override(ComponentToken)
returns (uint256 assets) {
    ...
    IOrderProcessor.PricePoint memory price = orderContract.latestFillPrice(d
shareToken, paymentToken);
    ...
    uint256 proceeds = ((dshares / precisionReductionFactor) * precisionReduc
tionFactor * 1 ether) / price.price;
    ...
    return proceeds - fees;
}
```

Assets:

- DinariAdapterToken.sol [https://github.com/dinaricrypto/sbt-contracts]

Status: Accepted

Classification

Impact: 3/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Consider integrating trusted oracle and implementing validation that the price is not stale.

```
IOrderProcessor.PricePoint memory price = orderContract.latestFillPrice(dshareToken, paymentToken);
if (block.timestamp - price.blocktime > threshold) revert OutdatedPrice(block.timestamp, price.blocktime);
```

Resolution: The Finding is partially fixed in the commit `d2af1d51230b73249acba101f76c6365e1c62689`.

The `_getDSharePrice` function wraps the `orderContract.latestFillPrice` external call with the necessary return value validations.

Usage of `latestFillPrice` as price source may lead to third party is able to manipulate the conversion rate impacting the system security and integrity. It is highly recommended to integrate or implement a trusted Oracle to enhance the price security.

Observation Details

F-2024-7625 - Public Functions that Can be External - Info

Description: Functions intended to be invoked exclusively from external sources should be designated as `external` rather than `public`.

This is essential to enhance both Gas efficiency and the overall security of the contract.

The visibility of the following functions can be restricted from `public` to `external`.

- `processNextSubmittedOrder`
- `processSubmittedOrders`
- `getNextSubmittedOrder`
- `getSubmittedOrderInfo`
- `initialize`

Assets:

- DinariAdapterToken.sol [https://github.com/dinaricrypto/sbt-contracts]

Status: Fixed

Recommendations

Remediation: Consider updating functions which are exclusively utilized by external entities from their current `public` visibility to `external` visibility.

Resolution: The Finding is fixed in the commit `a22f003e9ee21b83d349e03f6afee03ea44c5f38`.

The functions visibility modifiers are changed to `external`.

F-2024-7626 - Floating Pragma - Info

Description: The contracts use floating pragma `^0.8.25`.

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, contracts might be deployed using an outdated pragma version which may include bugs that affect the system negatively.

Assets:

- DinariAdapterToken.sol [https://github.com/dinaricrypto/sbt-contracts]

Status: Fixed

Recommendations

Remediation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs for the compiler version that is chosen.

Resolution: The Finding is fixed in the commit `842badab61436398c5d1dcd06c34ef30f5236786`.

The compiler version in pinned to `0.8.25`.

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/dinaricrypto/sbt-contracts
Initial Commit	1fa1a298373cefc252694984640db62d8f3a6b33
Second Commit	d327ea9a6fc5f86caf73769dd1983b5333e774b8
Whitepaper	https://assets.dinari.com/forms/dinari-whitepaper.pdf
Requirements	Notes shared internally
Technical Requirements	README.md

Asset	Type
DinariAdapterToken.sol [https://github.com/dinaricrypto/sbt-contracts]	Smart Contract

Appendix 3. Additional Valuables

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.