SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Date: 21 June, 2025

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 NSDQ
Approved By	Svyatoslav Nadozirny | Solidity SC Auditor
Auditor company	Coders Valley Ltd. 63-66 Hatton Garden Fifth Floor, Suite 23 EC1N 8LE - London London (GB) United Kingdom
Type	ERC-1400 Security Token
Platform	Ethereum Mainnet
Language	Solidity ^0.8.30
Methodology	Referenced document for audit methodology
ChangeLog	June 21, 2025 - initial release

Table of contents

Introduction

The Customer engaged our company to evaluate the **NSDQ** smart-contract for security, code quality and compliance with ERC-1400 best practices. This report summarizes our findings and provides actionable recommendations.

Scope

The scope of the project includes the following smart contracts from the file:

Contracts: https://drive.google.com/file/d/1Y72QoajCX6_LCZv73pSbV91wwCdscPAk/view

- **NSDQ.sol** – contains the entire ERC-1400 implementation, supporting partitioned transfers, EIP-1820 integration, role-based access control, migration logic, and token initialization

Live Code: Not provided

Technical Documentation: Not provided

Tests: Not provided

Environment: Not provided

SHA256 Hash

SHA256 hash of the source code provided:

77f945667433475803c57f65fcdbaba8cee5c801063da93d7d02b15e4c5255bb….. NSDQ.sol

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation by external or internal actors.
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 by external or internal actors.
Medium	Medium vulnerabilities are usually limited to state manipulations but cannot lead to asset loss. Major deviations from best practices are also in this category.
Low	Low vulnerabilities are related to outdated and unused code or minor Gas optimization. These issues won't have a significant impact on code execution but affect code quality.

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality Score is 8 out of 10.

- **Functional requirements** are provided in https://docs.google.com/presentation/d/1KHvku-ghghFn1563Wooa9So8eQSIo8u6cUmKTt3OQFh0/edit?slide=id.g3619fa5bf2d_0_0#slide=id.g3619fa5bf2d_0_0 Token name, symbol, initial and sale supply, unlimited minting, controllability and migration features provided. (Score: 5/5).
- **Technical Requirements**: Compiler version and ERC-1820 registry address specified; deployment instructions and environment details are absent. (Score: 3/5).
- **NatSpec Adherence**: NatSpec comments are not used, which reduces readability for auditors and developers.

Code quality

The total Code Quality Score is 6 out of 10.

- **Development Environment:** No configuration files or scripts (Hardhat/Truffle) provided. (Score: 2/5).
- **Solidity Style Guide Compliance**: Code is consistently formatted, follows OpenZeppelin patterns; missing explicit visibility on internal functions. (Score: 5/5).

Security score

The security Score is 0 out of 10.

Despite correct adherence to ERC-1400 semantics, the presence of a critical reentrancy vector and a compilation-breaking modifier issue drives the score to the minimum. No unit tests were provided (0 % branch coverage). (Score: 0/10).

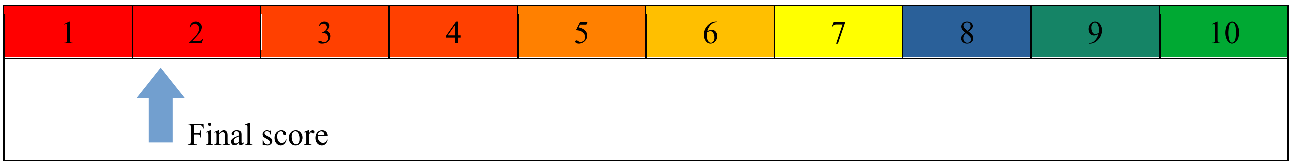
- **Critical Issues**: 1
  - Reentrancy in *transfer*/*transferFrom*/*transferWithData*/*transferFromWithData* hooks.
- **High Issues**: 1
  - Duplicate *nonReentrant* modifier in *operatorRedeemByPartition*, causing compilation failure.
- **Medium Issues**: None
- **Low Issues**: 2.

- Missing explicit *internal*/*private* visibility on many helper functions.
- Use of hex-only error codes (e.g. "*52*") instead of descriptive messages or custom errors.

Summary

According to the assessment, the Customer's smart contract has the following score: **2.1**.

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



Breakdown:

- Documentation Quality: 8/10
- Code Quality: 7/10
- Security Level: 0/10
- Test Coverage: Not provided (requires unit tests for scoring).

Note: The final score is weighted according to the methodology (Documentation weighted at 1.0, Code Quality at 2.0, Security at 7.0), and the absence of unit tests impacts the overall score.

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
21 June, 2025	2	0	1	1

Risks

1. Critical re-entrancy vector in transfer, transferFrom, transferWithData, and transferFromWithData.
An attacker could recursively re-enter the same operation before the initial balance update is finalized, effectively “printing” tokens or draining pools. This would allow large-scale theft or permanent locking of user funds, wiping liquidity and crashing the token price, which in turn could trigger emergency delistings. A likely exploit involves a malicious proxy contract whose tokensReceived() (or similar) callback repeatedly calls one of the vulnerable transfer functions as long as gas permits, siphoning value on every recursion.

2. Duplicate nonReentrant modifier in operatorRedeemByPartition, causing a compilation break.
The contract may fail to compile, or a hurried hot-fix could be pushed that removes the guard altogether. Either outcome opens a window for unauthorized calls or redeployments. Investors face launch delays, missed exchange listings, or—even worse—a community-created fork without the protective modifier that later becomes the canonical token and is then drained through the same re-

entrancy route described above.

3. Missing explicit internal/private visibility on several helper functions.
Utility methods intended only for internal bookkeeping are callable from the outside, exposing contract internals and letting anyone spam them with arbitrary data. This can leak business logic, inflate gas costs, and clog critical buy-back or dividend workflows. A bot could hammer such a helper with oversize payloads, consistently pushing blocks toward the gas limit and slowing or outright stalling time-sensitive user transactions.

4. Hex-only error codes ("52", "6a") instead of descriptive strings or custom errors.
Opaque return codes make it hard for wallets, bridges, and CEX/DEX listing engines to diagnose failures. Users see unexplained reverts, support tickets rise, and exchanges may temporarily flag the token as suspicious. In a plausible scenario, a user's transfer reverts with "0x52"; unable to decode it, they spread FUD on social media, while an exchange's monitoring script also detects the unknown code and pauses deposits until manual review—damaging liquidity and reputation in the interim.

System Overview

NSDQ is an ERC-1400 security token will be deployed on Ethereum Mainnet. Upon construction, the contract:

- Issues a fixed initial supply of **22,976,190 NSDQ** (multiplied by 10^{18}) to the designated owner address.
- Transfers **16,083,333 NSDQ** (70 % of initial issuance) from the deployer to the seller address for sale.

Key characteristics:

- **Unlimited Minting:** The *MinterRole* allows authorized minters (and the owner) to issue additional tokens until the owner calls *renounceIssuance*().
- **Partitioned Transfers:** All tokens reside in a single default partition (*NSDQ_DEFAULT_PARTITION*). Transfers use partition logic for fine-grained control and ERC-20 compatibility via default partition fallback.
- **Granularity:** Token granularity is set to 1, enforcing that token amounts are always multiples of 1.
- **Role-Based Control:** Owner and minters manage issuance; controllers and partition controllers enforce transfer restrictions when *_isControllable* is enabled.
- **ERC-1820 Integration:** Implements ERC1400, ERC20 interfaces in the ERC-1820 registry and supports dynamic extensions (validators, checkers, senders, recipients) via registry hooks.
- **Migration Support:** The *migrate* function registers a new contract address in ERC-1820 and can irrevocably disable the current implementation if invoked with *definitive=true*.
- **EIP-712 Domain Aware:** Implements a domain separator for off-chain signature verification supporting extensions that leverage signed certificates.

All contract logic resides in a single Solidity file (*NSDQ.sol*), facilitating a complete, on-chain audit scope.

Privileged roles

- **Owner:** Transfer ownership, renounce control/issuance, set controllers, set extensions, migrate.
- **Minters:** Addresses in *MinterRole* can issue new tokens until *renounceIssuance*().
- **Controllers:** Global and partition-specific operators when *_isControllable* is true.

Recommendations

To further enhance the quality and maintainability of the NSDQ contract, the following recommendations are made:

1. **Reentrancy Protection**
    - Add the *nonReentrant* modifier to all transfer functions. For example:

      ```
      - function transfer(address to, uint256 value) external override re
      turns (bool) {
      + function transfer(address to, uint256 value) external override no
      nReentrant returns (bool) {
            _transferByDefaultPartitions(msg.sender, msg.sender, to, value,
      "");
            return true;
          }
      ```

    - Similarly update `transferFrom`, `transferWithData`, and `transferFromWithData`.
    - Alternatively, reorder external hook calls (`_callRecipientExtension`) to occur after all state changes, but `nonReentrant` is simplest.
2. **Fix Duplicate Modifier**
    - In *operatorRedeemByPartition*, remove the extra *nonReentrant*:

      ```
      - function operatorRedeemByPartition(...) external override nonReen
      trant nonReentrant {
      + function operatorRedeemByPartition(...) external override nonReen
      trant {
            // ...
          }
      ```

3. **Explicit Visibility**
    - Add explicit visibility to all helper functions. Example:

      ```
      - function _transferWithData(address from, address to, uint256 valu
      e) internal {
      + function _transferWithData(address from, address to, uint256 valu
      e) internal {
            // ...
          }
      ```

    - Ensure no internal or private function lacks a visibility specifier.
4. **Descriptive Error Messages or Custom Errors**

- Replace hex-only *require* messages with descriptive strings:

  ```
  - require(_balances[from] >= value, "52");
  + require(_balances[from] >= value, "NSDQ: insufficient balance");
  ```

- Or define and use custom errors for gas savings:

  ```
  error InsufficientBalance(uint256 available, uint256 required);

  // ...
  if (_balances[from] < value) {
      revert InsufficientBalance(_balances[from], value);
  }
  ```

5. **Unit Test Coverage (100% Branch Coverage)**
   - Use Hardhat + solidity-coverage or Foundry.
   - Write tests covering:
     - Successful and failing transfers, minting, and redemption scenarios.
     - Reentrancy attempts (using mock recipient contracts).
     - Access control checks (minter, controller, owner).
   - Add coverage script in package.json:

     ```
     {
       "scripts": {
         "test": "hardhat test",
         "coverage": "hardhat coverage"
       }
     }
     ```

6. **CI/CD Integration**
   - Add a GitHub Actions workflow:

     ```
     name: CI
     on: [push, pull_request]
     jobs:
       build:
         runs-on: ubuntu-latest
         steps:
           - uses: actions/checkout@v2
           - uses: actions/setup-node@v2
             with:
               node-version: 16
           - run: npm ci
           - run: npx slither .
           - run: npm test
           - run: npm run coverage
     ```

   - Optionally include Mythril for additional static analysis.

7. **NatSpec Documentation**
   Add /// @notice, /// @param, /// @return for all public/external functions.

Checked Items

The contract was audited for commonly known and specific vulnerabilities. Here is a summary of the items considered:

Item	Type	Description	Status
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly.	**Failed**
Integer Overflow and Underflow	SWC-101	Solidity ^0.8.x includes built-in overflow and underflow protection.	Not relevant
Outdated Compiler Version	SWC-102	Uses recent Solidity version ^0.8.30.	Passed
Floating Pragma	SWC-103	Contracts should deploy with a fixed compiler version.	Passed
Unchecked Call Return Value	SWC-104	Ensures the return value of calls is checked.	Passed
Access Control & Authorization	CWE-284	Properly implemented without unauthorized access to protected functions.	Passed
SELFDESTRUCT Instruction	SWC-106	Contract does not contain self-destruct functionality.	Not Relevant
Check-Effect-Interaction	SWC-107	Follows the pattern to prevent reentrancy attacks..	**Failed**
Assert Violation	SWC-110	Proper code execution prevents reaching a failing assert statement.	Passed
Deprecated Solidity Functions	SWC-111	No deprecated functions are used.	Passed
Delegatecall to Untrusted Callee	SWC-112	No delegatecall usage to untrusted addresses.	Not Relevant
DoS (Denial of Service)	SWC-113 SWC-128	No risks of DoS attacks through contract design.	Passed
Race Conditions	SWC-114	No race conditions or transaction order dependencies identified.	Passed
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	Passed

Block values as a proxy for time	SWC-116	Block numbers are not used as time proxies.	Passed
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155	Not applicable, as the contract does not use message signatures..	Not Relevant
Shadowing State Variable	SWC-119	State variables are not shadowed.	Passed
Weak Sources of Randomness	SWC-120	Randomness is not generated using block attributes.	Not Relevant
Incorrect Inheritance Order	SWC-125	Inheritance order is carefully specified.	Passed
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	External calls are only performed to trusted addresses.	Passed
Presence of unused variables	SWC-131	The code should not contain unused variables if this is not justified by design. No unused variables found, ensuring efficient code.	Passed
EIP standards violation	EIP	The contract adheres to EIP standards, particularly ERC-20.	Passed
Assets integrity	**Custom**	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed
User Balances manipulation	**Custom**	Contract owners or any other third party should not be able to access funds belonging to users.	Passed
Data Consistency	**Custom**	Smart contract data should be consistent all over the data flow.	Passed
Flashloan Attack	**Custom**	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used.	Not Relevant
Token Supply manipulation	**Custom**	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the customer.	Passed
Gas Limit and Loops	**Custom**	Code is optimized to avoid high gas usage and	Passed

		unbounded loops.	
Style guide violation	**Custom**	Style guides and best practices should be followed.	Passed
Requirements Compliance	**Custom**	The code should be compliant with the requirements provided by the Customer.	Passed
Environment Consistency	**Custom**	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Not Relevant
Secure Oracles Usage	**Custom**	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be 100%, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	**Failed**
Stable Imports	**Custom**	The code should not reference draft contracts, that may be changed in the future.	Passed

Findings

Critical

1) **Reentrancy**

- **Description:** Functions *transfer*, transferFrom, *transferWithData*, and *transferFromWithData* call *tokensReceived* before updating balances, enabling a re-entrant attack.
- **Location:** ERC1400.transfer, ERC1400.transferFrom, ERC1400._transferByDefaultPartitions
- **Recommendation:** Apply *nonReentrant* or reorder external hook calls after state changes.

High

1) **Duplicate Modifier**

- **Description:** *operatorRedeemByPartition* declares *nonReentrant* twice, causing compilation errors.
- **Recommendation:** Remove the redundant modifier.

Medium

No issues

Low

1) **Implicit Visibility**

- **Description:** Internal helper functions lack explicit *internal*/*private* visibility.
- **Recommendation:** Declare all internal functions with explicit visibility.

2) **Error Messaging**

- **Description:** *require* uses only hex codes (e.g., "*52*"), hindering diagnosis.
- **Recommendation:** Use descriptive revert messages or custom errors.

Disclaimers

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.