PUBLIC

Code Assessment

of the NST Smart Contracts

June 13, 2024

Produced for

MAKER

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help MakerDAO with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of NST according to Scope to support you in forming an opinion on their security risks.

MakerDAO introduces a new stablecoin token (NST, rebranded DAI) along with a permissionless converter for 1:1 conversions between DAI and NST. The NST is an ERC-20-compliant token, and the converter, DaiNst, enables seamless exchanges. The project also features NstJoin, which is the NST equivalent of DaiJoin.

In this latest version, NST has been made upgradable. The previously separate deployment report has been integrated.

The most critical subjects covered in our audit are security, functional correctness and seamless integration with the existing system. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1
• Specification Changed	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the NST repository. No documentation was made available for the initial review. A README has been added in Version 2.

The scope consists of three solidity smart contracts:

```
./src/DaiNst.sol
./src/Nst.sol
./src/NstJoin.sol
```

In Version 2, deployment scripts have been added to the scope of the review:

```
./deploy/NstDeploy.sol
./deploy/NstInit.sol
./deploy/NstInstance.sol
```

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	11 May 2023	d7dbf877d2792e8b23cfca97dce60c9f0375ec58	Initial Version
2	18 September 2023	8dea00f1545925dce63511e1710e2ec3c7fc59d2	Deployment Scripts
3	16 October 2023	93abbc714ffa5662cdb264865829752e2ea63df9	Updated Version
4	29 May 2024	78e4589c0ec60d5a29b66f56ca02824bc34ca6f1	UUPS Proxy
5	11 June 2024	45c9e126ce65f22bfcac796902e2433eb010f286	Fixed Readme

For the solidity smart contracts, the compiler version `0.8.16` was chosen. In Version 4 the compiler version `0.8.21` was chosen.

2.1.1 *Excluded from scope*

Any other file not explicitly mentioned in the scope section. In particular tests, external dependencies, and configuration files are not part of the audit scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

MakerDAO implements a rebranded version of the DAI token (New Stablecoin Token, NST), an immutable and permissionless converter that converts DAI to NST with a fixed rate (1 DAI:1 NST) and vice versa. Both NST and DAI are permissionless IOUs of `vat.dai`.

2.2.1 NST

NST is an ERC-20-compliant token with 18 decimals. The contract is controlled by privileged roles `wards` initialized with `msg.sender` in the constructor. Any address in `wards` has access to:

- Add a new ward by `rely()`.
- Remove a ward by `deny()`.
- Mint any amount of NST tokens to an address by `mint()`.

Token transfers work the same way as a normal ERC-20 token but with a few restrictions. Specifically, transfers to the zero address (address(0)) or the contract itself are not allowed. A user can also burn its tokens by calling `burn()` with its own address. In case the address specified is different from the `msg.sender`, the user will burn on behalf of others if its allowance is sufficient.

NST supports unlimited allowance by approving `max(uint256)`. In addition, `permit()` (ERC-2612) is provided for setting allowances with signatures. Note that the functionality also supports contracts validating signatures with `isValidSignature()` according to EIP-1271. If the signature length is not equal to 65 bytes, it is assumed the allowance owner is a contract, which will be queried for signature validation.

2.2.2 NstJoin

NstJoin works like DaiJoin, which allows users to withdraw their NST from the system into a standard ERC-20 token NST or burn their ERC-20 NST. It exposes the following functions:

- `join()`: moves `vat.dai` from this to the user and burns the ERC-20 NST tokens.
- `exit()`: moves `vat.dai` from msg.sender to this and mints new ERC-20 NST tokens.
- `dai()`: returns the address of ERC-20 NST address.
- `nst()`: returns the address of ERC-20 NST address.
- `vat()`: returns the address of the Vat.

Contrary to DaiJoin NstJoin implements no `cage()` and hence cannot be paused.

2.2.3 DaiNst

DaiNst is an immutable and permissionless converter between DAI and NST tokens with a fixed conversion rate 1 to 1.

- `daiToNst()` converts DAI to NST ERC-20 tokens. First, ERC-20 DAI tokens are transferred from msg.sender to this contract. Then, it calls `join()` on the DaiJoin and `exit()` on the NstJoin.
- `nstToDai()` converts NST to DAI ERC-20 tokens in a similar way.

2.2.4 Changes in Version 2

Deployment scripts have been added to the scope of the review.

The systems are deployed in two steps:

1. Some EOA deploys the contracts and - if necessary - changes the owner of these contracts to the `PauseProxy`.
2. A governance `Spell` with quorum executes the initialization of the contracts through the `PauseProxy`.

After deployment, the owner of the `Nst` contract is changed to the `PauseProxy`. `NstJoin` is deployed with the given `Nst` address and the address of the `Vat`. `DaiNst` is then deployed with the address of the `DaiJoin` and newly deployed `NstJoin` contracts.

During the initialization of the contracts, the deployment parameters are checked for validity. The owner of the `Nst` contract is switched from the `PauseProxy` to the `NstJoin` contract so that it is allowed to call `mint()` and `burn()`.

Finally, the addresses are added to the chainlog.

2.2.5 Changes in Version 3

- Functions `increaseAllowance` and `decreaseAllowance` have been removed. Only functions `approve` and `permit` remain to modify allowances.
- Permit functionality: Now, when validating a signature with a contract, if there is no code deployed at the given address, the transaction will revert with a clear error message.

2.2.6 Changes in Version 4

NST is now intended to be deployed behind an ERC-1967 proxy. For upgrading the UUPS pattern of ERC-1822 is used. The contract now inherits from OpenZeppelins UUPSUpgradeable which provides all required functionality.

- `_authorizeUpgrade()` has been overwritten to add access control, only `wards` (the Governance PauseProxy is expected to be the only ward capable of initiating updates) can upgrade the implementation.
- `getImplementation()` has been added returning the address of the current implementation which is retrieved from the defined storage slot.

2.2.7 Roles and Trust Model

Wards: Privileged roles in the NST contract. Fully trusted to not misbehave and never act against the interest of system users. Expected to be the Governance PauseProxy and NstJoin only.

Users: Users interacting with the public functions of the system. Untrusted.

Deployer: Executes the deployment scripts deploying the contracts. Untrusted, governance must inspect the deployment before accepting the initialization vote.

Governance (DSPauseProxy): Fully trusted. Must verify the deployment and initialize the system with the correct parameters.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Readme Is Outdated About Wards of NST Specification Changed

6.1 Readme Is Outdated About Wards of NST

Correctness Low Version 4 Specification Changed

CS-NST-001

README.md states the following for the NST token:

```
There should be only one `wards(address)` set, and that needs to be the `NstJoin` implementation.
```

To make use of the upgradeability feature introduced in Version 4, the Governance Pause proxy needs to be set as a ward.

Specification changed:

This sentence has been removed from README.md.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Deployment Verification

Note **Version 2**

Since deployment of the contracts is not performed by the governance directly, special care has to be taken that all contracts have been deployed correctly. While some variables can be checked upon initialization through the `PauseProxy`, some things have to be checked beforehand.

We therefore assume that all mappings in the deployed contracts are checked for any unwanted entries (by verifying the bytecode of the contract and then looking at the emitted events). This is especially crucial for the `wards` mappings.

7.2 Deviations From ERC Standards

Note **Version 4**

Parts of the code technically do not satisfy certain ERC standards and deviate from the specification. Note that these deviations are common.

As noted in the readme, the following proxy scheme is implemented:

```
The token uses the ERC-1822 UUPS pattern for upgradeability and the ERC-1967 proxy storage slots standard.
```

These standards have conflicting specifications, furthermore, the OZ implementation used does not follow ERC-1822 strictly. Note that the ERC-1822 specification contradicts itself in various parts.

- Using the storage slot defined in ERC-1967 (`bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1)`) to store the implementation address contradicts the ERC-1822 specification which requires the address to be stored at slot `keccak256("PROXIABLE")`.
- `proxiable()` and `updateCodeAddress()` are not implemented. However, note that while the standard specifies `proxiable()`, `proxiableUUID()` is used in the sample implementation.
- `proxiableUUID` is implemented and returns the address of the implementation stored at the slot defined by EIP-1967, not EIP-1822 (which corresponds to the slot actually used).

Similarly, the specification of the `permit` functionality (ERC-2612) may only approve if and only if:

```
r, s and v is a valid secp256k1 signature from owner of the message
```

NST extends the idea of ERC-2612 by also accepting "signatures" from smart contracts according to ERC-1271 (Standard Signature Validation Method for Contracts) and hence this may not hold. Note that other token contracts such as Lido's stETH or USDC have a similar deviation from EIP-2612.

7.3 NstToDai Can Be Paused if DaiJoin Is Paused

Note **Version 1**

The DaiNst converter itself is permissionless. However, if DaiJoin is paused, `NstToDai()` will be indirectly paused as `exit()` will revert on DaiJoin.

Note that this theoretically possible situation does not apply to the existing Maker's DaiJoin deployed at 0x9759A6Ac90977b93B58547b4A71c78317f391A28.

The only ward of this contract is its deployer contract DaiJoinFab, which is immutable and does not have the functionality to pause the DaiJoin by calling `cage()` nor to add more wards.