PUBLIC

# Code Assessment

## of the Savings USDS Smart Contracts

September 30, 2024

Produced for


by

# 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 Savings USDS according to Scope to support you in forming an opinion on their security risks.

MakerDAO implements Savings USDS, a tokenized implementation of a savings rate for USDS.

The most critical subjects covered in our audit are functional correctness, security of the assets and the proxy/upgradability pattern. Security regarding all the aforementioned subjects is high.

The general subjects covered include the specification, adherence to the ERC standards and optimisations.

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 | 0 |

# 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 Savings USDS repository based on the documentation files.

The scope consists of the two solidity smart contracts:

```
./src/ISNst.sol
./src/SNst.sol
```

and the deployment scripts:

```
./deploy/SNstDeploy.sol
./deploy/SNstInit.sol
./deploy/SNstInstance.sol
```

As of Version 4, the files have been renamed as a result of a rebranding. The files below are in Scope:

```
./src/ISUsds.sol
./src/SUsds.sol
```

and

```
./deploy/SUsdsDeploy.sol
./deploy/SUsdsInit.sol
./deploy/SUsdsInstance.sol
```

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

In Version 5, the following have been added to for the deployment of SUSDS token on L2:

```
./src/l2/SUsds.sol
./deploy/l2/SUsdsDeploy.sol
```

| V | Date | Commit Hash | Note |
|---|---|---|---|
| 1 | 5 June 2024 | eea30a68bbed2dc808698b19c96e8f4561701294 | Initial Version |
| 2 | 11 June 2024 | 47bfad404140e6c0ab0c5867b3f8345eac4e7556 | After Intermediate Report |
| 3 | 03 July 2024 | 79812602f210731bd4fba5e14e84ea2e27588563 | Finalization |
| 4 | 27 August 2024 | e5660deac9434c79d48ecde879c29f37fb89f2dc | Renaming |

| | | | |
|---|---|---|---|
| 5 | 13 September 2024 | e1d160aba17e95e8cec3d6bf50f310fbed9f28d6 | L2 Token |

For the solidity smart contracts, 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.

At the end of this report section we have added subsections for each of the changes accordingly to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

MakerDAO implements a tokenized (ERC-4626) implementation of a savings rate for NST. Compared to Savings DAI which utilized the Pot (DAI Saving Rate) this implementation combines all functionality in one sNST contract. Savings NST is designed to be deployed using an ERC-1967 Proxy, for upgradability the UUPS pattern is used.

### 2.2.1 sNST

The contract implements the sNST Token (ERC-4626, 18 decimals, "Savings NST") representing shares of NST. `chi`, the rate accumulator increases over time based on the rate `nsr`, this allows the distribution of savings reward. Calculation of the current rate can be done whenever needed.

Anyone may trigger an update of the savings rate and provision of the funds by calling `drip()`. The NST tokens are provided as follows: Unbacked debt (`sin`) is allocated to the VOW and the sNST's `dai` balance in the VAT increases. The tracking of the bad debt and total debt is updated accordingly. No limits are enforced. Using the `dai` balance in the VAT accounting, NST tokens are exited through NstJoin making these funds available at this contract.

An event `Drip` is emitted containing the new rate accumulator and the difference in balance.

Note that the debt at the VOW can be erased, if a sufficient surplus exists, by calling `VOW.heal()`.

The sNST implements the tokenized vaults standard (ERC-4626) with the NST as the underlying asset to track the users' share of the NST held by the sNST, which includes the rewards. The shares are computed as normalized amounts with the formula `depositedNstAmount * RAY / chi`, where `chi` is the always increasing rate accumulator which can be calculated in real-time.

The state-changing functions related to EIP 4626 are:

- `deposit(uint256 assets, address receiver)`: Mints sNST to the receiver for the given amount of NST transferred from `msg.sender`. Emits the `Deposit` and `Transfer` events.
- `mint(uint256 shares, address receiver)`: Mints the amount of shares to the receiver, transfers the corresponding amount of NST from `msg.sender`. Emits the `Deposit` and `Transfer` events.
- `withdraw(uint256 assets, address receiver, address owner)`: Withdraws the given amount of NST to `receiver` while burning shares from the `owner`. `Owner` must be `msg.sender`,

or `msg.sender` must have an allowance of `owner` to spend the sNST. Emits the `Transfer` and `Withdraw` events.

- `redeem(uint256 shares, address receiver, address owner)`: Redeems the given amount of sNST of `owner`. `Owner` must be `msg.sender`, or `msg.sender` must have an allowance of `owner` to spend the sNST. Transfers the corresponding amount of NST to `receiver`. Emits the `Transfer` and `Withdraw` events.

Additionally, the following functions have been added:

- `deposit(uint256 assets, address receiver, uint16 referral)`: Wrapper for the default `deposit()` function, additionally emits the `Referral` event.
- `mint(uint256 shares, address receiver, uint16 referral)`: Wrapper for the default `mint()` function, additionally emits the `Referral` event.

View functions related to EIP 4626:

- `asset()`: returns the address of the NST.
- `totalAsset()`: returns the totalSupply of sNST converted into NST.
- `convertToShares(uint256 assets)`: returns the amount of shares the contract would exchange for the amount of assets provided.
- `convertToAssets(uint256 shares)`: returns the amount of assets the contract would exchange for the amount of shares provided.
- `maxDeposit(address)`: hardcoded to `type(uint256).max;`.
- `previewDeposit(uint256 assets)`: returns the number of shares one would receive for this amount of assets at this block.
- `maxMint(address)`: hardcoded to `type(uint256).max;`.
- `previewMint(uint256 shares)`: returns the amount of assets needed to mint the given amount of shares at this block.
- `maxWithdraw(address owner)`: returns `convertToAssets(balanceOf[owner]);`, the maximum assets this address can withdraw.
- `previewWithdraw(uint256 assets)`: returns the exact amount of shares that would be burned if the caller withdraws this amount of asset in this block.
- `maxRedeem(address owner)`: returns `balanceOf[owner]`, the maximum amount of shares this address can redeem (its current balance).
- `previewRedeem(uint256 shares)`: returns the amount of assets the caller would receive if he spends this amount of shares in this block.

The calculation of the view functions is real-time since they calculate the current rate accumulator `chi` based on the time elapsed between the current block timestamp and the last update (`rho`).

All standard ERC-20 functions (`transfer`, `transferFrom`, `approve`) are implemented. Savings NST extends the ERC-20 features with ERC-2612 (Permit Extension) and further ERC-1271 (Standard Signature Validation for Contracts). Two entry points for the permit functionality are available allowing to pass either the aggregated signature as bytes or `v`, `r` and `s` separately. Signatures for permits can either be from EOAs or, using ERC-1271, be validated by a contract that allows contracts to act as they "signed" permits.

Furthermore, the following permissioned functions, callable by addresses bearing the `ward` role, are available:

- `rely(address usr)`: Adds an address to the `wards` mapping. Emits the `Rely` event.
- `deny(address usr)`: Removes an address from the `wards` mapping. Emits the the `Deny` event.

- `file(bytes32 what, uint256 data)`: allows to update the savings rate parameter (`nsr`). Enforced to be `>= RAY`. Emits the `File` event.

### 2.2.2 Upgradeability

sNST inherits from Openzeppelin's UUPSUpgradeable which provides all functionality for UUPS Proxies implementation contracts to facilitate upgradeability.

sNST overrides `_authorizeUpgrade()` adding access control to restrict implementation upgrades by `wards` (assumed to be the Governance Pause proxy exclusively) only. Furthermore `getImplementation()` has been added returning the address of the current implementation which is retrieved from the defined storage slot.

For the sNST Proxy the widely used OpenZeppelin implementation of ERC1967Proxy is used. All calls are executed as delegatecall to the implementation contract, the address of the implementation contract is stored at slot calculated as `bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1))`.

### 2.2.3 Deployment

Savings NST is deployed in two steps:

1. Some EOA deploys the contracts (sNST and a ERC1967Proxy). The owner of the proxy is switched to the `PauseProxy`.
2. A governance `Spell` with quorum executes the initialization of the contracts through the `PauseProxy`.

SNstDeploy implements `deploy()` which deploys the implementation contract sNST and an ERC1967Proxy. The constructor parameters are `nstJoin` (passed as input parameter) and `VOW` (queried from the hardcoded chainlog address) for the sNST implementation contract. For the proxy, the constructor parameters are the address of the newly deployed implementation contract and the function selector of `initialize`. After deployment, the owner of the Proxy is switched to the Governance PauseProxy.

During the governance spell, the following sanity checks are performed:

1. The version of the implementation is ensured to be Version 1.
2. The implementation is validated to be the expected address of `sNstImp`
3. The sNst instance's VAT, nstJoin, nst and VOW addresses are validated to be the expected ones.

After the sanity checks, the following actions are performed:

1. The `nsr` is set.
2. sNST is added as ward in the VAT.
3. The addresses of SNST and SNST_IMP are added to the chainlog

### 2.2.4 Changes in Version 4

NST has been renamed to USDS. Hence, sNST has been renamed to sUSDS. Additionally, `drip` is now called before setting the savings rate in the initialization script.

### 2.2.5 Changes in Version 5

As of this version, a new SUSDS token contract is introduced to be used as the L2 token for SUSDS. Different from L1, the SUSDS on L2 is a standard UUPS upgradeable token without staking logic. The implementation is a common ERC-20 token (same implementation as the USDS contract with renamings to sUSDS, see audit report).

Note that the deployment script for the L2 token is slightly different from the L1 token. The process is described below:

1. Deploy the L2 SUSDS contract as the implementation contract (same as for the L1 token).
2. Deploy the `ERC1967Proxy` contract and initialize the contract. The implementation contract is the deployed L2 USDS contract (same as for the L1 token).
3. Switch the owner to the intended owner (same as for the L1 token).
4. Note that in contrast to the L1 token, no init script is provided. Namely, that is due to the L2 bridge spells performing `rely` on the tokens (bridge is minter, see for example OP Token Bridge). However, some sanity checks are not performed and should be performed by governance before voting on a spell (e.g. version check, implementation check).

### 2.2.6 Roles & Trust Model

Wards: Privileged roles in the SUSDS contract. Fully trusted. On L1, it is expected to be the Governance PauseProxy only. On L2, it is expected to be the L2 Governance Relay and the L2 Token Bridge. Addresses holding the ward role can update the implementation of the Proxy and hence change the behavior and state of the contract. Further, wards can add/remove more wards and update the `ssr` parameter.

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:

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 | 0 |
| Informational Findings | 1 |

- Discrepancy With NST Code Corrected

## 6.1 Discrepancy With NST

Informational Version 1 Code Corrected

*CS-SNST-003*

Parts of the functionality of the sNST should be identical to the corresponding implementation in the NST. However, there are a few discrepancies:

1. `_isValidSignature`: The NST checks that the code size of the call target is greater than zero before the static call `isValidSignature` is made. Additionally, `encodeWithSelector` is used instead of `encodeCall`.
2. Additionally, the comments arguing about the safety of unchecked operations are missing in the sNST contract.

While the impact on security is minimal, consistency between the contracts could help.

---

**Code corrected:**

The code has been updated to align with the NST code.

# 7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

## 7.1 More Events in Initialization Possible

**Informational** **Version 1** **Acknowledged**

*CS-SNST-001*

The `initialize` function only emits the `Rely` event. However, for the initialization of `nsr` and `chi` events could be emitted (`File` and `Drip` respectively).

---

**Acknowledged:**

MakerDAO replied:

```
This is the typical pattern in Maker's constructors, to usually just emit the Rely event.
We are aiming for the initialize function to resemble that. We don't view it as very
important either way.
```

## 7.2 Vow Immutable

**Informational** **Version 1** **Acknowledged**

*CS-SNST-002*

The sNST contract has an immutable `vow` which is inconsistent with other contracts that, in contrast, have a mutable `vow`. However, due to the upgradeable nature of the contract, the `vow` can still be updated by a contract upgrade.

---

**Acknowledged:**

MakerDAO replied:

```
The chances to update vow exist but they are really low (never happened so far).
As sNST is upgradable we can still update it through changing the implementation in case is needed.
```

# 8 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.

## 8.1 Deployment Verification

Note Version 1

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 the initcode, bytecode, traces and storage (e.g. mappings) are checked for unintended entries, calls or similar. This is especially crucial for any value stored in a mapping array or similar (e.g. could break access control which could lead to unexpected contract implementation upgrades and hence result in stealing of funds).

Additionally, it is of utmost importance that no allowance is given to unexpected addresses (e.g. NST approval to arbitrary addresses could have been given in the constructor).

## 8.2 Deviations From ERC Standards

Note Version 1

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.

## 8.3 Drip Before Changing NSR

**Note** **Version 1**

Governance should be aware that in governance spells `drip()` should typically be called before changing the NSR with `file()`. If not, some unexpected behaviour may occur. For example, `chi` could increase too much (or not enough).

---

As of Version 3, it is enforce that `drip()` has been called before `file()` and, thus, the side-effects cannot occur anymore. As of Version 4, the init script calls `drip()` before the `file()`. Thus, it is not required to call it separately.

## 8.4 End Considerations

**Note** **Version 1**

Governance should be aware that, in case of a governance-assisted shutdown, the sNST should be handled. For example, the NSR could be set to one (RAY) to not generate any further yield.