PUBLIC

Code Assessment

of the Sky stUSDS

Smart Contracts

August 12, 2025

Produced for

Sky

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

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

Sky implements stUSDS, an ERC-4626 compliant token that provides capital for SKY staking leverage hence segregates the risk of SKY-backed borrowing.

The most critical subjects covered in our audit are functional correctness, access control, and precision of arithmetic operations. Security regarding all the aforementioned subjects is high.

The general subjects covered are front-running and correct integration with other SKY system components. 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	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 Sky stUSDS repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	04 August 2025	09b7eee515866d725955894ce1d00a5c09ce23b0	Initial Version

For the solidity smart contracts, the compiler version `0.8.21` was chosen. The `evm_version` is set to `shanghai`.

The following files are in scope of this review:

```
src/
    StUsds.sol
    StUsdsMom.sol
    StUsdsRateSetter.sol

deploy/
    StUsdsDeploy.sol
    StUsdsInit.sol
    StUsdsInstance.sol
```

2.1.1 Excluded from scope

All files not listed above including the tests are out of scope.

stUSDS is expected to work with an updated Lockstake Engine Clipper, for more details please consult the Lockstake review.

The tokenomics and parameter selection is out of scope.

Further, it is assumed `Jug.base==0` holds.

2.2 System Overview

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

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

Sky offers stUSDS, an ERC-4626 compliant token that provides capital for SKY staking leverage hence segregates the risk of SKY-backed borrowing.

2.2.1 stUSDS

In general, stUSDS follows the design of sUSDS (Savings USDS):

- It follows Sky's standard authorization mechanism: privileged functions with auth modifier are restricted to wards configured with `rely()` and `deny()`.
- It is an ERC-20 token with 18 decimals and represents shares of underlying USDS.
- It extends 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.
- It is ERC-4626 compliant with USDS as the underlying token. Wrappers for deposit and mint are further provided to additionally emit the Referral events.
- `chi`, the rate accumulator increases over time based on the rate `str` (Staked USDS rate), allows the distribution of staking reward. Calculation of the current rate can be done whenever needed.

To fund staked-SKY-backed borrowing in the Lockstake Engine, stUSDS implements a different tokenomics:

- Its interest rate is determined by the utilization ratio, namely the total debt generated in Lockstake over the supplied USDS in stUSDS.
- Only idle USDS can be withdrawn from stUSDS since an equivalent amount of Lockstake debt (active debt and debt on auction) is locked.
- If bad debt occurs during a Lockstake liquidation, the deposit of stUSDS will be slashed to cover it as much as possible.
- Governance may also directly trigger a slash of stUSDS deposit.

Two risk parameters are introduced in stUSDS:

- `cap`: restricts the max deposits.
- `line`: restricts the max borrow ceiling.

A general overview of the integration and interaction flow is outlined below:

1. Deposit: Users can deposit USDS for stUSDS according to the up-to-date rate accumulator `chi`. The deposit cap is respected and `line` (debt limit) of Lockstake is increased.
2. Rate Update: The rate accumulator can be updated permissionlessly with `drip()`, which accrues with last interest rate `str` and mints USDS with Vat and Join interactions.
3. Slash: A privileged function `cut()` is implemented to slash USDS deposit by decreasing the rate accumulator `chi`, the Lockstake's `line` is updated accordingly.
4. Withdrawal: Users can burn stUSDS in exchange for USDS with the current rate accumulator `chi`. Funds that back the existing Lockstake debt cannot be withdrawn and the `line` of Lockstake is decreased.

A privileged function `file(bytes32 what, uint256 data)` allows wards to update the staking rate `str` (enforced to be `>= RAY`), the `line` and the `cap`.

2.2.2 StUsdsRateSetter

The staking rate (`str`) of stUSDS is expected to keep close track of the utilization ratio, while there is no feedback loop upon changes of stUSDS deposits or Lockstake debt. Hence a permissioned keeper approach is adopted with the StUsdsRateSetter contract to "synchronize" the staking rate and the ilk stability fee, and update the cap / line.

Similarly, this contract follows Sky's standard authorization mechanism: privileged functions with auth modifier are restricted to wards configured with `rely()` and `deny()`. Wards are responsible to set the parameters with `file()`:

- `bad`: Circuit breaker, to pause the rate update with `set()`.
- `tau`: Cooldown period between rate changes in seconds.
- `toc`: Last time when rates were updated (Unix timestamp).
- `maxLine`: The maximum `line` can be configured in stUSDS.
- `maxCap`: The maximum `cap` can be configured in stUSDS.
- `min`: The minimum annual BPS of either `str` or `duty`.
- `max`: The maximum annual BPS of either `str` or `duty`.
- `step`: The maximum delta in BPS that can be allowed in one `str` or `duty` update.

Keepers (`buds`) can be configured with `kiss()` and `diss()`. The keepers are expected to frequently run offchain algorithms given the current stUSDS deposits and Lockstake debt to update the stUSDS's `str`, `cap`, `line` and the Lockstake's `ilk.duty`:

- Consecutive updates should follow the cooldown period.
- Each update can set four parameters: `line`, `cap`, `str`, and `duty`.
- The Conv contract is consulted for conversions between BPS and per second rate.

2.2.3 stUsdsMom

The stUsdsMom contract is controlled by an owner who has privilege to transfer the ownership (`setOwner()`) and set the authority (`setAuthority()`). It implements `isAuthorized` access control where owner or ones who pass `canCall()` may trigger the following functions:

- `dissRateSetterBud`: Remove a keeper from the StUsdsRateSetter.
- `haltRateSetter`: Halt the StUsdsRateSetter by setting its `bad` flag to 1.
- `zeroCap`: Pause deposits to stUSDS by settings its `cap` to 0.
- `zeroLine`: Pause new borrowing on Lockstake by settings its `ilk.line` to 0.

2.2.4 Upgradeability

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

stUSDS 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 stUSDS Proxy the widely used OpenZeppelin implementation of ERC1967Proxy is used. All calls are executed as delegatecalls to the implementation contract, the address of the implementation contract is stored at slot calculated as:

`bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1))`.

2.2.5 Deployment

Savings stUSDS is deployed in two steps:

1. Some EOA deploys the contracts (stUSDS implementation, a ERC1967Proxy, the StUsdsRateSetter, and the StUsdsMom). The owner of the proxy is switched to the `PauseProxy`

and an StUsdsInstance is returned. Chainlog is queried to retrieve the system contract during the deployment.

2. A governance `Spell` with quorum executes the initialization (StUsdsInit) of the contracts through the `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 `stUsdsImp`
3. The stUSDS instance's constructor parameters UsdsJoin, Jug, Clip, Vow addresses are validated to be the expected ones.
4. The `str` to be configured is validated to be within 0~10000 BPS.
5. The `rateSetter` is wired correctly to the stUSDS and SP-BEAM's converter, and the mom is wired to the stUSDS.

After the sanity checks, the following actions are performed:

1. stUSDS is added as ward in the Vat.
2. The Lockstake ilk is removed from AutoLine.
3. The `str`, `cap`, and `line` are set.
4. The rateSetter is added as ward in the Jug to set the `duty`.
5. The rateSetter is added as ward in the stUSDS to set the `str`.
6. The `tau`, `maxLine`, and `maxCap` are configured in rateSetter.
7. The `max`, `min`, and `step` are configured for both `STR` and ilk in rateSetter.
8. Keepers are added with `kiss()`.
9. The mom contract is configured by being ward of stUSDS and rateSetter. The `MCD_ADM` is set as its authority.
10. The addresses of stUSDS proxy, implementation, rateSetter, and the mom are added to the chainlog.

2.3 Trust Model

Wards of stUSDS: fully trusted; assumed to be the Governance PauseProxy, the stUsdsMom, the rateSetter, and the updated Lockstake Clipper. Addresses holding the ward role has fully control over the contract and its underlying:

1. Update the implementation of the Proxy and hence change the behavior and state of the contract
2. Add/remove more wards and update the `str`, `cap`, and `line` parameters.
3. Slash the USDS deposits.

Wards of StUsdsRateSetter: fully trusted; assumed to be the Governance PauseProxy. It has the privilege to set the buds and configurations hence manipulate stUSDS's `str` and Lockstake's `ilk.duty`.

Buds of StUsdsRateSetter: semi-trusted; plays the keeper role to periodically set the stUSDS's `line`, `cap`, `str` and `ilk.duty`. In the worst case they may DoS updates or tweak the updates in its owner favor. A compromised keeper is expected to be removed by the Mom or with a governance spell.

Owner of stUSDS Mom: fully trusted; assumed to be the Governance PauseProxy. Can trigger the circuit breaker in different levels for stUSDS and Sky backed borrowing.

Authority of stUSDS Mom: fully trusted; assumed to be the `MCD_ADM`.

Suppliers of stUSDS: untrusted. Expected to deposit and withdraw to maximize their returns.

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 Open Findings

In this section, we describe our findings. 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 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.

6.1 Bad Debt May Not Be Socialized

Note Version 1

By design stUSDS aims to ensure all staked-SKY-backed borrowing is funded by segregated risk capital: the USDS deposited in stUSDS. However, it is not guaranteed the bad debt can be covered by the stUSDS depositors. In the worst case, the bad debt will be accrued on Vow, the system surplus buffer. Several scenarios are outlined below:

1. At launch time there is existing debt on Lockstake Engine. If not enough USDS deposits are incentivized, potential bad debt may still be accrued on the Vow in case of a large liquidation.
2. Further, it is assumed the USDS deposited to stUSDS are eventually backed by other collaterals. If the deposited USDS is minted from thin air (i.e. generated from the allocation system), upon slashing it eventually becomes system bad debt.

6.2 Deposit Risks

Note Version 1

As clearly documented in the project README, the depositors of stUSDS are exposed to several risks, in particular:

- Slashing: `cut()` can happen upon bad debt during liquidation or governance invocation. In such cases, depositors may lose a part of or all their funds.
- Withdrawal: Only idle funds can be withdrawn, so users may not be able to withdraw their funds until sufficient new deposits are made or debt is repaid. In particular, during the launch phase before deposits catch up with existing debt, users will not be able to withdraw. During normal operation, the borrow and supply rate mechanisms are designed to adjust so that there should generally be free funds available (out of scope of this review).
- Griefing: Users may not be able to deposit if someone front-runs and deposit to cap; similarly users may not be able to withdraw if the remaining idle funds are withdrawn or borrowed by front-running.

6.3 ERC-4626 Considerations

Note Version 1

stUSDS is ERC-4626 compliant, however, the following behaviors should be considered for 3rd-party integrations:

- `maxDeposit()` and `maxMint()` will return 0 if the cap is reached, however, calling `deposit()` or `mint()` with 0 amount may still revert if the existing total assets already exceed the cap.
- `maxWithdraw()` and `maxRedeem()` will return 0 if there are no idle funds available, however, calling `withdraw()` or `redeem()` with 0 amount may still revert if the amount to be locked already exceeds the total assets.

- If `chi==0` due to slashing, `deposit()` and `withdraw()` with non-zero assets will always revert due to division by zero. `mint()` and `redeem()` will revert due to the shares correspond to 0 asset.
- In general, the total assets used in the computation above or returned by `totalAssets()` may be over-estimated due to the potential bad debt generated from the ongoing liquidations.

6.4 Vat Line Should Consider rateSetter maxLine

Note **Version 1**

In stUSDS, `_setLine()` only updates the individual `line` in the Vat, leaving the `Line` unchanged. It is assumed that `vat.Line` is big enough that won't require constant updates.

Given the individual line is frequently changed by stUSDS, the `Line` should factor in the `rateSetter.maxLine`. Otherwise, USDS minting with other ilks may be blocked by staked-SKY-backed borrowing.

6.5 `str` and `duty` Considerations

Note **Version 1**

The keepers are expected to trigger the rates (`str` and `duty`), `cap` and `line` updates with an offchain algorithms according to the current and past onchain status. In particular:

- `duty` is expected be greater than `str`, otherwise the system could have a net loss.
- Both rates should be less than `conv.MAX_BPS_IN` otherwise the `btor()` conversion will fail.