

sUSDS Distribution Contract Security Assessment

Forward Research

Version 1.1 – June 2, 2025

Prepared By
Parnian Alimi

Prepared For
Ivan Morozov
Christian Smorra

1 Executive Summary

Synopsis

In May of 2025, Forward Research engaged NCC Group to conduct a security assessment of an Ethereum *Distribution* contract that utilizes the sUSDS vault on the Sky ecosystem to collect interest on the user deposited values. Users of the program deposit USDS tokens in the contract in exchange for AO tokens. The vault funnels these funds into the sUSDS vault in order to earn interest. While the *Distribution* contract is still upgradeable, its owner can upgrade the contract and configure the proxy admin. A refunder authority (or its fallback address) can proactively eject a user.

This engagement was delivered remotely by one consultant over 2 person-days. In a subsequent phase, all findings were retested and are considered to be fixed.

Scope

NCC Group's evaluation included the *Distribution* and the *Refunder* Solidity contracts in the `sky-prebridge` repository at commit `49a1337`.

Limitations

The NCC Group consultant focused their review on the scope above, paying attention to common programming pitfalls and security issues. Note that this vault is part of a larger bridge, which was not reviewed in its entirety. The consultant achieved good coverage of the in-scope items.

Key Findings

The assessment uncovered 3 findings:

- Finding "User's Recorded Deposit Amount May be Inaccurate" explains a potentially inaccurate user deposit accounting during withdrawals.
- Finding "Vault's Deposit Amount Should be Validated" recommends that the staking logic ensures that the vault only withdraws the expected USDS token amount.
- Finding "Potential Reentrancy Risk" recommends that the functions that invoke an external contract follow the Checks-Effects-Interactions pattern.

One minor implementation note is captured in the Implementation Notes section.

Strategic Recommendations

Consider addressing the findings presented in this review.

2 Dashboard

Target Data

Name	AO Prebridge for USDS with SSR Integration
Type	Source Code
Platforms	Solidity
Environment	Ethereum

Engagement Data

Type	Implementation Review
Method	Code-Assisted
Dates	2025-05-27 to 2025-05-30
Consultants	1
Level of Effort	2 person-days

Targets

***Distribution* Contract**	An ownable and upgradeable contract that stakes user's deposits in the sUSDS vault.
***Refunder* Contract**	An ownable contract that can proactively eject a batch of users.

Finding Breakdown

Critical issues	0
High issues	0
Medium issues	0
Low issues	3
Informational issues	0
Total issues	**3**

Category Breakdown

Data Validation	1
Security Improvement Opportunity	1
Session Management	1

Critical High Medium Low Informational



3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Potential Reentrancy Risk	Fixed	2UE	Low
Vault's Deposit Amount Should be Validated	Fixed	QKB	Low
User's Recorded Deposit Amount May be Inaccurate	Fixed	6Q2	Low



4 Finding Details

Low

Potential Reentrancy Risk

Overall Risk	Low	**Finding ID**	NCC-E025131-2UE
Impact	Low	**Category**	Security Improvement Opportunity
Exploitability	Low	**Status**	Fixed

Impact

Provided that the sUSDS vault is implemented as a proxy, all logic in the *Distribution* contract that interact with the sUSDS vault must follow the checks-effects-interactions pattern to prevent reentrancy type attacks.

Description

Reentrancy in `_stake()`

In the following code snippet, the `sefeTransferFrom()` call on line 188 and the `_depositToVault()` call on line 191 effectively invoke external contracts (the USDS ERC20 token contract and the sUSDS vault contract respectively). These external calls are followed by state updates and an event emission. This order of operations creates a vulnerability where the `depositToken` contract or the vault contract could reenter the contract through the `stake()` function or other public API, thereby exploiting the contract's un-updated state.

```
183 function _stake(address user_, uint256 amount_, bytes32 arweaveAddress_) private {
184     require(amount_ > 0, "DS: nothing to stake");
185
186     UserData storage userData = _usersData[user_];
187
188     IERC20(depositToken).safeTransferFrom(_msgSender(), address(this), amount_);
189
190     // only credit the user for the actual amount that landed in the SSR
191     amount_ = _depositToVault(amount_);
192
193     require((userData.deposited + amount_) >= minimalStake, "DS: amount too low");
194
195     totalDepositedInVault += amount_;
196     lastUpdate = uint128(block.timestamp);
197
198     // Update user data
199     userData.lastStake = uint128(block.timestamp);
200     userData.deposited += amount_;
201
202     emit UserStaked(user_, amount_, arweaveAddress_);
203 }
```

The severity of the exploit is constrained by the amount the user (`_msgSender()`) has approved the *Distribution* contract to spend. Additionally, it's important to note that the *Distribution* contract's initializer approves the vault to transfer unlimited amounts from the USDS token contract on behalf of the *Distribution* contract. As a result, the user's approved amount becomes the primary limiting factor.

To mitigate reentrancy risks, it is essential to follow the Checks-Effects-Interactions pattern as recommended by the solidity documentation.



Reentrancy in `ejectStakedFunds()`

The `ejectStakedFunds()` implementation largely adheres to the Checks-Effects-Interactions pattern, with one exception: it emits the `UserWithdrawn` event after invoking an external contract. This seemingly minor deviation can be exploited to emit events out of order if a malicious `depositToken` contract reenters the *Distribution* contract during the external call.

```
334 function ejectStakedFunds(address user) external onlyRefunder {
335     require(user != address(0), "DS: user is the zero address");
336
337     UserData storage userData = _usersData[user];
338     uint256 recordedAmount_ = userData.deposited;
339     if (recordedAmount_ == 0) {
340         return;
341     }
342
343     // Determine the assets that are currently redeemable
344     uint256 looseAssets_ = IERC20(depositToken).balanceOf(address(this));
345     uint256 shareAssets_ = vault.convertToAssets(vault.balanceOf(address(this)));
346     uint256 redeemable_ = looseAssets_ + shareAssets_;
347
348     // Cap transfer to what is actually redeemable
349     uint256 amountToTransfer_ = recordedAmount_ > redeemable_ ? redeemable_ :
    ↳ recordedAmount_;
350     require(amountToTransfer_ > 0, "DS: no assets redeemable");
351
352     // Update user and global accounting BEFORE any external calls
353     userData.deposited = recordedAmount_ - amountToTransfer_;
354     totalDepositedInVault -= amountToTransfer_;
355     lastUpdate = uint128(block.timestamp);
356
357     // Pull additional assets from the vault only if required
358     if (amountToTransfer_ > looseAssets_) {
359         _withdrawFromVault(amountToTransfer_ - looseAssets_);
360     }
361
362     // Transfer the redeemable assets to the user
363     IERC20(depositToken).safeTransfer(user, amountToTransfer_);
364
365     // Use zero bytes32 for compatibility with existing analytics tooling
366     emit UserWithdrawn(user, amountToTransfer_, 0x0);
367 }
```

Recommendation

Consider following the Checks-Effects-Interactions pattern or implement a reentrancy guard as described in the OpenZeppelin documentations.

Location

- *Distribution.sol*#L183-L203
- *Distribution.sol*#L334-L367

Retest Results

2025-06-02 – Fixed

As part of PR #1, the recommended reentrancy guard pattern has been implemented in the *Distribution* contract. Therefore, this finding is considered fixed once this PR merges.

Low

Vault's Deposit Amount Should be Validated

Overall Risk	Low	**Finding ID**	NCC-E025131-QKB
Impact	Low	**Category**	Data Validation
Exploitability	Undetermined	**Status**	Fixed

Impact

The *Distribution* contract pre-approves the sUSDS vault to withdraw an unbounded amount from its USDS balance. Despite this, the staking implementation should enforce checks to ensure that the vault does not withdraw more than the expected amount from the *Distribution* contract's account.

Description

The `stake()` API accepts a user's deposit and stakes it in the vault. The `_depositToVault()` helper function, depicted below, should ensure that the actual deposited amount (`actualDeposited_`) is equal to or slightly greater than the user's deposit amount. If the actual deposited amount significantly exceeds the expected amount, the call should revert to prevent unexpected behavior.

```
141 function _depositToVault(uint256 amount_) private returns (uint256) {
142     require(amount_ > 0, "DS: nothing to join");
143
144     uint256 preBalance_ = IERC20(depositToken).balanceOf(address(this));
145
146     // Deposit USDS into the vault using the configured referral code; the call reverts on
        ↳ failure (eg. insufficient allowance)
147     vault.deposit(amount_, address(this), depositReferralCode);
148
149     uint256 postBalance_ = IERC20(depositToken).balanceOf(address(this));
150     uint256 actualDeposited_ = preBalance_ - postBalance_;
151     require(actualDeposited_ > 0, "DS: SSR deposit failed");
152
153     return actualDeposited_;
154 }
```

The sUSDS vault at the `0xa3931d71877C0E7a3148CB7Eb4463524FEc27fbD` address is a proxy contract and can be upgraded in the future. This introduces the risk of it being replaced with malicious code. Therefore, it is important to implement safeguards to mitigate potential unauthorized or unexpected withdrawals.

It is important to note that this measure does not prevent the vault from draining the Distribution contract's USDS balance without an explicit invocation from the contract itself. This is an acknowledged risk and the reason this finding is classified as low severity.

Recommendation

Ensure that `actualDeposited_ - amount_` remains within a predefined acceptable margin to account for minor rounding or slippage.

Location

- *Distribution.sol*#L141



Retest Results

2025-06-02 – Fixed

As part of PR #1, the recommended check has been implemented in the *Distribution* contract. Therefore, this finding is considered fixed once this PR merges.



Low

User's Recorded Deposit Amount May be Inaccurate

Overall Risk	Low	**Finding ID**	NCC-E025131-6Q2
Impact	Low	**Category**	Session Management
Exploitability	None	**Status**	Fixed

Impact

Failure to accurately account for a user's withdrawal may result in inaccurate deposit accounting.

Description

The `_withdraw()` implementation, copied below, subtracts the minimum of the `amount_` or `depositTokenContractBalance_` values from the user's deposited balance (`userData.deposited`). However, according to the EIP-4626's description the `convertToAssets()` API returns an average value of the assets, and therefore the `amount_` may be less[1] than the `ssrExitedAmount_`, which will result in an inaccurate user balance accounting.

```
214 function _withdraw(address user_, uint256 amount_, bytes32 arweaveAddress_) private {
215     require(amount_ > 0, "DS: nothing to withdraw");
216     UserData storage userData = _usersData[user_];
217
218     uint256 deposited_ = userData.deposited;
219     require(deposited_ > 0, "DS: user isn't staked");
220
221     if (amount_ > deposited_) {
222         amount_ = deposited_;
223     }
224
225     uint256 newDeposited_;
226     require((block.timestamp > userData.lastStake + withdrawLockPeriodAfterStake), "DS:
    ↳ withdraw is locked");
227
228     uint256 depositTokenContractBalance_ =
    ↳ vault.convertToAssets(vault.balanceOf(address(this)));
229     if (amount_ > depositTokenContractBalance_) {
230         amount_ = depositTokenContractBalance_;
231     }
232
233     newDeposited_ = deposited_ - amount_;
234     require(newDeposited_ >= minimalStake || newDeposited_ == 0, "DS: invalid withdraw
    ↳ amount");
235
236     // Update user data
237     userData.deposited = newDeposited_;
238
239     totalDepositedInVault -= amount_;
240     lastUpdate = uint128(block.timestamp);
241
242     uint256 ssrExitedAmount_ = _withdrawFromVault(amount_);
```

1. The EIP-4626's security considerations section specifies that in order to avoid ambiguity and ensure consistency, the `convertTo` implementations MUST always round their results towards 0.



```
243     IERC20(depositToken).safeTransfer(user_, ssrExitedAmount_);
244
245     emit UserWithdrawn(user_, amount_, arweaveAddress_);
246 }
```

Recommendation

Update the user's data with the `ssrExitedAmount_` value instead of the `amount_` value. Alternatively, consider using the EIP-4626's `previewWithdraw()` API.

Location

- *Distribution.sol*#L237

Retest Results

2025-06-02 – Fixed

As part of PR #1, the call to the `convertToAssets()` has been updated to `previewRedeem()`, as recommended by this finding. Therefore, this finding is considered fixed once this PR merges.

Client Response

Why the feared mismatch is practically impossible:

1. EIP-4626 specification
   - `convertToAssets()` MUST round **down** towards 0 (see "Rationale", §Methods, EIP-4626).
   - Therefore its error is strictly less than 1 smallest-unit of USDS.
2. `vault.withdraw(amount_, …)` (invoked by `_withdrawFromVault`) must send **exactly** `amount_` assets or revert; it never returns a different figure. Consequently `ssrExitedAmount_ == amount_` unless the call reverts.
3. Both per-user (`userData.deposited`) and global (`totalDepositedInVault`) counters are decremented by that same `amount_`, keeping bookkeeping exact to the wei. If the vault really holds (say) 100 USDS but `convertToAssets` rounds it to 99, a user trying to withdraw the last 100 USDS will be limited to 99 and may need to:
   - wait for the next yield tick (exchange-rate ↑ by ≥ 1 wei), or
   - withdraw in two transactions.

No principal is lost; at most a temporary UX nuisance exists.

Impact:

- Financial loss: < 1 wei per edge case – negligible.
- Exploitability: none; rounding is deterministic, not attack-controlled.
- Operational risk: brief "can't withdraw full balance" until yield accrues.



5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.



Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.



6 Implementation Notes

This section consists of additional notes, comments, and observations from the review that do not present security concerns, but that may nevertheless be of interest.

Ignored Return Value

The `_withdrawFromVault()` helper function withdraws staked sUSDS tokens from the vault which is utilized by the `_withdraw()` (to withdraw user's stake), `ejectStakedFunds()` (to eject users), and `bridgeOverplus()` (to transfer the overplus to the AO wallet) functions. The `_withdrawFromVault()` function returns the actual exited balance from the vault:

```
160 function _withdrawFromVault(uint256 amount_) private returns (uint256) {
161     require(amount_ > 0, "DS: nothing to exit");
162
163     uint256 preBalance_ = IERC20(depositToken).balanceOf(address(this));
164
165     // Pull assets out - falls back to vault's rounding rules internally.
166     vault.withdraw(amount_, address(this), address(this));
167
168     uint256 postBalance_ = IERC20(depositToken).balanceOf(address(this));
169     uint256 actualExited_ = postBalance_ - preBalance_;
170     require(actualExited_ > 0, "DS: SSR withdraw failed");
171
172     return actualExited_;
173 }
```

However, the `ejectStakedFunds()` function's call to `_withdrawFromVault()` ignores its returned value, and assumes that the exited amount is exactly `amountToTransfer_ - looseAssets_`. If the exited amount is less than that, the *Distribution* contract's token balance will be less than the `amountToTransfer_`, and the call will fail. This edge case may not be rare since the `vault.convertToAssets()` call on line 345 may return an estimate of the assets' value according to the EIP-4626 documentation:

> The `convertTo` functions serve as rough estimates that do not account for operation specific details like withdrawal fees, etc. They were included for frontends and applications that need an average value of shares or assets, not an exact value possibly including slippage or other fees. For applications that need an exact value that attempts to account for fees and slippage we have included a corresponding `preview` function to match each mutable function. These functions must not account for deposit or withdrawal limits, to ensure they are easily composable, the `max` functions are provided for that purpose.

```
334 function ejectStakedFunds(address user) external onlyRefunder {
335     require(user != address(0), "DS: user is the zero address");
336
337     UserData storage userData = _usersData[user];
338     uint256 recordedAmount_ = userData.deposited;
339     if (recordedAmount_ == 0) {
340         return;
341     }
342
343     // Determine the assets that are currently redeemable
344     uint256 looseAssets_ = IERC20(depositToken).balanceOf(address(this));
345     uint256 shareAssets_ = vault.convertToAssets(vault.balanceOf(address(this)));
346     uint256 redeemable_ = looseAssets_ + shareAssets_;
347
348     // Cap transfer to what is actually redeemable
```

```
349     uint256 amountToTransfer_ = recordedAmount_ > redeemable_ ? redeemable_ :
        ↳ recordedAmount_;
350     require(amountToTransfer_ > 0, "DS: no assets redeemable");
351
352     // Update user and global accounting BEFORE any external calls
353     userData.deposited = recordedAmount_ - amountToTransfer_;
354     totalDepositedInVault -= amountToTransfer_;
355     lastUpdate = uint128(block.timestamp);
356
357     // Pull additional assets from the vault only if required
358     if (amountToTransfer_ > looseAssets_) {
359         _withdrawFromVault(amountToTransfer_ - looseAssets_);
360     }
361
362     // Transfer the redeemable assets to the user
363     IERC20(depositToken).safeTransfer(user, amountToTransfer_);
364
365     // Use zero bytes32 for compatibility with existing analytics tooling
366     emit UserWithdrawn(user, amountToTransfer_, 0x0);
367 }
```

The `bridgeOverplus()` function also disregards the `_withdrawFromVault()` function's return value. However, since this function only transfers the available loose assets on line 313, the ignored return value on line 305 does not impose a risk. It is worth noting that the `calculateOverplus()` helper function (which is invoked on line 298) emits a `CalculateOverplusResult` event which includes the estimated overplus value and not the actual transferred amount. Similar to the `ejectStakedFunds()` function's case, the `calculateOverplus()` function calls the vault's `vault.convertToAssets()` API, and thereby the calculated overplus should largely match the actual exited amount. As such this observation is left as a note. NCC Group recommends that this behavior is documented to avoid confusion in the future.

```
297 function bridgeOverplus() external {
298     uint256 overplus_ = calculateOverplus();
299     require(overplus_ > 0, "DS: overplus is zero");
300
301     uint256 looseAssets_ = IERC20(depositToken).balanceOf(address(this));
302
303     // Pull only what is missing (rounded up internally by the vault).
304     if (looseAssets_ < overplus_) {
305         _withdrawFromVault(overplus_ - looseAssets_);
306     }
307
308     uint256 transferable_ = IERC20(depositToken).balanceOf(address(this));
309
310     // By design `transferable_` can only exceed `overplus_` by vault-rounding dust.
311     // Transferring the full loose balance guarantees that *all* yield (incl. dust)
312     // is bridged, never touching user principal which remains invested in sUSDS.
313     IERC20(depositToken).safeTransfer(aoDistributionWallet, transferable_);
314
315     emit OverplusBridged(transferable_);
316 }
```

7 Contact Info

The team from NCC Group has the following primary members:

- Parnian Alimi – Consultant
  parnian.alimi@nccgroup.com
- Javed Samuel – Practice Director, Cryptography Services
  javed.samuel@nccgroup.com

The team from Forward Research has the following primary members:

- Christian Smorra
  christian@dataos.so
- Ivan Morozov
  ivan@dataos.so