Neutrl: InstantUnstaking.sol

Security Review

Cantina Managed review by:
Kurt Barry, Lead Security Researcher

March 7, 2026

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A security review is a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While the review endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that a security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Mar 3rd to Mar 5th the security researchers conducted a review of contracts on commit hash 821f3581. A total of **7** issues were identified:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	1	1	0
Gas Optimizations	2	2	0
Informational	4	3	1
Total	**7**	**6**	**1**

2.1 Scope

The security review had the following components in scope for contracts on commit hash 821f3581:

```
src/InstantUnstaking.sol
```

3 Findings

3.1 Low Risk

3.1.1 Fees Are Rounded In Favor of the User Instead of the Protocol

Severity: Low Risk

Context: InstantUnstaking.sol#L204

Description: Instantly redeeming very small amounts can avoid the fee entirely due to truncating division.

Recommendation: Rounding up (i.e. in favor of the protocol) is the recommended best practice. While gas costs likely make any exploit attempts unprofitable given that NUSD has 18 decimals, it's still a good idea to future-proof the code and shouldn't have a meaningful impact for normal unstake amounts.

Neutrl: Fixed in PR 59.

Kurt Barry: Fix verified.

3.2 Gas Optimization

3.2.1 Redundant Check in Constructor

Severity: Gas Optimization

Context: InstantUnstaking.sol#L75

Description: The `MAX_FEE` check is also done in `_setFee()`, called later in the constructor.

Recommendation: Remove the redundant check.

Neutrl: Fixed in PR 61.

Kurt Barry: Fix verified.

3.2.2 Unnecessary Use of `safeTransferFrom()`

Severity: Gas Optimization

Context: InstantUnstaking.sol#L145

Description: The SNUSD contract is known to be ERC20-compliant, so this use of `safeTransferFrom()` is unnecessary. Replacement with a plain `transferFrom()` call resulted in gas savings when running tests.

Recommendation: Use `transferFrom()` as `safeTransferFrom()` is not needed and increases gas costs.

Neutrl: Fixed in PR 63.

Kurt Barry: Fix verified.

3.3 Informational

3.3.1 `quoteInstantUnstake()` Ignores the Pause Status

Severity: Informational

Context: InstantUnstaking.sol#L180-L187

Description: When the contract is paused, the `instantUnstake()` function reverts but the `quoteInstantUnstake()` function does not change its behavior, returning as if instant unstaking were possible. This could make smart contract and frontend integrations more difficult (many developers will expect quote functions like this one to behave similarly to `maxRedeem/maxWithdraw` in ERC-4626 contracts, which are supposed to account for pauses and other limits).

Recommendation: Return zero or revert from `quoteInstantUnstake()` when the contract is paused.

Neutrl: Fixed in PR 64.

Kurt Barry: Fix verified.

3.3.2 Authorization Notes

Severity: Informational

Context: InstantUnstaking.sol#L39-L40

Description: The `InsantUnstaking` contract defines one privileged role, the `MANAGER_ROLE`, which can set fees and pause or unpause the contract. The constant value used (`keccak256("MANAGER_ROLE")`) is shared with other contracts in the system, e.g. the the `AssetLock` contract, being redefined in each contract. Further, other contracts like `AssetLock` define a separate `PAUSER_ROLE`; it is unclear what requirements dictate when the `MANAGER_ROLE` should also be allowed to pause/unpause and when a separate role is needed.

Recommendation:

1. For roles that are re-used in multiple contracts, consider defining them in a dedicated file (e.g. `Roles.sol`) and importing them where used for clarity, especially if this reflects how the backend system works (i.e. the same hot wallet is granted the `MANAGER_ROLE` across all contracts).
2. Evaluate the security model and consider whether a distinct `PAUSER_ROLE` makes sense for the `InstantUnstaking` module.

Neutrl: Fixed in PR 65.

Kurt Barry: Fix verified. A `PAUSER_ROLE` was added to control pausing and unpausing.

3.3.3 Return NUSD Quantity After Instantly Unstaking

Severity: Informational

Context: InstantUnstaking.sol#L141

Description: The `instantUnstake()` function does not return the NUSD amount minted to the caller. This would be a convenient value for smart contract integrations or frontends to have returned, as it prevents them from having to check a balance before and after the call to be sure how much was received.

Recommendation: Return the amount of NUSD minted to the caller from an `instantUnstake()` execution.

Neutrl: Fixed in PR 62.

Kurt Barry: Fix verified.

3.3.4 `instantUnstake()` Reverts When the Cooldown Duration Is Zero

Severity: Informational

Context: InstantUnstaking.sol#L149

Description: The `instantUnstake()` function always calls `SNUSD.cooldownShares()`; this function reverts if there is a zero cooldown duration. While users can instead call `SNUSD.unstake()` directly in this case, it creates extra work in particular for smart contract integrations, and runs counter to expectations (`instantUnstake()` should always succeed if possible).

Recommendation: Consider checking `SNUSD.cooldownDuration() == 0` and adding a code path that calls `unstake()` directly when this condition is `true` (and also avoids fee enforcement) for a maximally user-friendly implementation and easier integrations.

Neutrl: Acknowledged.

Kurt Barry: Acknowledged.

4 Appendix

4.1 Phantom Cooldown Balance Analysis

Because the `InstantUnstaking` module never calls `SNUSD.unstake()`, a "phantom" cooldown balance accrues over time. The main objective of the review was to determine if this posed any special risks. This finding aggregates all analysis of potential issues. Note that there are no recommended smart contract changes at this time, as the risks are all deemed negligible.

The cooldown balance is accounted in the underlying asset, not shares, and thus does not accrue rewards, eliminating a major point of concern. Two remaining risks were identified: cooldown balance overflow, and offchain confusion.

Risk 1: Cooldown Balance Overflow: An attacker could stake and instantly unstake repeatedly in order to trigger overflow on sNUSD.sol#L280.

Thus bricking the `InstantUnstake` contract, forcing it to be redeloyed, at which point the attacker could repeat the attack to continue griefing the system. The first defense against this is the instant unstaking fee--even a tiny fee will make this prohibitively expensive. But the contract does support a fee of zero so it's necessary to analyze this scenario in case it occurs in practice:

- The attacker needs to overflow a container of size $2^{152} \approx 10^{45}$.
- Assume the attacker has 10 billion (10^{10}) NUSD available to stake and instantly unstake (maybe via flashloans etc... this is probably more than realistic but if we show that the attack doesn't work even for this amount, we don't need to worry about practical amounts either).
- The attacker needs to stake-unstake $10^{45}/(10^{10} * 10^{18}) = 10^{17}$ times to cause overflow.
- Assume the attacker can get 1000 stake-unstake cycles into a single block (this should also be an upper bound, the real amount should be much, much lower but the block gas limit may increase in the future etc).
- Then $10^{17}/1000 = 10^{14}$ blocks are needed to achieve an overflowing state. With a 12 second block time, this is $(10^{14} \times 12)/(365.25 \times 86400) \approx 38$ million years of continuously buying all ETH block space, which is obviously unfeasible.

In summary, this overflow can only be a practical problem if both of the following conditions are true:

1. The instant unstaking fee is zero for the attacker.
2. The attacker can access extremely high quantities of NUSD at zero cost for the attack, e.g. if uncapped, unbacked flash loans are ever added as a feature to NUSD.

Risk 2: Offchain Confusion: The `cooldownShares()` function emits an `UnstakeRequested` event that could be confusing to offchain systems that are tracking the state of the protocol. Such systems need to be updated to conditionally interpret these events based on whether the event was emitted due to an instant unstaking action or a normal cooldown request.