SPEARBIT

Neutrl Contracts Security Review

Auditors

0xRajeev, Lead Security Researcher

Kurt Barry, Lead Security Researcher

Chinmay Farkya, Associate Security Researcher

Report prepared by: Lucas Goiriz

August 4, 2025

Contents

1 About Spearbit **2**

2 Introduction **2**

3 Risk classification **2**
- 3.1 Impact . . . 2
- 3.2 Likelihood . . . 2
- 3.3 Action required for severity levels . . . 2

4 Executive Summary **3**

5 Findings **4**
- 5.1 Medium Risk . . . 4
  - 5.1.1 Users can decrease their asset lock end times . . . 4
- 5.2 Low Risk . . . 4
  - 5.2.1 Granting multiple roles to the same address is risky . . . 4
  - 5.2.2 Removed yield tokens can never be added again affecting protocol yield strategy . . . 5
  - 5.2.3 Using a single EOA deployer for all privileged roles across protocol is risky . . . 5
  - 5.2.4 Users can take advantage of share values changing suddenly when the vesting period is increased . . . 6
  - 5.2.5 If `coolDownDuration` is lowered, users may be able to reduce their cooldown end times . . . 7
  - 5.2.6 Yield accrual in `YieldDistributor` will fail if NUSD mint limit is exceeded . . . 8
- 5.3 Gas Optimization . . . 9
  - 5.3.1 `mintRequestId` and `redemptionRequestId` can be cached . . . 9
  - 5.3.2 `getPendingMintRequests()` and `getPendingRedemptionRequests()` may hit OOG exception 9
  - 5.3.3 Redundant order type checks in `mint()` and `redeem()` . . . 10
  - 5.3.4 Redundant getters . . . 10
  - 5.3.5 Unnecessary storage field . . . 10
  - 5.3.6 Unnecessary transfer of sNUSD shares in `ELPDistribution.proceedToELPDistribution()` 11
- 5.4 Informational . . . 11
  - 5.4.1 Missing event emission for privileged functions reduces offchain transparency . . . 11
  - 5.4.2 Unused code constructs reduce readability . . . 11
  - 5.4.3 Centralized roles and responsibilities across the protocol are susceptible to misuse . . . 11
  - 5.4.4 Event parameters can be indexed for efficient lookups . . . 13
  - 5.4.5 Symbol instead of name passed to `ERC20Permit` constructor in `sNUSD` . . . 13
  - 5.4.6 Missing zero-address checks on critical protocol addresses . . . 13
  - 5.4.7 Miscellaneous issues reduce readability . . . 13
  - 5.4.8 Use `_msgSender()` instead of `msg.sender` in `SingleAdminAccessControl` . . . 14
  - 5.4.9 Unused imports . . . 14
  - 5.4.10 Consider additional validation checks . . . 14
  - 5.4.11 Misleading benefactor value in `Redeem` event when a queued redemption request is served . 15
  - 5.4.12 Yield should be accrued before removing a `yieldToken` from `YieldDistributor` . . . 15
  - 5.4.13 Incorrect comments . . . 15
  - 5.4.14 `AUTHORIZED` role is not set up in `AssetReserve` constructor . . . 16
  - 5.4.15 Pending request listing functions include expired requests in the returned arrays . . . 16
  - 5.4.16 Subcontract-specific constructs do not belong in base contracts . . . 16
  - 5.4.17 Inconsistent use of errors in `BaseMintRedeem` and derived contracts . . . 17

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Neutrl is a market-neutral synthetic dollar designed to unlock untapped yield opportunities in OTC and altcoin markets. Neutrl leverages OTC arbitrage, funding rate inefficiencies, and DeFi-native market-neutral strategies to provide a single, high-yield access point for capital allocators.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Neutrl Contracts according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

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

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 9 days in total, Neutrl engaged with Spearbit to review the neutrl-contracts protocol. In this period of time a total of **30** issues were found.

Summary

Project Name	Neutrl
Repository	neutrl-contracts
Commit	00949211
Type of Project	Stablecoin, Yield
Audit Timeline	Jul 21st to Jul 30th

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	6	5	1
Gas Optimizations	6	3	2
Informational	17	13	4
Total	**30**	**22**	**7**

5 Findings

5.1 Medium Risk

5.1.1 Users can decrease their asset lock end times

Severity: Medium Risk

Context: AssetLock.sol#L161

Description: The `AssetLock.lockAsset()` function allows users to add assets to an existing locked position. They are also able to specify a new duration which is applied to the position without regard for the existing expiration:

```
lockEndTime: uint64(block.timestamp + _lockDuration)
```

It's possible for the new lock end time to be earlier than the original lock end time. This applies as well to the `lockAssetOnBehalf()` function added in PR 20

Recommendation: Use the maximum of the current `lockEndTime` and `block.timestamp + _lockDuration` as the updated `lockEndTime`.

Neutrl: Fixed in PR 35.

Spearbit: Fix verified.

5.2 Low Risk

5.2.1 Granting multiple roles to the same address is risky

Severity: Low Risk

Context: AssetLock.sol#L82-L84, Router.sol#L129-L131, NUSD.sol#L41-L42, sNUSD.sol#L103-L104, YieldDistributor.sol#L53-L54

Summary: Granting multiple roles, which have different levels/types of authorization, to the same address is risky because it defeats the motivation for role based access control to impose separation of privileges.

Finding Description: Different protocol components define different roles to enforce role based access control. However, all the roles are initialized to a single address/account. This defeats the purpose of role based access control:

1. NUSD assigns `_admin` to both `DEFAULT_ADMIN_ROLE` and `DENYLIST_MANAGER_ROLE`.
2. SNUSD assigns `_owner` to both `DEFAULT_ADMIN_ROLE` and `PAUSER_ROLE`.
3. Router assigns `_admin` to `DEFAULT_ADMIN_ROLE`, `WHITELISTER_ROLE`, and `PAUSER_ROLE`.
4. AssetLock assigns `_admin` to `DEFAULT_ADMIN_ROLE`, `MANAGER_ROLE`, and `PAUSER_ROLE`.
5. YieldDistributor assigns `_admin` to both `DEFAULT_ADMIN_ROLE` and `YIELD_TOKEN_MANAGER_ROLE`.

These roles have different levels/types of privileges and should be granted to different actors. `DEFAULT_ADMIN_ROLE`, which has the highest level of privilege should not be given to addresses that also control lower privileged roles because their risk profiles are different which allows their wallet operational security to be managed proportionately.

Impact Explanation: High, because if that one account is compromised then protocol functionality controlled by all its roles is compromised.

Likelihood Explanation: Very low, assuming that single account has the highest level of operational security and is not compromised.

Recommendation: Consider using different addresses/accounts to initialize different roles within the protocol components.

Neutrl: PR 31

Spearbit: Reviewed that PR 31 removes multiple assignments of roles to the same address and instead moves those role assignments to the deployment script where different addresses (dummy zero addresses for now which are noted to be replaced later) are assigned to these different roles.

5.2.2 Removed yield tokens can never be added again affecting protocol yield strategy

Severity: Low Risk

Context: YieldDistributor.sol#L71-L89

Summary: Removed yield tokens from `YieldDistributor` can never be added again, which negatively affects protocol yield strategy and distribution thereafter.

Finding Description: `YieldDistributor` manages and distributes yield from accepted stablecoins (USDC, USDT and USDe) to sNUSD holders. While the yield generation happens offchain, the generated yield in USDC, USDT and USDe is converted to NUSD via `convertYieldToNUSD()` and then distributed to sNUSD holders via `distributeYield()`.

The protocol manages the accepted yield tokens via `addYieldToken()` and `removeYieldToken()`. `addYieldToken()` prevents addition of duplicated yield tokens by checking if the one being added already exists in the `yieldTokens` array. `removeYieldToken()` removes a yield token not by removing it from the array but only marking it as inactive via `yieldTokens[i].isActive = false`.

Given this differing logic during addition and removal, once a yield token is removed by marking it as inactive, it can never be added back again because `addYieldToken()` only checks for its presence and not for `isActive == true`.

Impact Explanation: Low, because not being able to add a yield token back will negatively impact protocol yield strategy and distribution thereafter.

Likelihood Explanation: Low, assuming it is unlikely for the admin to remove a yield token and then want to add it back later for some reason.

Recommendation: Consider checking for `yieldTokens[i].isActive == true` along with `yieldTokens[i].token == _yieldToken` in `addYieldToken()` logic, so that:

1. If a token is present but `isActive == false` then it can be set to `true`.
2. If a token is present and `isActive == true` then it can revert.
3. If a token is absent then it can be added.

Neutrl: Fixed in PR 22.

Spearbit: Fix verified.

5.2.3 Using a single EOA deployer for all privileged roles across protocol is risky

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Summary: `DeployProtocol` script currently uses an EOA deployer address for all privileged roles across the protocol, which is extremely risky.

Description: `DeployProtocol` script derives a deployer address from a private key read from an environment variable and uses that in calls to `deployDeterministic(deployer)` and `deployOthers(deployer)`. `deployDeterministic(deployer)` uses that deployer address to initialize all privileged roles across NUSD and sNUSD contracts. `deployOthers(deployer)` uses the same deployer address to initialize all privileged roles across `Router`, `StableMinter`, `Redeemer`, `AssetReserve`, `AssetLock`, and `YieldDistributor` contracts.

Impact Explanation: High, because if that single EOA is compromised, the entire protocol can be critically impacted given the centralized roles and responsibilities as documented in "Centralized roles and responsibilities across the protocol are susceptible to misuse"

Likelihood Explanation: Very low, assuming that this is a placeholder for now, will be replaced by a safer alternative and transferred immediately after deployment with a combination of multisig/mpc wallets (as communicated).

Recommendation: Consider:

1. Using safer alternatives than reading the deployer key from an environment variable as described in Foundry's key management best practices
2. Transferring the EOA deployer to a combination of multisig/mpc wallets immediately after deployment.
3. Revoking all access from the EOA deployer immediately (and atomically) after step (2).

Neutrl: PR 30

Spearbit: Reviewed that PR 30 adds a `mpcAdminWallet` (dummy zero-address for now noting that this will be replaced by the actual MPC admin wallet once its setup is complete) to which the admin roles of all deployed contracts are immediately transferred from the earlier used deployer address in `run()`.

5.2.4 Users can take advantage of share values changing suddenly when the vesting period is increased

Severity: Low Risk

Context: sNUSD.sol#L125-L130

Description: The `setVestingPeriod()` function checks that the `getUnvestedAmount()` function returns zero and reverts if not. However, an increase of the vesting period can still affect the unvested rewards, because neither `vestingAmount` nor `lastDistributionTimestamp` are updated. This will lead to a discontinuous change in the return of the `totalAssets()` function and thus affect the value per share. Specifically, if the vesting period is lengthened, and the new value of `lastDistributionTimestamp + vestingPeriod` is in the future, a fraction of the `vestingAmount` value will become unvested again.

Users can take advantage of this as follows:

1. Withdraw just prior to a vesting period increase. This will withdraw some amount of the vested rewards.
2. Redeposit after the increase, re-earning a portion of the previously vested rewards at the expense of other users.

Proof of Concept: The following test can be added to `test/unit/concrete/sNUSD/authorized/sNUSD_authorized.t.sol` and run via `forge test --match-test test_ChangingVestingPeriodChangesUnvestedRewards`:

```
function test_ChangingVestingPeriodChangesUnvestedRewards() external {
    address user1 = address(0x1111);
    address user2 = address(0x2222);

    deal(address(nusd), user1, 1 ether);
    deal(address(nusd), user2, 1 ether);

    // Both users deposit.

    vm.startPrank(user1);
    nusd.approve(address(sNusd), type(uint256).max);
    sNusd.deposit(1 ether, user1);
    vm.stopPrank();

    vm.startPrank(user2);
    nusd.approve(address(sNusd), type(uint256).max);
    sNusd.deposit(1 ether, user2);
    vm.stopPrank();

    // First transfer to create unvested rewards
    vm.startPrank(rewarder);
    deal(address(nusd), rewarder, 1 ether);
    nusd.approve(address(sNusd), 1 ether);
    sNusd.transferInRewards(1 ether);
    vm.stopPrank();

    uint256 vestingPeriod = sNusd.vestingPeriod();
    vm.warp(sNusd.lastDistributionTimestamp() + vestingPeriod);

    // All rewards have vested.
    assertEq(sNusd.getUnvestedAmount(), 0);

    // The two users, having made identical deposits at the identical time, have equal assets.
    assertEq(sNusd.convertToAssets(sNusd.balanceOf(user1)),
    ↪  sNusd.convertToAssets(sNusd.balanceOf(user2)));

    // user1 withdraws just before the vesting period update
    // (for simplicity, we are in instant withdrawal mode)

    vm.startPrank(users.admin);
    sNusd.setCooldownDuration(0);
    vm.stopPrank();

    vm.startPrank(user1);
    sNusd.redeem(sNusd.balanceOf(user1), user1, user1);
    vm.stopPrank();

    uint256 totalAssetsBefore = sNusd.totalAssets();

    vm.startPrank(users.admin);
    sNusd.setVestingPeriod(2 * vestingPeriod);
    vm.stopPrank();

    // This should still be zero but instead it has gone back to half of the unvestedRewards value.
    assertEq(sNusd.getUnvestedAmount(), 0.5e18);

    // Total assets has changed discontinuously
    assertEq(totalAssetsBefore - 0.5e18, sNusd.totalAssets());

    // user1 redeposits, including their accrued interest, to take advantage of the devalued share
    ↪  price.
    vm.startPrank(user1);
    sNusd.deposit(nusd.balanceOf(user1), user1);
    vm.stopPrank();

    // Warp to end of the new vesting period.
    vm.warp(sNusd.lastDistributionTimestamp() + 2 * vestingPeriod);

    // user1, despite merely withdrawing and re-depositing, and with no new rewards added,
    // now has significantly more assets than user2, because the previously added rewards have
    // been redistributed in their favor.
    assertEq(sNusd.convertToAssets(sNusd.balanceOf(user1)), 1799999999999999998);
    assertEq(sNusd.convertToAssets(sNusd.balanceOf(user2)), 1200000000000000001);
}
```

Recommendation: Set `vestingAmount` to zero whenever the vesting period is updated (or just when it is lengthened, as this issue does not apply to shortening the vesting period).

Neutrl: Fixed in PR 32.

Spearbit: Fix verified.

5.2.5 If `coolDownDuration` is lowered, users may be able to reduce their cooldown end times

Severity: Low Risk

Context: sNUSD.sol#L270, sNUSD.sol#L285

Description: Both `sNUSD.cooldownAssets()` and `sNUSD.cooldownShares()` update the user's cooldown end time as follows:

```
cooldowns[msg.sender].cooldownEnd = uint104(block.timestamp) + cooldownDuration;
```

This overwrites the previous value. If the `cooldownDuration` is lowered soon enough after a user initiates a cooldown, then the user may be able to lower their `cooldownEnd` value by initiating another cooldown (potentially depositing more assets if need be).

Recommendation: Set the `cooldownEnd` value to the maximum of the current value and `uint104(block.timestamp) + cooldownDuration` in both `cooldownAssets()` and `cooldownShares()`.

Neutrl: Fixed in PR 33.

Spearbit: Fix verified.

5.2.6 Yield accrual in `YieldDistributor` will fail if NUSD mint limit is exceeded

Severity: Low Risk

Context: BaseMintRedeem.sol#L172-L175, YieldDistributor.sol#L97-L118

Description: `convertYieldtoNUSD()` function in `YieldDistributor.sol` is used to accrue all `yieldToken` balances. It first calculates the total amount in terms of NUSD using collateral prices from `ROUTER.quoteDeposit()` and then calls the ROUTER to mint the calculated NUSD amounts for each yieldToken.

The call flow is `convertYieldtoNUSD() => ROUTER.mint() => StableMinter.mint()`.

`StableMinter.sol` inherits from `BaseMintRedeem.sol` which does some checks on this mint order parameters. One such check is `_belowMaxMintPerBlock()`.

```
function _belowMaxMintPerBlock(uint256 mintAmount) internal view {
    if (mintedPerBlock[block.number] + mintAmount > maxMintPerBlock) {
        revert MaxMintPerBlockExceeded();
    }
}
```

This check ensures that the already minted NUSD in the actual block plus the amount to be minted is below the `maxMintPerBlock` value. This mint limit is shared by all users that are trying to mint NUSD from the minter contract of that particular collateral asset.

The `convertYieldtoNUSD()` loops over all yieldTokens, so the whole yield distribution can fail if mint limit of any one collateral asset's minter has been exceeded.

This can cause yield distribution failure, and delay in accruing yield to sNUSD in a timely manner. This can also be purposefully DOS'ed by an attacker.

This will be problematic for integrators that depend on timely yield accrual as any withdrawals from sNUSD will still succeed even though the yield that was expected at the time was purposefully delayed by an attacker ⇒ leading to potential loss of yield even though they contributed with their NUSD properly.

Recommendation: Consider documenting this potential threat for integrators and monitoring mint usage over time to dynamically adjust mint limits such that yield operations do not fail.

Neutrl: We acknowledge this potential scenario where yield distribution could be disrupted by attackers exhausting mint limits for specific collateral assets. If this situation occurs, we have several mitigation strategies available:

1. Whitelist Management.
   - Since we operate with whitelisting controls, we can remove malicious actors from the whitelist to prevent further abuse.
2. Dynamic Threshold Adjustment.

- We can increase the maxMintPerBlock threshold for affected collateral assets to ensure yield distribution operations can proceed.

3. Operational Monitoring.

- We will monitor mint usage patterns to proactively identify and address potential DoS attempts before they impact yield distribution.

These mitigation measures provide us with the operational flexibility to maintain timely yield accrual while protecting against malicious interference.

Spearbit: Acknowledged.

5.3 Gas Optimization

5.3.1 `mintRequestId` and `redemptionRequestId` can be cached

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: `_requestMint()` and `_requestRedemption()` load `mintRequestId` and `redemptionRequestId` from storage three times consecutively. Given these are frequently used user flows, these repeated storage reads unnecessarily leads to more gas usage than required.

```
mintRequests[mintRequestId] = _order;
mintRequestStatus[mintRequestId] = RequestStatus.PENDING;
emit RequestMint(mintRequestId, _order);
```

```
redemptionRequests[redemptionRequestId] = _order;
redemptionRequestStatus[redemptionRequestId] = RequestStatus.PENDING;
emit RequestRedemption(redemptionRequestId, _order);
```

Recommendation: Consider caching these storage variables to avoid repeated storage reads.

Neutrl: Acknowledged.

Spearbit: Acknowledged.

5.3.2 `getPendingMintRequests()` and `getPendingRedemptionRequests()` may hit OOG exception

Severity: Gas Optimization

Context: Router.sol#L361-L406

Description: `getPendingMintRequests()` and `getPendingRedemptionRequests()` loop through every mint and redemption requests made in the protocol lifetime to check which of them are pending to return their details.

While this may work initially, this iterative approach is likely to hit Out-of-Gas (OOG) exception at some point in the future when the number of requests exceed a certain number. Even if this is only used by the protocol for offchain monitoring and accounting, the logic is suboptimal for gas usage.

Recommendation: Consider:

1. Adding a starting index parameter to these functions so that successive calls can optionally ignore the previously determined pending requests.
2. Caching `mintRequestId` and `redemptionRequestId` instead of loading them repeatedly from storage.

Neutrl: Fixed in PR 26 and PR 44.

Spearbit: Reviewed that PR 26 mitigates the issue as recommended in (1) by adding another set of functions that take starting index as a parameter. All functions now also return only pending mint/redemption requests that expire beyond `block.timestamp`. This PR however introduced an issue in `getPendingRedemptionRequests()` functions where `mintRequests[i].expiry` was incorrectly used instead of `redemptionRequests[i].expiry`. This was later resolved in PR 44.

5.3.3 Redundant order type checks in `mint()` and `redeem()`

Severity: Gas Optimization

Context: BaseMintRedeem.sol#L162-L168, Redeemer.sol#L65-L66, StableMinter.sol#L50-L51

Description: `mint()` and `redeem()` check their order types to be `OrderType.MINT` and `OrderType.REDEEM` respectively. However, they call `_verifyOrder(order)` which also checks for `if (order.orderType != OrderType.MINT && order.orderType != OrderType.REDEEM) revert InvalidInput()`.

These checks are redundant and the consolidated check in `_verifyOrder()` can be removed in favor of the specific checks already performed at the two call sites.

Recommendation: Consider:

1. Removing `if (order.orderType != OrderType.MINT && order.orderType != OrderType.REDEEM) revert InvalidInput()` from `_verifyOrder()`.
2. Moving the duplicated check `if (order.expiry < block.timestamp) revert OrderExpired()` from `mint()` and `redeem()` to `_verifyOrder()`.

Neutrl: PR 28.

Spearbit: Reviewed that PR 28 fixes the issue as recommended in (1).

5.3.4 Redundant getters

Severity: Gas Optimization

Context: AssetLock.sol#L65-L69, AssetLock.sol#L201-L206, AssetLock.sol#L219-L225

Description: The `getAssetInfo()` and `getUserLock()` getters in the `AssetLock` contract are redundant with the `public` fields `assetInfo` and `userLocks`. Having both increases bytecode size (and thus deployment costs) and can also increase function dispatch costs on calls to the contract since the function lookup logic will be more extensive.

Recommendation: Either remove the unnecessary getters or make the associated fields `private`.

Neutrl: Fixed in PR 34.

Spearbit: Fix verified.

5.3.5 Unnecessary storage field

Severity: Gas Optimization

Context: AssetLock.sol#L16

Description: The `isSupported` field of the `AssetInfo` struct used in the `AssetLock` contract isn't strictly necessary--the same effect could be achieved by allowing a `maxLockCapacity` of zero denote an unsupported asset. This would eliminate a full storage slot per locked position.

Recommendation: Remove `AssetLock.isSupported` and rely on `maxLockCapacity` alone to reduce gas costs.

Neutrl: Acknowledged. We prefer to keep it for offchain SDK purposes.

Spearbit: Acknowledged.

5.3.6 Unnecessary transfer of sNUSD shares in `ELPDistribution.proceedToELPDistribution()`

Severity: Gas Optimization

Context: ElpDistribution.sol#L152-L153

Description/Recommendation: In `ELPDistribution.proceedToELPDistribution()`, funds are deposited into the `sNUSD` contract and then the resulting shares are transferred to a multisig address for storage:

```
sNUSD.deposit(totalAllocated, address(this));
sNUSD.transfer(multisig, totalAllocated);
```

The second parameter of `deposit()` is the address that receives the shares, so the code could be simplified to:

```
sNUSD.deposit(totalAllocated, multisig);
```

eliminating an external call and thus saving gas.

Neutrl: Fixed in PR 41.

Spearbit: Fix verified.

5.4 Informational

5.4.1 Missing event emission for privileged functions reduces offchain transparency

Severity: Informational

Context: AssetLock.sol#L245-L247, BaseMintRedeem.sol#L111-L121, YieldDistributor.sol#L67-L69, YieldDistributor.sol#L97-L126

Description: While many of the privileged functions across the protocol emit events to enable offchain monitoring and increased transparency, some are missing such event emits. This reduces offchain transparency, which is a key aspect of Neutrl protocol to strengthen user confidence.

Recommendation: Consider emitting missed events in all privileged functions.

Neutrl: Fixed in PR 21.

Spearbit: Fix verified.

5.4.2 Unused code constructs reduce readability

Severity: Informational

Context: AssetLock.sol#L35, AssetLock.sol#L50, BaseMintRedeem.sol#L21

Description: There are some code constructs such as declared errors and events which are unused and reduce readability.

Recommendation: Consider revisiting their intended usage or remove them for clarity.

Neutrl: Fixed in PR 23.

Spearbit: Fix verified.

5.4.3 Centralized roles and responsibilities across the protocol are susceptible to misuse

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The protocol has several centralized roles and responsibilities across components, which are susceptible to misuse/abuse if compromised, such as:

1. NUSD `MINTER_ROLE` can mint arbitrary NUSD amounts to arbitrary addresses.

2. NUSD `REDEEMER_ROLE` can burn arbitrary NUSD amounts from arbitrary addresses.
3. NUSD `DENYLIST_MANAGER_ROLE` can prevent arbitrary NUSD transfers.
4. Router `WHITELISTER_ROLE` can prevent arbitrary mint/redeem requests.
5. Router `KEEPER_ROLE` can serve/cancel arbitary mint/redeem requests.
6. Router admin can configure arbitrary minters/redeemers.
7. Router admin can set arbitrary order waiting periods affecting order expiry.
8. Minter/Redeemer admin can add/remove support for arbitrary assets.
9. Minter/Redeemer admin can set arbitrary min/max price for supported collateral assets.
10. Minter/Redeemer admin can set arbitrary max mints/redeems per block.
11. Admins can set arbitrary asset reserve and router addresses.
12. `PAUSER_ROLE` can pause/unpause various key functions.
13. YieldDistributor admin can add/remove support for arbitrary yield tokens.
14. YieldDistributor admin can recover arbitrary ERC20 tokens.
15. SNUSD `BLACKLIST_MANAGER_ROLE` can prevent arbitrary sNUSD transfers.
16. SNUSD admin can set arbitrary cooldown durations for unstaking/withdrawing.
17. In PR 20, the `MANAGER_ROLE` can indefinitely extend a user's lock end time by locking a tiny amount of tokens just before each lock expiration while specifying the largest possible duration to update the end time far into the future.

Recommendation: Consider:

1. Enforcing the best privilege separation possible across roles/actors.
2. Implementing the highest degree of operational security for wallet accounts controlling them.
3. Implementing timelocks for some of the most critical actions.

Neutrl: We acknowledge the centralized roles identified in the protocol and recognize the potential security risks associated with compromised administrative keys. To mitigate the risk of compromised admin keys, we will implement the following security measures:

Security Mitigation Strategy:

1. Robust Private Key Management.
   - Strict operational security (OpSec) practices for all admin wallets and role-based wallets.
   - Industry-standard key storage and access protocols.
2. Multi-Party Computation (MPC) Wallet Implementation.
   - All administrative roles managed through MPC wallets with strict role-based policies.
   - Multi-signature requirements for critical operations.
3. Advanced Transaction Protection.
   - Integration with third-party security solutions to detect and automatically reject malicious transactions on MPC wallets.
   - Real-time monitoring and anomaly detection for administrative actions.

These comprehensive security measures will effectively mitigate the risks associated with centralized roles while maintaining the operational capabilities necessary for protocol functionality.

Spearbit: Acknowledged.

5.4.4 Event parameters can be indexed for efficient lookups

Severity: Informational

Context: sNUSD.sol#L47-L49

Description: Events `LockedAmountRedistributed`, `Unstaked` and `RequestUnstaked` do not use indexed attribute on their address parameters for more efficient lookups.

Recommendation: Consider adding `indexed` attribute to these event parameters.

Neutrl: Fixed in PR 24.

Spearbit: Fix verified.

5.4.5 Symbol instead of name passed to `ERC20Permit` constructor in `sNUSD`

Severity: Informational

Context: sNUSD.sol#L101

Description: The name, rather than the symbol, is intended to be passed to the `ERC20Permit` constructor (ERC20Permit.sol#L37).

However, the `sNUSD` contract passes the symbol. This is in contrast to the `NUSD` contract, which correctly uses the token name.

Recommendation: Pass the name (`"Staked NUSD"`) rather than the symbol to the `ERC20Permit` constructor in `sNUSD`.

Neutrl: Fixed in PR 25.

Spearbit: Fix verified.

5.4.6 Missing zero-address checks on critical protocol addresses

Severity: Informational

Context: AssetReserve.sol#L113-L116, BaseMintRedeem.sol#L188-L191, BaseMintRedeem.sol#L213-L215, Redeemer.sol#L41-L42

Description: While the protocol implements zero-address checks in several places, this is missing in some of the initializations and setter functions.

Recommendation: Consider adding zero-address checks on all critical protocol addresses to reduce accidental errors.

Neutrl: Fixed in PR 27.

Spearbit: Fix verified.

5.4.7 Miscellaneous issues reduce readability

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: There are miscellaneous minor issues related to naming, NatSpec, function visibility, missing checks etc. across the codebase that reduce readability.

Recommendation: Consider:

1. Adding a non-zero check for `amountToDistribute` in `redistributeLockedAmount()`.
2. Making `5e18` a declared constant or a settable storage value to allow modifying the acceptable slippage in `convertYieldToNUSD()`.
3. Renaming `startTime` as `lockStartTime` in `AssetLock.UserLock`.

4. Renaming `_asset` as `_collateralAsset` in various Router functions.
5. Making `Router.quoteRedemption()` `external` visibility instead of `public`.
6. Making `SingleAdminAccessControl.owner()` `external` visibility instead of `public` as in IERC5313.
7. Changing `@notice` of `_setMintWhitelisted()` to `/// @notice Internal function to set a user's mint whitelist status.`
8. Changing `@notice` of `_setMintWhitelistEnforcement()` to `/// @notice Internal function to set the mint whitelist enforcement.`
9. Adding `@param` for `_nusd` in `Router.constructor`.
10. Adding `@param` for `_multisig` in `ElpDistribution.constructor`.
11. Using named mapping parameters for Router `minters` and `redeemers`.

Neutrl: Fixed in PR 29.

Spearbit: Fix verified.

5.4.8 Use `_msgSender()` instead of `msg.sender` in `SingleAdminAccessControl`

Severity: Informational

Context: SingleAdminAccessControl.sol#L30-L39

Description/Recommendation: Consider using `_msgSender()` instead of `msg.sender` in `SingleAdminAccess-Control` for consistency with the OZ base class implementation (AccessControl.sol#L170).

While currently the two options are equivalent, if there's ever a contract that inherits `SingleAdminAccessCon-trol` and overrides `_msgSender()`, then bugs could result from using `msg.sender` in `SingleAdminAccessControl` instead of `_msgSender()`.

5.4.9 Unused imports

Severity: Informational

Context: Redeemer.sol#L5, Redeemer.sol#L14, StableMinter.sol#L4-L5, StableMinter.sol#L15

Description: The following imports are unused:

- StableMinter.sol#L4-L5: The `IERC20` and `SafeERC20` imports are only used in the `using` statement on line 15; they can be removed along with the `using` statement.
- Redeemer.sol#L5: The `SafeERC20` import is only used in the `using` statement on line 14; it can be removed along with the `using` statement.

Recommendation: Remove unused imports and related lines for clarity and readability.

Neutrl: Acknowledged. We prefer to keep it as a boilerplate on a future iteration of minter /redeemer, it avoids to forget.

Spearbit: Acknowledged.

5.4.10 Consider additional validation checks

Severity: Informational

Context: BaseMintRedeem.sol#L210-L218, Router.sol#L152, Router.sol#L422-L438

Description/Recommendation:

- In `Router.configureAsset()`, consider validating that `_asset` is not NUSD as NUSD should never be its own collateral.

- In `Router._setMinter()` and `Router._setRedeemer()`, consider verifying that the minter or redeemer supports the specified asset.
- In `BaseMintRedeem._addAsset()`, consider validating that `_asset` is not NUSD as NUSD should never be its own collateral.

Neutrl: Fixed in PR 39.

Spearbit: Fix verified.

5.4.11 Misleading benefactor value in `Redeem` event when a queued redemption request is served

Severity: Informational

Context: Router.sol#L235, Router.sol#L511

Description: When a queued redemption request is served, the `order.benefactor` value is changed to be the `Router` since that's where the NUSD tokens were transferred when the request was created. This modification is necessary for the internal `_redeem()` function to work correctly, but it obscures the original benefactor in the emitted `Redeem` event--it will now be the `Router` instead of the original NUSD source. This makes this event parameter less informative for an asynchronously executed redemption.

Recommendation: Consider modifying the code to always emit the `Redeem` event using the original benefactor.

Neutrl: Fixed in PR 37.

Spearbit: Fix verified.

5.4.12 Yield should be accrued before removing a `yieldToken` from `YieldDistributor`

Severity: Informational

Context: YieldDistributor.sol#L82-L89

Description: The `YieldDistributor` contract has methods to accrue and distribute yield according to yield token balances accumulated in the contract. The `YIELD_TOKEN_MANAGER_ROLE` is supposed to periodically call `convertYieldToNUSD()` to convert all token balances into NUSD.

There is also a way to remove a registered yield token from the distribution mechanisms, via the `removeYieldToken()` function which is controlled by `DEFAULT_ADMIN_ROLE`.

The problem here is that if the yield token is removed from the `yieldTokens` array without accruing its existing balance into NUSD yield, then the existing balance will be left out of the distribution mechanism of the sNUSD contract.

The `yieldToken` can also not be added again because of another issue, and can only be recovered. Since `YIELD_TOKEN_MANAGER_ROLE` and `DEFAULT_ADMIN_ROLE` are assumed to be separate privileged addresses, this can break the yield distribution process.

Recommendation: Protocol has to make sure that these two calls : `convertYieldToNUSD()` and `removeYieldToken()` are always made in the specified order by privileged roles.

Neutrl: Acknowledged.

Spearbit: Acknowledged.

5.4.13 Incorrect comments

Severity: Informational

Context: BaseMintRedeem.sol#L117-L118, sNUSD.sol#L177, YieldDistributor.sol#L17

Description: There are several instances of incorrect/ incomplete comments across the codebase :

- sNUSD.sol#L177: "This function is called by the `StakingRewardsDistributor` contract" should be changed to "This function is called by the `YieldDistributor` contract".

- YieldDistributor.sol#L17: "A contract that manages and distributes yield from accepted stablecoins (USDC & USDT) to sNUSD holders" should be changed to "A contract that manages and distributes yield from accepted stablecoins (USDC,USDT & USDe) to sNUSD holders".
- BaseMintRedeem.sol#L117-L118: These two comment lines should use the term `price` instead of `slippage`; while the terms are logically related, "slippage" usually denotes a relative price difference rather than an absolute price (which is what is actually accepted as the argument to this function and stored in the contract).

Recommendation: Consider modifying the mentioned comments.

Neutrl: Fixed in PR 38.

Spearbit: Fix verified.

5.4.14 `AUTHORIZED` role is not set up in `AssetReserve` constructor

Severity: Informational

Context: AssetReserve.sol#L48-L50

Description: Across the codebase, all roles are set up in the constructor, which ensures proper role allotment at deployment. In one instance in `AssetReserve` contract, the `AUTHORIZED` role has not been granted by default. This role is required to move collateral assets from AssetReserve contract to custodian addresses. This can lead to no address having the `AUTHORIZED` role initially, affecting protocol strategy as user funds sit in `AssetReserve` contract.

Recommendation: Consider granting `AUTHORIZED` role at deployment via constructor, for consistency.

Neutrl: Acknowledged. Most of the roles have been removed according to finding "Granting multiple roles to the same address is risky" They are defined at deployment, for the authorized role, we are aware of this we need first some MPC wallets being created.

Spearbit: Acknowledged.

5.4.15 Pending request listing functions include expired requests in the returned arrays

Severity: Informational

Context: Router.sol#L358-L406

Description: In the `Router` contract the functions `getPendingMintRequests()` and `getPendingRedeemRequests()` both include pending but expired requests in the arrays they return. However, expired requests cannot be served, only cancelled. Depending on the use cases for these functions, including expired requests may be undesirable and lead to confusion.

Recommendation: Consider whether including expired requests is desirable. For maximum flexibility, a boolean parameter could be added to these functions specifying whether to include expired requests or not.

Neutrl: Fixed in PR 26.

Spearbit: Fix verified.

5.4.16 Subcontract-specific constructs do not belong in base contracts

Severity: Informational

Context: BaseMintRedeem.sol#L19-L20, BaseMintRedeem.sol#L34-L35, BaseMintRedeem.sol#L53-L71, BaseMintRedeem.sol#L93-L103, BaseMintRedeem.sol#L111-L121, BaseMintRedeem.sol#L148-L158, BaseMintRedeem.sol#L170-L184, BaseMintRedeem.sol#L193-L205

Description: The `BaseMintRedeem` contract defines a number of fields and functions that are specific to whether the derived contract is a minter or redeemer; in either case, the fields for the other type of derived contract are entirely unused in the derived contract. Specifically, the fields `minPrice`, `maxPrice`, `mintedPerBlock`, `redeemedPerBlock`, `maxMintPerBlock`, and `maxRedeemedPerBlock`, and the associated functions that set or check these values, are all minter- or redeemer-specific.

This is undesirable for several reasons:

- Gas costs: Derived contracts possess all of these fields and functions. This increases bytecode size (hence deployment costs), and can increase function resolution costs at runtime.
- Possibility for mistakes: It creates the possibility of accidentally calling the wrong function and having a transaction succeed when the intended goal was not accomplished. Concretely, imagine a governance action that is supposed to set `minPrice` on the minter and `maxPrice` on the redeemer, but accidentally swaps which contract each field is set on. The action will succeed with no obvious indication that a mistake was made, whereas it would fail instantly if minters and redeemers didn't share functions unnecessarily.
- Maintainability: It violates expectations of best practices, making the code less readable and maintainable.

Recommendation: Since the desire appears to be capable of supporting multiple minters and redeemers in the future, refactor the inheritance structure, with an extra layer of base contracts that are specific to minters and redeemers:

```
.             BaseMinter -- all minter contracts
             /
BaseMintRedeem
             \
              BaseRedeemer -- all redeemer contracts
```

Move the minter- and redeemer-specific fields and functions to the appropriate intermediate base contract.

Neutrl: Fixed in PR 42.

Spearbit: Fix verified.

5.4.17 Inconsistent use of errors in `BaseMintRedeem` and derived contracts

Severity: Informational

Context: BaseMintRedeem.sol#L22-L23, BaseMintRedeem.sol#L164-L167, Redeemer.sol#L124

Description: The `BaseMintRedeem` class defines a custom error `ZeroInput()` but uses `InvalidInput()` for cases where the input is zero; only in the derived class `Redeemer` is `ZeroInput()` is used, for the exact same sort of error that `BaseMintRedeem` is using `InvalidInput()` for.

Recommendation: Consider making the error usage more regular and consistent with the rest of the codebase. In particular, use `ZeroInput()` when the input is zero, and `InvalidInput()` for other problems with function arguments.

Neutrl: Fixed in PR 40.

Spearbit: Fix verified.