STBL Peg Mechanism Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Stalin

ChainDefenders (1337web3, PeterSRWeb3)

December 13, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

STBL Protocol is a yield-bearing stablecoin platform that tokenizes Real World Assets (RWAs) through a dual-token architecture. The protocol enables users to deposit yield-bearing assets and receive fungible stablecoin tokens alongside yield-bearing NFTs.

The protocol supports two distinct asset categories:

- Rebasing Tokens (PT1 and LT1) – Designed for tokens that implement automatic compounding via rebasing mechanisms. Yield accrues through periodic increases in the token's price/index (e.g., USDY_PT1, OUSG_-PT1).
- Non-Rebasing Tokens (DPT1 and DLT1) – Designed for tokens that distribute yield externally (e.g., via airdrops or direct transfers) without altering the token price. Example: Benji Token.

The STBL Protocol implements a dynamic peg-stabilization mechanism that automatically adjusts minting and withdrawal fees for assets used to mint USST, with the objective of maintaining a 1:1 USD peg.

- When USST trades below the $1.00 target, the mechanism raises minting (deposit) fees to discourage new supply while lowering burning (withdrawal) fees to encourage redemptions, thereby contracting circulating supply.
- When USST trades above the $1.00 target, it lowers minting fees to stimulate deposits and new supply while increasing burning fees to discourage redemptions, thereby expanding circulating supply until the peg is restored.

The peg mechanism dynamically adjusts minting and burning fee rates to defend a target peg for USST. `STBL_-USSTDepegController` reads price signals (preferentially a Curve TWAP), computes a normalized deviation from the peg, maps that deviation into tolerance/scaling/crisis zones, and converts the result into asset-level fee rates which are published to the protocol fee registry. The primary use case is to create configurable economic incentives that discourage destabilizing supply changes when the price deviates from the peg. End users (depositors/withdrawers) are affected through variable mint/burn fees.

Architecture Summary

The protocol separates policy, price discovery, and fee application into distinct components:

- `STBL_USSTOracle` reads from an external Curve pool and returns a normalized price to the controller
- `STBL_USSTDepegController` uses the `STBL_DepegRateCalculator` logic to convert the oracle price and stored per-asset parameters into minting and burning fee rates. The controller iterates over configured assets and calls `iSTBL_Register.setFees(assetID, depositFee, withdrawFee, yieldFee, insuranceFee)` to publish the computed mint/burn rates; downstream vaults and issuers then read those fees when processing user deposits and withdrawals. Token flows are not performed by the controller — economic effect is achieved by changing fee rates (i.e., costs applied on mint/withdraw), while actual fee collection and distribution occur inside the Register and asset contracts
- Access control is role based: `DEFAULT_ADMIN_ROLE` manages parameters and oracle settings, `OPERATOR_ROLE` is authorized to run periodic updates, and the oracle/register are immutable or permissioned integration points that must be correctly provisioned

Overall, the architecture enforces a clear separation of concerns: oracle → controller (policy math) → register (fee application) → asset flows (vaults/issuers).

5 Audit Scope

The audit scope is distributed across two separate repositories.

1. The first repository is the STBL Protocol at commit f00c091d7d85e62a6fb64886808b5ae447d7afa3.

```
contract/contracts/Assets/DLT1/lib/STBL_OracleLib.sol
contract/contracts/Assets/DLT1/STBL_DLT1_Issuer.sol
contract/contracts/Assets/DLT1/STBL_DLT1_Oracle.sol
contract/contracts/Assets/DLT1/STBL_DLT1_Vault.sol
contract/contracts/Assets/DLT1/STBL_DLT1_YieldDistributor.sol
contract/contracts/Assets/DPT1/lib/STBL_OracleLib.sol
contract/contracts/Assets/DPT1/STBL_DPT1_Issuer.sol
contract/contracts/Assets/DPT1/STBL_DPT1_Oracle.sol
contract/contracts/Assets/DPT1/STBL_DPT1_Vault.sol
contract/contracts/Assets/DPT1/STBL_DPT1_YieldDistributor.sol
```

On the STBL Protocol repository, we got a request to include in the scope the following files as per commit 87ddc80f8aa2cf9d2080a34d3489ab392b0316c6

- The changes in comparisson to the last audit are the same as the changes for the files of the above scope, the only difference between the T1 and T2 contracts is that T1 supports tokens that distributed yield automatically whilst T2 contracts supports tokens where yield has to be claimed manually.

```
contract/contracts/Assets/DLT2/lib/STBL_OracleLib.sol
contract/contracts/Assets/DLT2/STBL_DLT2_Issuer.sol
contract/contracts/Assets/DLT2/STBL_DLT2_Oracle.sol
contract/contracts/Assets/DLT2/STBL_DLT2_Vault.sol
contract/contracts/Assets/DLT2/STBL_DLT2_YieldDistributor.sol
contract/contracts/Assets/DPT2/lib/STBL_OracleLib.sol
contract/contracts/Assets/DPT2/STBL_DPT2_Issuer.sol
contract/contracts/Assets/DPT2/STBL_DPT2_Oracle.sol
contract/contracts/Assets/DPT2/STBL_DPT2_Vault.sol
contract/contracts/Assets/DPT2/STBL_DPT2_YieldDistributor.sol
```

2. The second repository is the Peg Mechanism at commit e1fe2e6766b3be4fe584ba1ed49164dded219991.

```
peg-contracts/contracts/lib/STBL_Structs.sol
peg-contracts/contracts/peg/DepegRateCalculator.sol
peg-contracts/contracts/peg/USSTDepegController.sol
peg-contracts/contracts/peg/USSTOracle.sol
peg-contracts/contracts/splitter/STBL_YLDSplitter.sol
```

6 Executive Summary

Over the course of 10 days, the Cyfrin team conducted an audit on the STBL Peg Mechanism smart contracts provided by STBL. In this period, a total of 27 issues were found.

During the audit we identified 2 Critical, 5 High and 5 Medium severity issues with the remainder being low and informational.

The first critical-severity vulnerability stems from an inverted usage of the calculates minting and burning rates, which they were set on the register as if they would've been discounts, when in reality, they are set as fees. Turning minting fees into subsidies (making deposits artificially cheap or free) and withdrawal fees into rewards. This fully inverts economic incentives, enabling arbitrage, unintended token dilution, and massive fee-revenue loss.

The second critical-severity vulnerability occurs when splitting a `YLD_NFT`. The original `depositTimestamp` is not copied to the newly minted NFTs, causing them to inherit the default zero value and be immediately flagged as expired, permanently locking the associated assets and preventing any withdrawal.

A high-severity vulnerability arises from incorrectly swapping minting and burning rates when a boolean flag is enabled: the swap routine overwrites one rate with the other without preserving the original value, causing both rates to collapse to the same value and effectively nullifying the intended distinction between minting and burning fees.

Two high-severity DoS vulnerabilities affect fee updates via the `PegController`. In both cases the system falls in DoS.

- Raw 18-decimal calculated rates are passed instead of downscaling to the 9-decimal format expected by the `Register`, causing all updates to revert and permanently disabling `PegController` fee adjustments.
- Fee-update loop aborts entirely if any single registered asset reverts, preventing fee updates for all remaining assets and fully blocking `PegController` from updating fees to maintain the Peg.

Another high-severity vulnerability rises when splitting a `YLD_NFT` by lot; The final lot and any remainder from `stableValueNet` are not correctly minted, resulting in loss of corresponding principal and yield for the last lot of the original `YLD_NFT`.

The 5th high-severity is derived from the the updates to `STBL_DLT1, DPT1, DLT2, and DPT2` contracts to support airdrop-style yield tokens (where yield is delivered by directly increasing the holder's balance, either automatically or via manual claim), any tokens transferred to the respective `YieldDistributor` contracts continue to accrue yield while awaiting user claims. Because these `YieldDistributor` contracts lack any mechanism to track, harvest, or redistribute this secondary yield — and provide no administrative or user-facing function to withdraw excess tokens — all yield generated on unclaimed balances becomes permanently stranded and irreversibly lost to users.

The medium-severity issues include the following oversights:

- Not initializing the owner on the `STBL_YLDSplitter`
- Unintentionally overwritting the `yieldFee` and `insuranceFee` on the Register when the fees are updated by the `PegController`
- Mechanism to pull price from a Cuve Oracle lacking robustness to prevent price manipulation
- Incorrect calculations when splitting a `YLD_NFT` by lot causing execution to revert because an underflow

Post Audit Recommendations

It is strongly recommended to expand the test suite with comprehensive end-to-end tests that cover all critical execution paths across the entire contract system. Due to the significant number of Critical & High severity findings it is statistically likely that more serious vulnerabilities still remain which could not be discovered during the 10-day audit window. Hence it is recommended that prior to deploying significant capital on-chain in a production environment, another audit be conducted during which no Critical or High severity findings should be found. In addition to the previous comment, it is also recommended to revisit the inline documentation and update it accordingly to match the latest state of the codebase.

Summary

Project Name	STBL Peg Mechanism
Repository	contract
Commit	f00c091d7d85...
Repository 2	peg-contracts
Commit	23f77b48d93b...
Audit Timeline	Nov 24th - Dec 5th, 2025
Methods	Manual Review

Issues Found

Critical Risk	2
High Risk	5
Medium Risk	5
Low Risk	7
Informational	8
Gas Optimizations	0
Total Issues	27

Summary of Findings

[C-1] Splitting a `YLD_NFT` causes all the newly minted position to be instantly marked as expired because `depositTimestamp` is not copied over from the original `YLD_NFT` to all new minted `YLD_NFTs`	Resolved
[C-2] Peg mechanism doesn't correctly update `depositFees` and `withdrawalFees` to achieve the goal of maintaining the peg, all the opposite, it systematically amplifies deviation from target peg	Resolved
[H-1] DOS risk in `updateRates` due to `register.setFees` failure	Resolved
[H-2] Total `stableValueNet` is not conserved (always short by one full lot size)	Resolved
[H-3] Updating fees via the `PegController` will always revert because the calculated rate always exceeds the upper limit for a valid fee on the `STBL_Register`	Resolved
[H-4] Assets configured with inversion will have incorrect rate calculations applied	Resolved
[H-5] Unaccounted yield accumulation in `STBL_DLT1_YieldDistributor` leads to loss of tokens	Resolved
[M-1] Pulling price from Curve's Oracle can be hardened to prevent price manipulation	Resolved
[M-2] `updateRates` sets yieldFee and insurenceFee to 0	Resolved
[M-3] `owner` in `STBL_YLDSplitter` is never initialized	Resolved

[M-4] Incorrect remainder calculation causing underflow in `splitNftByLot`	Resolved
[M-5] Incorrect value for `DEFAULT_ADMIN_ROLE` on `DPT2` and `DLT2` causes calls restricted to default admin only to revert	Resolved
[L-1] Recipient parameter ignored in `splitNftByPart` and `splitNftByRatio`	Resolved
[L-2] Missing operator approval check despite error message	Resolved
[L-3] Invalid NFT with zeroed fields minted when remainder is equal to 0	Resolved
[L-4] Arithmetic underflow in `withdrawERC20` when there is a lower price of asset token prevents governance from using `insuranceFunds` to make investor's whole	Acknowledged
[L-5] Rounding errors in `splitNftByPart` and `splitNftByRatio`	Acknowledged
[L-6] `VaultData` values not reset on emergency withdraw	Acknowledged
[L-7] Inconsistent minimum rate handling in `calculateUnifiedRate`	Resolved
[I-1] `STBL_YLDSplitter` not emitting events on important state changes and actions	Resolved
[I-2] Missing `recipient` zero address check	Resolved
[I-3] `FeesWithdrawn` event emitted even when `feesToWithdraw` equals zero	Resolved
[I-4] Outdated fields on `YLD_Metadata` no longer required given dynamic fees based on new `Peg` mechanism	Acknowledged
[I-5] Typo in error name `SpiltRejected` in `STBL_YLDSplitter`	Resolved
[I-6] Wrong docstring and comment regarding percentage conversion	Resolved
[I-7] Unused constant declaration	Resolved
[I-8] Lack of events for important actions in `STBL_YLDSplitter`	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Splitting a `YLD_NFT` causes all the newly minted position to be instantly marked as expired because `depositTimestamp` is not copied over from the original `YLD_NFT` to all new minted `YLD_NFTs`

Description: The `depositTimestamp` property of all `YLD_NFTs` helps to determine if a position has expired or not. When a deposit occurs, a new `YLD_NFT` is minted, and the `depositTimestamp` is set to `block.timestamp`, indicating that the position will remain valid for a specific period after that point before it expires.

```
function generateMetaData(
    uint256 assetValue
) internal view returns (YLD_Metadata memory MetaData) {
    ...
    MetaData.depositTimestamp = block.timestamp;
    ...

    return MetaData;
}
```

When withdrawing, the execution checks if the position has expired. If it has, the withdrawal reverts. The only way to retrieve those assets is via the `Issuer::withdrawExpired` function, where all the withdrawn resources are routed to the treasury instead of the position's owner.

```
function iWithdraw(uint256 _tokenID, address _sender) internal isSetupDone {
    AssetDefinition memory AssetData = registry.fetchAssetData(assetID);

    YLD_Metadata memory MetaData = iSTBL_YLD(registry.fetchYLDToken())
        .getNFTData(_tokenID);
    ...

    //should revert when withdraw duration has expired
    if (
        (MetaData.depositTimestamp + MetaData.Fees.duration) <
        block.timestamp
    ) revert STBL_Asset_WithdrawDurationNotReached(assetID, _tokenID);

    ...
}
```

The problem when splitting a `YLD_NFT` is that the `depositTimestamp` of the original `YLD_NFT` is not copied into all the new `YLD_NFTs` that get minted, which means that all of the new positions have their `depositTimestamp` with the default value (0), which means that all those positions are automatically marked as expired.

Impact: All newly minted positions will be instantly marked as expired because `depositTimestamp` from the original NFT is not copied to the new positions, causing the associated assets to the newly minted NFTs not to be able to be withdrawn by the NFT's owner.

Recommended Mitigation: Copy over the `depositTimestamp` from the original NFT onto the metadata of all the new positions

STBL: Fixed in commit 1d46e1c

Cyfrin: Verified. `depositTimestamp` value is copied to the metada of all new positions.

7.1.2 Peg mechanism doesn't correctly update `depositFees` and `withdrawalFees` to achieve the goal of maintaining the peg, all the opposite, it systematically amplifies deviation from target peg

Description: `STBL_DepegRateCalculator::updateRates` sets the fees for the assets of the `STBL_Register` based on the deviation between the current price of USST against each asset listed on the Register that is allowed for minting USST.

The purpose of the `STBL_DepegRateCalculator` contract is to dynamically adjust the fees charged for deposits and withdrawals, ensuring the price maintains its peg. To achieve this, the fees should be set in the following way so that the system correctly incentivizes arbitrageurs and users to do, or not do, certain actions when the peg is either above or below:

- When peg is above, the system needs more supply of USST, meaning:
  - Minting should be cheap; therefore, `depositFees` should be 0 (incentivize minting)
  - Burning should be expensive; `withdrawalFees` should be increased (disincentivize burning)
- When peg is below, the system needs less supply of USST, meaning:
  - Minting should be expensive; therefore, `depositFees` should be increased (disincentivize minting)
  - Burning should be cheap; `withdrawalFees` should be 0 (incentivize burning)

The problem is that `STBL_DepegRateCalculator::updateRates` sets the fees in a way that causes the system to incentivize the opposite behavior than expected, meaning that instead of helping to maintain peg stability, the `STBL_DepegRateCalculator` contract will systematically amplify the deviation, exacerbating the depeg even further.

The problem arises because the `mintingRate` and `burnRate` are calculated inversely, as if they were used as discounts, when in reality, they are used as fees.

1. Minting and Burning rates, `depositFees` and `withdrawalFees`, respectively, are calculated as follows:

```
// STBL_DepegRateCalculator.sol
    function updateRates() external onlyRole(OPERATOR_ROLE) {
        ...

        // Loop through all assets to update their rates
        for (uint256 assetID = 0; assetID < maxAssets; assetID++) {
           ...

//@> A positive delta means peg is above
//@> A negative delta means peg is below
            // Calculate delta
            int256 delta = calculateDelta(
                currentPrice,
                rateParams[assetID].targetPrice
            );

            ...

//@> When delta is negative, peg is below, minting rate is set as 0
//@> When delta is positive, peg is above, burning rate is set as 0
            // Calculate rates for this asset
            uint256 assetMintRate = calculateMintingRate(assetParams, delta);
            uint256 assetBurnRate = calculateBurningRate(assetParams, delta);

//@> fees on the register are updated, `assetMintRate` updates `depositFees`, and, `assetBurnRate`
↪  updates `withdrawFees`
            ...
            // Set Fees in register contract
            register.setFees(assetID, assetMintRate, assetBurnRate, 0, 0);

            ...
        }
```

```
        ...
    }

    function calculateDelta(
        uint256 currentPrice,
        uint256 targetPrice
    ) internal pure returns (int256 delta) {

        //  = (p - p0) / p0
//@> A positive delta means peg is above
        if (currentPrice >= targetPrice) {
            delta = int256(
                ((currentPrice - targetPrice) * PRECISION) / targetPrice
            );
        } else {
//@> A negative delta means peg is below
            delta = -int256(
                ((targetPrice - currentPrice) * PRECISION) / targetPrice
            );
        }
    }

    function calculateMintingRate(
        RateParams memory params,
        int256 delta
    ) internal pure returns (uint256 rate) {
        // Only mint when price is above peg (positive delta)

//@> When delta is negative, peg is below, minting rate is set as 0
        if (delta <= 0) {
            return 0;
        }
        ...
    }

    function calculateBurningRate(
        RateParams memory params,
        int256 delta
    ) internal pure returns (uint256 rate) {
        // Only burn when price is below peg (negative delta)

//@> When delta is postive, peg is above, burning rate is set as 0
        if (delta >= 0) {
            return 0;
        }
        ...
    }
```

```
// STBL_Register.sol

    function setFees(
        uint256 _id,
        uint256 _depositFee,
        uint256 _withdrawFee,
        uint256 _yieldFee,
        uint256 _insuranceFee
    ) external onlyRole(REGISTER_ROLE) {
        ...

        /** Sets value */
//@> `assetMintRate` updates `depositFees`, and, `assetBurnRate` updates `withdrawFees`
        assetData[_id].depositFees = _depositFee;
```

```
        assetData[_id].withdrawFees = _withdrawFee;
        ...
    }
```

2. Once fees are updated, minting and burning (deposits and withdrawals) will charge those fees. **Key concept to understand is that these are fees, not discounts**, meaning, the value of the fees is charged to the user, either in the form of:

- When minting: The user receives fewer USST because the fees are discounted from what can be minted to the user
- When withdrawing: The user receives fewer assets because the fees are discounted from the final value of USST that the user could redeem for assets.

Impact: Incorrect updates to fees based on peg deviation exacerbate the peg deviation from the target, as the incentives are opposite to maintaining the peg.

- **When peg is below, the system should disincentivize minting and incentivize burning.**
  - Instead, it incentivizes minting and disincentivizes burning.
- **When peg is above, the system should incentivize minting and disincentivize burning.**
  - Instead, it disincentivizes minting and incentivizes burning.

Recommended Mitigation: Consider inverting the first conditional on `calculateMintingRate` and `calculateBurningRate`. **Note:** When calculating the minting rate, consider calculating the absolute delta of the `delta` as the initialization value of the `absDelta` variable.

```
function calculateMintingRate(
    RateParams memory params,
    int256 delta
) internal pure returns (uint256 rate) {

-   // Only mint when price is above peg (positive delta)
-   if (delta <= 0) {
-       return 0;
-   }

-   uint256 absDelta = uint256(delta);

+   if (delta >= 0) {
+       return 0;
+   }

+   uint256 absDelta = abs(delta);

    ...
}

function calculateBurningRate(
    RateParams memory params,
    int256 delta
) internal pure returns (uint256 rate) {

-   if (delta >= 0) {
+   if (delta <= 0) {
        return 0;
    }
}
```

- When peg is above (delta is positive):
  - Minting should be cheap; therefore, `depositFees` should be 0 (incentivize minting)

  - Burning should be expensive; `withdrawalFees` should be increased (disincentivize burning)
- When peg is below (delta is negative):
  - Minting should be expensive; therefore, `depositFees` should be increased (disincentivize minting)
  - Burning should be cheap; `withdrawalFees` should be 0 (incentivize burning)

STBL: Fix in commits 7e30bec && 758c611.

Cyfrin: Verified. Assets will have the `inversion` flag configured as true by default. Calculated rates will be inverted at the moment of being set as fees on the Register.

7.2 High Risk

7.2.1 DOS risk in `updateRates` due to `register.setFees` failure

Description: The `updateRates` function iterates over all assets and updates their minting and burning rates based on the current price. For each asset, it calls `register.setFees(assetId, assetMintRate, assetBurnRate, 0, 0)` to enforce the new rates.

According to the register contract logic, `setFees` can **revert** if:

1. The asset is **disabled**, or
2. `assetMintRate + assetBurnRate + assetData[_id].cut > 10000`.

Because `updateRates` does **not handle exceptions** from `setFees`, a single failing asset will **revert the entire loop**, preventing all other assets from being updated. This creates a **denial-of-service (DoS) risk** for rate updates.

Impact: If one asset triggers a revert, **none of the assets will have their rates updated**, potentially leaving USST peg interventions stale. This could allow large deviations from the peg to persist, undermining system stability.

Here's a cleaned-up and more professional version of your recommendation, with corrected spelling, improved flow, and clearer guidance:

Recommended Mitigation

Wrap the `setFees` call in a `try/catch` block and verify that the asset is enabled (using `register::fetchAssetData`) before attempting to update its fees. If the asset is not enabled, skip the call entirely.

```
Register.AssetDefinition memory data = register.fetchAssetData(assetId);
if (data.enabled) {
    try register.setFees(assetId, assetMintRate, assetBurnRate, 0, 0) {
        // Successfully updated fees
    } catch {
        // Log the failure but continue processing remaining assets
        emit RatesUpdateFailed(assetId, assetMintRate, assetBurnRate);
    }
}
```

STBL: Fixed in commit 549fa3

Cyfrin: Verified. `setFees()` is now called in a try-catch block to prevent execution from reverting when the fees can not be updated on a single asset; If the update fails, an event is emitted to log the failure.

7.2.2 Total `stableValueNet` is not conserved (always short by one full lot size)

Description: The `STBL_YLDSplitter::splitNftByLot` function aims to divide the original NFT's `stableValueNet` (referred to as `oldStableNet`) into multiple smaller NFTs, each with a `stableValueNet` equal to the specified lot size (`parts_ratio_lot`, abbreviated as `lot`), while handling any remainder if `oldStableNet` is not perfectly divisible by `lot`. It calculates `numLots` as `oldStableNet / lot` using integer division, which means `oldStableNet` equals `numLots * lot + remainder`, where `remainder` is between 0 and `lot - 1`. The code then burns the original NFT and mints new ones: a loop creates `numLots - 1` NFTs each with `stableValueNet = lot`, totaling `(numLots - 1) * lot`. If `remainder > 0`, it mints an additional NFT with `stableValueNet = remainder`. This results in a grand total of `(numLots - 1) * lot + remainder`, which simplifies to `oldStableNet - lot` because it omits one full lot's worth of value. This systematic under-minting occurs regardless of whether there's a remainder, leading to a consistent loss of exactly one lot's value in the redistributed NFTs.

Impact: The original NFT's value is not fully preserved after burning, as the minted NFTs collectively hold less `stableValueNet` than the original, violating the function's documented purpose of splitting into equal lots with proper remainder handling. This will result in permanent value loss for the user and accounting discrepancies in the contract's ecosystem.

Recommended Mitigation: Adjust the minting loop to run for `i < numLots` instead of `i < numLots - 1`, ensuring all `numLots` full lots are minted with `stableValueNet = lot` each, totaling `numLots * lot`. Then, conditionally mint

a remainder NFT only if `remainder > 0`, setting its `stableValueNet = remainder` to capture any leftover value. This approach guarantees the total minted `stableValueNet` equals `numLots * lot + remainder`, fully matching `oldStableNet` without loss.

STBL: Fixed in commit 6701d17.

Cyfrin: Verified. `STBL_YLDSplitter::splitNftByLot` function has been removed.

7.2.3 Updating fees via the `PegController` will always revert because the calculated rate always exceeds the upper limit for a valid fee on the `STBL_Register`

Description: `STBL_USSTDepegController::updateRates` calculates the minting and burning rates according to the `currentPrice` and `targetPrice` of each asset registered on the Register. Then, using the calculated rates, the `PegController` updates the fees on the register; The goal is to dynamically adjust the fees to maintain the peg of the `USST` stablecoin.

The problem is that the calculated rates have 18 decimals of precision, whereas the fees on the register have a hardcoded upper limit of at most 1e9.

All attempts made by the `PegController` to update the fees on the register will be reverted because the rates will exceed the maximum allowed value of `FEES_CONSTANT` (1e9), thereby breaking the purpose of the peg mechanism entirely.

Impact: `PegController` will be incapable of updating the minting and burning (deposit/withdrawal) fees on the `STBL_Register` to dynamically maintain the peg of their USST stablecoin.

Recommended Mitigation: Normalize the calculated rates down to `PERCENTAGE_PRECISION` decimals (1e9) before passing the calculated rates to the `STBL_Register`.

Consider using the `STBL_USSTDepegController::rateToPercentage` function to get the normalized value of the calculated rate.

STBL: Fixed in commit 0fab575

Cyfrin: Verified. Minting and Burning rates are converted to Percentages with 9 decimals using the `STBL_USSTDepegController::rateToPercentage` before updating the fees on the Register.

7.2.4 Assets configured with inversion will have incorrect rate calculations applied

Description: In the `STBL_USSTDepegController::updateRates` function, when `rateParams[assetID].inversion` is true, the code attempts to swap the `assetMintRate` and `assetBurnRate` variables. However, the swap is implemented incorrectly:

```
assetMintRate = assetBurnRate;
assetBurnRate = assetMintRate;
```

After the first assignment, both variables hold the same value (the original `assetBurnRate`). The second assignment then copies this same value back, resulting in both rates being equal to the original `assetBurnRate`. The original `assetMintRate` value is permanently lost.

Impact: When rate inversion is enabled for an asset, the minting rate is set to the burning rate value, and the burning rate incorrectly remains unchanged. This defeats the purpose of the inversion flag, which should swap the application of minting and burning rates. Assets configured with inversion will have incorrect rate calculations applied.

Recommended Mitigation: Use a temporary variable to properly swap the two values:

```
if (rateParams[assetID].inversion) {
    uint256 temp = assetMintRate;
    assetMintRate = assetBurnRate;
    assetBurnRate = temp;
}
```

STBL: Fix in commits 7e30bec && 758c611.

Cyfrin: Verified. Depending on the inversion flag, rates are now calculated by calling the corresponding function; swapping the rates is no longer needed.

7.2.5 Unaccounted yield accumulation in `STBL_DLT1_YieldDistributor` leads to loss of tokens

Description: The current yield distribution design for `STBL_DLT1` introduces a subtle but critical flaw that can result in permanent loss of yield for users.

Current Flow

- Users deposit Benji tokens into `STBL_DLT1_Vault` to mint USST.
- The vault holds these Benji tokens, which continue to accrue yield over time.
- Periodically, the Franklin Templeton team distributed yield by airdropping Benji tokens onto the user's accounts, meaning the Vault will receive more Benji tokens on its balance.
- `STBL_DLT1_Vault::distributeYield()`, transfers the accrued Benji yield to the `STBL_DLT1_YieldDistributor`.
- The YieldDistributor records the amount of yield available at the moment of transfer and allows users to claim their proportional share based on their deposits.

Once the earned yield on the Vault is distributed to the `STBL_DLT1_YieldDistributor` by using the `STBL_DLT1_Vault::distributeYield` function, the tokens corresponding to the earned yield are transfered onto the YieldDistributor, from where users will be able to claim a proportional amount based on their respective deposits.

The problem is that once yield is transferred to the `YieldDistributor`, the `Benji` tokens sit on its balance and continue to generate additional yield (since Benji is a yield-bearing asset). However, the `YieldDistributor` does not track or account for yield earned on tokens that remain unclaimed. This creates a growing discrepancy between the accounting (fixed at the time of distribution) and the actual token balance of the YieldDistributor.

Concrete Example (Two Equal Stakers) Assume two users have deposited identical amounts, each entitled to 50% of any distributed yield.

1. 100 Benji tokens of accrued yield are sent from the Vault to the YieldDistributor. → Accounting: each staker is entitled to 50 Benji.
2. Staker A claims immediately and receives their 50 Benji. → 50 Benji remain unclaimed in the YieldDistributor.
3. Time passes. The remaining 50 Benji in the YieldDistributor continue earning yield. → Suppose 10% yield is generated → the contract receives an extra 5 Benji tokens.
4. Staker B later claims their share. → According to the original accounting, they received exactly 50 Benji.
5. Result:

- The 5 newly accrued Benji tokens are now stranded in the YieldDistributor.
- No user is credited for them.
- The contract considers all yield fully distributed → the excess tokens become permanently stuck and effectively lost to users.

Without addressing this issue, a non-trivial portion of the protocol's generated yield will systematically leak and become unclaimable by users.

Impact: Yield generated on unclaimed tokens in the YieldDistributor is irreversibly lost.

Recommended Mitigation:

1. The less invasive mitigation to prevent assets from getting stuck on the YieldDistributor is to implement a "sweep" function that forwards any excess tokens above recorded accounting to a new distribution round or treasury, with transparent tracking.

2. A more elaborate mitigation would involve a major refactoring of the accounting to allow the YieldDistributor to be capable of tracking any extra yield earned on its balance.

STBL: Fixed as of commit e5a63a3.

Cyfrin: Verified. DLT1, DLT2, DPT1 and DPT2 contracts have introduced a new function to account for any yield received by the tokens on their balance, such yield is added as rewards to be distributed to users. Each time a user claims, this new function is called to re-calculate indexes and distribute rewards evenly to all users with pending rewards.

7.3 Medium Risk

7.3.1 Pulling price from Curve's Oracle can be hardened to prevent price manipulation

Description: The `getCurrentPrice` function fetches the USST/USDC price from a Curve pool using `get_dy`. According to Curve documentation, `get_dy(i, j, dx)` returns the output amount **after the swap fee** has been applied. The current implementation treats this post-fee value as the spot price. You can find more here.

`getCurrentPrice` is used as a **final fallback** in `getOraclePrice`, which attempts to fetch prices from `price_oracle` first, then `last_prices`, and finally falls back to `getCurrentPrice` if the other sources fail. Therefore, when fallback occurs, the returned price reflects a **net execution price** (post-fee), potentially diverging from the expected oracle price.

Impact: * When `getCurrentPrice` is used as a fallback, the returned price is slightly **lower than the theoretical spot price**, because fees are already deducted.

- Downstream consumers of `getOraclePrice` may assume the fallback price is consistent with the official oracle price, introducing bias.

Proof of Concept:

```
function getOraclePrice() public view returns (uint256 price) {
    try ICurvePool(curvePool).price_oracle() returns (uint256 oraclePrice) {
        return oraclePrice; // official oracle price
    } catch {
        try ICurvePool(curvePool).last_prices() returns (uint256 lastPrice_) {
            return lastPrice_; // last recorded price
        } catch {
            // Fallback: post-fee price from get_dy
            return getCurrentPrice();
        }
    }
}

// getCurrentPrice uses get_dy:
// uint256 usdcReceived = ICurvePool(curvePool).get_dy(usstIndex, usdcIndex, usstAmount);
// price = (usdcReceived * PRECISION) / (10 ** usdcDecimals);
```

Here, the final fallback returns a **post-fee price**, which may not match the pre-fee spot prices returned by the oracle or `last_prices`.

Pulling the current price `CurvePool::price_oracle` doesn't check for the time that has passed since the last update, nor does it check for a difference between the current price and the last reported price; this can lead to using stale prices and/or allow attackers to manipulate the price on the pool.

Recommended Mitigation: Replace or supplement the `getCurrentPrice` fallback with a Chainlink price feed, which provides decentralized, tamper-resistant pricing and avoids post-fee discrepancies. Normalize decimals to 18 for consistency and ensure the feed is fresh before using it.

Consider pulling the price from `CurvePool::last_prices` and verifying the time since the last update. Also, consider implementing a safe diff threshold not to allow operations if the current price and last price have deviated beyond that threshold. See an example of the proposed implementation here.

STBL: Fixed in commit 595646b && c34a46c.

Cyfrin: Verified. Prices pulled from the Curve's Oracle are now validated to ensure they do not use stale data or manipulated prices exceeding a specified threshold. The returned price is the EMA Price, which is less susceptible to manipulation compared to the spot price.

7.3.2 `updateRates` sets yieldFee and insurenceFee to 0

Description: The `setFees` function, when called through `updateRates`, overwrites multiple fee fields inside the asset's register entry. Although the intention appears to be updating only **assetMintRate** and **assetBurnRate**, the function also sets `insuranceFee` and `yieldFee` to **zero** for the asset.

This results in **silent loss of configured fee parameters**, breaking assumptions elsewhere in the system that depend on these fields retaining their configured values.

Any existing non-zero insurance or yield fees are effectively erased every time `setFees` is called.

Impact: Insurance fee and yield fee are forcibly reset to 0, eliminating two income streams from the protocol. Currently these two fees are not used in the protocol, but in the future this will be changed. Also the `USSTDepegController` could not be upgraded.

Recommended Mitigation: Only update the fields that should actually change (e.g., mintRate, burnRate).

STBL: Fixed in commit 4ec6d1

Cyfrin: Verified. `yieldFee` and `insuranceFee` are preserved by passing the current values when updating the fees on the Register; Only `depositFee` and `withdrawFee` are updated.

7.3.3 `owner` in `STBL_YLDSplitter` is never initialized

Description: The contract's `owner` variable is **never initialized** in the constructor. Because `owner` defaults to the zero address, the `onlyOwner` modifier permanently reverts for all protected functions such as `transferOwnership` and `setMinAfterSplitValue`.

This makes the contract effectively **owner-less and unmanageable** from deployment onward.

Impact: * **All owner-restricted functionality becomes unusable.**

- `transferOwnership` and `setMinAfterSplitValue` always revert.
- No address can ever obtain ownership or modify critical configuration values.
- The contract cannot be administered after deployment, which may prevent required updates, emergency responses, or parameter adjustments.

Recommended Mitigation: Initialize `owner` inside the constructor:

```
constructor(address _yldAddress) {
    owner = msg.sender;         // FIX: set contract owner
    yldContract = iSTBL_YLD(_yldAddress);
}
```

Or use `Ownable2Step`.

STBL: Fixed in commit 9de8d0.

Cyfrin: Verified.

7.3.4 Incorrect remainder calculation causing underflow in `splitNftByLot`

Description: When calculating the remainder NFT metadata in `splitNftByLot`, the code multiplies `oldYldMetadata` fields by `numLots` instead of properly calculating the remaining amount. This causes severe underflow as it attempts to subtract a much larger value from a smaller one.

```
newYldMetadataRemainder.stableValueGross =
    oldYldMetadata.stableValueGross -
    (oldYldMetadata.stableValueGross * numLots);

newYldMetadataRemainder.depositfeeAmount =
    oldYldMetadata.depositfeeAmount -
    (oldYldMetadata.depositfeeAmount * numLots);

// Same pattern for all other fields...
```

Impact:

- Arithmetic underflow causing transaction reversion

- `splitNftByLot` function becomes completely unusable when remainder > 0
- Users cannot split their NFTs using the lot-based method
- Loss of core functionality for the contract

Proof of Concept:

```
// User has NFT with:
// stableValueGross = 1000
// stableValueNet = 950
// parts_ratio_lot = 90 (lot size)

// Calculations:
numLots = 950 / 90 = 10
remainder = 950 % 90 = 50

// When calculating remainder metadata:
newYldMetadataRemainder.stableValueGross =
    1000 - (1000 * 10) =
    1000 - 10000 =
    -9000  //  UNDERFLOW - Transaction reverts

// Expected calculation should be:
// 1000 - (100 * 9) = 1000 - 900 = 100
// Where 100 = oldYldMetadata.stableValueGross / numLots
```

Recommended Mitigation: Correct the remainder calculations to use division instead of multiplication:

```
if (remainder > 0) {
    newYldMetadataRemainder.assetID = oldYldMetadata.assetID;
    newYldMetadataRemainder.uri = oldYldMetadata.uri;
    newYldMetadataRemainder.additionalBuffer = oldYldMetadata.additionalBuffer;
    newYldMetadataRemainder.isDisabled = oldYldMetadata.isDisabled;

    newYldMetadataRemainder.stableValueNet = remainder;

    // Correct calculation: subtract what was already allocated to (numLots-1) lots
    newYldMetadataRemainder.assetValue =
        oldYldMetadata.assetValue -
        (newYldMetadata.assetValue * (numLots - 1));

    newYldMetadataRemainder.stableValueGross =
        oldYldMetadata.stableValueGross -
        (newYldMetadata.stableValueGross * (numLots - 1));

    newYldMetadataRemainder.depositfeeAmount =
        oldYldMetadata.depositfeeAmount -
        (newYldMetadata.depositfeeAmount * (numLots - 1));

    newYldMetadataRemainder.haircutAmount =
        oldYldMetadata.haircutAmount -
        (newYldMetadata.haircutAmount * (numLots - 1));

    newYldMetadataRemainder.haircutAmountAssetValue =
        oldYldMetadata.haircutAmountAssetValue -
        (newYldMetadata.haircutAmountAssetValue * (numLots - 1));

    newYldMetadataRemainder.withdrawfeeAmount =
        oldYldMetadata.withdrawfeeAmount -
        (newYldMetadata.withdrawfeeAmount * (numLots - 1));

    newYldMetadataRemainder.insurancefeeAmount =
        oldYldMetadata.insurancefeeAmount -
```

```
        (newYldMetadata.insurancefeeAmount * (numLots - 1));
}
```

STBL: Fixed in commit 6701d17.

Cyfrin: Verified. `STBL_YLDSplitter::splitNftByLot` function has been removed.

7.3.5 Incorrect value for `DEFAULT_ADMIN_ROLE` on `DPT2` and `DLT2` causes calls restricted to default admin only to revert

Description: On DLT2 and DPT2, the value of the `DEFAULT_ADMIN_ROLE` variable is set as follows:

```
contract STBL_DPT2_Vault is
   ...
{
   ...

    /** @notice Role identifier for contract DEFAULT_ADMIN_ROLE authorization */
    bytes32 public constant DEFAULT_ADMIN_ROLE =
@>    keccak256("DEFAULT_ADMIN_ROLE");
}
```

The problem is that calls restricted to the DEFAULT_ADMIN_ROLE are validated on the Register, and, the value for the `DEFAULT_ADMIN_ROLE` in the Register is the one set on the `AccessControlUpgradeable` contract, which is:

```
abstract contract AccessControlUpgradeable is Initializable, ContextUpgradeable, IAccessControl,
↪  ERC165Upgradeable {
    ...

    bytes32 public constant DEFAULT_ADMIN_ROLE = 0x00
}
```

The difference between the two will cause calls on the DLT2 and DPT2 to revert because the requested role to be checked would be different than the actual value of the `DEFAULT_ADMIN_ROLE` as per the Register.

```
    function setClaimContract(address _claim) external {
@<      if (!registry.hasRole(DEFAULT_ADMIN_ROLE, _msgSender()))
            revert STBL_UnauthorizedCaller();

        ...
    }
```

Impact: Calls restricted to DEFAULT_ADMIN_ROLE from the DPT2 and DLT2 will revert.

Recommended Mitigation: Update the value assigned to `DEFAULT_ADMIN_ROLE` on DPT2 and DLT2 to be the same value as the one on the Register (`0x00`).

STBL: Fixed in commit e03f003

Cyfrin: Verified.

7.4 Low Risk

7.4.1 Recipient parameter ignored in `splitNftByPart` and `splitNftByRatio`

Description: The `YLDSplitStruct` includes a `recipient` field to specify who should receive the split NFTs. However, in `splitNftByPart()` and `splitNftByRatio()`, this parameter is completely ignored and NFTs are always minted to `msg.sender`. Only `splitNftByLot()` correctly uses the `recipient` parameter. This creates inconsistent behavior across split methods.

```
struct YLDSplitStruct {
    uint256 ID;
    SplitType splitType;
    address recipient; // Intended recipient
    uint256 parts_ratio_lot;
}

function splitNftByPart(YLDSplitStruct calldata splitParams) internal {
    // ...
    for (uint256 i = 0; i < splitParams.parts_ratio_lot; i++) {
        yldContract.mint(msg.sender, newYldMetadata);
    }
}

function splitNftByRatio(YLDSplitStruct calldata splitParams) internal {
    // ...
    yldContract.mint(msg.sender, newYldMetadata1);
    yldContract.mint(msg.sender, newYldMetadata2);
}

function splitNftByLot(YLDSplitStruct calldata splitParams) internal {
    // ...
    yldContract.mint(splitParams.recipient, newYldMetadata); //  Uses recipient
}
```

Impact:

- Users cannot split NFTs directly to another address using BY_COUNT or BY_RATIO methods
- Inconsistent behavior across different split types creates confusion
- Users must perform an extra transfer transaction after splitting, increasing gas costs
- The `recipient` parameter in the struct becomes misleading for 2 out of 3 split methods
- Integration issues for protocols expecting consistent recipient behavior

Recommended Mitigation: Update `splitNftByPart` and `splitNftByRatio` to use the `recipient` parameter consistently.

STBL: Fixed in commit b27e38

Cyfrin: Verified.

7.4.2 Missing operator approval check despite error message

Description: All split functions check only NFT ownership but not operator approvals, despite the error message being named `NotOwnerNorApproved`. The current implementation prevents approved operators (via `approve` or `setApprovalForAll`) from splitting NFTs, even though the error name suggests this should be supported.

```
function splitNftByPart(YLDSplitStruct calldata splitParams) internal {
    address nftOwner = yldContract.ownerOf(splitParams.ID);
    if (nftOwner != msg.sender) revert NotOwnerNorApproved();
    // ...
}
```

```
// Same pattern in splitNftByRatio and splitNftByLot
```

Impact:

- Approved operators cannot split NFTs on behalf of owners
- Breaks composability with other contracts that receive operator approval
- Misleading error message suggests approval should work but doesn't

Recommended Mitigation: Implement proper approval checking consistent with ERC721 standards:

```
function splitNftByPart(YLDSplitStruct calldata splitParams) internal {
    address nftOwner = yldContract.ownerOf(splitParams.ID);

    // Check if caller is owner, approved operator, or has specific token approval
    bool isAuthorized = nftOwner == msg.sender ||
                        yldContract.isApprovedForAll(nftOwner, msg.sender) ||
                        yldContract.getApproved(splitParams.ID) == msg.sender;

    if (!isAuthorized) revert NotOwnerNorApproved();

    // ... rest of function ...
}

// Apply same pattern to splitNftByRatio and splitNftByLot
```

Alternatively, if the design intentionally restricts splits to owners only, rename the error for clarity:

```
error NotOwner(); // More accurate if approvals are intentionally not supported
```

And update the validation:

```
if (nftOwner != msg.sender) revert NotOwner();
```

STBL: Fixed in commit 7b7284

Cyfrin: Verified

7.4.3 Invalid NFT with zeroed fields minted when remainder is equal to 0

Description: The `STBL_YLDSplitter::splitNftByLot` function declares `YLD_Metadata memory newYldMetadataRemainder;` at the beginning but only initializes its fields (e.g., `assetID`, `uri`, `additionalBuffer`, `isDisabled`, `stableValueNet`, `assetValue`, `stableValueGross`, `depositfeeAmount`, `haircutAmount`, `haircutAmountAssetValue`, `withdrawfeeAmount`, `insurancefeeAmount`) inside the `if (remainder > 0)` block. If `remainder == 0`, this block is skipped, leaving `newYldMetadataRemainder` uninitialized. In Solidity, uninitialized memory structs default to zero values for all fields (e.g., `stableValueNet = 0`, `assetValue = 0`, and strings/booleans at defaults). The final `yldContract.mint(splitParams.recipient, newYldMetadataRemainder);` call is placed outside the `if` block, so it executes regardless, minting this zeroed-out metadata as a new NFT. This occurs after minting `numLots - 1` equal-lot NFTs via the loop, but without properly handling the case where there is no remainder, resulting in an extra invalid NFT being created instead of completing the intended split.

Impact: Minting an uninitialized, zero-value NFT when `remainder == 0` introduces an invalid token into the system, which could fail downstream validations. This invalid NFT might circulate, leading to errors in ownership tracking, value calculations, or interactions like transfers, merges, or withdrawals. Transactions will succeed but produce unexpected results, eroding trust in the NFT splitting mechanism.

Recommended Mitigation: Refactor the logic to avoid minting an uninitialized struct by conditionally handling the final mint based on the remainder. After the loop that mints `numLots - 1` equal lots using `newYldMetadata`, use:

```
if (remainder > 0) {
    /* set newYldMetadataRemainder fields as before */
    yldContract.mint(splitParams.recipient, newYldMetadataRemainder);
```

```
}
else {
    yldContract.mint(splitParams.recipient, newYldMetadata);
}
```

This ensures that when `remainder == 0`, the last full lot is minted using the pre-calculated `newYldMetadata` (a valid equal lot), completing the split without creating a zero-value NFT.

STBL: Fixed in commit 67ac0d5.

Cyfrin: Verified. An extra NFT is minted only if remainder is > 0.

7.4.4 Arithmetic underflow in `withdrawERC20` when there is a lower price of asset token prevents governance from using `insuranceFunds` to make investor's whole

Description: The yield of the underlying token (Benji) is distributed in an aidrop-based model to the user's accounts, meaning the raw balance of the contract will increment, as opposed to price rebasing in perpetuity. While the underlying collateral has the lowest risks, capital is still susceptible to risk, as pointed by Franklin Templeton.

The vault tracks deposits using deposit-time pricing but calculates withdrawals using current market pricing. When prices decrease, the withdrawal calculation requires more tokens than the vault's accounting system has tracked as available, more over, the arithmetic underflow would prevent governance from donating directly to the Vault contract because that donation wouldn't be reflected on the internal accounting, meaning, even if governance donates tokens to cover any negative price, the underflow would still occur because the donation doesn't update the internal accounting.

The consequences of a lower price (requiring more asset units for the same amount of USD) will impact the withdrawal flow when subtracting the `withdrawAssetValue (and fee)` from the `VaultData.assetDepositNet`, withdrawals will hit an underflow on that operation because the amount of asset units that will be discounted will be greater than the amount of asset units on the `VaultData.assetDepositNet`.

```
    function withdrawERC20(
        address _to,
        YLD_Metadata memory MetaData
    ) external isValidIssuer {
        AssetDefinition memory AssetData = registry.fetchAssetData(assetID);

//@audit => In case of a lower price, governance can make investors whole by donating Benji from the
↪  Insurance Fund
        if (
            iSTBL_DLT1_AssetOracle(AssetData.oracle).fetchForwardPrice(
                IERC20(AssetData.token).balanceOf(address(this))
            ) < (VaultData.depositValueUSD)
        )
            revert STBL_Asset_InsufficientVaultValue(
                iSTBL_DLT1_AssetOracle(AssetData.oracle).fetchForwardPrice(
                    IERC20(AssetData.token).balanceOf(address(this))
                ),
                VaultData.depositValueUSD
            );

        uint256 withdrawfeeAmount = calculateWithdrawFees(
            MetaData,
            AssetData.withdrawFees
        );

        // Calculate Withdraw asset value
        uint256 withdrawAssetValue = iSTBL_DLT1_AssetOracle(AssetData.oracle)
            .fetchInversePrice(
                ((MetaData.stableValueNet + MetaData.haircutAmount) -
                    withdrawfeeAmount)
```

```
        ); //@audit gets the latest price -> if price is lower, withdrawAssetValue can be greater
        ↪  than assetDepositNet

    ...


VaultData.assetDepositNet -= (withdrawAssetValue +
        withdrawFeeAssetValue); //@audit if the above happens, assetDepositNet will underflow

    ...
```

Impact: Denial of service on user withdrawals in the event of lower NVA for `Benji` token and preventing governance from ensuring users are make whole via donations using the `insuranceFunds`

Recommended Mitigation: Consider removing the `VaultData.assetDepositNet` variable from the DLT1 and DLP1 vault contracts. For this contract, it is not required to track the `assetDepositNet` deposited on the Vault, given that the underlying asset of the Vault doesn't rebase.

Benji distributes yield by airdropping tokens into users' accounts, as opposed to rebasing the token's price.

This change aligns the vault's internal accounting model with the actual economic behavior of the non-rebasing asset and airdrop-based yield distribution, while enhancing operational transparency for risk-mitigation processes.

Exclusion of `assetDepositNet` in `DLT1` and `DPT1` aligns the contracts' state management with the actual tokenomics of the non-rebasing, airdrop-driven yield asset (Benji Token), thereby reducing storage footprint, simplifying logic, and lowering deployment and operational gas costs without sacrificing accounting accuracy or governance oversight capabilities.

STBL: Acknowledged. In case of a depeg and bank run, the last withdrawer can be settled off-chain.

7.4.5 Rounding errors in `splitNftByPart` and `splitNftByRatio`

Description: When splitting NFTs by part or ratio, integer division is used to calculate the new NFT values. This truncates decimal values, causing the sum of the newly minted NFTs' values to be **less than the original NFT's value**.

Impact: * Small discrepancies in `assetValue`, `stableValueNet`, or other financial fields may accumulate over multiple splits.

- NFTs could end up with values below `minStableValueNet`, causing splits to fail or creating inconsistent valuations.
- Could result in unintentional loss of value from the original NFT.

Proof of Concept: Assume an NFT with the following metadata:

Field	Value
assetValue	100
stableValueNet	100

We attempt to split this NFT into **3 equal parts** (`parts_ratio_lot = 3`) using `splitNftByPart`:

```
newYldMetadata.assetValue = 100 / 3;        // 33
newYldMetadata.stableValueNet = 100 / 3;    // 33
```

Observation:

NFT Part	assetValue	stableValueNet
Part 1	33	33
Part 2	33	33
Part 3	33	33

- **Sum of assetValue**: 33 + 33 + 33 = 99 (original 100 → **1 lost**)
- **Sum of stableValueNet**: 33 + 33 + 33 = 99 (original 100 → **1 lost**)

The remaining value is effectively lost due to truncation.

Recommended Mitigation: Allocate the remainder from integer division to the **last NFT** (or first, consistently).

```
uint256 baseValue = oldYldMetadata.assetValue / splitParams.parts_ratio_lot;
uint256 remainder = oldYldMetadata.assetValue % splitParams.parts_ratio_lot;

for (uint256 i = 0; i < splitParams.parts_ratio_lot; i++) {
    YLD_Metadata memory newYld = baseMetadata;
    newYld.assetValue = baseValue;
    if (i == splitParams.parts_ratio_lot - 1) {
        newYld.assetValue += remainder; // Add remainder to last NFT
    }
    yldContract.mint(msg.sender, newYld);
}
```

STBL: Acknowledged. This will be dust amount which will be remaining which is acceptable unless its create a big problem as 10^18 is token decimal which is taken into consideration.

7.4.6 `VaultData` values not reset on emergency withdraw

Description: During an emergency withdrawal (`STBL_DLT1_Vault::emergencyWithdraw`, `STBL_DPT1_Vault::emergencyWithdraw`), the values of `VaultData` are not reset. This means that after an emergency withdrawal occurs, the total values and other metrics related to the vault's state remain unchanged. This, combined with the fact that the vault does not pause, can lead to incorrect accounting or transactions reverting.

Recommended Mitigation: To address this issue, it is recommended to reset the relevant fields in `VaultData` to their initial state or to appropriate values after an emergency withdrawal is executed. This could involve setting some of the metrics to zero or adjusting them based on the amount withdrawn. Implementing this reset will ensure that the vault's state accurately reflects its financial position following an emergency withdrawal, thus maintaining the integrity of the protocol's operations.

STBL: Acknowledged.

7.4.7 Inconsistent minimum rate handling in `calculateUnifiedRate`

Description: In `calculateUnifiedRate` there is inconsistent behavior when compared to `calculateBurningRate` and `calculateMintingRate`.

Specifically, when `abs(delta) <= toleranceThreshold`:

- **calculateUnifiedRate** returns **rate = 0**
- **calculateBurningRate** returns **minRate**
- (and likewise `calculateMintingRate` returns **minRate**)

This means that for the same price deviation, depending on which function is used, the protocol produces two different intervention intensities.

Here is the `calculateUnifiedRate`

```
function calculateUnifiedRate(
        RateParams memory params,
        int256 delta
    ) public pure returns (uint256 rate, bool isMinting) {
        uint256 absDelta = abs(delta);

        // If within tolerance, no intervention
        if (absDelta <= params.toleranceThreshold) {
            return (0, delta > 0);
        }

        // Determine if minting or burning
        isMinting = delta > 0;

        // If beyond depeg threshold, use maximum rate
        if (absDelta >= params.depegThreshold) {
            return (params.maxRate, isMinting);
        }
```

and the `calculateBurningRate`:

```
function calculateBurningRate(
        RateParams memory params,
        int256 delta
    ) internal pure returns (uint256 rate) {
        // Only burn when price is below peg (negative delta)
        if (delta >= 0) {
            return 0;
        }

        uint256 absDelta = abs(delta);

        // If within tolerance threshold, use minimum rate
        if (absDelta <= params.toleranceThreshold) {
            return params.minRate;
        }
        // If beyond depeg threshold, use maximum rate
        else if (absDelta >= params.depegThreshold) {
            return params.maxRate;
        }
```

Impact: * **Protocol incentives behave inconsistently depending on which function path is used**, potentially leading to weaker peg-restoration forces.

- **Deviation near the tolerance threshold results in no corrective action** when using the unified rate, while burning/minting logic expects at least minimal intervention.

Recommended Mitigation: Align the behavior of `calculateUnifiedRate` with the logic used in `calculateBurningRate` and `calculateMintingRate` by returning `minRate` instead of `0` when within tolerance:

```
if (absDelta <= params.toleranceThreshold) {
    return (params.minRate, delta > 0);
}
```

STBL: Fixed in commit 35f122.

Cyfrin: Verified. `getUnifiedRate()` and `calculateUnifiedRate()` were removed.

7.5 Informational

7.5.1 `STBL_YLDSplitter` not emitting events on important state changes and actions

Description: Several state-changing functions on `STBL_YLDSplitter` (notably `transferOwnership`, `setMinAfterSplitValue`, and `splitYldNft`/specific split helpers) do not emit events. These functions update contract state or perform important actions (ownership change, configuration change, token splitting/minting/burning) but the contract itself does not log those actions with events.

Recommended Mitigation: Emit explicit events for each important state change and action:

- `OwnershipTransferred(address indexed previousOwner, address indexed newOwner)`
- `MinStableValueUpdated(uint256 indexed oldValue, uint256 indexed newValue)`
- `YLDSplitExecuted(address indexed operator, uint256 indexed originalTokenId, SplitType splitType, address indexed recipient, uint256 parts_ratio_lot, uint256 mintedCount)`
- Optionally: `YLDSplitMinted(address indexed recipient, uint256 indexed newTokenId, uint256 stableValueNet)`

STBL: Fixed in commit 1bdcc0d.

Cyfrin: Verified.

7.5.2 Missing `recipient` zero address check

Description: In `STBL_YLDSplitter` the `splitParams.recipient` address is not validated to ensure it is not a zero address (0x0) before being used in the `mint` function calls within the `splitNftByRatio`, `splitNftByLot` and `splitNftByPart` functions. A zero address is typically used to represent an invalid or non-existent address in Ethereum, and sending tokens or NFTs to a zero address results in the loss of those assets, as they cannot be recovered.

Recommended Mitigation: Implement a check at the beginning of the `splitNftByPart`, `splitNftByRatio`, and `splitNftByLot` functions to validate that `splitParams.recipient` is not the zero address. If it is, revert the transaction with an appropriate error message. For example:

```
if (splitParams.recipient == address(0)) revert InvalidRecipient();
```

Define a new error for clarity:

```
error InvalidRecipient();
```

This validation should be placed before any operations that involve the `recipient` address to prevent any unintended loss of tokens or NFTs.

STBL: Fixed in commit 17849ef.

Cyfrin: Verified.

7.5.3 `FeesWithdrawn` event emitted even when `feesToWithdraw` equals zero

Description: The `FeesWithdrawn` event is emitted in the `STBL_DLT1_Vault::iWithdrawFees` function even when no fees are actually withdrawn (i.e., when `feesToWithdraw` equals zero). Emitting events when no action has taken place can lead to confusion for users and off-chain systems that rely on these events for tracking fee withdrawals.

Recommended Mitigation: Modify the `iWithdrawFees` function to emit the `FeesWithdrawn` event only when `feesToWithdraw` is greater than zero. This can be achieved by wrapping the event emission in a conditional statement:

```
if (feesToWithdraw > 0) {
    IERC20(AssetData.token).safeTransfer(
        treasury,
        feesToWithdraw.convertFrom18Decimals(
```

```
            DecimalConverter.getTokenDecimals(AssetData.token)
        )
    );
    emit FeesWithdrawn(
        treasury,
        _depositFee,
        _withdrawFee,
        _yieldFee,
        _insuranceFee,
        feesToWithdraw
    );
}
```

STBL: Fixed in commit 8aeb416.

Cyfrin: Verified.

7.5.4 Outdated fields on `YLD_Metadata` no longer required given dynamic fees based on new `Peg` mechanism

Description: `YLD_Metadata` struct contains fields that are no longer a must-have given that these fields are now dynamically updated on the Registry to be compatible with the Peg mechanism, specifically, the field `MetaData.Fees.depositFee` and `MetaData.Fees.withdrawFee`, these fees are no longer fixed for the duration of the Position; instead, they are adjusted dynamically based on the Peg mechanism to either incentivize minting or burning, thereby maintaining the Stability of the USST stablecoin's Peg.

Recommended Mitigation: Consider removing the outdated fields from the YLD_Metadata struct. This will keep the code cleaner and more concise.

STBL: Acknowledged.

7.5.5 Typo in error name `SpiltRejected` in `STBL_YLDSplitter`

Description: The custom error `SpiltRejected` in is misspelled in `STBL_YLDSplitter`. The intended spelling is `SplitRejected`.

Recommended Mitigation: Rename the error to `SplitRejected` consistently across the contract.

STBL: Fixed in commit 49588d2.

Cyfrin: Verified.

7.5.6 Wrong docstring and comment regarding percentage conversion

Description: The `STBL_USSTDepegController::rateToPercentage` function's documentation claims the output has 2 decimal places, but the actual implementation produces a result with 7 decimal places. The docstring states:

- `@dev` comment: "Converts from 18 decimal precision to percentage with 2 decimal places"
- `@return` comment: "The rate as percentage with 2 decimals (e.g., 500 = 5.00%)"
- `@custom:example`: "Input: 5e16 (0.05 * 1e18) → Output: 500 (5.00%)"

However, the calculation `(rate * PERCENTAGE_PRECISION) / PRECISION` actually multiplies by `PERCENTAGE_PRECISION`, resulting in 7 decimal places of precision, not 2. The example is incorrect—if the input is `5e16`, the output would be `5e25 / 1e18 = 5e7 = 50000000`, which represents 7 decimals of precision.

Recommended Mitigation: Update the docstring and example to accurately reflect that the function outputs a value with 7 decimal places of precision.

1. Correct the documentation to state "7 decimal places" instead of "2 decimal places"
2. Provide an accurate example showing the actual output format with 7 decimals

STBL: Fixed in commit 9fa5e96.

Cyfrin: Verified.

7.5.7 Unused constant declaration

Description: The `PERCENTAGE_PRECISION` constant is declared in the `STBL_USSTOracle` contract but is never referenced or used anywhere within this contract. An identical constant with the same value (`1e9`) exists in `STBL_USSTDepegController` where it is actively used in the `rateToPercentage` function for percentage calculations.

This represents code duplication and unnecessary storage of an unused constant in the oracle contract.

Recommended Mitigation: Remove the unused `PERCENTAGE_PRECISION` constant from STBL_USSTOracle.sol:

```
contract STBL_USSTOracle is AccessControl, iSTBL_USSTOracle {
    /**
     * @notice Precision constant for calculations
     * @dev Used to scale prices to 18 decimal places
     */
    uint256 public constant PRECISION = 1e18;

    // ...existing code...
    // REMOVE: uint256 public constant PERCENTAGE_PRECISION = 1e9;

    address public immutable curvePool;
    // ...existing code...
}
```

The constant is already properly defined and used in `STBL_USSTDepegController` where it belongs, so no functionality will be lost by removing it from the oracle contract.

STBL: Fixed in commit 8a45c65.

Cyfrin: Verified.

7.5.8 Lack of events for important actions in `STBL_YLDSplitter`

Description: The `STBL_YLDSplitter` contract performs several critical state-changing operations without emitting any events. This makes it difficult for off-chain systems, UIs, and monitoring tools to track important actions in real-time. The affected functions are:

1. `transferOwnership()` - Changes the contract owner but does not emit an event to signal this critical ownership change
2. `setMinAfterSplitValue()` - Updates the minimum stable value threshold but provides no event notification
3. `splitYldNft()` - Performs NFT splitting operations (which internally call `splitNftByPart()`, `splitNftByRatio()`, or `splitNftByLot()`) but does not emit events to indicate successful splits

The lack of events creates opacity in contract activity and prevents external systems from efficiently indexing and monitoring state changes. This is particularly problematic for NFT operations where tracking mint/burn events is essential for users and dApps.

Recommended Mitigation: Define and emit the following events:

```
// Add event definitions to the contract
event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
event MinStableValueUpdated(uint256 newMinValue);
event NFTSplitByCount(uint256 indexed originalNFTId, uint256 numParts, address indexed recipient);
event NFTSplitByRatio(uint256 indexed originalNFTId, uint256 ratio, address indexed recipient);
event NFTSplitByLot(uint256 indexed originalNFTId, uint256 lotSize, uint256 numLots, address indexed
↪  recipient);
```

Then emit these events in their respective functions:

- Emit `OwnershipTransferred` in `transferOwnership()` with the old and new owner addresses
- Emit `MinStableValueUpdated` in `setMinAfterSplitValue()` with the new minimum value
- Emit `NFTSplitByCount`, `NFTSplitByRatio`, or `NFTSplitByLot` (as appropriate) in each split function with details about the original NFT ID, split parameters, and recipient address

This enables proper event indexing and provides transparency for all significant contract operations.

STBL: Fixed in commit 4e3d45c.

Cyfrin: Verified.