STBL Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

Stalin

September 10, 2025

Contents

1 About Cyfrin 2

2 Disclaimer 2

3 Risk Classification 2

4 Protocol Summary 2

5 Audit Scope 2

6 Executive Summary 3

7 Findings 6

7.1 Medium Risk . . . 6

7.1.1 Incorrect haircut asset value conversion in `STBL_PT1_Issuer::generateMetaData` . . . 6

7.1.2 High centralization risk in `STBL_USST::bridgeBurn` . . . 6

7.1.3 Front-running attack on yield distribution allows potential theft of accumulated rewards . . . 6

7.1.4 Arithmetic underflow in `withdrawERC20` when there is a negative rebasing of asset tokens . . 7

7.2 Low Risk . . . 9

7.2.1 Missing zero address checks in `STBL_Register::setupAsset` . . . 9

7.2.2 Insufficient fee validation in `STBL_Register::setupAsset` can cause underflow . . . 9

7.2.3 Insufficient duration validation in `STBL_Register::setupAsset` can lock user withdrawals . . 9

7.2.4 Disabling an asset via `STBL_Register::disableAsset` with active deposits can DOS user withdrawals . . . 10

7.2.5 `STBL_Register::addAsset` does not check for non-empty asset name . . . 10

7.2.6 Race condition at duration expiration boundary during withdrawals . . . 10

7.2.7 Treasury cannot withdraw expired assets if NFT is disabled . . . 11

7.3 Informational . . . 12

7.3.1 Misleading `ContractType` field suggests that assets can be non ERC20 tokens . . . 12

7.3.2 Missing event emissions for critical oracle parameter changes . . . 12

7.3.3 Mismatching variable naming for `Metadata.depositBlock` . . . 12

7.3.4 CEI pattern violation in `STBL_PT1_Vault::withdrawFees` . . . 12

7.3.5 Misleading inline comment in `STBL_LT1_Issuer::withdrawExpired` function . . . 13

7.3.6 Anybody can deposit and withdraw on behalf of users . . . 13

7.4 Gas Optimization . . . 14

7.4.1 Unnecessary usage of `_msgSender()` to validate if caller is the `Issuer` on the `STBL_PT1_YieldDistributor` . . . 14

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 like USDY (Ondo's yield-bearing USD) and OUSG (Ondo Short-Term US Government Bond Token) and receive fungible stablecoin tokens alongside yield-bearing NFTs.

The protocol supports two distinct asset categories: *PT1 Assets (Principal Token - Type 1)*

Designed for assets like USDY that appreciate through rebasing/compound interest Examples: USDY_PT1, OUSG_PT1 Yield comes from asset price appreciation over time

LT1 Assets (Liquidity Token - Type 1)

Designed for assets with more complex yield mechanisms Examples: USDY_LT1, OUSG_LT1 Similar yield distribution but different internal mechanics

5 Audit Scope

Following contracts were part of the scope for the current audit.

- STBL_OracleLib.sol
- STBL_LT1_Issuer.sol
- STBL_LT1_Oracle.sol
- STBL_LT1_Vault.sol
- STBL_LT1_YieldDistributor.sol
- STBL_OracleLib.sol
- STBL_PT1_Issuer.sol

- STBL_PT1_Oracle.sol
- STBL_PT1_Vault.sol
- STBL_PT1_YieldDistributor.sol
- STBL_Core.sol
- STBL_Register.sol
- STBL_AssetDefinitionLib.sol
- STBL_DecimalConverter.sol
- STBL_Decoder.sol
- STBL_MetadataLib.sol
- STBL_Token.sol
- STBL_USST.sol
- STBL_YLD.sol
- Lock.sol

6 Executive Summary

Over the course of 10 days, the Cyfrin team conducted an audit on the STBL smart contracts provided by STBL. In this period, a total of 18 issues were found.

STBL is a decentralized finance protocol that enables users to deposit various assets and receive fungible stablecoin tokens (USST) alongside yield-bearing non-fungible tokens (YLD). The protocol employs a sophisticated fee structure, yield distribution mechanism, and supports various asset lifecycles with time-based withdrawal restrictions.

Primary issues discovered during the audit included arithmetic underflows during negative rebasing of assets, front-running MEV attacks around yield distrubution, incorrect asset value conversions based on oracle price and high centralization risks in bridge burning functions. **All reported issues were successfully resolved by the protocol and verified by Cyfrin.**

The audit revealed excellent test coverage throughout the protocol, including comprehensive fuzz testing implementations. Robust fuzz tests covering core functionality, asset lifecycles, registry operations, and token behaviors significantly strengthening the protocol's security posture.

The protocol contains several crucial administrative functions that require careful governance and security considerations. Some key administrative functions that significantly impact protocol users are:

- `emergencyWithdraw`: Allows emergency extraction of funds from the protocol
- `disableAssets`: Can disable specific assets, potentially affecting user withdrawals
- `withdrawExpired`: Enables treasury to withdraw expired user assets

In light of the above, we recommend establishing comprehensive documentation for operational procedures and emergency response protocols. The security infrastructure for administrative wallets and deployment processes should also be thoroughly reviewed and documented before mainnet launch.

Summary

Project Name	STBL
Repository	contract
Commit	568c9211c9c1...
Audit Timeline	Aug 19th - Sep 1st
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	4
Low Risk	7
Informational	6
Gas Optimizations	1
Total Issues	18

Summary of Findings

[M-1] Incorrect haircut asset value conversion in `STBL_PT1_Issuer::generateMetaData`	Resolved
[M-2] High centralization risk in `STBL_USST::bridgeBurn`	Resolved
[M-3] Front-running attack on yield distribution allows potential theft of accumulated rewards	Acknowledged
[M-4] Arithmetic underflow in `withdrawERC20` when there is a negative rebasing of asset tokens	Resolved
[L-1] Missing zero address checks in `STBL_Register::setupAsset`	Resolved
[L-2] Insufficient fee validation in `STBL_Register::setupAsset` can cause underflow	Resolved
[L-3] Insufficient duration validation in `STBL_Register::setupAsset` can lock user withdrawals	Resolved
[L-4] Disabling an asset via `STBL_Register::disableAsset` with active deposits can DOS user withdrawals	Resolved
[L-5] `STBL_Register::addAsset` does not check for non-empty asset name	Resolved
[L-6] Race condition at duration expiration boundary during withdrawals	Resolved
[L-7] Treasury cannot withdraw expired assets if NFT is disabled	Resolved
[I-1] Misleading `ContractType` field suggests that assets can be non ERC20 tokens	Resolved
[I-2] Missing event emissions for critical oracle parameter changes	Resolved

[I-3] Mismatching variable naming for `Metadata.depositBlock`	Resolved
[I-4] CEI pattern violation in `STBL_PT1_Vault::withdrawFees`	Acknowledged
[I-5] Misleading inline comment in `STBL_LT1_Issuer::withdrawExpired` function	Resolved
[I-6] Anybody can deposit and withdraw on behalf of users	Acknowledged
[G-1] Unnecessary usage of `_msgSender()` to validate if caller is the `Issuer` on the `STBL_PT1_YieldDistributor`	Resolved

7 Findings

7.1 Medium Risk

7.1.1 Incorrect haircut asset value conversion in `STBL_PT1_Issuer::generateMetaData`

Description: `STBL_PT1_Issuer::generateMetaData` (and its `STBL_LT1_Issuer` counterpart) uses the wrong oracle conversion function when calculating `haircutAmountAssetValue`, resulting in mathematically incorrect values being stored in NFT metadata. The function uses `fetchForwardPrice()` that expects the input (`MetaData.haircutAmount`) to be in the asset currency but haircut amount is already converted into USD (when applied on `stableValueGross` which is in USD terms).

The issue stems from a unit conversion error where USD amounts are incorrectly passed to a function expecting asset amounts:

```
function generateMetaData(uint256 assetValue) internal view returns (YLD_Metadata memory MetaData) {
    // ... other calculations ...

    // @audit Step 1: Convert asset to USD using fetchForwardPRice
    MetaData.stableValueGross = iSTBL_PT1_AssetOracle(AssetData.oracle)
        .fetchForwardPrice(MetaData.assetValue);

    // @audit Step 2: Calculate haircut in USD terms
    MetaData = MetaData.calculateDepositFees(); // Sets haircutAmount in USD

    // @audit BUG - Wrong conversion function used
    MetaData.haircutAmountAssetValue = iSTBL_PT1_AssetOracle(AssetData.oracle)
        .fetchForwardPrice(MetaData.haircutAmount);  // @audit inverse of price needs to be used
        //                 ^^^^^^^^^^^^^^^^^^^^
        // @audit This expects ASSET AMOUNT but receives USD AMOUNT
}
```

Correct conversion here is by applying the inverse of the oracle price to get the haircut value in asset token denomination.

Impact: While the core vault accounting logic is unaffected, `haircutAmountAssetValue` metadata is incorrect. Off-chain systems reading metadata get wrong values.

Proof of Concept: Here is a mock calculation:

```
Step 1: stableValueGross = fetchForwardPrice(100e18)
       = (110000000 * 100e18) / 1e8 = 110e18 USD

Step 2: haircutAmount = (110e18 * 500) / 10000 = 5.5e18 USD

Step 3 (WRONG): haircutAmountAssetValue = fetchForwardPrice(5.5e18)
       = (110000000 * 5.5e18) / 1e8 = 6.05e18

Step 3 (CORRECT): haircutAmountAssetValue = fetchInversePrice(5.5e18)
       = (5.5e18 * 1e8) / 110000000 = 5e18
```

Recommended Mitigation: Consider using the `fetchInversePrice` to calculate `haircutAmountAssetValue`

STBL: Fixed in commit 1adc1f2.

Cyfrin: Verified.

7.1.2 High centralization risk in `STBL_USST::bridgeBurn`

Description: Current implementation of `STBL_USST::bridgeBurn` allows `BRIDGE_ROLE` to burn tokens from any arbitrary address without user approval. This creates unnecessary centralization risk when a safer approach is already demonstrated in the protocol's own `STBL_Token` contract.

```
// STBL_USST.solfunction bridgeBurn(
    address _from,       // @audit can be any address
    uint256 _amt,
    bytes memory _data
) external whenNotPaused onlyRole(BRIDGE_ROLE) {
    _burn(_from, _amt);  // @audit Burns from arbitrary address without consent
    emit BridgeBurn(_from, _amt, _data);
}
```

The above approach allows the `BRIDGE_ROLE` (initialized to the `DEFAULT_ADMIN`) to burn tokens from any address. An alternate implementation already implemented in `STBL_TOKEN` is much safer:

```
// STBL_Token.sol
function bridgeBurn(uint256 _amt) external whenNotPaused onlyRole(BRIDGE_ROLE) {
    _burn(_msgSender(), _amt);  // @audit Only burns caller's (bridge's) own tokens
}
```

Impact: Bridge contract compromise can allow mass token burning in a single transaction. This also significantly increases the trust assumptions and centralization risk around the bridge contract creating a single point of failure for entire token ecosystem.

Recommended Mitigation: Consider using the `STBL_Token` code for implementing `bridgeBurn` functionality. Alternatively, first transfer tokens from the caller into the contract and then burn them.

STBL: Fixed in commit a737746

Cyfrin: Verified.

7.1.3 Front-running attack on yield distribution allows potential theft of accumulated rewards

Description: Current yield distribution has following properties:

1. Periodic distribution - yield is calculated and distributed once every `yieldDuration` seconds. Onchain contracts show that `yieldDuration` for assets is 2,592,000 (30 days).
2. Proportionate rewards - all depositors receive yield proportional to their stake size regardless of deposit duration.
3. Predictable price changes - the 30-day distribution cycle coupled with external oracle updates allow MEV hunters to create exploitation opportunities.

The `distributeReward` function allows rewards to be distributed once every `yieldDuration` seconds and the rewards are proportional to the stake amounts:

```
// STBL_LT1_YieldDistributor.sol

function distributeReward(uint256 reward) external {
    // @audit once every yieldDuration seconds
    if (previousDistribution + AssetData.yieldDuration >= block.timestamp)
        revert STBL_Asset_YieldDurationNotReached(assetID, previousDistribution);

    // @audit proportional distribution without time weighting
    rewardIndex += (reward * MULTIPLIER) / totalSupply;
    previousDistribution = block.timestamp;
}
```

The `distributeYield` function calculates price differential using the updated oracle prices:

```
  function distributeYield() external {
        AssetDefinition memory AssetData = registry.fetchAssetData(assetID);
        uint256 differentialUSD = iCalculatePriceDifferentiation(); //@audit USD value calculated based
        ↪  on latest price
         if (differentialUSD > 0) {
               iSTBL_LT1_AssetYieldDistributor(AssetData.rewardDistributor)
                .distributeReward(
                    yieldAssetValue.convertFrom18Decimals(
                        DecimalConverter.getTokenDecimals(AssetData.token)
                    )
                );
       }
}
```

The above design allows for the following attack vector (for simplicity, lets assume a 10% fees and Alice as the only depositor)

- Alice deposits $100,000, receives NFT representing her stake
- Price appreciation accumulates over yield duration - 30 days (e.g., $10,000 in OUSG price gains or USDY compound interest)
- Alice expects to receive ~$9,000 yield (after 10% protocol fees)
- Bob monitors `previousDistribution + yieldDuration` to identify the exact distribution window (Bob also monitors the oracle updates and knows the price difference gives a significant upside on yield)
- Bob deposits $10,000,000 immediately before `distributeYield` is called
- Distribution occurs proportionally. Bob captures the bulk of rewards even though Alice staked for the entire 30 days

Impact: Longer term depositors can systematically lose accumulated rewards to opportunitistic whales who are OK to lock capital for yield duration.

Proof of Concept: Add the following test to `STBL_Test.t.sol`

```
    /**
     * @notice Demonstrates the yield frontrunning attack
     * @dev Shows how a whale can capture majority of yield rewards despite minimal stake duration
     */
    function test_YieldMEVAttack() public {
        // Test amounts
        address alice = address(0x1001); // Long-term depositor
        address bob = address(0x1002);   // MEV attacker
        uint256 aliceDeposit = 100000e18; // $100,000 equivalent
        uint256 bobDeposit = 10000000e18; // $10,000,000 (whale deposit)

        // Step 1: Alice deposits early and holds for the full period
        console.log("--- Step 1: Alice (Long-term depositor) deposits $100k ---");

        // Get LT1 contracts (using asset ID 2)
        STBL_LT1_Issuer issuerContract = STBL_LT1_Issuer(getAssetIssuer(2));
        STBL_LT1_Vault vaultContract = STBL_LT1_Vault(getAssetVault(2));
        STBL_LT1_YieldDistributor yieldDistributorContract =
        ↪  STBL_LT1_YieldDistributor(getAssetYieldDistributor(2));
        STBL_LT1_TestOracle oracleContract = STBL_LT1_TestOracle(getAssetOracle(2));
        STBL_LT1_TestToken token = STBL_LT1_TestToken(getAssetToken(2));

        // Mint tokens to Alice and approve
        vm.startPrank(admin);
        token.mintVal(alice, aliceDeposit);
        vm.stopPrank();

        vm.startPrank(alice);
        token.approve(address(vaultContract), aliceDeposit);
        uint256 aliceNftId = issuerContract.deposit(aliceDeposit);
        vm.stopPrank();

        console.log("Alice's NFT ID:", aliceNftId);
```

```solidity
console.log("Alice's deposit amount:", aliceDeposit);

// Step 2: Wait almost full yield duration and generate significant yield
console.log("--- Step 2: Wait 29 days, generate significant yield ---");

// Get current yield duration from asset config
AssetDefinition memory assetData = registry.fetchAssetData(2);
uint256 yieldDuration = assetData.yieldDuration;
console.log("Yield duration (seconds):", yieldDuration);

// Fast forward most of the duration (leave small window for Bob)
uint256 timeToWait = yieldDuration - 86400; // 1 day before yield can be distributed
vm.warp(block.timestamp + timeToWait);
console.log("Time advanced by:", timeToWait, "seconds (~29 days)");

// Simulate significant price appreciation
uint256 initialPrice = oracleContract.fetchPrice();
console.log("Initial oracle price:", initialPrice);

// Generate 10% price appreciation (1000 * 0.01 = 10)
for(uint256 i = 0; i < 1000; i++) {
    oracleContract.setPrice();
}

uint256 appreciatedPrice = oracleContract.fetchPrice();
console.log("Final oracle price:", appreciatedPrice);
console.log("Price appreciation:", ((appreciatedPrice - initialPrice) * 100) / initialPrice,
↪  "%");

// Step 3: Bob monitors and deposits just before yield distribution window
console.log("--- Step 3: Bob monitors yield distribution window ---");

// Bob waits until yield can be distributed
vm.warp(block.timestamp + 86401); // Now at yieldDuration exactly
console.log("Advanced to yield distribution window");

// Bob deposits whale amount right before distribution
vm.startPrank(admin);
token.mintVal(bob, bobDeposit);
vm.stopPrank();

vm.startPrank(bob);
token.approve(address(vaultContract), bobDeposit);
uint256 bobNftId = issuerContract.deposit(bobDeposit);
vm.stopPrank();

console.log("Bob's NFT ID:", bobNftId);
console.log("Bob deposits 100x more capital than Alice:", bobDeposit);

// Step 4: Check distribution ratios before yield distribution
console.log("--- Step 4: Analyzing stake distribution ---");

(uint256 aliceStake,,,) = yieldDistributorContract.stakingData(aliceNftId);
(uint256 bobStake,,,) = yieldDistributorContract.stakingData(bobNftId);
uint256 totalStake = yieldDistributorContract.totalSupply();

console.log("Alice stake:", aliceStake);
console.log("Bob stake:", bobStake);
console.log("Total stake:", totalStake);

uint256 aliceSharePct = (aliceStake * 100) / totalStake;
uint256 bobSharePct = (bobStake * 100) / totalStake;
console.log("Alice will receive:", aliceSharePct, "% of yield");
```

```
        console.log("Bob will receive:", bobSharePct, "% of yield");

        // Step 5: Trigger yield distribution
        console.log("--- Phase 5: Yield Distribution ---");

        uint256 expectedYield = vaultContract.CalculatePriceDifferentiation();
        console.log("Expected yield amount:", expectedYield);

        if (expectedYield > 0) {
            // Treasury distributes yield
            vm.prank(treasury);
            vaultContract.distributeYield();

            console.log("Yield distributed");
        }

        // Phase 6: Analyze unfair results
        console.log("--- Step 6: Attack Results ---");

        uint256 aliceRewards = yieldDistributorContract.calculateRewardsEarned(aliceNftId);
        uint256 bobRewards = yieldDistributorContract.calculateRewardsEarned(bobNftId);
        uint256 totalRewards = aliceRewards + bobRewards;

        console.log("Alice earned rewards:", aliceRewards);
        console.log("Bob earned rewards:", bobRewards);

        if (totalRewards > 0) {
            uint256 alicePct = (aliceRewards * 100) / totalRewards;
            uint256 bobPct = (bobRewards * 100) / totalRewards;

            console.log("UNFAIR DISTRIBUTION RESULTS:");
            console.log("Alice (30-day holder):", alicePct, "% of rewards");
            console.log("Bob (last-minute attacker):", bobPct, "% of rewards");

            // Verify attack succeeded
            assertTrue(bobRewards > aliceRewards, "MEV attack failed - Bob should earn more");
        }
```

Recommended Mitigation: Consider distributing yield more frequently - higher the frequency, lesser is the MEV possibility. We recommend atleast a weekly frequency to largely mitigate economic incentives for such an attack.

STBL: Acknowledged. This we will handled it via configuration and keep it in mind while setting config for yield duration.

Cyfrin: Acknowledged.

7.1.4 Arithmetic underflow in `withdrawERC20` when there is a negative rebasing of asset tokens

Description: The vault's accounting is based on the premise that the price of the underlying collateral will grow in perpetuity. While the underlying collateral has the lowest risks, capital is still susceptible to risk, especially when bonds are sold before maturity. For eg., if ONDO is ever forced to sell underlying bonds at market price and the bond's price is lower than the time they were purchased, that will cause a loss that will most likely mean a negative rebase on the price of `USDY` or `oUSG`.

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.

The consequences of a negative rebase (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);

        // Calculate Withdraw asset value
      uint256 withdrawAssetValue = iSTBL_LT1_AssetOracle(AssetData.oracle)
            .fetchInversePrice(
                ((MetaData.stableValueNet + MetaData.haircutAmount) -
                    MetaData.withdrawfeeAmount)
            ); //@audit gets the latest price -> if price is negative withdrawAssetValue can be greater
            ↪ than assetDepositNet

        ...


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

        ...
    }
```

Impact: Potential denial of service on user withdrawals in the even of negative rebasing of asset prices .

Proof of Concept: Add the following function to `STBL_PT1_TestOracle`

```
    function decreasePriceByPercentage(uint256 basisPoints) external {
        require(basisPoints <= 9999, "Cannot decrease more than 99.99%");
        price = (price * (10000 - basisPoints)) / 10000;
    }
```

Then add the following test to `STBL_Test.sol`:

```
   /// @notice Test deposit -> large price decrease -> attempted yield distribution -> withdrawal
    /// @dev This tests accounting integrity under negative price movements and potential underflow
    ↪ risks
    function test_DepositWithdraw1PctPriceDecrease() public {
        console.log("=== 1% PRICE DECREASE TEST ===");

        // Get contracts
        STBL_PT1_TestToken assetToken = STBL_PT1_TestToken(getAssetToken(1));
        STBL_PT1_Issuer stblPT1Issuer = STBL_PT1_Issuer(getAssetIssuer(1));
        STBL_PT1_Vault stblPT1Vault = STBL_PT1_Vault(getAssetVault(1));
        STBL_PT1_TestOracle stblPT1TestOracle = STBL_PT1_TestOracle(getAssetOracle(1));
        STBL_PT1_YieldDistributor yieldDist = STBL_PT1_YieldDistributor(getAssetYieldDistributor(1));

        // === PHASE 0: ENSURE VAULT LIQUIDITY ===
        // Add significant extra tokens directly to vault to ensure liquidity
        // This prevents withdrawal failures due to insufficient vault balance after price drop
        console.log("--- PHASE 0: ENSURE VAULT LIQUIDITY ---");
        uint256 extraLiquidity = 100000e6; // 100,000 extra tokens
        vm.startPrank(admin);
        assetToken.mintVal(address(stblPT1Vault), extraLiquidity);
        vm.stopPrank();

        // Setup treasury
        vm.startPrank(admin);
        registry.setTreasury(admin);
        vm.stopPrank();
```

```
// === PHASE 1: INITIAL DEPOSIT ===
console.log("--- PHASE 1: DEPOSIT ---");
uint256 depositAmount = 10000e6;

vm.startPrank(user1);
assetToken.approve(address(stblPT1Vault), depositAmount);
uint256 nftId = stblPT1Issuer.deposit(depositAmount);
vm.stopPrank();

// Log initial state
console.log("Initial deposit NFT ID:", nftId);
console.log("Initial oracle price:", stblPT1TestOracle.fetchPrice());
console.log("Initial vault token balance:", assetToken.balanceOf(address(stblPT1Vault)));
console.log("User USST balance:", usst.balanceOf(user1));

// Get initial metadata and vault state
YLD_Metadata memory initialMetadata = yld.getNFTData(nftId);
VaultStruct memory initialVaultData = stblPT1Vault.fetchVaultData();
console.log("Asset value:", initialMetadata.assetValue);
console.log("Stable value net:", initialMetadata.stableValueNet);
console.log("Initial vault asset deposit net:", initialVaultData.assetDepositNet);
console.log("Initial vault deposit value USD:", initialVaultData.depositValueUSD);

// === PHASE 2: ADVANCE TIME FOR YIELD ELIGIBILITY ===
console.log("\n--- PHASE 2: ADVANCE TIME FOR YIELD ELIGIBILITY ---");
vm.warp(block.timestamp + initialMetadata.Fees.yieldDuration + 1);
console.log("Advanced time by:", initialMetadata.Fees.yieldDuration + 1, "seconds");

// === PHASE 3: LARGE PRICE DECREASE ===
console.log("\n--- PHASE 3: LARGE PRICE DECREASE ---");

uint256 initialPrice = stblPT1TestOracle.fetchPrice();
console.log("Price before decrease:", initialPrice);

// Simulate 10% price decrease
stblPT1TestOracle.decreasePriceByPercentage(100);

uint256 newPrice = stblPT1TestOracle.fetchPrice();
console.log("Price after decrease:", newPrice);
console.log("Percentage change:", stblPT1TestOracle.calculatePercentageChange(initialPrice,
↪   newPrice));

// === PHASE 4: CHECK YIELD CALCULATION AFTER PRICE DECREASE ===
console.log("\n--- PHASE 4: YIELD CALCULATION AFTER PRICE DECREASE ---");

// Check vault state before potential yield distribution
VaultStruct memory preYieldVaultData = stblPT1Vault.fetchVaultData();
console.log("Vault asset deposit net (pre-yield attempt):", preYieldVaultData.assetDepositNet);
console.log("Vault deposit value USD (pre-yield attempt):", preYieldVaultData.depositValueUSD);

// Calculate price differential - should be 0 for price decrease
uint256 priceDifferential = stblPT1Vault.CalculatePriceDifferentiation();
console.log("Price differential calculated:", priceDifferential);

// Verify that no yield is distributed when price decreases
assertEq(priceDifferential, 0, "Price differential should be 0 when price decreases");

// Attempt yield distribution - should be a no-op
vm.startPrank(admin);
stblPT1Vault.distributeYield();
vm.stopPrank();

// Check vault state after yield distribution attempt
```

```
        VaultStruct memory postYieldVaultData = stblPT1Vault.fetchVaultData();
        console.log("Vault asset deposit net (post-yield attempt):",
        ↪   postYieldVaultData.assetDepositNet);
        console.log("Vault yield fees collected:", postYieldVaultData.yieldFees);

        // Verify no changes occurred during yield distribution
        assertEq(postYieldVaultData.assetDepositNet, preYieldVaultData.assetDepositNet, "assetDepositNet
        ↪   should not change");
        assertEq(postYieldVaultData.yieldFees, preYieldVaultData.yieldFees, "yieldFees should not
        ↪   change");

        // === PHASE 5: WITHDRAWAL UNDER DECREASED PRICE CONDITIONS ===
        console.log("\n--- PHASE 5: WITHDRAWAL UNDER DECREASED PRICE ---");

        // Check if user can still withdraw when asset price has decreased
        uint256 usstBalance = usst.balanceOf(user1);
        console.log("USST balance before withdrawal:", usstBalance);

        vm.startPrank(user1);
        usst.approve(address(usst), usstBalance);
        yld.setApprovalForAll(address(yld), true);

        uint256 preWithdrawVaultBalance = assetToken.balanceOf(address(vault));
        uint256 preWithdrawUserBalance = assetToken.balanceOf(user1);
        console.log("Vault balance before withdrawal:", preWithdrawVaultBalance);
        console.log("User balance before withdrawal:", preWithdrawUserBalance);

        // Attempt withdrawal
        vm.expectRevert();
        stblPT1Issuer.withdraw(nftId, user1);
        vm.stopPrank();

    }
```

Recommended Mitigation: Consider modifying `withdrawERC20` to prevent underflows by capping withdrawals to available accounting balance:

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

    uint256 withdrawAssetValue = iSTBL_PT1_AssetOracle(AssetData.oracle)
        .fetchInversePrice(
            ((MetaData.stableValueNet + MetaData.haircutAmount) - MetaData.withdrawfeeAmount)
        );

    uint256 withdrawFeeAssetValue = iSTBL_PT1_AssetOracle(AssetData.oracle)
        .fetchInversePrice(MetaData.withdrawfeeAmount);

    uint256 totalWithdrawal = withdrawAssetValue + withdrawFeeAssetValue;

    // @audit Cap withdrawal to available balance
    if (VaultData.assetDepositNet < totalWithdrawal) {
        uint256 availableWithdrawal = VaultData.assetDepositNet > withdrawFeeAssetValue
            ? VaultData.assetDepositNet - withdrawFeeAssetValue
            : 0;

        withdrawAssetValue = availableWithdrawal;
        VaultData.assetDepositNet = 0; //@audit effectively force this to 0

    } else {
        VaultData.assetDepositNet -= totalWithdrawal;
    }
    // Rest of function continues normally...
}
```

STBL: Fixed in commit c540943

Cyfrin: Verified. We note that in the unlikely event of negative rebasing, the current fix reverts when vault becomes insolvent - however, in such a scenario this fix allows early users to withdraw (before insolvency occurs) while it causes denial of service for later users attempting withdrawal.

7.2 Low Risk

7.2.1 Missing zero address checks in `STBL_Register::setupAsset`

Description: Several asset addresses are initialized without zero address checks in `STBL_Register::setupAsset` :

- `_contractAddr`
- `_issuanceAddr`
- `_distAddr`
- `_vaultAddr`
- `_oracle`

Notably, the same variables when set via their respective setter functions, eg,. `setOracle` have these validations. It is also important to note that an asset can only be setup once.

Recommended Mitigation: Consider adding zero address checks in the `setupAsset`

STBL: Fixed in commit a737746

Cyfrin: Verified.

7.2.2 Insufficient fee validation in `STBL_Register::setupAsset` can cause underflow

Description: The `STBL_Register::setupAsset` and `STBL_Register::setFees` functions validate individual fees but ignore their cumulative impact. Consider the following logic in `STBL_MetadataLib.calculateDepositFees` where the `depositfeeAmount`, `haircutAmount` and `insurancefeeAmount` are calculated on the gross stable value:

```
function calculateDepositFees(YLD_Metadata memory data) internal pure returns (YLD_Metadata memory) {
    // All deposit-time fees calculated on same base (stableValueGross)
    data.depositfeeAmount = (data.stableValueGross * data.Fees.depositFee) / 10000;
    data.haircutAmount = (data.stableValueGross * data.Fees.hairCut) / 10000;
    data.insurancefeeAmount = (data.stableValueGross * data.Fees.insuranceFee) / 10000;
    data.withdrawfeeAmount = (data.stableValueGross * data.Fees.withdrawFee) / 10000;
    return data; //@audit depositfeeAmount, haircutAmount and insurancefeeAmount are calculated on
    ↪  stableValueGross
}
```

Both the `STBL_LT1_Issuer` and `STBL_PT1_Issuer` contracts calculate the net stable value as follows:

```
   MetaData.stableValueNet = (MetaData.stableValueGross -
            (MetaData.depositfeeAmount +
                MetaData.haircutAmount +
                MetaData.insurancefeeAmount));
```

This would mean that if the sum of all fees in basis points exceeds 10000, the metadata logic will always revert when computing the net stable value.

Impact: A combination of fees where the sum of deposit fee, hair cut and insurance fee exceeds 10000 can cause underflow in net stable value calculation.

Recommended Mitigation: Consider introducing a cumulative check in the `STBL_Register::setupAsset` and `STBL_Register::setFees` functions.

STBL: Fixed in commit 1adc1f2

Cyfrin: Verified.

7.2.3 Insufficient duration validation in `STBL_Register::setupAsset` can lock user withdrawals

Description: Lack of duration validation in `STBL_Register::setupAsset` and `STBL_Register::setDurations` allows asset configurations where `yieldDuration > duration`, creating mathematically impossible withdrawal conditions that lock user funds.

The withdrawal logic requires users to wait at least `yieldDuration` while also withdrawing before duration expires. When `yieldDuration > duration`, no valid time window exists for user withdrawals.

The `iWithdraw` function in `STBL_LT1_Issuer` has the following checks

```
function iWithdraw(uint256 _tokenID, address _sender) internal isSetupDone {
    YLD_Metadata memory MetaData = iSTBL_YLD(registry.fetchYLDToken()).getNFTData(_tokenID);
    // code..

    //ensures that users can't withdraw after duration has passed
        if ((MetaData.depositBlock + MetaData.Fees.duration) < block.timestamp)
            revert STBL_Asset_WithdrawDurationNotReached(assetID, _tokenID);

        //ensures that users must wait for yield duration to withdraw assets
        if (
            (MetaData.depositBlock + MetaData.Fees.yieldDuration) >
            block.timestamp
        ) revert STBL_Asset_YieldDurationNotReached(assetID, _tokenID);

    // @audit withdrawal proceeds only if BOTH conditions are false...
   //@audit when yieldDuration > duration, no withdrawal window exists for users
}
```

Impact: Incorrectly configured asset durations can prevent user withdrawals.

Recommended Mitigation: Consider adding duration relationship validation in `setupAsset` and `setDurations`

STBL: Fixed in commit c540943

Cyfrin: Verified.

7.2.4 Disabling an asset via `STBL_Register::disableAsset` with active deposits can DOS user withdrawals

Description: The `STBL_Register::disableAsset` does not check for active deposits before disabling an asset, creating a Denial of Service (DOS) condition where users cannot withdraw their deposited funds.

`STBL_Register::disableAsset` only checks if the asset is currently enabled but ignores whether there are active user deposits:

```
// STBL_Register.sol - disableAsset()
function disableAsset(uint256 _id) external onlyRole(REGISTER_ROLE) {
    if (assetData[_id].status != AssetStatus.ENABLED)
        revert STBL_AssetNotActive();
    assetData[_id].status = AssetStatus.DISABLED;  // @audit does not check for active deposits
    emit AssetStateUpdateEvent(_id, true);
}
```

However, the withdrawal flow requires the asset to be in ENABLED status to complete successfully. The `STBL_Core::exit` function uses the `isValidIssuer` modifier which checks asset status:

```
// STBL_Core.sol
modifier isValidIssuer(uint256 _assetID) {
    AssetDefinition memory AssetData = registry.fetchAssetData(_assetID);
    if (!AssetData.isIssuer(_msgSender())) revert STBL_UnauthorizedIssuer();
    if (!AssetData.isActive()) revert STBL_AssetDisabled(_assetID);  // @audit blocks exit on disabled
    ↪  assets
    _;
}
```

```
function exit(uint256 _assetID, address _from, uint256 _tokenID, uint256 _value)
    external isValidIssuer(_assetID) {  // @audit Modifier prevents exit on disabled assets
    // ... burn tokens and decrement deposits
}
```

Impact: Disabled asset with active deposits prevents user withdrawals.

Recommended Mitigation: Consider preventing disabling of assets with active deposits.

STBL: Acknowledged.

Cyfrin: Acknowledged.

7.2.5 `STBL_Register::addAsset` does not check for non-empty asset name

Description: `STBL_Register::setupAsset` allows an assets to be created with empty names. However, elsewhere in the code, the `STBL_AssetDefinitionLib::isValid` function marks an an asset with empty name field as invalid. This creates an inconsistency where assets can be successfully created and enabled in the protocol but would fail validation checks via the `isValid` function.

```
// STBL_Register.sol - addAsset function
function addAsset(
    string memory _name,    // @audit No validation on name field
    string memory _desc,
    uint8 _type,
    bool _aggType
) external onlyRole(REGISTER_ROLE) returns (uint256) {
    unchecked {
        assetCtr += 1;
    }
    assetData[assetCtr].id = assetCtr;
    assetData[assetCtr].name = _name;        // @audit Can be empty string
    assetData[assetCtr].description = _desc;
    assetData[assetCtr].contractType = _type;
    assetData[assetCtr].isAggreagated = _aggType;
    assetData[assetCtr].status = AssetStatus.INITIALIZED;

    emit AddAssetEvent(assetCtr, assetData[assetCtr]);
    return assetCtr;
}
```

`isValid` function marks this asset invalid:

```
// STBL_AssetDefinitionLib.sol
function isValid(AssetDefinition memory asset) internal pure returns (bool) {
    return
        asset.id != 0 &&
        bytes(asset.name).length > 0 &&  //  Requires non-empty name
        asset.token != address(0) &&
        asset.issuer != address(0) &&
        asset.rewardDistributor != address(0) &&
        asset.vault != address(0);
}
```

The `isValid()` function is currently unused in the main protocol contracts but is imported in test files, suggesting it was intended for validation but never properly integrated into the protocol flow.

Impact: Inconsistent validation logic and potential off-chain integration issues.

Recommended Mitigation: Consider adding name validation to `addAsset` or remove name requirement from `isValid`

STBL: Fixed in commit 4a187a5

Cyfrin: Verified.

7.2.6 Race condition at duration expiration boundary during withdrawals

Description: The `STBL_LT1_Issuer` contract contains inconsistent boundary condition logic in the withdrawal validation checks that creates a race condition when `block.timestamp == depositBlock + duration`.

In the `iWithdraw()` function (user withdrawal):

```
if ((MetaData.depositBlock + MetaData.Fees.duration) < block.timestamp)
    revert STBL_Asset_WithdrawDurationNotReached(assetID, _tokenID);
```

In the `withdrawExpired()` function (protocol withdrawal):

```
if ((MetaData.depositBlock + MetaData.Fees.duration) > block.timestamp)
    revert STBL_Asset_WithdrawDurationNotReached(assetID, _tokenID);
```

When `depositBlock + duration == block.timestamp`, both user and protocol can withdraw. This creates a race condition where whichever transaction is mined first will succeed and the other reverts.

Impact: Users may have their valid withdrawal transactions fail unexpectedly due to protocol intervention at the exact expiration moment

Proof of Concept: TBD.

Recommended Mitigation: Consider modifying boundary condition to give users clear priority at the expiration timestamp.

```
// withdrawExpired()
if ((MetaData.depositBlock + MetaData.Fees.duration) >= block.timestamp) //@audit modify > to >=
    revert STBL_Asset_WithdrawDurationNotReached(assetID, _tokenID);
```

STBL: Fixed.

Cyfrin: Verified.

7.2.7 Treasury cannot withdraw expired assets if NFT is disabled

Description: The `STBL_LT1_Issuer::withdrawExpired` function attempts to claim rewards for disabled NFTs, but the `STBL_LT1_YieldDistributor::claim` function explicitly reverts when called on disabled NFTs.

In `STBL_LT1_Issuer.withdrawExpired`:

```
// If NFT is disabled then claim for yield is not done
if (MetaData.isDisabled) {
    iSTBL_LT1_AssetYieldDistributor(AssetData.rewardDistributor).claim(_tokenID);
}
```

In `STBL_LT1_YieldDistributor.claim`:

```
if (MetaData.isDisabled) revert STBL_YLDDisabled(id);
```

Impact: `withdrawExpired` will cause denial of service when metadata is disabled

Recommended Mitigation: Consider modifying the existing `claim()` function to allow issuer calls on disabled NFTs:

```
function claim(uint256 id) external returns (uint256) {
  // current logic

   // @audit Allow issuer to claim for disabled NFTs, but block regular users
    if (MetaData.isDisabled && msg.sender != AssetData.issuer) {
        revert STBL_YLDDisabled(id);
    }
}
```

STBL: Fixed in commit 1adc1f2

Cyfrin: Verified.

7.3 Informational

7.3.1 Misleading `ContractType` field suggests that assets can be non ERC20 tokens

Description: The `AssetDefinition` struct includes a `contractType` field that suggests support for ERC20, ERC721, and Custom asset types, but the entire codebase implicitly assumes all assets are ERC20 tokens. This creates misleading documentation and potential integration issues.

The `AssetDefinition` struct defines a `contractType` field with multiple options:

```
// STBL_Structs.sol
struct AssetDefinition {
    // ...
    /** @notice Type of contract (ERC20 = 0, ERC721 = 1, Custom = 2) */
    uint8 contractType;
    // ...
}
```

However, throughout the codebase, all asset token interactions use the IERC20 interface, implicitly assuming the `contractType` is ERC20.

Impact: Misleading documentation and potential confusion while setting up assets.

Recommended Mitigation: Consider removing the `contractType` field as it is inconsistent with the asset handling in the codebase that implicitly assumes ERC20 type.

STBL: Acknowledged. This is kept in place for future use as there can be protocols which support erc 721 or erc1155 based debt instrument.

Cyfrin: Acknowledged.

7.3.2 Missing event emissions for critical oracle parameter changes

Description: The oracle contracts (`STBL_PT1_Oracle` and `STBL_LT1_Oracle`) do not emit events when critical parameters are modified by admin functions.

Following admin functions modify critical parameters but do not emit events:

- `setPriceDecimals`
- `setPriceThreshold`
- `enableOracle`
- `disableOracle`

Recommended Mitigation: Consider adding event declarations for the above functions.

STBL: Fixed in commit c540943

Cyfrin: Verified.

7.3.3 Mismatching variable naming for `Metadata.depositBlock`

Description: Based on the inline doc of ‘Metadata.depositBlock‘ implies it saves the block number when the deposit was created, but in reality, this variable saves the timestamp.

Recommended Mitigation: Consider renaming the variable `Metadata.depositBlock` to a more accurate name, i.e., `Metadata.depositTimestamp`

STBL: Fixed in commit c540943.

Cyfrin: Verified.

7.3.4 CEI pattern violation in `STBL_PT1_Vault::withdrawFees`

Description: The `STBL_PT1_Vault::withdrawFees` function violates the Checks-Effects-Interactions (CEI) pattern by performing external token transfers before resetting the fees to 0.

The function resets fee counters (`VaultData.depositFees`, `VaultData.withdrawFees`, etc.) to zero after executing the `safeTransfer()` call to the treasury. It is notable that this function has no access control.

Recommended Mitigation: Consider implementing a proper CEI pattern. Emit the event, reset counters and then make the transfer.

STBL: Acknowledged.

Cyfrin: Acknowledged.

7.3.5 Misleading inline comment in `STBL_LT1_Issuer::withdrawExpired` function

Description: The `STBL_LT1_Issuer::withdrawExpired` function contains a misleading inline comment that directly contradicts the actual code logic and intended functionality.

```
function withdrawExpired(uint256 _tokenID) external isSetupDone {
    // ... validation checks ...


>    //ensures that protocol can't withdraw after duration has passed //@audit incorrect comment
    if ((MetaData.depositBlock + MetaData.Fees.duration) > block.timestamp)
        revert STBL_Asset_WithdrawDurationNotReached(assetID, _tokenID);

    // ... withdrawal logic ...
}
```

The comment states: "ensures that protocol can't withdraw after duration has passed". However, the actual code logic does the exact opposite.

Also, the revert in the `iWithdraw` function is misleading:

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

Recommended Mitigation: Consider updating the inline comment to accurately reflect the code logic. Also consider replacing the error `STBL_Asset_WithdrawDurationNotReached` in `iWithdraw` to `STBL_Asset_Withdraw-DurationExpired`.

STBL: Fixed.

Cyfrin: Fixed.

7.3.6 Anybody can deposit and withdraw on behalf of users

Description: Issuer contracts allow anybody to call deposits and withdrawals on behalf of any account, without restrictions.

While no way was found to leverage this to perform a more harmful attack, it is still not ideal that anybody can operate on behalf of other users without their approval.

While the approval of the underlying collateral is required for deposits, it is not uncommon for users to grant infinite allowance through wallet UI's. So anybody could re-deposit assets once the user withdraws them or as soon as the user gets any of those assets in their balance.

For withdrawals, users' assets could be withdrawn before the asset's owner meant to withdraw them.

Recommended Mitigation: Consider whitelisting `operators` who can call on behalf of depositor/withdrawer instead of allowing anyone.

STBL Acknowledged. This is by design to facilitate integration by other protocols.

Cyfrin: Acknowledged.

7.4 Gas Optimization

7.4.1 Unnecessary usage of `_msgSender()` to validate if caller is the `Issuer` on the `STBL_PT1_YieldDistributor`

Description: On the `STBL_PT1_YieldDistributor` contract, the functions `enableStaking()` and `disableStaking()` use the modifier `isIssuer()` to validate if the caller is the authorized issuer of the `assetId` configured for the YieldDistributor.

`STBL_PT1_YieldDistributor.isIssuer()` calls the internal `_msgSender()` which calls `ERC2771ContextUpgradeable._msgSender()` and eventually will call `ContextUpgradeable._msgSender()`.

It is more optimal to use `msg.sender` than having the execution go through the `ERC2771ContextUpgradeable._msgSender()` because the `TrustedForwarder` won't be the `YieldDistributor`, the issuer makes a direct call to the `YieldDistributor`.

```
    modifier isIssuer() {
        AssetDefinition memory AssetData = registry.fetchAssetData(assetID);
@>      if (!AssetData.isIssuer(_msgSender()))
            revert STBL_Asset_InvalidIssuer(assetID);
        _;
    }


    function _msgSender()
        internal
        view
        override(ERC2771ContextUpgradeable)
        returns (address)
    {
@>      return ERC2771ContextUpgradeable._msgSender();
    }
```

The same applies for `distributeReward()`:

```
    function distributeReward(uint256 reward) external {
        ...
@>      if (!AssetData.isVault(_msgSender()))
            revert STBL_Asset_InvalidVault(assetID);
        ...
        IERC20(AssetData.token).safeTransferFrom(
@>          _msgSender(),
            address(this),
            reward
        );
        ...
    }
```

Recommended Mitigation: Consider using `msg.sender` instead of calling `_msgSender()`

STBL: Fixed in commit c540943

Cyfrin: Verified.