Syntetika Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

Jorge

August 1, 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

Syntetika provides an innovative "real yield" solution that bridges traditional Bitcoin holdings with DeFi and sophisticated trading strategies to generate sustainable "real yield". At its core, the protocol operates on a three-tier token system (`BTC/wBTC <-> hBTC <-> shBTC`) that enables users to earn yield on their Bitcoin without sacrificing liquidity or control.

The protocol begins with users depositing either native `BTC` or wrapped `BTC` through a compliance-checked minting process. Upon deposit, users receive `hBTC` tokens at a 1:1 ratio, representing their claim on the underlying Bitcoin. Users can also permissionlessly buy `hBTC` using decentralized exchanges.

The synthetic token `hBTC` maintains full 1:1 redeemability for native or wrapped `BTC`, while enabling participation in the protocol's yield generation mechanism. The deposited Bitcoin capital is then transferred to Hilbert Group, a professional proprietary trading firm, which deploys these funds across various trading strategies to generate profits.

To access yield opportunities, users can permissionlessly stake their `hBTC` tokens in the `StakingVault`, receiving `shBTC` (staked `hBTC`) shares in return. These vault shares represent proportional ownership in the yield-generating pool and appreciate in value as trading profits are distributed back to the vault. This mechanism creates a sustainable yield source directly tied to real trading performance rather than inflationary token emissions.

Security and compliance are paramount in the protocol's design. The system incorporates multiple layers of protection including KYC/KYB verification through Galactica's SBT-based compliance system, blacklist functionality, and carefully managed role-based access control. The `StakingVault` includes additional safeguards such as "first depositor attack" protection, cooldown periods for withdrawals (configurable up to 90 days) to deter "just in time" yield theft, minimum transaction requirements to prevent hackers manipulating the vault via 1 wei transactions, and restrictions on `shBTC` direct transfers between users. These measures protect both the protocol and its users while maintaining regulatory compliance.

The economic model creates aligned incentives where Hilbert Group's trading performance directly benefits `shBTC` holders. As long as the trading strategies remain profitable, `shBTC` continuously appreciates against both `hBTC` and native Bitcoin, providing holders with real, sustainable yield derived from actual trading profits rather than token inflation or unsustainable farming rewards. This design positions the protocol as a bridge between traditional finance expertise and decentralized finance accessibility, offering Bitcoin holders a way to earn yield while maintaining exposure to their preferred store of value asset.

5 Audit Scope

The scope of the audit was limited to:

```
deposit-registry/contracts/ComplianceChecker.sol
deposit-registry/contracts/CompliantDepositRegistry.sol
issuance/src/helpers/Blacklistable.sol
issuance/src/helpers/TokensHolder.sol
issuance/src/helpers/Whitelist.sol
issuance/src/minter/Minter.sol
issuance/src/token/HilBTC.sol
issuance/src/vault/StakingVault.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Syntetika smart contracts provided by Syntetika Labs. In this period, a total of 34 issues were found.

The findings consist of 1 High, 1 Medium and 7 Low severity issues with the remainder being informational and gas optimizations.

- Single High 7.1.1 related to incorrect exchanges in the `Minter` contract between `baseAsset` and `hilBTCToken` (`hBTC`) that could occur from a depeg of `baseAsset` or other factors, allowing users to mint more `hBTC` than they should receive and use that to drain `x/hBTC` liquidity pools
- Single Medium 7.2.1 was a high-impact but low-probability edge case that resulted in unvested yield becoming permanently stuck inside `StakingVault`
- 7 Lows were a mix of issues mostly related to inconsistent applications of blacklisting, whitelisting and compliance functionality

As part of the audit we wrote an invariant fuzz testing suite focused on the solvency of `StakingVault`; this was provided to the client as an additional deliverable which the client added to their repository.

We also:

- made numerous recommendations for increasing the defensiveness of `StakingVault` by reverting for transaction input/output patterns which have been used in previous vault exploits but which no legitimate user would perform; all recommendations were implemented
- provided many safe gas optimizations especially related to eliminating identical storage reads; all recommendations were implemented

Negative Yield Risk Factor

If the off-chain trading strategy incurs significant sustained losses, then the 1:1 redemption peg will break in both the DEX pool and the client's platform, as the client won't be able to continue supplying sufficient `BTC`, `wBTC` (or other assets used to exit) which enables `hBTC` holders to redeem out at 1:1.

The protocol is able to socialize trading losses to `StakingVault` stakers by calling `HiiBTC::burnFrom(address(stakingVault), amountOfLosses)` and distribute trading profits to stakers by calling `StakingVault::distributeYield`. In networks with public mempools ideally both of these calls would be done through services which prevent front-running.

The success of the protocol in large part depends on the ability of the off-chain trading strategy to generate profits or at least avoid significant sustained losses.

Summary

Project Name	Syntetika
Repository	monorepo
Commit	337a4e9aa8ce...
Fix Commit	9fccd3b18f15...
Audit Timeline	July 28th - Aug 1st, 2025
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	1
Low Risk	7
Informational	19
Gas Optimizations	6
Total Issues	34

Summary of Findings

[H-1] `Minter` doesn't account for depegs, exchange rates and decimal precision mismatch between `baseAsset` and `hilBTCToken`, which can result in total loss scenarios for both users and protocol	Resolved
[M-1] Yield tokens can be permanently locked in the vault when all stakers withdraw during vesting window	Resolved
[L-1] Blacklisted users can claim withdrawn assets after the cooldown period	Resolved
[L-2] Non-compliant users can claim withdrawn assets after the cooldown period	Resolved
[L-3] Missing check if `receiver` is whitelisted in `StakingVault::mint, deposit`	Resolved
[L-4] Inability for users to permissionlessly stake and earn yield	Resolved
[L-5] Unbounded `depositAddresses` can cause `CompliantDepositRegistry::challengeLatestBatch` to revert due to out of gas	Resolved
[L-6] Malicious user holding `HilBTC` tokens can front run blacklisted transaction	Acknowledged
[L-7] Owner can not burn tokens from blacklisted addresses	Resolved
[I-01] Use named mapping parameters to explicitly note the purpose of keys and values	Resolved
[I-02] Don't initialize to default values in Solidity	Resolved
[I-03] Prefer explicit `uint` sizes	Resolved

[I-04] Use named imports	Resolved
[I-05] Consider using `SafeCast` when downcasting amounts	Resolved
[I-06] `StakingVault::claimWithdraw` should revert if `assets` are zero	Resolved
[I-07] Remove obsolete `return` statements when using named return variables	Resolved
[I-08] Emit missing event information	Resolved
[I-09] Missing call to `_setGlobalWhitelist` in `Minter.sol`	Resolved
[I-10] Enforce minimum transaction amounts in `StakingVault`	Resolved
[I-11] Missing `redeem` convenience function in the `StakingVault.sol`	Resolved
[I-12] `Minter::ownerMint` bypasses whitelist requirement and increases `totalDeposits` without actually transferring any tokens	Acknowledged
[I-13] Revert if `StakingVault::deposit, mint, redeem, withdraw` would return zero	Resolved
[I-14] Not check for timestamp in `distributeYield` could DoS the distribute rewards	Acknowledged
[I-15] `ComplianceChecker::isCompliant` incorrectly returns `true` if compliance options have no required soul bound tokens	Acknowledged
[I-16] Consider using a staking rewards distributor to efficiently space out staking rewards, further deterring just-in-time attacks	Acknowledged
[I-17] Enforce that `StakingVault::decimals` is greater or equal to the underlying asset decimals	Resolved
[I-18] `StakingVault::distributeYield` should revert when there are no vault shares	Resolved
[I-19] Roles not set in `deposit-registry` contract constructors	Resolved
[G-1] Cache storage to prevent identical storage reads	Resolved
[G-2] Variables only set once in `constructor` of non-upgradeable contracts should be declared `immutable`	Resolved
[G-3] Use named return variables when this eliminates local variables	Resolved
[G-4] Use `ReentrancyGuardTransient` for faster `nonReentrant` modifiers	Resolved
[G-5] Remove `from` parameter from `Minter:redeem` and `_onlySender` function	Resolved
[G-6] Small functions only used once should be inlined into their parent functions	Acknowledged

7 Findings

7.1 High Risk

7.1.1 `Minter` doesn't account for depegs, exchange rates and decimal precision mismatch between `baseAsset` and `hilBTCToken`, which can result in total loss scenarios for both users and protocol

Description: `Minter` supports conversion between two key assets, `baseAsset` and `hilBTCToken`:

```
/// @notice The token address of the base asset accepted for deposit (e.g., WBTC or a stablecoin)
IERC20 public immutable baseAsset;

/// @notice The token address of the HilBTC ERC-20 contract (which the Minter will mint/burn)
IMintableERC20 public immutable hilBTCToken;
```

- `baseAsset` explicitly can be `wBTC`, `USDC` or `USDT`; these can have various precision such as 8, 6 or 18 depending on the chain
- `hilBTCToken` is explicitly always the protocol's synthetic bitcoin token `hBTC` having 8 decimal precision
- `Minter::mint` allows users to supply input `amount` tokens in `baseAsset` and receive output `amount` tokens in `hilBTCToken` with no adjustment for different decimal precision, exchanges rates or depeg events:

```
function mint(
    address to,
    uint256 amount
) external onlyWhitelisted(msg.sender) onlyWhitelisted(to) nonReentrant {
    baseAsset.safeTransferFrom(msg.sender, address(this), amount);
    hilBTCToken.mint(to, amount);
    totalDeposits += amount;
    emit Minted(to, amount);
}
```

Impact: There are multiple negative scenarios that can arise, but the most significant examples are:

1) `baseAsset` represents a wrapped form of bitcoin which uses 8 decimal places (eg wBTC) but has currently depegged and is not worth anywhere near the actual bitcoin price:

- user buys wBTC very cheap from a decentralized exchange due to the depeg
- user calls `Minter::mint` passing `amount = 1e8` (normally worth 1 BTC but now worth far less due to the depeg)
- user receives 1e8 worth of `hBTC` where 1 BTC is 1e8
- user can then drain x/hBTC liquidity pools since they were credited 1 BTC worth of hBTC even though they didn't provide 1 BTC worth of wBTC since wBTC has depegged
- in the kick-off call notes it states that hBTC can always be redeemed for BTC at a 1:1 ratio, so this may also be another way to drain reserves though this likely involves off-chain components
- user could also stake the hBTC to earn more yield than they should though this is less immediately impactful

2) `baseAsset` represents a stablecoin such as USDC:

- user deposits 1e6 ($1)
- user receives 1e6 worth of `hBTC` which is currently worth around $1,180
- user can then drain x/hBTC liquidity pools and other similar scenarios as 1) above

Recommended Mitigation: `Minter` in its current form can only be safely used with wrapped BTC representations that use 8 decimal places - the first option is to enforce this is the case in the `constructor` and remove the comment stating that `baseAsset` can be multiple different assets.

However as noted `baseAsset` is intended to be many other different assets eg:

```
    /// @notice The token address of the base asset accepted for deposit (e.g., WBTC or a stablecoin)
    IERC20 public immutable baseAsset;
```

To support different `baseAsset` as the code currently intends, the `mint` and `redeem` functions will need to account for:

- differences in decimal precision between `baseAsset` and `hilBTCToken`
- exchange rates between `baseAsset` and `hilBTCToken`
- alternatively rename `hilBTCToken` to `hilSyntheticToken` and ensure that the synthetic token is always the equivalent of `baseAsset`

The code also needs to handle depeg events where the `baseAsset` even if `wBTC` can depeg and be worth far less than actual bitcoin, but `hBTC` is always redeemable 1:1 for native `BTC`. So if a depeg has occurred minting or redeem should revert. The ideal way to implement this check is via Chainlink price feeds, reverting if a depeg has occurred.

If Chainlink is not available on specific chains a secondary option could be Uniswap V3 TWAP. A third option could be making the `Minter` contract pausable and having an off-chain bot monitor `baseAsset` for depegs then pause the contract should a depeg occur - but this introduces additional risk related to the offchain bot not functioning correctly.

Syntetika: In commits a70110c, db8c139 we:

- changed the comments and renamed `hilBTCToken` to `hilSyntheticToken` so it is more generic to support different synthetic token types for different instances of `Minter`
- implemented a decimal matching check inside the constructor such that `baseAsset` and `hilSyntheticToken` must use the same decimal precision

The intention was that the `Minter` contract shouldn't support stablecoin -> `hBTC` exchanges, but that in each instance of the `Minter` contract the `baseAsset` would be paired with a matching synthetic token.

In commits 20045e0, 2e308e8 we added pausing functionality to core `Minter` functions. An off-chain bot will monitor the `baseAsset` of each `Minter` instance and pause that instance if it detects a depeg.

Cyfrin: Verified; we note that the introduction of the off-chain bot responsible for monitoring `baseAsset` for depegs and pausing the correct `Minter` instance introduces an additional risk if this process malfunctions.

7.2 Medium Risk

7.2.1 Yield tokens can be permanently locked in the vault when all stakers withdraw during vesting window

Description: Yield tokens can be permanently locked in the vault when all stakers withdraw during vesting window.

Proof of Concept: Add test to `StakingVaultTest.t.sol`:

```
function test_allStakersWithdrawDuringVestingWindow_LockedTokensInVault() public {
    // Start fresh - check initial state
    assertEq(vault.totalSupply(), 100 ether); // Owner already has 100 from setup

    // Setup: 2 more stakers
    address staker1 = address(0x1);
    address staker2 = address(0x2);

    uint256 stakeAmount = 100 ether;

    // Setup stakers
    vm.startPrank(owner);
    sbtContract.setVerified(staker1, true);
    sbtContract.setVerified(staker2, true);
    asset.mint(staker1, stakeAmount);
    asset.mint(staker2, stakeAmount);
    vm.stopPrank();

    // Each stakes 100 HILBTC
    vm.prank(staker1);
    asset.approve(address(vault), stakeAmount);
    vm.prank(staker1);
    vault.stake(stakeAmount);

    vm.prank(staker2);
    asset.approve(address(vault), stakeAmount);
    vm.prank(staker2);
    vault.stake(stakeAmount);

    // Vault now has 300 HILBTC total (100 owner + 200 from stakers), 300 shares
    assertEq(vault.totalSupply(), 300 ether);
    assertEq(asset.balanceOf(address(vault)), 300 ether);

    // Owner distributes 30 HILBTC yield
    vm.startPrank(owner);
    asset.approve(address(vault), 30 ether);
    vault.distributeYield(30 ether, block.timestamp);
    vm.stopPrank();

    // After 4 hours (50% vested)
    vm.warp(block.timestamp + 4 hours);

    console.log("Before withdrawals:");
    console.log("Vault balance:", asset.balanceOf(address(vault)));
    console.log("Unvested:", vault.getUnvestedAmount());
    console.log("Total assets:", vault.totalAssets());
    console.log("Total shares:", vault.totalSupply());

    // Owner withdraws their 100 shares
    vm.prank(owner);
    vault.redeem(100 ether, owner, owner);

    console.log("\nAfter owner withdrawal:");
    console.log("Vault balance:", asset.balanceOf(address(vault)));
    console.log("Unvested:", vault.getUnvestedAmount());
    console.log("Total assets:", vault.totalAssets());
    console.log("Total shares:", vault.totalSupply());

    // Check if remaining stakers can withdraw
    uint256 staker1Shares = vault.balanceOf(staker1);
    uint256 maxRedeem1 = vault.maxRedeem(staker1);
    console.log("\nStaker1 shares:", staker1Shares);
    console.log("Max redeemable:", maxRedeem1);

    // Try withdrawal
    vm.prank(staker1);
    vault.redeem(staker1Shares, staker1, staker1);

    console.log("\nAfter staker1 withdrawal:");
    console.log("Vault balance:", asset.balanceOf(address(vault)));
    console.log("Total assets:", vault.totalAssets());
    console.log("Total shares:", vault.totalSupply());

    // Can staker2 still withdraw?
    uint256 staker2Shares = vault.balanceOf(staker2);
    uint256 maxRedeem2 = vault.maxRedeem(staker2);
    console.log("\nStaker2 shares:", staker2Shares);
    console.log("Max redeemable:", maxRedeem2);
    console.log("Would receive:", vault.previewRedeem(staker2Shares));

    // Final withdrawal
    vm.prank(staker2);
    vault.redeem(staker2Shares, staker2, staker2);

    console.log("\nFinal state:");
    console.log("Vault balance:", asset.balanceOf(address(vault)));
    console.log("Total shares:", vault.totalSupply());

    // Wait for vesting to complete
    vm.warp(block.timestamp + 4 hours); // Now 8 hours total

    console.log("\nAfter vesting completes:");
    console.log("Vault balance:", asset.balanceOf(address(vault)));
    console.log("Unvested:", vault.getUnvestedAmount());

    // 15000000000000000001 still locked in the vault
    console.log("Total assets:", vault.totalAssets());
    // 0 shares
    console.log("Total supply:", vault.totalSupply());

    // Owner deposits 100 HILBTC
    vm.startPrank(owner);
    asset.approve(address(vault), 100 ether);
    uint256 sharesReceived = vault.deposit(100 ether, owner);
    vm.stopPrank();

    console.log("\nAfter owner re-deposits 100 HILBTC:");
    console.log("Shares received:", sharesReceived);
    console.log("Vault balance:", asset.balanceOf(address(vault)));
    console.log("Total assets:", vault.totalAssets());
    console.log("Share price:", vault.totalAssets() * 1e18 / vault.totalSupply());

    // Owner withdraws all
    vm.prank(owner);
    uint256 withdrawn = vault.redeem(sharesReceived, owner, owner);

    console.log("\nOwner withdrew:", withdrawn);
    // 16428571428571428571 still locked in the vault greater than
```

```
    // 15000000000000000001 previously locked!
    console.log("Vault balance:", asset.balanceOf(address(vault)));
    // 0 shares
    console.log("Total shares:", vault.totalSupply());
}
```

Recommended Mitigation: Don't allow all shares to be withdrawn; a common technique is on the first deposit to mint 1000 shares to a burn address. Using this technique the owner doesn't need to make the first deposit either.

Syntetika: Fixed in commits 315fd62, 8c6deb1 - `StakingVault::deposit` now burns an amount of shares equivalent to the minimum allowed deposit amount (~$10) as the first deposit - the owner is effectively paying a $10 fee to secure the vault.

Cyfrin: Experimenting with the new code and modifying our PoC, we found the issue can still manifest even with the dead shares:

```
function test_vestingWithDeadShares() public {
    // Simulate the dead shares implementation
    uint256 DEAD_SHARES = 10000;

    // First deposit by owner triggers dead shares
    vm.startPrank(owner);
    asset.approve(address(vault), 100 ether);
    vault.stake(100 ether);

    // Simulate minting dead shares (in reality this would be in deposit function)
    uint256 actualOwnerShares = vault.balanceOf(owner);
    console.log("Owner shares received:", actualOwnerShares);
    console.log("Total supply (includes dead shares):", vault.totalSupply() + DEAD_SHARES);
    assertEq(actualOwnerShares + DEAD_SHARES, vault.totalSupply());
    vm.stopPrank();

    // Setup 2 more stakers
    address staker1 = address(0x1);
    address staker2 = address(0x2);

    uint256 stakeAmount = 100 ether;

    // Setup and stake
    vm.startPrank(owner);
    asset.mint(staker1, stakeAmount);
    asset.mint(staker2, stakeAmount);
    vm.stopPrank();

    vm.startPrank(staker1);
    asset.approve(address(vault), stakeAmount);
    vault.stake(stakeAmount);
    vm.stopPrank();

    vm.startPrank(staker2);
    asset.approve(address(vault), stakeAmount);
    vault.stake(stakeAmount);
    vm.stopPrank();

    // Current state: 300 HILBTC deposited, 300 shares (+ 1000 dead shares)
    uint256 totalUserShares = vault.totalSupply(); // This would be 1300 with dead shares
    console.log("\nBefore yield distribution:");
    console.log("Vault balance:", asset.balanceOf(address(vault)));
    console.log("Total user shares:", totalUserShares);

    // Owner distributes 30 HILBTC yield
    vm.startPrank(owner);
    asset.approve(address(vault), 30 ether);
```

```
vault.distributeYield(30 ether, block.timestamp);
vm.stopPrank();

// After 4 hours (50% vested)
vm.warp(block.timestamp + 4 hours);

console.log("\nDuring vesting (4 hours):");
console.log("Vault balance:", asset.balanceOf(address(vault)));
console.log("Unvested amount:", vault.getUnvestedAmount());
console.log("Total assets:", vault.totalAssets());

// All users withdraw during vesting
uint256 ownerShares = vault.balanceOf(owner);
vm.prank(owner);
vault.redeem(ownerShares, owner, owner);

uint256 staker1Shares = vault.balanceOf(staker1);
vm.prank(staker1);
vault.redeem(staker1Shares, staker1, staker1);

uint256 staker2Shares = vault.balanceOf(staker2);
vm.prank(staker2);
vault.redeem(staker2Shares, staker2, staker2);

console.log("\nAfter all users withdraw:");
console.log("Vault balance:", asset.balanceOf(address(vault)));
console.log("Unvested amount:", vault.getUnvestedAmount());
console.log("Total assets:", vault.totalAssets());
console.log("Remaining shares (should be just dead shares):", vault.totalSupply());

// Wait for vesting to complete
vm.warp(block.timestamp + 4 hours);

console.log("\nAfter vesting completes:");
// 15000000000000010501 tokens locked in vault
console.log("Vault balance:", asset.balanceOf(address(vault)));
console.log("Unvested amount:", vault.getUnvestedAmount());
console.log("Total assets:", vault.totalAssets());
// only 10000 dead shares remain
console.log("Dead shares remain:", vault.totalSupply());

// New user tries to deposit and recover locked funds
address newUser = address(0x3);
vm.startPrank(owner);
asset.mint(newUser, 100 ether);
vm.stopPrank();

vm.startPrank(newUser);
asset.approve(address(vault), 100 ether);
uint256 sharesReceived = vault.deposit(100 ether, newUser);

console.log("\nNew user deposits 100 HILBTC:");
// 66673
console.log("Shares received:", sharesReceived);
// 115000000000000010501
console.log("Vault balance:", asset.balanceOf(address(vault)));
console.log("Total assets:", vault.totalAssets());

// New user immediately withdraws to see how much they can recover
uint256 withdrawAmount = vault.redeem(sharesReceived, newUser, newUser);
console.log("\nNew user withdraws all shares:");
// user received 99999934788846293400 tokens
console.log("Amount received:", withdrawAmount);
```

```
    // user lost 65211153706600 tokens
    console.log("Loss:", 100 ether - withdrawAmount);
    // vault locked balance increased to 15000065211153717101
    console.log("Vault balance (still locked):", asset.balanceOf(address(vault)));
    vm.stopPrank();
}
```

Recommended Mitigation: * The simplest option is to remove the 8-hour vesting period and allow the yield to be collected when it is deposited into the contract. As long as there is a sufficiently long cooldown period (default 90 days), this deters "just in time" attacks where users deposit large amounts to collect most of the yield then immediately withdraw. The yield distribution transaction could also be run through particular services designed to prevent front-running to further protect against "just in time" attacks

- A more complicated option is to:

1) have the owner deposit in addition to the dead shares, such that any "locked" tokens effectively accrue to the owner's stake as well as the dead shares. In this case ensure that the owner's deposit is much greater than the dead shares. If the owner is the last one to withdraw, then don't distribute anymore yield afterwards
2) in `_withdraw`, check if this transaction would result in only the dead shares remaining
3) if true, then check if the unvested amount > 0
4) if true, reset the unvested amount and send the unvested amount to the contract owner (or another contract)

```
// After the operation, check if only dead shares remain
if (totalSupply() - shares == deadShares) {
    uint256 remainingUnvested = getUnvestedAmount();
    if(remainingUnvested > 0) {
        vestingAmount = 0;
        lastDistributionTimestamp = 0;
        IERC20(asset()).safeTransfer(owner(), remainingUnvested);
    }
}
```

Additionally consider implementing a "sunset" feature where:

- the admin can initiate the "sunset" process; this prevents new deposits and mints but allows users to redeem or withdraw. It also sets a `sunsetTimestamp` 6 months into the future
- once either only the `DEAD_SHARES` remain (all stakers have withdrawn) OR `block.timestamp > sunsetTimestamp`, the admin can call a special function that performs the "rescue" transferring all asset tokens to the admin

This provides a nice way to "sunset" the vault and collect any remaining tokens, while giving users plenty of time to withdraw and protecting them from a rugpull.

Syntetika: Fixed in commits 1625c09, 8113753, 347330a by:

- mint 1000 "dead shares" to the burn address with the first deposit
- the first deposit can only be done by the admin, who will do a significantly bigger (at least 10x) deposit than the dead shares
- the admin will keep this stake active such that any "lost" unvested amounts accrue to the admin and the dead shares
- implemented the "rescue" mechanism in `_withdraw` when all non-dead shares are burned while there is a positive unvested amount, the positive unvested amount gets sent to the owner

Cyfrin: Verified though ideally the transfer in `StakingVault::_withdraw` would use `safeTransfer` instead of `transfer`.

7.3 Low Risk

7.3.1 Blacklisted users can claim withdrawn assets after the cooldown period

Description: `StakingVault::_update` uses modifiers `notBlacklisted(from) notBlacklisted(to)` to prevent blacklisted users from performing most actions.

But `StakingVault::claimWithdraw` does not use the `notBlacklisted` modifier. Hence a user who has been blacklisted after they first withdrew/redeemed can still claim those assets once the cooldown period has expired.

Recommended Mitigation: `StakingVault::claimWithdraw` should have at least `notBlacklisted(msg.sender)` and possibly also `notBlacklisted(receiver)`, though the second one is less effective since the user can input an arbitrary address.

Syntetika: Fixed in commit d98afbf.

Cyfrin: Verified.

7.3.2 Non-compliant users can claim withdrawn assets after the cooldown period

Description: `StakingVault::redeem, withdraw` use the `onlyWhitelisted` modifier to verify that `msg.sender` is whitelisted or is compliant, as this modifier ends up calling `Whitelist::isAddressWhitelisted`:

```
function isAddressWhitelisted(address user) public view returns (bool) {
    if (manualWhitelist[user] || globalWhitelist) {
        return true;
    }

    return complianceChecker.isCompliant(user);
}
```

If a user was whitelisted or was compliant when they created the withdrawal/redemption, but was then removed from the whitelist or became non-compliant, they will still be able to call `StakingVault::claimWithdraw` to withdraw their assets after the cooldown period.

Recommended Mitigation: `StakingVault::claimWithdraw` should use modifier `onlyWhitelisted(msg.sender)` to ensure that the caller is still whitelisted or compliant; `onlyWhitelisted(receiver)` could also be used to enforce that the destination address is also whitelisted.

Syntetika: Fixed in commit 86384fe by removing the whitelist functionality entirely from `StakingVault` to resolve finding L-4.

Cyfrin: Verified.

7.3.3 Missing check if `receiver` is whitelisted in `StakingVault::mint, deposit`

Description: `StakingVault::mint, deposit` only validates that `msg.sender` is whitelisted but fails to check if the receiver parameter is whitelisted. Since non-whitelisted addresses cannot withdraw, redeem, or transfer shares, any shares minted to non-whitelisted receivers become permanently locked and unusable.

```
 function mint(
        uint256 shares,
        address receiver //@audit receiver could be not whitelisted?
    ) public override onlyWhitelisted(msg.sender) returns (uint256 assets) {
        ...
    }
```

Impact: Permanent loss of user funds or temporary if owner give whitelisted permissions.

Proof of Concept: Run the next proof of concept in `StakingVault.sol`:

```
function test_mint_non_whitelist_receiver() public {
        uint256 amount = 100 ether;
        vm.startPrank(user1);
        asset.approve(address(vault), amount);

        //create a non whitelisted receiver
        address bob = makeAddr("receiver");

        // 1. Alice (whitelisted) mints shares to Bob (non-whitelisted)
        vault.mint(1000 * 1e8, bob); // Success - only checks Alice is whitelisted

        vm.stopPrank();

        // 2. Bob tries to withdraw - REVERTS
        vm.prank(bob);
        vault.withdraw(1000 * 1e8, bob, bob); // Reverts: not whitelisted
        // Result: 1000 shares worth of HilBTC permanently locked
    }
```

Recommended Mitigation: Add a whitelist check for the receiver in the mint() function:

```
function mint(
    uint256 shares,
    address receiver
+ ) public override onlyWhitelisted(msg.sender) onlyWhitelisted(receiver) returns (uint256 assets) {
-   ) public override onlyWhitelisted(msg.sender) returns (uint256 assets) {
 ...
}
// Similar fix to `deposit`
```

Syntetika: Fixed in commit 86384fe by removing the whitelist functionality entirely from `StakingVault` to resolve finding L-4.

Cyfrin: Verified.

7.3.4 Inability for users to permissionlessly stake and earn yield

Description: The intention of the protocol as specified in the kick-off call and in discussion with the client is that users should be able to permissionlessly:

- buy hBTC from a decentralized exchange
- stake/unstake hBTC in a permissionless manner via `StakingVault`

Impact: In the current implementation `StakingVault` uses `onlyWhitelisted` modifiers on many core functions which prohibits users who permissionlessly bought hBTC using a decentralized exchange from subsequently staking their hBTC and earning yield.

To enable this the admin would need to call `setGlobalWhitelist` which would effectively disable the whitelist and compliance checks anyway.

Recommended Mitigation: Consider using only the "blacklist" functionality in `StakingVault` but removing the "whitelist" functionality to allow users to permissionlessly participate in staking and earning yield.

Syntetika: Fixed in commit 86384fe.

Cyfrin: Verified.

7.3.5 Unbounded `depositAddresses` can cause `CompliantDepositRegistry::challengeLatestBatch` to revert due to out of gas

Description: `CompliantDepositRegistry::challengeLatestBatch` contains an unbounded loop that removes deposit addresses from the latest batch by calling `depositAddresses.pop()` repeatedly. When a large batch of deposit addresses is added via `CompliantDepositRegistry::addDepositAddresses`, challenging this batch could consume excessive gas, potentially exceeding the block gas limit and causing the transaction to revert. This creates a Denial of Service (DoS) vulnerability where legitimate challenges cannot be executed.

```
function challengeLatestBatch() public onlyRole(CANCELER_ROLE) {
        require(latestBatchUnlockTime >= block.timestamp, NoChallengeAfterUnlock());

        uint256 _finalizedAddressesLength = finalizedAddressesLength;
        // Get rid of the challenged batch by removing it from the list
        uint256 batchLength = depositAddresses.length - _finalizedAddressesLength;
        for (uint256 i; i < batchLength; i++) {
            depositAddresses.pop();
        }<---------

        // Reset the challenge period to allow a new batch to be generated
        latestBatchUnlockTime = block.timestamp;

        emit BatchChallenged(_finalizedAddressesLength, block.timestamp, batchLength);
    }
```

Impact: Large batches become unchallengeable, allowing malicious or incorrect deposit addresses to be finalized.

Proof of Concept: Recommended Mitigation: Implement limits on the amount of addresses that can be added through `CompliantDepositRegistry::addDepositAddresses`.

Syntetika: Fixed in commit 319e7ea by changing `challengeLatestBatch` to allow cancelling in batches.

Cyfrin: Verified.

7.3.6 Malicious user holding `HilBTC` tokens can front run blacklisted transaction

Description: When the `blacklister` attempts to blacklist a malicious user holding `HilBTC` tokens, the maliciois user can monitor the memepool and front-run the blacklist tx by quickly redeeming their `HilBTC` tokens through the `minter` contract before the blacklist takes effect. Once redeemed, the user successfully exits their position with the underlying assets, completely bypassing the intended blacklist enforcement mechanism.

```
function redeem(
        uint256 amount
    ) external onlyWhitelisted(msg.sender) nonReentrant {
        hilBTCToken.burnFrom(msg.sender, amount);
        baseAsset.safeTransfer(msg.sender, amount);
        totalDeposits -= amount;
        emit Redeemed(msg.sender, amount);
    }
```

Note that this could not happen in the `StakingVault.sol`

Impact: Blacklist can be avoided if a malicious user front run the blacklisted call

Proof of Concept: Recommended Mitigation:

Consider add a cooldown also in the `redeem` function of the minter

Syntetika: Acknowledged; this is generally the case with all blacklisting mechanisms. Protocols can perform these transactions via services which prevent front-running.

We don't think it is worth adding extra complexity to the code to deal with this issue, we'll just use a service for running blacklist txns through if this is a real concern. In practice this "attack vector" is pretty much never an issue; I'm not aware of a single instance where this attack has occurred.

7.3.7 Owner can not burn tokens from blacklisted addresses

Description: The `HilBTC.sol` contract contains a logical inconsistency in its `burnFrom()` function regarding blacklisted addresses. While the function grants the `owner` special privileges to burn tokens from any user without requiring allowance (bypassing the `_spendAllowance` check), the underlying `_burn()` function calls `_update()` which contains the `notBlacklisted(from)` modifier. This prevents the owner from burning tokens from blacklisted addresses:

```
function burnFrom(address from, uint256 amount) external {
       address spender = msg.sender;
       if (spender != from && spender != owner()) {
           _spendAllowance(from, spender, amount);
       }
       _burn(from, amount);
   } //@audit owner can not burn from Blacklisted

   function _update(address from, address to, uint256 amount)
       internal
       override
       notBlacklisted(from)
       notBlacklisted(to)
   {
       super._update(from, to, amount);
   }
```

Impact: Owner can not burn tokens from blacklisted addresses.

Recommended Mitigation: If you want the owner to have the capability to burn blacklisted user tokens, consider creating a special access control function where the direct `_balances` of the backlisted user is reduced.

Syntetika: Fixed in commit dc14ad2.

Cyfrin: Verified.

7.4 Informational

7.4.1 Use named mapping parameters to explicitly note the purpose of keys and values

Description: Use named mapping parameters to explicitly note the purpose of keys and values:

- Issuance:

```
vault/StakingVault.sol
37:     mapping(address => UserCooldown) cooldowns;

helpers/Blacklistable.sol
8:     mapping(address => bool) internal _blacklisted;

helpers/Whitelist.sol
6:     /// @notice A mapping of specific user addresses that are allowed to bypass SBT checks
8:     mapping(address => bool) public manualWhitelist;
```

- Deposit-Registry:

```
CompliantDepositRegistry.sol
21:     mapping(address => uint) public investorDepositMap;
```

Syntetika: Fixed in commit 6f77988.

Cyfrin: Verified.

7.4.2 Don't initialize to default values in Solidity

Description: Don't initialize to default values in Solidity:

- Deposit-Registry:

```
ComplianceChecker.sol
44:         for (uint i = 0; i < complianceOptions.length; i++) {
58:             uint optionIndex = 0;
65:                 uint sbtIndex = 0;

CompliantDepositRegistry.sol
133:             uint i = 0;
157:         for (uint i = 0; i < newDepositAddresses.length; i++) {
200:         for (uint i = 0; i < batchLength; i++) {
```

Syntetika: Fixed in commit 7c69e94.

Cyfrin: Verified.

7.4.3 Prefer explicit `uint` sizes

Description: Prefer explicit `uint` sizes:

- Issuance

```
interfaces/minter/IBridgeMinter.sol
14:         uint id,
29:         uint id;

interfaces/galactica/IVerificationSBT.sol
45:         uint _expirationTime,
55:         uint _expirationTime

interfaces/token/IMintableERC20.sol
7:     function mint(address to, uint amount) external;
8:     function burn(address from, uint amount) external;
```

```
vault/StakingVault.sol
143:        uint assetsRedeemed = _redeemTo(shares, address(tokensHolder));
289:    ) internal returns (uint shares) {
306:    ) internal returns (uint assets) {
```

- Deposit-Registry:

```
interfaces/ICompliantDepositRegistry.sol
44:        uint indexed startIndex,
45:        uint challengePeriodEnd,
48:    event BatchChallengePeriodSet(uint newChallengePeriod);
50:        uint indexed startIndex,
51:        uint challengePeriodEnd,
52:        uint batchLength
100:        uint startIndex,
101:        uint count
116:    function setBatchChallengePeriod(uint newChallengePeriod) external;

ComplianceChecker.sol
44:        for (uint i = 0; i < complianceOptions.length; i++) {
58:            uint optionIndex = 0;
65:                uint sbtIndex = 0;

CompliantDepositRegistry.sol
27:    uint public nextDepositAddressIndex;
30:    uint public batchChallengePeriod;
32:    uint public latestBatchUnlockTime;
34:    uint public finalizedAddressesLength;
44:        // Block the first deposit address so that the default uint does not point to a valid address
124:        uint startIndex,
125:        uint count
127:        uint returnLength = count;
133:            uint i = 0;
156:        uint startIndex = depositAddresses.length;
157:        for (uint i = 0; i < newDepositAddresses.length; i++) {
174:    function _setBatchChallengePeriod(uint newChallengePeriod) internal {
184:        uint newChallengePeriod
199:        uint batchLength = depositAddresses.length - finalizedAddressesLength;
200:        for (uint i = 0; i < batchLength; i++) {
```

Syntetika: Fixed in commit cb00843.

Cyfrin: Verified.

7.4.4 Use named imports

Description: Use named imports; this is already being done in some places but not others:

- Issuance:

```
minter/Minter.sol
4:import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
5:import "@openzeppelin/contracts/access/AccessControl.sol";
6:import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";

vault/StakingVault.sol
5:import "../interfaces/vault/IStakingVault.sol";
7:import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

helpers/TokensHolder.sol
4:import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

```
token/HilBTC.sol
4:import "@openzeppelin/contracts/access/AccessControl.sol";
```

- `Deposit-Registry`:

```
interfaces/ICompliantDepositRegistry.sol
4:import "@openzeppelin/contracts/access/IAccessControl.sol";
5:import "./IComplianceChecker.sol";

interfaces/IComplianceChecker.sol
4:import "@openzeppelin/contracts/access/IAccessControl.sol";
5:import "@galactica-net/zk-certificates/contracts/interfaces/IVerificationSBT.sol";

ComplianceChecker.sol
4:import "@openzeppelin/contracts/access/AccessControl.sol";
5:import "./interfaces/IComplianceChecker.sol";

CompliantDepositRegistry.sol
4:import "@openzeppelin/contracts/access/AccessControl.sol";
5:import "./interfaces/IComplianceChecker.sol";
6:import "./interfaces/ICompliantDepositRegistry.sol";
```

Syntetika: Fixed in commit a8b4853.

Cyfrin: Verified.

7.4.5 Consider using `SafeCast` when downcasting amounts

Description: Consider using SafeCast when downcasting amounts:

- `StakingVault.sol`:

```
144:        cooldowns[msg.sender].underlyingAmount += uint152(assetsRedeemed);
165:        cooldowns[msg.sender].underlyingAmount += uint152(assets);
```

Syntetika: Fixed in commit 8d7987c.

Cyfrin: Verified.

7.4.6 `StakingVault::claimWithdraw` should revert if `assets` are zero

Description: `StakingVault::claimWithdraw` should revert if `assets` are zero.

Syntetika: Fixed in commit 2fe18df.

Cyfrin: Verified.

7.4.7 Remove obsolete `return` statements when using named return variables

Description: Remove obsolete `return` statements when using named return variables:

- `StakingVault::_withdrawTo, redeemTo`

Syntetika: Fixed in commit bd4bb12.

Cyfrin: Verified.

7.4.8 Emit missing event information

Description: Emit missing event information:

- `YieldDistributed` should have the `timestamp` parameter in addition to the amount

Syntetika: Fixed in commit f4305a6.

Cyfrin: Verified.

7.4.9 Missing call to `_setGlobalWhitelist` in `Minter.sol`

Description: The `Minter.sol` contract inherits the `Whitelist.sol` abstract contract, which manages access control for the `mint` and `redeem` functions through the `onlyWhitelisted` modifier:

```
 modifier onlyWhitelisted(address addr) {
       require(isAddressWhitelisted(addr), AddressNotWhitelisted());
       _;
   }

   /// @notice Checks if an address is whitelisted.
   /// @param user The address to check.
   /// @return bool True if the address is whitelisted, false otherwise.
   function isAddressWhitelisted(address user) public view returns (bool) {
       if (manualWhitelist[user] || globalWhitelist) { <-------
           return true;
       }

       return complianceChecker.isCompliant(user);
   }
```

The `onlyWhitelisted` modifier checks the `globalWhitelist` flag. The `StakingVault.sol` contract implements the `setGlobalWhitelist` function, which is crucial because the `StakingVault.sol` contract expects to use it. However, the `Minter.sol` contract, which mints `HilBTC` (the asset for `StakingVault.sol`), does not implement `setGlobalWhitelist`.

Impact: The `StakingVault.sol` contract will not work as expected when `setGlobalWhitelist` is enabled because `setGlobalWhitelist` is not implemented in `Minter.sol`.

Recommended Mitigation: Consider implementing `setGlobalWhitelist` in `minter.sol`:

```
function setGlobalWhitelist(bool enable) external onlyOwner {
       _setGlobalWhitelist(enable);
   }
```

Syntetika: Fixed in commits 1796e5e, 86c7b2e by removing the global whitelist functionality as it was not required by the `Minter` contract, and after the fix for L-4 it is not required at all.

Cyfrin: Verified.

7.4.10 Enforce minimum transaction amounts in `StakingVault`

Description: Some elaborate vault hacks have involved performing vault transactions using very small amounts such as 1 wei in order to manipulate the vault via rounding.

Normal users will never perform transactions using such small amounts; hence consider enforcing minimum transaction amounts to deprive attackers of this potential attack path.

Since hBTC uses 8 decimals and is 1:1 redeemable for BTC:

- 100000000 = 1 BTC ($118K)
- 10000 = 0.0001 BTC($11.87)

Consider making the minimum transaction limit a configurable parameter that the admin can change as the price of BTC fluctuates, so that it can remain around ~$10 (or even higher if preferred).

The best way to enforce this is likely overriding `ERC4626::_deposit, _withdraw` and reverting inside them if `assets` is smaller than the minimum transaction amount.

Syntetika: Fixed in commit 5ba3c19.

Cyfrin: Verified.

7.4.11 Missing `redeem` convenience function in the `StakingVault.sol`

Description: `StakingVault.sol` implements a number of convenience functions: `stake(uint256 assets)` ,`unstake(uint256 assets)` and `mint(uint256 shares)`:

```
function mint(uint256 shares) external returns (uint256) {
        return mint(shares, msg.sender);
    }

    function stake(uint256 assets) external returns (uint256) {
        return deposit(assets, msg.sender);
    }

    function unstake(uint256 assets) external returns (uint256 shares) {
        return withdraw(assets, msg.sender, msg.sender);
    }
```

But it does not have a convenience function for `redeem`, consider adding one such as:

```
function redeem(uint256 shares) external returns (uint256 shares) {
        return redeem(shares, msg.sender, msg.sender);
    }
```

Syntetika: Fixed in commit 1625c09.

Cyfrin: Verified.

7.4.12 `Minter::ownerMint` bypasses whitelist requirement and increases `totalDeposits` without actually transferring any tokens

Description: The regular functions `Minter::mint, redeem` enforce whitelist requirements and always transfer or burn tokens when incrementing or decrementing `totalDeposits`.

In contrast the admin function `Minter::ownerMint`:

- doesn't enforce whitelist requirements on `addressTo`
- increments `totalDeposits` without actually transferring any tokens into the contract

Impact: Misuse of this function can cause:

- tokens to be minted to a non-whitelisted address
- corruption of `totalDeposits` which can become different to the actual amount of tokens in the contract
- the admin could mint themselves infinite `hBTC` tokens which they could then use to drain pair tokens from any decentralized liquidity pools

Recommended Mitigation: Ideally `Minter::ownerMint` would require the admin to supply sufficient `baseAsset` tokens to the contract.

Syntetika: Acknowledged; this is the intended functionality of the `ownerMint` function.

7.4.13 Revert if `StakingVault::deposit, mint, redeem, withdraw` would return zero

Description: A common tactic of vault exploits is that the vault is manipulated such that:

- `deposit` returns 0 shares (user makes a deposit but gets no shares, effectively donating to the vault)
- `mint` returns 0 assets (user gets shares without depositing assets)

- `redeem` returns 0 assets (user burned their shares but got no assets)
- `withdraw` returns 0 shares (user withdrew assets without burning shares)

There is no legitimate user transaction which should succeed under any of the above conditions; to deny attackers these attack paths, revert if `StakingVault::deposit, mint, redeem, withdraw` would return 0.

Syntetika: Fixed in commit 2e72a57.

Cyfrin: Verified.

7.4.14 Not check for timestamp in `distributeYield` could DoS the distribute rewards

Description: The `distributeYield()` function in `StakingVault.sol` accepts a timestamp parameter without any validation checks. If an incorrect timestamp is provided (especially one set far in the future), it gets stored as `lastDistributionTimestamp` in the `_updateVestingAmount()` function. This can permanently break the vesting mechanism, as future calls to `distributeYield()` will fail at the `require(getUnvestedAmount() == 0, StillVesting())` check.

```
function _updateVestingAmount(uint256 newVestingAmount, uint256 timestamp) internal {
        require(getUnvestedAmount() == 0, StillVesting()); <-----

        vestingAmount = newVestingAmount;
        lastDistributionTimestamp = timestamp;
    }
```

Impact: Permanent denial of service for yield distribution mechanism if the owner set a incorrect timestamp value

Proof of Concept: **Recommended Mitigation:** Check for if the timestamp is less than some threshold in the future.

Syntetika: Acknowledged; users are already trusting the admin to provide yield so it is an even smaller thing to trust the admin to correctly set the `distributeYield` input.

7.4.15 `ComplianceChecker::isCompliant` incorrectly returns `true` if compliance options have no required soul bound tokens

Description: `ComplianceChecker::isCompliant` contains a logic flaw that allows universal bypass of all compliance requirements. When a compliance option exists with an empty `requiredSBTs` array (length = 0), the inner verification loop never executes, leaving the compliant variable as true by default

```
function isCompliant(address user) public view returns (bool) {
        // The user can be compliant with any option (KYC or KYB)
        uint256 complianceOptionsLength = _complianceOptions.length;
        for (
            uint256 optionIndex;
            optionIndex < complianceOptionsLength;
            optionIndex++
        ) {
            // But the address must have all the required SBTs for that option (e.g. non-sanctioned and
            ↪   age>18)
            bool compliant = true;
            uint256 requiredSBTsLength = _complianceOptions[optionIndex]
                .requiredSBTs
                .length;
            for (uint sbtIndex; sbtIndex < requiredSBTsLength; sbtIndex++) {
                if (
                    !_complianceOptions[optionIndex]
                        .requiredSBTs[sbtIndex]
                        .isVerificationSBTValid(user)
                ) {
                    compliant = false;
                    break;
                }
            }
            if (compliant) {
                return true; <-----
            }
        }
        return false;
    }
```

Malicious users can exploit this gaining unauthorized access to deposit addresses calling `registerDepositAddress` with different addressees.

Impact: Any address can gain access to deposit addresses without verification.

Recommended Mitigation: Add validation to prevent empty options in `setComplianceOptions`

Syntetika: Acknowledged; in practice this is a non-issue as compliance options are set by the admin and always have at least one required Soul Bound Token (SBT).

Cyfrin:

7.4.16 Consider using a staking rewards distributor to efficiently space out staking rewards, further deterring just-in-time attacks

Description: Consider using a staking rewards distributor to efficiently space out staking rewards instead of depositing a large amount in one transaction.

The current code uses a post-withdraw cooldown to deter "just in time" yield attacks where a user front-runs a call to `StakingVault::distributeYield` by depositing a large amount then staking it to get a large amount of the yield.

However this attack can still be executed just that the user must then wait for the cooldown to withdraw which can be as long as 90 days. The cooldown can be set by the admin as low as zero though which would enable "just in time" attacks.

Another option is to perform calls to `StakingVault::distributeYield` via services designed to prevent front-running.

Syntetika: Acknowledged.

7.4.17 Enforce that `StakingVault::decimals` is greater or equal to the underlying asset decimals

Description: EIP4626 states:

> Although the convertTo functions should eliminate the need for any use of an EIP-4626 Vault's decimals variable, it is still strongly recommended to mirror the underlying token's decimals if at all possible, to eliminate possible sources of confusion and simplify integration across front-ends and for other off-chain users.

And this set of vault property tests enforce that the vault's decimals are greater or equal to the underlying asset decimals:

```
        assertGte(
            vault.decimals(),
            asset.decimals(),
            "The vault's share token should have greater than or equal to the number of decimals as the
            ↪   vault's asset token."
        );
```

Recommended Mitigation: In `StakingVault::constructor`, revert if `IERC20Metadata(_asset).decimals() > decimals()`.

Syntetika: Fixed in commit ac97972 by enforcing decimal equality per the EIP4626 standard recommendation.

Cyfrin: Verified.

7.4.18 `StakingVault::distributeYield` should revert when there are no vault shares

Description: `StakingVault::distributeYield` should revert when there are no vault shares, and in the updated code when the vault shares are only `DEAD_SHARES`. This could be elegantly implemented as:

```
    function distributeYield(
        uint256 yieldAmount,
        uint256 timestamp
    ) external onlyOwner {
+       require(totalSupply() > DEAD_SHARES, NoStakers());
```

Syntetika: Fixed in commit 1b9d7f8.

Cyfrin: Verified.

7.4.19 Roles not set in `deposit-registry` contract constructors

Description: Roles in contracts that belong to the `issuance` contracts are being initialized in the constructors and contain proper functions to update the address for this roles (revoke and grant):

```
constructor(
        string memory _name,
        string memory _symbol,
        address _initialAdmin,
        address _minter
    ) ERC20(_name, _symbol) Ownable(_initialAdmin) {
        require(
            _minter != address(0) && _initialAdmin != address(0),
            AddressCantBeZero()
        );
        minter = _minter;
        blacklister = _initialAdmin;
        _grantRole(DEFAULT_ADMIN_ROLE, _initialAdmin);
        _grantRole(MINTER_ROLE, _minter);
    }

function setMinter(
        address newMinter
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(newMinter != address(0), AddressCantBeZero());
        revokeRole(MINTER_ROLE, minter);
        minter = newMinter;
        _grantRole(MINTER_ROLE, newMinter);
    }
```

But roles are not being initialized in the constructors of contracts in `deposit-registry`:

```
bytes32 public constant COMPLIANCE_ADMIN_ROLE = keccak256("COMPLIANCE_ADMIN_ROLE");

    constructor(address defaultAdmin) {
        // The defaultAdmin can grant other roles later
        _grantRole(DEFAULT_ADMIN_ROLE, defaultAdmin);
    }
```

Consider initializing `COMPLIANCE_ADMIN_ROLE` in the constructor of the `ComplianceChecker` contract. Consider initializing `CANCELER_ROLE` in the constructor of the `CompliantDepositRegistry` contract.

Syntetika: Fixed in commit 9fccd3b.

Cyfrin: Verified.

7.5 Gas Optimization

7.5.1 Cache storage to prevent identical storage reads

Description: Reading from storage is expensive; cache storage to prevent identical storage reads:

- `ComplianceChecker.sol`:

```
// cache `for` loop storage lengths in `isCompliant`
59:             optionIndex < _complianceOptions.length;
66:                 sbtIndex < _complianceOptions[optionIndex].requiredSBTs.length;
```

- `CompliantDepositRegistry.sol`:

```
// cache `investorDepositMap[investor]` in `getDepositAddress`
60:         require(investorDepositMap[investor] > 0, UnregisteredInvestor());
64:         return depositAddresses[investorDepositMap[investor]];

// cache `nextDepositAddressIndex` in `registerDepositAddress`
93:                 nextDepositAddressIndex < depositAddresses.length &&
102:        investorDepositMap[msg.sender] = nextDepositAddressIndex;

// cache `getDepositAddress(msg.sender)` in `registerDepositAddress`
// ideally do this by using a named return variable, assigning straight to it,
// using the named return variable to emit the event then deleting the obsolete
// `return` statement
104:        emit DepositAddressSet(msg.sender, getDepositAddress(msg.sender));
106:        return getDepositAddress(msg.sender);

// cache `depositAddresses.length` in `getDepositAddresses`
128:        if (startIndex + count > depositAddresses.length) {
129:            returnLength = depositAddresses.length - startIndex;
134:            i < count && startIndex + i < depositAddresses.length;

// cache `depositAddresses.length` in `addDepositAddresses`
// just invert the order of these two statements then use `startIndex`
// to set `finalizedAddressesLength`
154:        finalizedAddressesLength = depositAddresses.length;
156:        uint startIndex = depositAddresses.length;

// cache `block.timestamp + batchChallengePeriod` in `addDepositAddresses`
// and use it to set `latestBatchUnlockTime` and also to emit the event
161:        latestBatchUnlockTime = block.timestamp + batchChallengePeriod;
165:            latestBatchUnlockTime,

// cache `finalizedAddressesLength` and use `block.timestamp` instead of
// `latestBatchUnlockTime` when emitting event in `challengeLatestBatch`
199:        uint batchLength = depositAddresses.length - finalizedAddressesLength;
208:            finalizedAddressesLength,
209:            latestBatchUnlockTime,
```

- `Blacklistable.sol`:

```
// use input `_newBlacklister` when emitting event in `updateBlackLister`
74:        emit BlacklisterChanged(blacklister);
```

- `Minter.sol`:

```
// cache `custodian` in `transferToCustody`
135:        baseAsset.safeTransfer(custodian, amount);
136:        emit FundsTransferredToCustody(amount, custodian);
```

- `StakingVault.sol`:

```
// cache `cooldownDuration` in `redeem, withdraw`
137:        if (cooldownDuration == 0) {
142:            cooldownDuration;
159:        if (cooldownDuration == 0) {
164:            cooldownDuration;

// use input `duration` when emitting event in `setCooldownDuration`
220:        cooldownDuration = duration;
221:        emit CooldownDurationUpdated(previousDuration, cooldownDuration);

// cache `lastDistributionTimestamp` in `getUnvestedAmount` if first `return`
// statement is unlikely to be frequently triggered
271:        if (lastDistributionTimestamp > block.timestamp) {
275:            lastDistributionTimestamp;
```

Syntetika: Fixed in commits bc24502, 8560039, bfad835.

Cyfrin: Verified though ideally `StakingVault::withdraw` would also cache `cooldownDuration` similar to the fix made inside `redeem`.

7.5.2 Variables only set once in `constructor` of non-upgradeable contracts should be declared `immutable`

Description: Variables only set once in `constructor` of non-upgradeable contracts should be declared `immutable`:

- `CompliantDepositRegistry::complianceChecker`
- `Minter::baseAsset, hilBTCToken`

Syntetika: Fixed in commit 3d1e596.

Cyfrin: Verified.

7.5.3 Use named return variables when this eliminates local variables

Description: Use named return variables when this eliminates local variables:

- `CompliantDepositRegistry::getDepositAddresses`
- `StakingVault::redeem`

Syntetika: Fixed in commit f8f821d.

Cyfrin: Verified.

7.5.4 Use `ReentrancyGuardTransient` for faster `nonReentrant` modifiers

Description: Use ReentrancyGuardTransient for faster `nonReentrant` modifiers:

- `issuance/src/minter/Minter.sol`

Syntetika: Fixed in commit d5131f6.

Cyfrin: Verified.

7.5.5 Remove `from` parameter from `Minter:redeem` and `_onlySender` function

Description: `Minter:redeem` takes a `from` input parameter but then calls `_onlySender` to enforce that `from == msg.sender`.

In this case there is no need for the `from` input parameter for the `_onlySender` function; remove them both and just use `msg.sender` inside `Minter:redeem`.

Syntetika: Fixed in commit 94a2165.

Cyfrin: Verified.

7.5.6 Small functions only used once should be inlined into their parent functions

Description: Small functions only used once should be inlined into their parent functions:

- `StakingVault::_updateVestingAmount`

Syntetika: Acknowledged.