PUBLIC

Code Assessment

of the HilBTC Smart Contracts

October 31, 2025

Produced for

SYNTETIKA

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Syntetika with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of HilBTC according to Scope to support you in forming an opinion on their security risks.

Syntetika implements an ERC-20 token, `hBTC`, that is a synthetic BTC. The custodian can use the underlying funds to generate yield which is then forwarded to `hBTC` stakers whose staking shares are tokenized as `shBTC`.

The most critical subjects covered in our audit are asset solvency, functional correctness and access control. The general subjects covered are testing, documentation, gas efficiency, and upgradeability.

Security regarding most of the above is improvable.

Further, readers are advised to carefully read the report as several findings remain where the risk was accepted. Below the most notable ones are listed:

- The loss realization process is improvable, see totalAssets() Can Revert and Loss Realization Process, Inaccurate Loss Realization and Loss Realization Can Be DoSed.
- Users can be penalized even when their cooldown window has passed, see Users Outside of Cooldown Penalized.

Additionally, the *tests are insufficient* as several issues could have been caught by testing more extensively.

Hence, security regarding and quality regarding the aforementioned subjects is improvable. In summary, we find that the codebase provides an improvable level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	2
• Code Corrected	2
Medium-Severity Findings	6
• Code Corrected	5
• Risk Accepted	1
Low-Severity Findings	13
• Code Corrected	4
• Code Partially Corrected	2
• Risk Accepted	5
• Acknowledged	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the HilBTC repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	15 Sep 2025	a276b312162abee5068fe4dcd86fe5570ff3c574	Initial Version
2	29 Sep 2025	a95fe630fe1e91186c229df878c52bf084b6fa15	First fixes
3	06 Oct 2025	bf9d3c6bc1437a2c1830468576d64b93b914eb8b	Second fixes
4	23 Oct 2025	1adf43be5320b79bd06a312171218550e8ff2117	Third fixes
5	29 Oct 2025	59cda6d4df2155978f0228122943a4060a970d90	Fourth fixes

For the solidity smart contracts, the compiler version `0.8.28` was chosen.

The contracts in the following files are in the scope of the review:

```
deposit-registry/contracts:
    ComplianceChecker.sol
    CompliantDepositRegistry.sol
    Imports.sol

issuance/src:
    helpers:
        Blacklistable.sol
        TokensHolder.sol
        Whitelist.sol
    minter:
        Minter.sol
    token:
        HilBTC.sol
    vault:
        StakingVault.sol
```

After V2, the scope was updated as follows:

Renamed:

```
issuance:
    src:
        helpers:
            Blacklistable.sol  -->  BlacklistableUpgradeable.sol
```

Added:

```
deposit-registry:
    contracts:
        ComplianceCheckerUpgradeable.sol
```

2.1.1 Excluded from scope

Any contracts not explicitly listed above are out of the scope of this review. Third-party libraries are out of the scope of this review. More specifically, `openzeppelin-contracts` is expected to work as intended and is out of the scope of this review. The Galactica protocol, responsible for managing compliance checks and issuing the SBTs, is expected to work as intended and is out of the scope of this review.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes according to the versions. However, note that the system overview, even including the changelog, might thus be inaccurate for the latest version.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Syntetika offers an ERC20 version of `BTC` (`HilBTC`) with a blacklist functionality, and a way to stake it for increased yield. The issuance of the `HilBTC` is done by a minter contract that enforces either a manual whitelist or compliance checks provided by Galactica. All the contracts are deployed behind a transparent proxy.

2.2.1 Compliance Checker

This contract allows querying whether a user is compliant with a configured set of rules by calling the `isCompliant()` function. The `COMPLIANCE_ADMIN_ROLE` can configure the different compliance options, each option contains a list of `VerificationSBT` representing compliance checks. Examples of compliance options and their SBTs:

- Option 1: user resides in the USA, user is over 21
- Option 2: user resides in EU, user is over 18

In order to be compliant, a user must have all the soulbound tokens (SBT) of at least one compliance option. The SBTs are issued by Galactica.

2.2.2 Compliant Deposit Registry

The registry offers a list of deposit addresses on another chain that whitelisted investors can register for and use. The addresses are added by the `DEPOSIT_ADDRESS_CREATOR_ROLE` and the new batch can be challenged by the `CANCELER_ROLE` for `batchChallengePeriod` seconds after it is submitted. If a batch is challenged, the addresses it contains will not be made available until they are submitted again. To be able to register, investors need to pass the `isCompliant()` check on the attached `ComplianceChecker`. Once an investor is linked to an address, they cannot unregister it or request another one.

2.2.3 HilBTC

The `HilBTC` is an ERC20 representation of `BTC` on an EVM chain with 8 decimals. It offers a blacklist feature that the `blacklister` address can manage, and the tokens are burnable. A blacklisted address is not able to receive, send nor burn funds. One exception to this rule is that the contract's owner is able to `burnFrom()` any address, without requiring any allowance. If the owner has the required allowance, they can also transfer from arbitrary addresses, even blacklisted ones.

The tokens can be minted by the `MINTER_ROLE`, which is expected to be held only by the Minter contract. The `HilBTC` is expected to be pegged 1:1 to its underlying asset.

2.2.4 Minter

The `Minter` contract is expected to bear the `MINTER_ROLE` on the HilBTC token and extends the `ComplianceChecker`. It is the entry point for allowed users to mint and redeem `HilBTC` against some `baseAsset` having the same decimals as `HilBTC`, e.g., `WBTC`. A user is allowed to mint/redeem if they are either part of a local whitelist managed by the `DEFAULT_ADMIN_ROLE`, or are compliant in the sense of the Compliance Checker. The contract tracks the amount of deposited `baseAsset`, which is increased/decreased upon mint/redeem in the `variable`. If not enough `baseAsset` is available for redemption, the function will revert.

The liquidity that is left in the `Minter` after a `mint()` can be transferred to the `custodian` address to generate some yield by the `OPERATOR_ROLE` with the `transferToCustody()` function.

The `DISTRIBUTOR_ROLE` can call the `distributeYield()` function when yield has been accrued from the liquidity transferred to the `custodian`. This will increase the total deposited amount and trigger `distributeYield()` on the Staking Vault.

The contract can be paused by the `PAUSER_ROLE`, when paused, the `mint()`, `redeem()` and `transferToCustody()` functions are not available. Note that the `DEFAULT_ADMIN_ROLE` can mint unbacked `HilBTC`.

2.2.5 Staking Vault

The `StakingVault` is an ERC4626 vault whose underlying asset is `HilBTC`, it implements a blacklist similar to `HilBTC`, delayed withdrawals and redemptions (max 90 days), and optionally early exits with a fee. A blacklisted address is not able to receive, send nor redeem shares. Holders of `HilBTC` can deposit their tokens in the vault and the value of their shares will increase every time yield is distributed. The first deposit must be made by the contract's owner and `1000` shares will be sent to `address(0xdead)`, the vault cannot be used prior to this action. The assets amount upon `deposit()` and `mint()` must be greater than the configurable `minAssetsAmount`.

The `totalAssets()` function computes the difference between the vault's assets balance and the currently unvested amount to be distributed. The vesting amount is updated when yield is distributed through `distributeYield()`, the vested amount is unlocked linearly over a period of 8 hours. Yield can be distributed only if the previous vesting has finished and the total supply of shares is above the number of dead shares.

Upon `withdraw()` or `redeem()`, a cooldown starts for the caller. Their shares are burned, and the assets are sent to a special `TokensHolder` contract. When the cooldown duration has elapsed, users can call the `claimWithdraw()` function. If multiple withdrawals or redemptions are done before the previous ones are claimed, the whole amount will be subject to the latest cooldown period. If the `cooldownDuration` is set to `0`, withdrawals and redemptions are instantaneous and do not enter any cooldown queue.

In the case where early exits are enabled, the `claimWithdraw()` function can be called right after the cooldown phase started. The fee taken for an early exit starts at `maxEarlyExitFeeBps` and then decreases linearly with time until the end of the cooldown. The fee is `0 <= maxEarlyExitFeeBps < 10%`. The fee is then sent to the `earlyExitFeeRecipient`. By

default, the earlyExitFeeRecipient is the StakingVault itself, this configuration has the effect that the early exit fee increases the value of a share in the vault.

If the custodian incurs a loss in their strategy, Syntetika plans to burn HilBTC from the vault, as HilBTC should keep its peg. Syntetika plans to have an insurance fund designed as a drawdown protection mechanism to which they allocate a portion of revenues to.

2.2.6 Changes in V2

- The proxy architecture has been set to UUPS. The ComplianceDepositRegistry and the TokensHolder are not upgreadeable.
- The CANCELER_ROLE in CompliantDepositRegistry can only cancel the full unfinalized batch, partial challenge is not possible anymore.

2.2.7 Changes in V3

Assets in the cooldown can now be slashed. Note that more changes were implemented as a response to fixes.

2.2.8 Changes in V4-V5

To correct the slashing, TokensHolder now implements ERC-4626. Changes were implemented accordingly.

2.3 Trust Model

- Users are fully untrusted.
- Bearers of the DEFAULT_ADMIN_ROLE in ComplianceChecker are fully trusted. They are expected to manage the critical roles of the contract correctly and in a non-adversarial manner. In the worst case, they can grant the DEFAULT_ADMIN_ROLE or the COMPLIANCE_ADMIN_ROLE to malicious actors.
- Bearers of the COMPLIANCE_ADMIN_ROLE in ComplianceChecker are fully trusted. They are expected to manage the compliance options correctly and in a non-adversarial manner. In the worst case, they can register options that either DOS the contract or wrongly grant compliance.
- Bearers of the DEFAULT_ADMIN_ROLE in CompliantDepositRegistry are fully trusted. They are expected to manage the critical roles of the contract correctly and in a non-adversarial manner. In the worst case, they can grant the DEFAULT_ADMIN_ROLE, DEPOSIT_ADDRESS_CREATOR_ROLE and CANCELER_ROLE to malicious actors. Additionally, they can set the batch challenge period to unreasonable values.
- Bearers of the DEPOSIT_ADDRESS_CREATOR_ROLE in CompliantDepositRegistry are partially trusted. They are expected to manage the batches of new deposit addresses correctly and in a non-adversarial manner. They are expected to not collude with bearers of the CANCELER_ROLE. In the worst case, they can register dead or already used addresses, but they are expected to be challenged by the CANCELER_ROLE.
- Bearers of the CANCELER_ROLE in CompliantDepositRegistry are fully trusted. They are expected to challenge the batches correctly and in a non-adversarial manner. They are expected to not collude with bearers of the DEPOSIT_ADDRESS_CREATOR_ROLE. In the worst case, they can challenge and remove all the addresses, even the finalized ones.
- Bearers of the DEFAULT_ADMIN_ROLE in HilBTC are fully trusted. They are expected to manage the critical roles of the contract correctly and in a non-adversarial manner. In the worst case, they can grant the MINTER_ROLE to malicious actors.

- Bearers of the `MINTER_ROLE` in `HilBTC` are fully trusted. They are expected to mint tokens correctly and in a non-adversarial manner. In the worst case, they can mint unlimited amounts of `HilBTC`. This role is expected to be held only by the `Minter` contract.
- The `owner` of the `HilBTC` is fully trusted. They are expected to not burn from arbitrary addresses without a good reason and to manage the `blacklister` address correctly. In the worst case, they can burn arbitrary amounts from arbitrary addresses.
- The `blacklister` of the `HilBTC` is fully trusted. They are expected to blacklist and unblacklist addresses in the best interest of the system and in a non-adversarial manner. In the worst case, they choose to not blacklist an address that should be blacklisted, or unblacklist it if they previously blacklisted it. This would violate the system's compliance.
- Bearers of the `DEFAULT_ADMIN_ROLE` in `Minter` are fully trusted. They are expected to manage the critical roles and parameters of the contract correctly and in a non-adversarial manner. In the worst case, they can grant the `OPERATOR_ROLE`, `PAUSER_ROLE` and `DISTRIBUTOR_ROLE` to malicious actors, whitelist arbitrary addresses or set a wrong `ComplianceChecker`. Additionally, they have the power to perform arbitrary minting of `HilBTC`.
- Bearers of the `OPERATOR_ROLE` in `Minter` are partially trusted. They are expected to transfer assets to the custodian correctly and in a non-adversarial manner. In the worst case, they can transfer assets to the custodian as soon as some liquidity arrives in the `Minter`, DOSing the redemptions until their role is revoked.
- The `custodian` in `Minter` is fully trusted. They are the recipient of funds and are responsible for handling them securely. In the worst case, a loss of funds leading to a severe depeg could occur.
- Bearers of the `PAUSER_ROLE` in `Minter` are partially trusted. They are expected to pause and unpause the contract in the best interest of the system in a non-adversarial manner. In the worst case, they can choose to not pause the system in case of an emergency, or pause it to DOS the system until their role is revoked and the contract is unpaused.
- Bearers of the `DISTRIBUTOR_ROLE` in `Minter` are fully trusted. They are expected to distribute the yield correctly and in a non-adversarial manner. In particular, they are expected to not distribute more yield than what the custodian's strategy earned, and set a realistic timestamp. In the worst case, they distribute arbitrary amounts of yield, creating unbacked `HilBTC`, or DOS the yield distribution on the `StakingVault` by setting a timestamp that is far in the future.
- Bearers of the `DEFAULT_ADMIN_ROLE` in `StakingVault` are fully trusted. They are expected to manage the critical roles and parameters of the contract correctly and in a non-adversarial manner. In the worst case, they can grant the `DISTRIBUTOR_ROLE` or `DEFAULT_ADMIN_ROLE` to malicious actors.
- Bearers of the `DISTRIBUTOR_ROLE` in `StakingVault` are fully trusted. They are expected to distribute the yield correctly and in a non-adversarial manner. In the worst case, they can DOS the yield distribution by setting a timestamp far in the future.
- The `owner` of the `StakingVault` is fully trusted. They can set arbitrary cooldowns and minimum amounts. In the worst case, they can DoS users.
- The `blacklister` of the `StakingVault` is fully trusted. They are expected to blacklist and unblacklist addresses in the best interest of the system and in a non-adversarial manner. In the worst case, they choose to not blacklist an address that should be blacklisted, or unblacklist it if they previously blacklisted it. This would violate the system's compliance.

2.3.1 Changes in V2

- The `DEFAULT_ADMIN_ROLE` of the `StakingVault` and `Minter`, and the `owner` of `HilBTC` are also responsible for upgrading the contracts. They are trusted to not upgrade to malicious versions.

2.3.2 Changes in V5

- The owner can now upgrade `TokensHolder`.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- totalAssets() Can Revert and Loss Realization Process Risk Accepted

Low-Severity Findings	9

- Inaccurate Loss Realization Risk Accepted
- Loss Realization Can Be DoSed Risk Accepted
- Users Outside of Cooldown Penalized Risk Accepted
- Inconsistent Upgrader Acknowledged
- Initializer Problems Code Partially Corrected
- Challenge Timing Problems Code Partially Corrected Acknowledged
- Cooldown and Fee User Agreement Unclear and Unfair Risk Accepted
- ERC-4626 Violations Acknowledged
- Unnecessary Complexity in Access Control Risk Accepted

5.1 `totalAssets()` Can Revert and Loss Realization Process

Design Medium Version 1 Risk Accepted

CS-HBTC-006

In case of a loss, Syntetika specified they would burn `HilBTC` from the `StakingVault`. While this could work for small losses, bigger losses might not be fully realizable if they exceed the balance of the `StakingVault`. Additionally, the owner of the `HilBTC` is able to burn from arbitrary addresses, but needs to be trusted to burn only from the `StakingVault` for that purpose.

Implementing a dedicated loss realization callpath would crystallize the process and lower the amount of trust needed. Ultimately, issues arise due to a lack of such a well-defined process.

More specifically, the function `StakingVault.totalAssets()` carries the following computation:

```
IERC20(asset()).balanceOf(address(this)) - getUnvestedAmount();
```

If `getUnvestedAmount()` is greater than the contract's balance in the underlying `asset()`, the computation will underflow and the function will revert. This can happen if the vault incurs a loss while yield is being distributed. If the loss or the yield is big enough, the unvested amount will outweigh the balance.

If `StakingVault.totalAssets()` reverts, the vault is DOSed until `getUnvestedAmount()` returns an amount that is at most equal to the contract's balance. This could be fixed with a dedicated loss realization function taking care of this case.

Risk accepted:

Syntetika is aware of the issue and accepts the risk based on it being forked from the Ethena codebase.

Note that Syntetika introduced the function `Minter.realizeLosses()` which defines a clear entrypoint. In Version 5, `Minter.realizeLosses()` was improved to handle burning from the vault and the tokens holder. However, it only clarifies the loss realiziation process in code but does not consider the underlying issue.

5.2 Inaccurate Loss Realization

Design | Low | Version 5 | Risk Accepted

CS-HBTC-033

The underlying issue in Fundamentally Broken Loss Realization Process was resolved. However, the loss realization remains inaccurate:

- Due to a lack of on-chain computations, the `TokensHolder` and shBTC may be slashed unproportionally.
- Assuming that the off-chain computation computes it correctly, the validation of seen values is improper. More specifically, the supply of `TokensHolder` and shBTC are are validated to match the expected (seen) supplies during computation. However, the relevant factors are the assets held. While for `TokensHolder` this is typically only changing along with the shares, that is not the case for shBTC which can accure yield.

To summarize, the loss realization remains inaccurate.

Risk accepted:

Syntetika is aware and notes:

```
This is done intentionally because introducing proportions and percentages would introduce the potential for precision loss.
```

While the potential precision loss is true, it could have been defined that in case of rounding errors, the vault, for example, is penalized slightly more.

5.3 Loss Realization Can Be DoSed

Security | Low | Version 5 | Risk Accepted

CS-HBTC-034

`Minter.realizeLosses()` can be DoSed. More specifically, an attacker can trigger the following checks to revert:

```
require(
    IERC20(address(vault)).totalSupply() == expectedStakingVaultSupply,
    GuardNotPassed()
);
require(
    IERC20(vault.tokensHolder()).totalSupply() ==
        expectedTokensHolderSupply,
    GuardNotPassed()
);
```

In detail, the attacker could simply mint new shares on either of the two contracts to then trigger a revert. Note that this is rather unlikely. A longer DoS would require an attacker to spam the network long enough.

Risk accepted:

Syntetika is aware of the problem but prefers to perform the computations off-chain and to implement front-running protection this way.

5.4 Users Outside of Cooldown Penalized

Design | Low | Version 3 | Risk Accepted

CS-HBTC-032

The loss realization process will affect both the `StakingVault` and the `TokensHolder`. However, users with registered withdrawals outside of the cooldown period will also be penalized.

Ultimately, the cooldown mechanism does not prevent users from getting slashed after the cooldown period has ended.

Risk Accepted

Syntetika acknowledged the issue and accepts the risk.

Users should exit the system as soon as possible to prevent getting penalized unnecessarily.

5.5 Inconsistent Upgrader

Design | Low | Version 2 | Acknowledged

CS-HBTC-028

For `Minter` and `StakingVault`, `DEFAULT_ADMIN_ROLE` is the only address that passes `_authorizeUpgrade()`. In contrast, for `HilBTC` this is `owner`.

Ultimately, the access control is unnecessarily inconsistent.

Acknowledged:

Syntetika is aware of this inconsistency and chose to not change the code.

5.6 Initializer Problems

Design | Low | Version 2 | Code Partially Corrected

CS-HBTC-029

The fixes for Proxied contracts cannot set state in constructor introduced initializers for several contracts. However, the following problems are present:

1. `__ComplianceChecker_init()` performs `__AccessControl_init()`. Also, `Minter.initialize()` does the same while also invoking `__ComplianceChecker_init()`. Thus, the functionality for initializing the access control is duplicated (however, it is a no-op).
2. Similarly, `Minter.initialize()` grants the `DEFAULT_ADMIN_ROLE` which is already done in `__ComplianceChecker_init()`.
3. `__ComplianceChecker_init()` is an internal initializer but uses the `initializer()` modifier instead of `onlyInitializing()`.
4. `__Blacklistable_init()` performs `__Ownable_init()`. Also, `StakingVault.initialize()` does the same while also invoking `__Blacklistable_init()`. Thus, the functionality for initializing owner is duplicated.
5. Similarly, `HilBTC.initialize()` sets the blacklister storage variable which is already done in `__Blacklistable_init()`.
6. `__Blacklistable_init()` is an internal initializer but uses the `initializer()` modifier instead of `onlyInitializing()`.
7. `__AccessControl_init()` is often used even though it is a no-op. Thus, that indicates that even no-op initializers of ancestor contracts should be used (as recommended by the library docs). However, various ancestor initializers are ignored (`NoncesUpgradeable`, `ContextUpgradeable`, ...). Ultimately, code is inconsistent.

Code partially corrected:

1. Corrected: `__AccessControl_init()` in `Minter.initialize()` was removed.
2. Corrected: granting of the `DEFAULT_ADMIN_ROLE` in `Minter.initialize()` was removed.
3. Not corrected.
4. Corrected: `__Ownable_init()` in `StakingVault.initialize()` was removed.
5. Corrected: `HilBTC.initialize()` does not set the blacklister storage variable anymore.
6. Corrected: the modifier was changed to `onlyInitializing()`.
7. Not corrected.

5.7 Challenge Timing Problems

Correctness | Low | Version 1 | Code Partially Corrected | Acknowledged

CS-HBTC-008

`CompliantDepositRegistry.challengeLatestBatch()` performs the following:

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

However, note that the intention behind this is unclear:

1. Assuming that the full batch was dropped, this could be reasonable to ensure that new batches can be added. However, `addDepositAddresses()` requires that the timestamp is in the past. Thus, one cannot immediately add new addresses.
2. After a challenge, more addresses can still be dropped (similar reasons as above).
3. Partial dropping immediately finalizes the addresses. Thus, the canceller could for example cancel 0 addresses and does immediately make all added one finalized.

Additionally, `addDepositAddresses()` allows adding a small amount of addresses (e.g. no addresses). This updates the challenging time. Additionally, the canceller's monitoring will be triggered. That leads to the following consequences:

1. `DEPOSIT_ADDRESS_CREATOR_ROLE` can disturb operations.
2. `DEPOSIT_ADDRESS_CREATOR_ROLE` can trigger unnecessary compute and thus cost for the `CANCELER_ROLE` who is expected to monitor the adding of addresses.

Code partially corrected:

The fix for Arbitrary Challenging Problems mitigates the partial dropping issue, a check enforcing that a new batch should have at least one address was added in `addDepositAddresses()`. However, after challenging a batch, the next batch cannot be added immediately and will have to wait for the next block.

Acknowledged:

Syntetika is aware of the remaining inconsistency and chose to not change the code.

5.8 Cooldown and Fee User Agreement Unclear and Unfair

Design | Low | Version 1 | Risk Accepted

CS-HBTC-009

The contracts implement an unclear and unfair agreement regarding exit times and fees in `StakingVault.claimWithdrawal()`.

Unfair Cooldown

An unfair cooldown mechanism is implemented. More specifically, the following conditional allows for potentially unfair execution:

```
if (
    !earlyExitEnabled &&
    (block.timestamp >= userCooldown.cooldownEnd ||
        cooldownDuration == 0)
) {
    tokensHolder.withdraw(receiver, assets);
}
```

Consider the following scenario:

1. Alice withdraws while the `cooldownDuration == 90 days`.
2. The next day, `cooldownDuration = 10 days` is set.
3. Bob initiates a withdrawal directly after that.
4. Ultimately, Alice will have to wait for 89 days while Bob can enjoy the new cooldown.

Ultimately, the agreement is unfair (e.g. here the cooldown in favor of the user could make sense).

Unclear Agreement

Note that the other conditional within the function creates confusion in the agreement:

```
else if (earlyExitEnabled) {
    (uint256 fee, uint256 withdrawAmount) = getEarlyExitAmount(
        userCooldown.cooldownStart,
        userCooldown.cooldownEnd,
        assets
    );
    tokensHolder.withdraw(receiver, withdrawAmount);
}
```

Note how the cooldown that was agreed upon on the withdrawal initiation is considered. This contrasts the following:

- The previous conditional where the current cooldown is relevant (e.g. here the cooldown in favor of the user could make sense).
- The early exit fee might have increased or decreased since the initiation of the withdrawal (e.g. here the fee in favor of the user could make sense).

Unfair Fee

Additionally, when both conditionals are combined, the fee might be unfair. Consider the following scenario:

1. Alice initiates a withdrawal when early exits are not enabled. Assume the cooldown is 10 days.
2. Directly after that, early exits are enabled where the fee is at most 10%.
3. After 5 days, the cooldown is set to 0.
4. Bob does initiate a withdrawal which immediately withdraws without paying fees.
5. If Alice was to withdraw, she would have to pay fees.

Summary

The agreement regarding the cooldown period is inconsistent, unclear and unfair in some cases. The main reasons are:

- Unfair logic (e.g. can be in favor of user)
- Valid but inconsistent configuration (e.g. could be performed as part of one function)

Risk accepted:

Syntetika is aware of the issue but chose not to resolve it fully.

Only the case where `cooldownDuration == 0` is set leads now to immediate withdrawals for all.

5.9 ERC-4626 Violations

Design | Low | Version 1 | Acknowledged

CS-HBTC-010

`StakingVault` is described as EIP-4626 compatible tokenized vault. However, several violations of the standard make it non-compatible and hard to integrate with.

Below, we provide *non-exhaustive* lists of violations and other potential problems:

- Violations

    1. The standard explicitly states that withdrawals and redemptions send out `assets`. That is not the case due to delayed withdrawals (i.e. cooldown). Note that implementing a different standard could be meaningful (e.g. EIP-7540)
    2. `withdraw()` and `redeem()` completely ignore the arguments `receiver` and `owner`. While that is somewhat required for a safe and efficient cooldown mechanism, it nonetheless violates the standard.
    3. The first deposit may pull more funds than specified as an argument in the `deposit` function. Namely, that is due to the first deposit first minting dead shares and then depositing accordingly.

- Violations depending on interpretation

    1. The `maxXYZ()` functions ignore the blacklist and may return non-zero for blacklisted addresses. Depending on interpretation, "blacklists" can be interpreted as user limits but could be interpreted as other reasons of reverts. Note that the inherited ERC4626, for example, technically blacklists `0x0` but does not adjust the functions.
    2. Similarly, the `previewXYZ()` function could or could not revert depending on interpretation.
    3. Similarly, the first deposit (i.e. `DEAD_SHARES` creation) is not considered in any of the `maxXYZ()` or `previewXYZ()` functions.
    4. Similarly, the `minAssetsAmount` is ignored by those functions. Depending on interpretation this may or may not be correct.

- Other behavior problematic for integrations and other:

    1. `deposit()` / `mint()` and `withdraw()` / `redeem()` define minimum amounts. That includes checks against `minAssetsAmount` and `0`. While these do not necessarily violate the EIP, they may lead to difficulties for integrators.
    2. `maxWithdraw()` / `maxRedeem()` and `previewWithdraw()` / `previewRedeem()` are unclear in terms of EIP-4626 on how they could comply with the standard due to the cooldown mechanism.

As of Version 5, the `TokensHolder` implements ERC-4626. However, it violates the standard in various ways and should not be treated as such.

Acknowledged:

Syntetika is aware of that integrators might run into issues when integrating with `shBTC` and chose to not modify the code.

5.10 Unnecessary Complexity in Access Control

Design | Low | Version 1 | Risk Accepted

CS-HBTC-013

Access control is unnecessarily complex. More specifically, the following increases its complexity:

1. Mix up of `AccessControl` and storage variables for roles. Note that this unnecessarily increases code size and complexity. Consider the following example in `HilBTC`:

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

Note how the function indicates that there should only be one minter. However, due to `AccessControl` multiple are possible (e.g. `grantRole()`). Additionally, note that all such occurrences are: `Minter.setOperator()`, `Minter.setPauser()`, `Minter.setCustodian()`, `HilBTC.setMinter()` and `StakingVault.setDistributor()`.

Note that it is advised that access control wrappers (e.g. `setMinter()` which wraps `grantRole()`) are used. Later computations expect the storage variables to be set appropriately.

Further, this design has lead to the below more objective problems:

- `Minter.constructor()`: `distributor` is not set but the role is assigned.
- `Minter`: There is no explicit function to set `distributor` (e.g. `setDistributor()`) while there are equivalent functions for other roles (e.g. `setOperator()`).
- `Minter`: The role `CUSTODIAN_ROLE` can be assigned but is never used for access control.
- `StakingVault.constructor()`: `distributor` is set but the role is not assigned.

1. Mixup of `AccessControl` and `Ownable` (inherited through `Blacklistable`) in `HilBTC` and `StakingVault` contracts. Note that this unnecessarily increases code size and complexity. More specifically, it creates the following confusion regarding the differences between `DEFAULT_ADMIN_ROLE` and `owner` (both hold significant power):
   - `owner` can call `updateBlacklister()`, `setCooldownDuration()` and `setMinAssetsAmount()` while `DEFAULT_ADMIN_ROLE` cannot.
   - `DEFAULT_ADMIN_ROLE` can call `setDistributor()`, `setEarlyExitEnabled()`, `setMaxEarlyExitFeeBps()`, `setEarlyExitFeeRecipient()` and manage other roles while `owner` cannot.

   Ultimately, the access control is mixed up which increases the complexity.

To summarize, access control is not implemented consistently which has led to problems. Optimally, either only `AccessControl` or only `Ownable` with custom storage variables should be used.

Risk accepted:

Syntetika is aware of the design and potential problems.

However, Syntetika corrected the following immediate resulting problems:

- Corrected : The address of the `distributor` is now set in the `initialize()` function of the `Minter` contract.
- Corrected: The `DISTRIBUTOR_ROLE` is set in the `initialize()` function of the `StakingVault`.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	2

- Fundamentally Broken Loss Realization Process Code Corrected
- Incorrect Early Exit Fees Code Corrected

Medium-Severity Findings	5

- Loss Realization Function Cannot Be Called Code Corrected
- Users Can Escape Losses Code Corrected
- Blacklisted Addresses Can Still Interact Code Corrected
- Missing Permit Feature Code Corrected
- Proxied Contracts Cannot Set State in Constructor Code Corrected

Low-Severity Findings	4

- Inconsistent Storage Locations Code Corrected
- Arbitrary Challenging Problems Code Corrected
- Incorrect Event Emissions Code Corrected
- Incorrect DEAD_SHARES Logic Code Corrected

Informational Findings	1

- StakingVault Has No Initial Blacklister Code Corrected

6.1 Fundamentally Broken Loss Realization Process

Correctness High Version 3 Code Corrected

CS-HBTC-031

The loss realization process introduced to protect against frontrunning attacks fundamentally breaks various properties for the loss realization.

More specifically, the process is now defined as follows:

1. Losses are recognized (e.g. off-chain strategy).
2. Losses are realized on-chain by burning from the vault. Thus, the vault is immediately penalized.
3. Users with unclaimed withdrawals are penalized by taking the minimum value of the shares burned and the amount withdrawn.
4. The remaining amount is donated to the vault.

Several problems arise:

1. **Incorrect Conversion of Burned Shares at Current Time.**

   In `StakingVault.claimWithdraw()`, an amount of shares is passed to `previewWithdraw()`, which expects amount of assets. As a result, the `assetsAfterCooldown` value will be too small if the share price increased and users will loose tokens when claiming their withdrawal. On the other hand, if a loss was incurred, `assetsAfterCooldown` will be too high and `assets` will not be capped, so users will not be affected by the loss.

2. **Incorrect Donation to Vault.**

   In `StakingVault.claimWithdraw()`, the unclaimed withdrawal from a user (created by the slashing), is donated to the vault. This creates a scenario where the vault profits from slashed amounts. However, slashed amounts should be burned. Otherwise, the slash is improperly realized.

3. **Loss Realization Unreasonable or Inaccurate.**

   Assuming both of the above points were resolved, the loss realization is either completely unreasonable or it could be possible that an accurate loss realization is impossible to achieve.

   1. *Loss Realization Unreasonable.* If a loss of 10 hBTC occurs and 10 hBTC are burned from the vault, then the penalization will exceed the intended amount. Consider the following example.

      1. Assume that both Alice and Bob hold 100 shBTC each at an exchange rate of 1 (i.e. 200 hBTC total assets).
      2. Now, Bob withdraws everything.
      3. Next, a loss of 10 is realized by burning 10 from the vault.
      4. Finally, when Bob claims the withdrawal, he will only receive 90 hBTC.
      5. Ultimately, 20 hBTC would have been slashed for a loss of 10.

      Note that more example can be constructed more extremely.

   2. *Loss Realization is Inaccurate.* As illustrated above, not 10 hBTC can be burned from the vault. Rather, it should be proportional according to the split of funds (ratio of total assets in vault and total assets in total (vault and tokens holder)). In the given example, burning 5 hBTC from the vault would have implied that Alice would have paid the remaining 5 (50% of assets in vault).

      However, that approach is inaccurate and unsuitable. Consider the adjusted example below:

      1. The same setup is expected with Bob and Alice as in the previous example.
      2. Syntetika recognizes the loss and computes the proportion to be 50% so that it should publish 5 hBTC to be burned from the vault.
      3. However, in the time between the on-chain burning and the computation of the amount, funds arrived in the vault (e.g. Charlie deposited 100 assets).
      4. Now, the loss is realized and the vault loses 5 hBTC.
      5. Finally, when Bob claims the withdrawal, he will receive 97.5 hBTC.
      6. Ultimately, the total burned amount is only 7.5 instead of 10 hBTC.

      To summarize, it is impossible to create accurate amounts to burn without the corresponding on-chain logic.

4. **Yield Generated for Tokens Holder.**

   Yield is still generated for users in the tokens holder. This fundamentally violates accounting and breaks the system. Consider the following example:

1. Assume that both Alice holds 10 and Bob holds 90 shBTC at an exchange rate of 1 (i.e. 100 hBTC total assets).
2. Assume that Bob withdraws everything.
3. Assume that an accurate and proportional slashing of 10 hBTC would occur (assuming all of the above done correctly). Meaning Bob could claim 81 hBTC and Alice would 9 hBTC underlying.
4. However, note that the slash is not realized yet. Now, assume that 1 hBTC yield is generated.
5. Ultimately, Alice is eligible for 10 hBTC again. Note that the exchange rate is 1 again. When Bob finally claims, he receives 90 hBTC due to that.
6. Finally, only a slashing of 1 hBTC occured.

To summarize, yield is generated for funds in tokens holder (if they were slashed), which is incorrect.

Summary: Various problems exist with the current approach, breaking the loss realization process in its entirety.

Code corrected:

The issue has been resolved. Now, the tokens holder implements shares logic.

1. The amount of shares withdrawn during the withdrawal/redemption is now passed.
2. No self-donation is performed.
3. Losses are now more fairly and more accurately accounted for.
4. No yield is generated for users in cooldown.

6.2 Incorrect Early Exit Fees

Correctness | High | Version 1 | Code Corrected

CS-HBTC-001

`StakingVault.claimWithdraw()` allows users to finalize their withdrawal. When early exits are enabled (`earlyExitEnabled == true`), the transfer of fees has been implemented incorrectly which leads to a severe loss of unclaimed user funds.

More specifically, the fees and the amount for the user are computed as follows:

```
function getEarlyExitAmount(...) public view returns (uint256 fee, uint256 withdrawAmount) {
    ...
    withdrawAmount = (assets * feePercent) / BPS;
    fee = assets - withdrawAmount;
}
```

When transferring, the transfers occur as follows:

```
} else if (earlyExitEnabled) {
    (uint256 fee, uint256 withdrawAmount) = getEarlyExitAmount(...);
    tokensHolder.withdraw(receiver, withdrawAmount);
    tokensHolder.withdraw(earlyExitFeeRecipient, assets - fee);
```

Note that the second withdrawal does not transfer `fee` but `assets - fee` which is equal to `withdrawAmount`.

As a consequence, the following is possible:

- Revert due to a lack of funds in `tokensHolder`.
- Loss of funds held in `tokensHolder` if the contract holds sufficient funds. Note that if the recipient is a governance-controlled address, the issue could be mitigated by manually transferring. However, `earlyExitFeeRecipient` could be `StakingVault` (and is set as such in the constructor). In that case, other `StakingVault` share holders would benefit and treat it as a donation.

Code corrected:

The amount transferred to `earlyExitFeeRecipient` has been set to `fee` instead of `assets - fee`.

6.3 Loss Realization Function Cannot Be Called

Correctness | Medium | Version 4 | Code Corrected

CS-HBTC-030

The function `Minter.realizeLosses()` will fail everytime because the `Minter` tries to burn `hBTC` from the `StakingVault` but does not have the required allowance. As an effect, the dedicated loss realization callpath is completely broken, but losses can still be realized by the owner of the `HilBTC` contract, even though it is not the intended use.

Code corrected:

Now, the `Minter` does not require any approval on `HilBTC` to `burnFrom` any user:

```
if (spender != from && spender != owner() && spender != $.minter) {
    _spendAllowance(from, spender, amount);
}
```

While the issue is resolved, it further highlights the need for access control to be improved, see Unnecessary Complexity in Access Control.

6.4 Users Can Escape Losses

Design | Medium | Version 2 | Code Corrected

CS-HBTC-026

In case they see an unrealized loss coming, users can escape it by redeeming their shares before the loss is realized and buying back the shares at a lower price later. This increases the percentage loss of the other vault share holders, as they now cover the escaped loss.

Consider the following scenario where there is an unrealized loss of 10%. User A holds 90% of all shares in the vault:

1. User A redeems all their shares.
2. The loss is realized.
3. User A deposits again (after cooldown)

Now, user A has a 0% loss, while other users in the vault have suffered a 100% loss.

This increased loss could lead to a situation similar to a bank run, as many users might try to redeem their shares before the loss is realized, leaving the remaining users to take the loss, and making the vault insolvent in the worst cases.

Code corrected:

The code has been corrected. Users cannot frontrun exits. More specifically, if the vault is penalized users with funds held in the tokens holder will be penalized equally.

Note that the solution, however, introduced other problems described in other issues.

6.5 Blacklisted Addresses Can Still Interact

Design | Medium | Version 1 | Code Corrected

CS-HBTC-002

Syntetika requires that a blacklisted address should not be able to move funds at all and that, optimally, the interactions possible should be kept at a minimum. However, that is not fully enforced:

1. The overridden `_update()` in `StakingVault` and `HilBTC` function called upon transfers only enforces that the `from` and `to` addresses are not blacklisted. By not checking that `msg.sender` is also not blacklisted, it allows a blacklisted address to use open allowances they might have, violating the requirement above.
2. Similarly, that in the `StakingVault` contract for functions `deposit()` / `mint()`. `msg.sender` could be blacklisted but the operations would be successful as `receiver` could not be blacklisted.
3. Additionally, the `receiver` for `claimWithdrawal()` could be blacklisted, but the operation could succeed. However, typically it is expected that the blacklist in `HilBTC` prevents this from happening.

Additionally, note that blacklisted addresses could still interact with the code. For example, they can give approvals to other addresses, but transfers will be blocked. Additionally, note that in some cases the blacklist will not be applied (e.g. `HilBTC` when `owner()` is `msg.sender`). However, that is expected to satisfy the requirements.

Code corrected:

The code was updated to ensure that `msg.sender` is not blacklisted in the functions mentioned above.

Note that, in `claimWithdrawal()`, the `receiver` could be blacklisted in `shBTC` but not in `hBTC` (unsynchronized blacklist). Thus, the staking vault could transfer `hBTC` to a blacklisted address.

However, Syntetika confirmed that this is intended since a user could withdraw to self and transfer the `hBTC` to the potentially blacklisted receiver (in `shBTC`).

6.6 Missing Permit Feature

Design | Medium | Version 1 | Code Corrected

CS-HBTC-004

The NatSpec of `HilBTC` mentions that the token implements the "permit functionality" (EIP-2612), but the contract does not implement the functionality.

Code corrected:

The `HilBTC` contract was updated to extend OpenZeppelin's `ERC20PermitUpgradeable`, which implement EIP-2612.

6.7 Proxied Contracts Cannot Set State in Constructor

Correctness | Medium | Version 1 | Code Corrected

CS-HBTC-005

Syntetika specified that the contracts will be deployed behind proxies. But all the contracts have a constructor that sets some state (e.g., roles, addresses, ...), this is incompatible with the use of proxies as the state will be set on the implementation contract, but the proxy storage will be untouched. As a consequence, the system is unusable from the proxies.

Code corrected:

The `HilBTC`, `Minter` and `StakingVault` contracts have been updted to extend OpenZeppelin's `UUPSUpgradeable`. Their constructors have been replaced with initializer functions and they override the `_authorizeUpgrade()` function. The `Blacklistable` contract was renamed `BlacklistableUpgradeable` and updated to inherit from the upgradeable version of its dependencies. The `ComplianceCheckerUpgradeable` contract has been added, it clones the core logic of the `ComplianceChecker` while being abstract and upgradeable.

6.8 Inconsistent Storage Locations

Design | Low | Version 2 | Code Corrected

CS-HBTC-027

The contracts `StakingVault`, `Minter`, `HilBTC`, `ComplianceCheckerUpgradeable`, and `BlacklistableUpgradeable` currently define their storage variables in a linear layout, without using the ERC-7201 storage slot standard. On the other hand, some of their parent contracts (e.g., `AccessControlUpgradeable`) uses ERC-7201.

This inconsistency in storage layout approaches may introduce potential upgradeability risks. Specifically, future upgrades may inadvertently overwrite storage variables or cause storage collisions.

Code corrected:

All the upgradeable contracts use EIP-7201 now.

6.9 Arbitrary Challenging Problems

Correctness | Low | Version 1 | Code Corrected

CS-HBTC-007

`CompliantDepositRegistry.challengeLatestBatch` should allow the canceller to drop the latest batch of addresses. While `challengeLatestBatch()` ensure that, `challengeLatestBatch(uint256 length)` does not consider the number of finalized addresses.

This leads to multiple problems:

1. *Partial Dropping*: It is possible to drop the unfinalized addresses partially. However, the implementation does not give sufficient flexibility.
2. *Dropping Finalized Addresses*: There is no enforcement that only unfinalized addresses can be dropped. Thus, an investor might have already claimed an address. As a consequence, a user might have his deposit address changed (or deleted). Additionally, this can lead to reverts and other problems.
3. *Dropping Initial Address*: The initial address might be dropped which should not occur.

Ultimately, several problems regarding the dropping of arbitrary entries exist.

Code corrected:

Partial dropping has been removed, only the full pending batch (`depositAddresses.length - finalizedAddressesLength`) can be challenged and dropped.

6.10 Incorrect Event Emissions

Correctness | Low | Version 1 | Code Corrected

CS-HBTC-011

`StakingVault._redeemTo()` emits the `Unstaked(address indexed user, uint256 assets)` event as follows:

```
emit Unstaked(msg.sender, shares);
```

Note that `shares` is emitted instead of `assets`.

Code corrected:

The code has been updated to emit the `assets`.

6.11 Incorrect DEAD_SHARES Logic

Correctness | Low | Version 1 | Code Corrected

CS-HBTC-012

The `StakingVault.deposit()` function burns shares for the first deposit as follows:

```
// burn shares on first deposit
if (totalSupply() == 0) {
    _checkOwner();
    super.deposit(DEAD_SHARES, BURN);
}
```

However, note that `super.deposit()` takes `assets` as an argument. That leads to the following amount of shares being created (and in this case "burned"):

```
shares = DEAD_SHARES.mulDiv(totalSupply() + 10 ** _decimalsOffset(), totalAssets() + 1, Math.Rounding.Floor)
       = 1000.mulDiv(0+1, totalAssets() + 1, Math.Rounding.Floor)
       = 1000 * 1 / (totalAssets() + 1)
```

Note that this not imply directly that `shares` is `DEAD_SHARES` since `totalAssets()` can be manipulated through donations.

Consequently, consider the following:

- `distributeYield()`: The check `totalSupply() > DEAD_SHARES` is inaccurate.
- `_withdraw()`: The check `totalSupply() - shares == DEAD_SHARES` is inaccurate.

While it is unlikely for the scenarios to occur they might still lead to incorrect executions and unexpected results.

Code corrected:

Now `deposit()` invokes `super.mint()` to mint the exact amount of dead shares. While this resolves the problem, the solution still leads to an ERC-4626 violation, see ERC-4626 Violations.

6.12 StakingVault Has No Initial Blacklister

Informational **Version 1** **Code Corrected**

CS-HBTC-023

`StakingVault` does not set the initial blacklister while `HilBTC` does.

Code corrected:

An initial blacklister is now set in the `StakingVault`'s `initialize()` function.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Events Emitted Even When State Is Not Updated

Informational | Version 1 | Acknowledged

CS-HBTC-014

Events should be emitted on each important storage update of a smart contract in order to allow external observers to track important events in the contract's life. Emitting an event when a value is replaced by itself or when no other important update was made can be avoided as no new information is gained, and also incurs an unnecessary gas cost. Below is a non-exhaustive list functions emitting such events:

- `Blacklistable.updateBlacklister()`
- `Blacklistable.blacklist()`
- `Blacklistable.unblacklist()`
- `Whitelist._whitelistAddress()`
- `Whitelist._setComplianceChecker()`
- all the `Minter.setXYZ()`
- `HilBTC.setMinter()`
- all the `StakingVault.setXYZ()`

Acknowledged:

Syntetika is aware of this and chose to not modify the code.

7.2 Gas Optimizations

Informational | Version 1 | Acknowledged

CS-HBTC-015

Below is a non-exhaustive list of potential gas optimizations:

1. In `CompliantDepositRegistry.registerDepositAddress()`, the `ComplianceChecker.requireCompliant()` is called twice when the investor receives a deposit address. Note that one such call is in `getDepositAddress()`.
2. Often access control is checked twice. For example, in the execution of `CompliantDepositRegistry.challengeLatestBatch()` the role checks are executed twice. Similarly, that is the case in `Minter.setOperator()`, `Minter.setPauser()`, `Minter.setCustodian()`, `HilBTC.setMinter()` and `StakingVault.setDistributor()` (hidden check in `revokeRole()`).
3. `Minter.stakingVault` could be immutable.

Acknowledged:

Syntetika is aware of this and chose to not modify the code.

7.3 Inconsistent Revert Statements

Informational | Version 1 | Acknowledged

CS-HBTC-016

Throughout the codebase, reverts and error raising are done in two different manners:

```
if(!cond) {
    revert CustomError();
}
```

and

```
require(cond, CustomError());
```

For the sake of code clarity and maintainability, it is recommended to use only one of the ways.

Acknowledged:

Syntetika is aware of this and chose to not modify the code.

7.4 Inconsistent Transfer in TokenHolder

Informational | Version 1 | Acknowledged

CS-HBTC-017

`TokenHolder.withdraw()` allows the staking vault to withdraw from the escrowed funds.

Note that `SafeERC20.safeTransfer()` is not used. While for HBTC this is not relevant, it contradicts the `StakingVault` which consistently uses `SafeERC20` for the same asset.

Ultimately, `StakingVault` suggests that the contract that the system is designed with some flexibility in mind. However, the `TokenHolder` contract is not consistent with that.

Acknowledged:

Syntetika is aware of this and chose to not modify the code.

7.5 Interfaces Problems

Informational | Version 1 | Code Partially Corrected

CS-HBTC-018

Below is a non-exhaustive list of functions missing in their respective interfaces:

1. `CompliantDepositRegistry`: `challengeLatestBatch(uint256 length)` is missing the interface definition.
2. `Whitelist`: `isAddressWhitelisted()` and `manualWhitelist()` are missing in the interface definition.
3. `TokensHolder`: `withdraw()` is missing in the interface definition.
4. `HilBTC` / `StakingVault` / `Minter`: Various functions are missing in the respective interfaces.

Additionally, the `IMinter` defines event `Deposit` which is never used.

Code partially corrected:

1. Corrected: the function was removed.
2. Not corrected.
3. Not corrected.
4. Not corrected.

The `Deposit` event was removed from the `IMinter` interface.

7.6 Lack of Events

Informational **Version 1** **Acknowledged**

CS-HBTC-019

Note that various constructors are missing event emissions. That includes:

1. `Minter`
2. `HilBTC`
3. `StakingVault`

Additionally, one could argue that `CompliantDepositRegistry` lacks an event for the invalid initial item.

Acknowledged:

Syntetika is aware of this and chose to not modify the code.

7.7 NatSpec Problems

Informational **Version 1** **Code Partially Corrected**

CS-HBTC-021

Below is a non-exhaustive list of functions missing NatSpec:

1. `CompliantDepositRegistry.challengeLatestBatch(uint256 length)` is missing NatSpec documentations.
2. `ICompliantDepositRegistry` does not annotate the getter functions with full NatSpec (e.g. `DEPOSIT_ADDRESS_CREATOR_ROLE()`)
3. NatSpec is missing for various constructors. Similarly, that is the case for events and custom errors.

4. While it generally it is fine not to provide NatSpec for internal and private functions, the NatSpec is inconsistent in that regard.
5. `Minter.constructor()`: `_distributor` and `_stakingVault` are undocumented.
6. `Minter.distributeYield()` / `Minter.pause()` / `Minter.unpause()`: No NatSpec provided.
7. `Minter.redeem()`: The NatSpec mentions redemption for "a specified address", but the function only accepts one `amount` parameter and the redemption is always done for `msg.sender`.
8. `StakingVault.distributeYield()`: `timestamp` is undocumented.
9. `StakingVault.getEarlyExitAmount()`: No NatSpec provided.

Further, note that `@inheritdoc` could be used to not need to copy-paste NatSpec from interface definitions.

Code partially corrected:

1. Corrected. The function has been removed.
2. Not Corrected. NatSpec is inconsistently implemented for external/public getter functions.
3. Not Corrected. NatSpec is inconsistently implemented for contrustor/initializer, errors and events.
4. Not Corrected. NatSpec is inconsistently implemented for internal/private functions.
5. Corrected. NatSpec updated.
6. Corrected. NatSpec added.
7. Corrected. NatSpec updated.
8. Corrected. NatSpec updated.
9. Corrected. NatSpec added.

7.8 Sanity Checks

Informational **Version 1** **Acknowledged**

CS-HBTC-022

The codebase performs sanity checks on various occasions. Thus, the lack of some checks implies an inconstency:

1. No checks against `0x0`:
    1. `CompliantDepositRegistry.constructor()`
    2. `ComplianceChecker.constructor()`
    3. `TokensHolder.constructor()`
    4. `StakingVault.constructor()` (`_distributor`)
2. `TokensHolder.constructor()` could retrieve `HILBTC` from `STAKING_VAULT.asset()`.
3. No `0x0` checks for the SBTs in the compliance options when they are added in `ComplianceChecker.setComplianceOptions()`.

Note that ultimately, the sanity checks could be more consistent.

Acknowledged:

Syntetika is aware of this and chose to not modify the code.

7.9 StakingVault Has No Initial Early Exit Fee

Informational | Version 1 | Acknowledged

CS-HBTC-024

The `maxEarlyExitFeeBps` storage variable in the `StakingVault` is not set in the constructor and the `setEarlyExitEnabled()` function does not check that it is non-zero. While a 0-value fee is a valid configuration, Syntetika needs to check and update the configuration of the contract prior to enabling early exits if they want to charge a fee.

Acknowledged:

Syntetika is aware of this and chose to not modify the code.

7.10 Unbound Array in Compliance Checker

Informational | Version 1 | Acknowledged

CS-HBTC-025

`ComplianceChecker._complianceOptions` is a two-dimensional array that is unbound in both dimensions. Note that operations could revert or become inefficient if the array is sufficiently large.

Acknowledged:

Syntetika is aware of this and chose to not modify the code.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Cooldown Increases for All Pending Withdrawals

Note Version 1

Users should be aware that the cooldown end increases for all pending withdrawals. Consider the following scenario:

1. Alice withdraws 10 assets. The cooldown is 10 days.
2. After 5 days, she withdraws 10 assets again. The cooldown is still 10 days.
3. Only after a total of 15 days can she withdraw her 20 assets. Note that the first withdrawal is hence additionally delayed.

8.2 Losses Can Be Front-Run When Cooldown Is Small

Note Version 3

When the cooldown period is small or even 0, it is possible for users to observe the mempool, or the on-chain state of the strategy if any, in order to front-run an incoming loss and escape it. Syntetika is expected to set a cooldown period that reflects the risk of the strategy to ensure fair operations in the vault.

8.3 Owner Special Cases

Note Version 1

The `owner` has special powers for token related operations in `HilBTC`. More specifically, they can call `burnFrom()` without needing allowance and even transfer from and to blacklisted addresses. Note that this is intended to be used for seizing funds and covering strategy loss scenarios.

Note that `StakingVault` does not implement such functionality intentionally due to weaker legal requirements.

8.4 Partial EIP-165 Support

Note Version 2

The contract `AccessControl[Upgradeable]` implements the EIP-165 (https://eips.ethereum.org/EIPS/eip-165) for interface detection. However, the contracts extending `AccessControl[Upgradeable]` (`ComplianceChecker`, `ComplianceCheckerUpgradeable`, `Minter`, `HilBTC`, `StakingVault`) do not extend the support by overriding the `supportsInterface()` function, limiting the interface detection to `AccessControl[Upgradeable]`.

8.5 Redeeming HilBTC

Note Version 1

Consider the following scenario:

1. Alice deposits by calling `Minter.mint()`. Note that this increases `totalDeposits`.
2. In a separate mechanism, Bob deposits to his deposit address. The owner initiates `Minter.ownerMint()` to mint the respective amount accordingly. That increases `totalDeposits`.
3. After some time Bob decides to withdraw. However, he does so by calling `Minter.redeem()`.
4. Alice can now not `redeem()` her HilBTC as they are backed by BTC on Bob's deposit address.

Users should be aware that:

- The above is intended.
- Syntetika owns and controls the deposit addresses.
- Off-chain rebalancing will be performed to ensure that funds can be accessed.
- If an address wants to receive BTC on Bitcoin, they will call `burn()`.

However, note that this mechanism is not described nor specified in detail and that it is expected to work correctly and that sufficient data is present for the mechanism to work properly.

8.6 Temporary DoS and Griefing of Deposit Registry

Note Version 1

Investors can claim deposit addresses with `CompliantDepositRegistry.registerDepositAddress()`. However, Syntetika should be aware that there might be temporary DoS and griefing possibilities.

More specifically, the set of deposit addresses is limited and needs to be pushed by a privileged role. The addresses can be claimed by addresses that satisfy the compliance requirements. However, these addresses could still be malicious with the intent to disturb operations. Hence, they could claim all unclaimed deposit addresses which leads to:

- Temporary DoS: To publish new addresses the time delay needs to be satisfied. Hence, a waiting time might be enforced.
- Griefing: Syntetika must pay for gas fees to publish the addresses. Thus, claiming addresses unnecessarily might lead to higher operating cost.

Nonetheless, such scenarios are unexpected since:

- The set of compliant addresses is expected to be small.
- Each compliant address can claim one deposit address at most.

To summarize, while unlikely it could still be possible to disturb the operations in the deposit registry.

8.7 User DoSed by Minima

Note Version 1

Note that users can be DoSed by the minimum `minAssetsAmount`. Below is a non-exhaustive list of example scenarios:

- A user has shares that have a value of 100 tokens. Then, the minimum is increased to 150 tokens. The user cannot withdraw.
- A user has shares worth 100 tokens and transfers 60 to another user. If the minimum is 50 tokens, the user cannot withdraw.
- A user sees a minimum of 10 tokens and wants to deposit 20. However, another pending transaction increasing the minimum to 30 is executed before. The user is griefed. Note that this scenario typically is not problematic as the user can always re-execute. However, integrators should not rely on being able to deposit any amount.

8.8 Vault Integration Considerations

Note Version 1

Syntetika, users and integrators should be aware that classical manipulation attacks are possible for `StakingVault`. However, these are limited by the underlying vault contract used (i.e. `ERC4626`) and the dead shares mechanism.

More specifically, all relevant parties should be aware that:

1. Share price manipulations are possible (e.g. donations to the contract).
2. Consequently, inflation attacks are possible.
3. Therefore, second depositor attacks are possible.

To summarize, all related parties should do their due diligence and ensure the safety of using the `StakingVault`.