PUBLIC

Code Assessment

of the Box Smart Contracts

December 15, 2025

Produced for

Steakhouse Financial

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Steakhouse team,

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

Steakhouse implements a modular vault system centered around `Box`, an ERC-4626 child vault that holds a base asset, invests in whitelisted ERC-20 tokens, and interacts with lending protocols through funding modules. Adapters connect `Box` to a parent Vault V2 by Morpho, enabling controlled deposits, withdrawals, and allocations. The system includes factories for deploying components and enforces role-based permissions and timelocks for managing operations and assets.

The most critical subjects covered in our audit are access control, integration with lending protocols, permissionless functions during winddown and slippage protection. We identified issues regarding insufficient input validation in the funding modules, see Funds can be locked in FundingAave during winddown and FundingMorpho.depledge() does not sanitize collateralToken. Moreover, the `swapper` selection by an allocator or by any user during winddown can be abused to extract value from the `Box` due to a lack of reentrancy protection, see Read-only reentrancy. These issues were addressed and fixed in the second version of the codebase. Security regarding all the aforementioned subjects is good.

The general subjects covered are timelock mechanics, shutdown procedure and functional correctness. Security regarding correctness is improvable, see Box cannot receive native currency. Security regarding all the remaining topics is high.

In summary, we find that the codebase provides a good 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	3
• Code Corrected	3
Medium-Severity Findings	7
• Code Corrected	5
• Risk Accepted	2
Low-Severity Findings	11
• Code Corrected	8
• Specification Changed	1
• Risk Accepted	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 Box 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	06 Oct 2025	543fd10269dc81693af9406d1e97952d791e9af7	Initial Version
2	27 Oct 2025	d4302fe8665f913f3e448a9fb82d10d3e86b544f	First Fixes
3	08 Dec 2025	d4fb13efd6445365d72beb10cac0f202b9431cb0	Second Fixes
4	15 Dec 2025	80a5779f326af2a99fb9ca617701c7057b2f70ff	Final Version

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

The following files are in the scope of this review:

```
src:
    Box.sol
    BoxAdapter.sol
    BoxAdapterCached.sol
    FundingAave.sol
    FundingMorpho.sol

    factories:
        BoxAdapterCachedFactory.sol
        BoxAdapterFactory.sol
        BoxFactory.sol
        FundingAaveFactory.sol
        FundingMorphoFactory.sol

    libraries:
        Constants.sol
        ErrorsLib.sol
        EventsLib.sol
```

After Version 3, the scope was updated as follows:

Added:

```
FundingBase.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, libraries related to `openzeppelin-contracts` and Morpho Vault V2, as well as the integrated protocols such as Morpho Blue and AaveV3 are expected to work as intended and are out of the scope of this review. Governance attacks on the `guardian` and implementation of the price oracles are out of the scope of this review. The system is expected to work only with ERC20 tokens without special behaviors such as reentrant tokens or tokens with transfer fees.

2.2 System Overview

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

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

Steakhouse implements `Box` and adapters for `Box`, together with funding modules and factories. `Box` is an ERC-4626 vault that holds a base asset, invests in other ERC-20 tokens, and can borrow or lend via funding modules. `Box` is designed to act as a child vault for a Vault V2 by Morpho. The parent vault interacts with `Box` through `BoxAdapter` or `BoxAdapterCached`.

2.2.1 Box

`Box` holds a primary asset and allows privileged allocator roles to invest in a whitelist of other tokens and borrow or lend through external funding modules. The vault is not intended for direct public use; deposits are restricted to whitelisted feeders, and operations are governed by `curator`, `guardian`, and `owner` roles with timelocked actions. Steakhouse intends for `BoxAdapter` and `BoxAdapterCached` to be the sole feeders in the system. Capabilities of each role are described in detail under the Trust Model section. `Box` also includes features such as slippage protection, shutdown, and a permissionless winddown process.

Swaps between the primary token and investment tokens, as well as between two investment tokens, can be executed by an `allocator` through `allocate()`, `deallocate()`, and `reallocate()`. Moreover, they can post collateral to a lending facility, pull the collateral, borrow, and repay debt by calling `pledge()`, `depledge()`, `borrow()`, and `repay()`. The `curator` can add or remove tokens from the whitelist for all the aforementioned operations. Whitelisting tokens is subject to a delay.

The following operations are timelocked and can be submitted to the timelock queue only by `curator`. Timelock delays are stored in a mapping called `timelock`. This mapping should be populated upon contract deployment; otherwise, `curator` could submit an operation and finalize execution within the same transaction:

- setting a new guardian
- decreasing the timelock delay for a function
- adding or removing a `feeder`
- setting the maximum allowed slippage, capped to 1%
- whitelisting a token for `Box`
- changing a token oracle, unless the system is in final winddown
- whitelisting a funding module
- whitelisting a funding facility
- whitelisting a token as collateral for a funding module

- whitelisting a token as a loan token for a funding module

Except for setting a new guardian, decreasing the timelock delay, and whitelisting a token as collateral, these operations can later be executed permissionlessly by anyone after the corresponding delay, unless `guardian` or `curator` removes them from the queue. The `curator` can remove tokens, funding modules, funding facilities, debt tokens and collateral tokens from their respective whitelists without delay. Note that in order to de-whitelist a token or a funding module, `Box` ensures that it doesn't hold any such token and that the funding module to remove is empty.

This contract employs a dual slippage protection mechanism for asset swaps. Each trade is validated against a maximum slippage tolerance based on oracle prices, reverting if the received amount is lower than expected. Additionally, realized slippage is normalized against the vault's total NAV and added to an `accumulatedSlippage` counter for the current epoch. If this value exceeds `maxSlippage` within `slippageEpochDuration`, the current swap and subsequent ones are blocked until the epoch resets. Epochs reset upon a call to the contract, leading to variable epoch durations. The same `maxSlippage` value is used for local and global slippage protection. Slippage tolerance increases linearly from 0% to 1% once the final winddown process starts. Moreover, the per-epoch slippage tolerance limit is ignored during winddown.

The shutdown process can be triggered by `curator` or `guardian`. This immediately blocks new deposits and begins a `shutdownWarmup` period during which the action can be cancelled by the `guardian`. Once the warmup period ends, the vault enters the irreversible winddown phase, a permissionless process designed to close all positions. During winddown, the allowed slippage for swaps increases linearly to 1% over `slippageEpochDuration`. The actions that can still be executed permissionlessly during winddown are:

- withdraw and redeem for investors to claim assets
- swap the primary asset for a token if there is debt for the token
- swap a token held by the vault for the primary asset if there is **no** debt for the token
- withdraw collateral from a lending facility
- repay borrowed tokens to a lending facility
- execute a flash operation

An `allocator`, or anyone during winddown, can let `Box` temporarily use its funds for complex operations by calling `Box.flash()`. After the caller grants an ERC-20 allowance, `Box.flash()` is called, the current NAV is cached and the specified amount of tokens are pulled from the caller. `Box` then executes a callback, allowing the caller to execute a sequence of actions such as swaps or interacting with lending modules. Once the callback is finished, the function returns the initial tokens to the caller via a direct transfer. During a flash operation, calls to `totalAssets()` revert, which blocks deposits and withdrawals to or from `Box` until the flash operation completes.

2.2.2 BoxAdapter and BoxAdapterCached

The parent vault (Vault V2 by Morpho) interacts with `Box` through `BoxAdapter` or `BoxAdapterCached`. The parent vault, the adapter, and `Box` all use the same underlying asset. Upon deployment, both the parent vault and `Box` are granted infinite allowance for that asset. `realAssets()` is computed by calling `Box.previewRedeem()` using the total balance of `Box` shares held by the adapter as input. `BoxAdapterCached` stores the last computed value instead of recalculating it on every call. An update is triggered whenever the parent vault allocates or deallocates, or when an allocator or sentinel of the parent vault calls `BoxAdapterCached.updateTotalAssets()`. Additionally, anyone can trigger an update if 24 hours have passed since the last one.

Whitelisted users first bring funds into the system through `VaultV2.deposit/mint()`. From there, assets can be allocated into `Box` through the adapter, or withdrawn back to the parent vault when needed. The movement of funds into `Box` works as follows:

1. Whitelisted users call `VaultV2.deposit()` to add the base asset into the parent vault.

2. An allocator of VaultV2 calls VaultV2.allocate() with the adapter and a specified amount of assets as input. This transfers the assets from VaultV2 to the adapter and invokes BoxAdapter.allocate().
3. As a whitelisted feeder, BoxAdapter then calls Box.deposit(). At that point, Box pulls the assets from the adapter and mints the corresponding amount of Box shares to the adapter.

When funds need to be returned from Box back to VaultV2, the following steps take place:

1. An allocator or sentinel of VaultV2 calls VaultV2.deallocate() with the adapter and a specified amount of assets. Users can also do it through VaultV2.forceDeallocate().
2. BoxAdapter calls Box.withdraw(), causing Box to send the assets back to the adapter and burn the corresponding amount of Box shares from the adapter.
3. Control flow then returns to VaultV2, which pulls the assets from the adapter.

It is worth mentioning that only the adapter holds Box shares.

2.2.3 Funding Modules

Funding modules are smart contracts used by a Box instance to interact with external lending protocols such as Aave and Morpho. Box delegates actions like pledge(), depledge(), borrow(), and repay() to a funding module, which then performs token approvals, protocol calls, and facility-specific checks (e.g., market params, collateral/debt token whitelists). Both funding modules include logic to add or remove facilities from the whitelist. FundingMorpho interacts with different Morpho Markets V1 by registering them as facilities. In the case of FundingAave, this mechanism cannot be used because Aave V3 does not provide isolated markets.

Both modules compute the loan-to-value ratio by using the formula borrowed_amount / collateral_amount, without taking into account the underlying pool or market's LLTV. In FundingAave, the input to ltv() is disregarded, whereas in FundingMorpho the input represents the encoded facility data (market params). FundingAave reports its NAV by querying the Aave pool and computing borrowed - collateral. If the borrowed value exceeds the supplied collateral, the reported NAV is set to 0. FundingMorpho iterates over each facility (market) and computes the difference between borrowed and collateral for each one. If the facilityNAV is negative, it is treated as zero. Additionally, FundingMorpho has an immutable lltvCap that further restricts the LLTV of each Morpho market as a multiplicative factor.

Box must be the owner of any deployed funding module due to the design of FundingMorpho and FundingAave. The owner of a funding module cannot be changed after deployment. The process for adding or removing facilities in FundingMorpho is as follows. To add a facility, the owner must first whitelist the underlying collateral and debt tokens. To remove a facility, the owner must first pull all collateral and repay all debt (i.e., close the position). To remove a collateral token, the owner must first remove all facilities that use that token as collateral. To remove a debt token, the owner must first remove all facilities that use that token as debt. FundingAave does not enforce these checks, other than preventing duplicate tokens in whitelists.

2.2.4 Factories

There are five factory contracts: BoxFactory, BoxAdapterFactory, BoxAdapterCachedFactory, FundingAaveFactory and FundingMorphoFactory. Each factory exposes a public method to deploy the corresponding smart contract. BoxFactory deploys a new Box instance using the CREATE2 opcode, and its createBox() function accepts a salt as a parameter. The other factories use the CREATE opcode instead.

Each factory maintains a public mapping that can be queried with an address to check whether it corresponds to a contract deployed by that factory. In addition, BoxAdapterFactory and BoxAdapterCachedFactory each hold a mapping that, given the parent vault address (Morpho Vault

V2) and the Box address, returns the address of the deployed BoxAdapter or BoxAdapterCached respectively.

2.2.5 Changes in Version 2

In Version 2, the following changes were made to the system:

- Skim functionality was added to both funding modules and to Box itself. Box can now invoke the skim method on each funding module.
- Remaining funds are sent back to Box if FundingAave.repay() does not utilize the full repayment amount.
- In addition to flash operations, NAV is now cached for swaps. This uses a depth-based mechanism that allows flash operations to occur within a swap operation.
- Virtual shares are introduced when the underlying token has fewer than 18 decimals.
- A multicall function was added to Box and the funding modules.
- Skim functionality for native token was added.

2.2.6 Changes in Version 3

In Version 3, the following changes were made to the system:

- Only the curator can execute changeTokenOracle() when the Box is not winding down.

2.3 Trust Model

Privileged roles for Box are owner, guardian, curator, allocator, feeder and skimRecipient. Additionally, Box exposes some methods that can be called permissionlessly by anyone, especially during winddown mode.

- **owner**: Semi-trusted. Can set the curator. If malicious, can acquire curator and then allocator privileges without a delay and swap all the funds for a malicious token. To this end, the malicious token must be first whitelisted, which is subject to a timelock and revocation by the guardian. If the token is already whitelisted, this can be achieved within a single transaction. Additionally, they can set the skimRecipient and transfer the ownership.
- **guardian**: Semi-trusted, they have priority over the curator. A malicious guardian can start the shutdown process, which can only be cancelled by them. Moreover, they can change the token oracle to any address for a whitelisted token after the shutdownSlippageDuration, during which anyone can call allocate() if the token has an open debt position, resulting in loss of funds or mispriced shares.
- **curator**: Semi-trusted. They can whitelist any address as allocator and swap the base asset for any whitelisted investment token within the same transaction. They can DOS the system by irreversibly abdicating the timelock for any method that is subject to a timelock without any delay. For completeness, they submit operations that are subject to the timelock via submit() as described in Box and they can perform the following with no delay:
  - cancel a pending timelocked operation
  - increase timelock delays (does not affect already pending operations)
  - permanently disable a function that is subject to the timelock
  - add or remove an allocator from the whitelist
  - initiate the shutdown process
  - remove a token from the whitelist
  - remove a funding module

    - remove a funding facility from a funding module
    - disable a token as collateral in a funding module
    - disable a token for borrowing in a funding module
- **allocator**: Semi-trusted. They are expected to rebalance the tokens in the best interest of the LPs. In the worst case, they can extract value up to the allowed maximum slippage. They can perform the following operations:
    - swap base assets for whitelisted investment tokens unless winddown has started (may choose any swapper address to call `sell()` on)
    - swap whitelisted investment tokens for base assets unless winddown has started (may choose any swapper address to call `sell()` on)
    - swap investment tokens for investment tokens unless winddown has started (may choose any swapper address to call `sell()` on)
    - post collateral through any whitelisted funding module unless winddown has started
    - withdraw collateral through any whitelisted funding module
    - take out a loan through any whitelisted funding module unless winddown has started
    - repay a loan through any whitelisted funding module
    - execute flash operations (may pick any `flashToken` that is neither a whitelisted investment token nor the base asset)
- **feeder**: Semi-trusted. They can deposit assets for new shares of `Box` unless during shutdown. They can perform inflation attacks on an empty vault. They are expected to be some `VaultV2` adapters that can be allocated only by trusted allocators.
- **skimRecipient**: Semi-trusted. They can pull any token from `Box` that is **neither whitelisted nor the base asset**. If the `Box` receives an airdrop due to a holding of its own and the received airdrop token is not whitelisted yet, `skimRecipient` could steal it.
- **users**: Not trusted. They can perform the following operations:
    - execute a timelocked operation after the delay has passed, except for setting a new guardian, decreasing the timelock delay, and whitelisting a token as collateral
    - withdraw assets if they hold the corresponding `Box` shares or were granted allowance by the share owner. Considering box adapters are the only entities holding `Box` shares, this does not apply.
    - swap the base asset for a whitelisted investment token during winddown if there is non-zero debt for the token (may choose any swapper address to call `sell()` on)
    - swap a whitelisted investment token back to the base asset during winddown if there is zero debt for the token (may choose any swapper address to call `sell()` on)
    - withdraw collateral through any whitelisted funding module during winddown
    - repay borrowed tokens through any whitelisted funding module during winddown
    - execute flash operations during winddown (may pick any `flashToken` that is neither a whitelisted investment token nor the base asset)

Privileged roles for both adapters are the owner of the parent vault, the `skimRecipient`, and the parent vault itself. For `BoxAdapterCached`, allocators and sentinels of the parent vault have one additional permission:

- **owner of the parent vault**: can set the `skimRecipient` to an arbitrary address
- **skimRecipient**: can pull any token from the adapter besides `Box` shares

- **parent vault**: can call `allocate()` and `deallocate()`. Since the adapter acts as a `feeder` in `Box`, this allows the parent vault to deposit into and withdraw from `Box`
- **allocator or sentinel of the parent vault** *(only for BoxAdapterCached)*: can update the cached total assets of `BoxAdapterCached`

All the aforementioned roles are semi-trusted in Steakhouse's system. However, the roles described by the parent vault may have additional trust assumptions with respect to the parent vault itself. Since parent vault implicitly has the `feeder` role in Steakhouse's system, an `allocator` of the parent vault can execute inflation attacks on `Box`.

Each funding module has a single privileged role, namely `owner`. Steakhouse intends to set this role as a `Box` instance. They are semi-trusted during normal operations, as the interaction between `Box` and each facility is limited to the allocators of `Box`. During winddown, anyone can call `Box.depledge()` to move collateral from the underlying market to `Box` and call `Box.repay()` to (partially) close a debt position. For completeness, the `owner` of a funding module can perform the following operations:

- add or remove facilities from the whitelist
- add or remove collateral/debt tokens from the respective whitelists
- call `pledge()`, `depledge()`, `borrow()` and `repay()`

Factory contracts do not have any privileged roles and `createXYZ()` can be called by anyone.

Other assumptions:

- the `guardian` is expected to be an `AragonGuardian` where the LPs of the `VaultV2` can take governance decisions such as revoking an action, recovering the `Box`, or updating an oracle after the `shutdownSlippageDuration`.
- Oracles will never revert. If an oracle reverts, `Box` is temporarily DOSed.
- Oracles should return the price of a token in the underlying asset denomination.
- Tokens with fees on transfer are not supported.
- Allocators of the `VaultV2` and `Box` are expected to manage the free liquidity to allow for smooth user operations.
- Allocators and sentinels of the `VaultV2` are expected to call `BoxAdapterCached.updateTotalAssets()` often enough and as soon as some important events change the share price in order to avoid arbitrage opportunities.

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:

- 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	2

- BoxAdapterCached Can Be Arbitraged Risk Accepted
- Gas Griefing Problems Risk Accepted

Low-Severity Findings	2

- Effects of Function Abdication Are Not Immediate Risk Accepted
- LTV of FundingAave Can Report More Than Managed Collateral Risk Accepted

5.1 BoxAdapterCached Can Be Arbitraged

Design Medium Version 1 Risk Accepted

CS-STKHSBX-003

`BoxAdapterCached` uses a caching mechanism to prevent expensive `_nav()` calculations in `Box` by storing a `totalAssets` value. An arbitrage possibility is introduced by this 24 hours caching period:

A user can monitor the `Box`'s total assets (by calling `totalAssets()`) and compare the value with the cached `totalAssets` in `BoxAdapterCached` (by calling `realAssets()`). If the difference between the two values is high, the user can deposit or withdraw funds into the Morpho VaultV2, which will use the cached `totalAssets` when calculating the value of a vault share, effectively extracting value from the other vault investors.

For example, if the `Box` made a significant profit in the past 24 hours, a user can deposit money into the VaultV2 before the increase in assets from the `Box` is reflected in the `BoxAdapterCached`. They will realize a profit as the value of their VaultV2 shares will increase once the cached value is updated. In case of a significant loss, a user can do the opposite and sell VaultV2 shares before the loss is reflected in the value of the shares.

Note that the impact of this issue can be reduced by `sentinel` or `allocator` calling `BoxAdapterCached`'s `updateTotalAssets()` in such situations to force the update of the cached `totalAssets` value.

Risk accepted:

Steakhouse accepted the risk.

5.2 Gas Griefing Problems

Design | Medium | Version 1 | Risk Accepted

CS-STKHSBX-007

In multiple places in the codebase, the logic makes sure that a token balance is empty. This opens the way to gas griefing attacks, as an attacker could send small amounts of that token to disrupt operations.

1. `Box.removeToken()` enforces a zero-balance in the `Box` for the token to be removed. An attacker can send a small amount of the token before the `removeToken()` call is done. This will force the allocators to swap it, incurring additional costs. The comment at line 34 states that this can be avoided by first deallocating and then removing the token atomically. This is not always possible as, if there is slippage, it could lead to a 100% slippage that would go over the maximum allowed slippage, making the swap impossible. Allocators would need to either inject liquidity themselves, or swap to the target token first and then swap back. Note that this attack can be repeated.
2. Some tokens or positions can be whitelisted, while being not currently used. In the different `nav()` functions, if the balance of such tokens is 0, then no further computation is done for it. Otherwise, a call to a price oracle and other computations are done. An attacker can send a small amount for each whitelisted but unused tokens, forcing more calls and computations. This could make the `nav()` computations very costly.
3. The function `_isFacilityUsed()` in the two funding modules returns `false` only if the facility has no collateral and no debt. An attacker can send a small amount of `aToken` or open a small Morpho position "on behalf of" the module to make the function return `true` and disrupt operations.

Risk accepted:

Steakhouse is aware of this griefing vector and accepted the risk.

5.3 Effects of Function Abdication Are Not Immediate

Design | Low | Version 1 | Risk Accepted

CS-STKHSBX-011

In the case where timelocked calls are pending and the target function is abdicated, those pending calls can still be executed. This can lead to unexpected behaviors as an abdicated function can still be called.

An exception is `decreaseTimelock()`: if the target function was abdicated during the timelock period, `decreaseTimelock()` cannot be called on the target function anymore.

Risk accepted:

Steakhouse accepted the risk and added the following comment before `Box.abdicateTimelock()`:

```
@dev Does not impact previsously queued changes
```

5.4 LTV of `FundingAave` Can Report More Than Managed Collateral

Design | Low | Version 1 | Risk Accepted

CS-STKHSBX-008

The `FundingAave.ltv()` function computes `collateral / debt`, but the `collateral` side can be inflated with donations, either in whitelisted tokens, or in other tokens that won't be recoverable unless whitelisted. Note that in normal mode (not isolated), gifted `aToken` will automatically be activated as collateral. This might make the allocators open debt positions against collateral tokens that are not whitelisted, and thus outside of the desired risk profile of the strategy.

Risk accepted:

Steakhouse accepted the risk and added the following comment before `FundingAave.ltv()`:

```
@dev ltv can also use non whitelisted collaterals (donated)
```

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	3

- Stale Total Assets Value During Flash Operation Code Corrected
- Funds Can Be Locked in FundingAave During Winddown Code Corrected
- Read-only Reentrancy Code Corrected

Medium-Severity Findings	5

- Box Cannot Receive Native Currency Code Corrected
- Discrepancy in Cumulated Slippage Computation Code Corrected
- Excessive Access Control for addFundingCollateral() Code Corrected
- FundingMorpho.depledge() Does Not Sanitize collateralToken Code Corrected
- Box Decimals Fixed to 18 Code Corrected

Low-Severity Findings	9

- Inconsistency in FundingAave facilities() Code Corrected
- ERC-4626 Violations Code Corrected
- Floating Pragma Code Corrected
- Inaccurate Naming, Comments and NatSpec Code Corrected
- Inconsistent and Missing Input Sanitization Code Corrected
- Rounding Direction Code Corrected
- Unreachable Code Code Corrected
- Unused Event Code Corrected
- Wrong LTV on Total Loss Specification Changed

Informational Findings	2

- Missing Constructor Parameter in Event Code Corrected
- Unchecked Block for Loop Iterator Code Corrected

6.1 Stale Total Assets Value During Flash Operation

Correctness High Version 2 Code Corrected

CS-STKHSBX-028

During a `flash()`, the NAV is cached to a avoid read-only reentrancy and `totalAssets()` returns the cached value. The cached value can be abused as it is not updated during `deposit()`, `mint()`,

`withdraw()` or `redeem()` operations, if two such functions are called during a flash, the second call will operate on a wrong `totalAssets` value. Example:

1. Attacker calls `flash()` and the NAV is cached.
2. During the flash callback, `withdraw()` is called (via `VaultV2.forceDeallocate()` for example) and uses a correct (cached) value for `totalAssets`, but the cached value of NAV is not updated with the new correct `totalAssets`. This sets `VaultV2.lastUpdate` to the current `blocktimestamp`.
3. During the flash callback, `withdraw()` is called again and uses the cached NAV, which allows the `VaultV2` to withdraw at an inflated price from the `Box`.

This allows an LP of some `VaultV2` to steal the liquidity of another `VaultV2` participating in the same `Box`. Note that the attack is not profitable right away, as the `VaultV2` will need time to distribute the newly acquired liquidity as `totalAssets`.

This attack can potentially be repeated and has the most chances to happen during a winddown, but note that a malicious allocator could also trigger it during normal operations.

This issue was uncovered independently by Steakhouse during the review.

Code corrected:

The code has been updated to block the functions `deposit()`, `mint()`, `withdraw()`, `redeem()` during a `flash()`.

6.2 Funds Can Be Locked in `FundingAave` During Winddown

Design | High | Version 1 | Code Corrected

CS-STKHSBX-001

If the box has an open debt position on Aave when winddown is active, funds can be locked in the `FundingAave` module, effectively incurring a loss to the `Box` and its associated `VaultV2`. This can be done because:

1. The function `Box.repay()` can be called with a `repayAmount` that is larger than the debt to repay. The `Box` sends `repayAmount` to the funding module and expects it to use the full amount. If `repayAmount` is greater than the debt amount, Morpho will revert as the associated number of shares will be higher than the balance (https://github.com/morpho-org/morpho-blue/blob/main/src/Morpho.sol#L286), but in AaveV3 the `repayAmount` will be capped to the debt amount (https://etherscan.io/address/0x97287a4f35e583d924f78ad88db8afce1379189a#code#F14#L174). This means that `repayAmount - debtAmount` will stay stuck in the `FundingAave` module.
2. During winddown, the `allocate()` function allows to swap more than the gap needed to repay the debt in `token` because it does not account for debt tokens already in the `Box`:

```
uint256 debtValue = _debtBalance(token).mulDiv(oraclePrice, ORACLE_PRECISION);
```

During winddown, by combining 1. and 2., an attacker could swap the underlying asset to the debt token in multiple swaps of size of the debt, and then call `Box.repay()` with the contract's balance in the debt token, which should be larger than the debt. This will lock the difference in the `FundingAave` module.

While the combination of 1. and 2. is the worst case scenario, 1. can already do some damage on its own.

Note that this risk also exists during normal operations, but with a lower criticality, as allocators can specify a `repayAmount` that is higher than the debt to repay. While the allocators are expected to be non-malicious, they can still make mistakes.

Code corrected:

Steakhouse addressed the issues as follows:

1. `Box.repay()` no longer allows one to repay more than the debt.
2. During winddown, the allocation mechanism now accounts for debt tokens already in the `Box`.

Additionally, `FundingAave.repay()` will transfer any unused tokens back to the `owner`.

6.3 Read-only Reentrancy

Design | High | Version 1 | Code Corrected

CS-STKHSBX-002

The swapping functions `allocate()`, `deallocate()` and `reallocate()` take an arbitrary `swapper` address to execute the swap, this opens a read-only reentrancy attack vector that allows to undervalue the total assets of the `Box` and mint cheap shares in a connected `VaultV2` or any other protocol that uses `Box` or `VaultV2` shares price. The attack on the `VaultV2` works as follows:

1. Call one of the swap functions with malicious swapper.
2. On the `sell()` call, pull the tokens out of the `Box`.
3. (If `BoxAdapterCache` is used) call `VaultV2.forceDeallocate()` for the `Box` with `assets = 0`. This will update the cached total assets (undervalued).
4. Mint cheap shares on the `VaultV2`. Shares are cheap because the NAV of the `Box` is missing the tokens to sell during the swap. A lower `totalAssets` is also recorded on the `VaultV2`.
5. Finish the swap.
6. (If `BoxAdapterCache` is used) call `VaultV2.forceDeallocate()` for the `Box` with `assets = 0`. This will update the cached total assets (correct value this time).

The freshly minted shares will not regain value immediately as the `VaultV2` `totalAssets` cannot increase by more than `maxRate` per second. This will give time to other users to profit from the cheap share price.

This attack can potentially be repeated and has the most chances to happen during a winddown, but note that a malicious allocator could also trigger it during normal operations.

Code corrected:

Steakhouse implemented a caching mechanism for NAV during swap operations.

6.4 Box Cannot Receive Native Currency

Design | Medium | Version 2 | Code Corrected

CS-STKHSBX-024

Box does not implement a `receive()` or `fallback()` function. As a result, attempting to skim native currency from a funding module will always revert.

Code corrected:

The `receive()` and `fallback()` functions have been added to the contract.

6.5 Discrepancy in Cumulated Slippage Computation

Design | Medium | Version 1 | Code Corrected

CS-STKHSBX-004

There is a discrepancy in the way the slippage is computed (`slippageValue.mulDiv(PRECISION, _navForSlippage())`) for the cumulated slippage percentage depending whether `_isInFlash` is active or not. Example:

In the normal case:

1. `totalAssets = X`
2. swap lost the equivalent of 100 assets
3. percentage is `100 * PRECISION / (X - 100)`

But in the flash case:

1. `totalAssets = X`
2. swap lost the equivalent of 100 assets
3. percentage is `100 * PRECISION / X` because the value is cached

In summary, the computed percentage is bigger in the non-flash case.

Code corrected:

The caching mechanism for NAV, introduced to fix the issue related to the read-only reentrancy exploit (Read-only reentrancy), also addresses this issue. In Version 2, the percentage is computed as `100 * PRECISION / X` in both cases.

6.6 Excessive Access Control for `addFundingCollateral()`

Design | Medium | Version 1 | Code Corrected

CS-STKHSBX-005

The function `Box.addFundingCollateral()` enforces a timelock and the `msg.sender` to be the `curator`. This design is too restrictive in comparison to similar `addFundingXYZ()` functions and considering that the timelock already ensures that the function call was submitted by the `curator`.

Code corrected:

Steakhouse removed the restrictive condition.

6.7 FundingMorpho.depledge() Does Not Sanitize collateralToken

Design Medium Version 1 Code Corrected

CS-STKHSBX-006

The `depledge()` method allows a caller to specify an arbitrary whitelisted collateral token. If a whitelisted token that is different from `market.collateral` is supplied, the function calls `morpho.withdrawCollateral()`, then transfers the requested `collateralAmount` of the supplied token to `Box` instead of the `market.collateral` token:

```
function depledge(bytes calldata facilityData, IERC20 collateralToken,
uint256 collateralAmount) external override {
    ...
    require(isCollateralToken(collateralToken), ErrorsLib.TokenNotWhitelisted());
    ...
    morpho.withdrawCollateral(market, collateralAmount, address(this), address(this));
    ...
    collateralToken.safeTransfer(owner, collateralAmount);
}
```

The tokens actually withdrawn from the market can remain held by the funding module while the specified `collateralToken` is transferred out. This causes the actual market collateral to become stuck in the funding module. The issue is severe in winddown mode because anyone can permissionlessly invoke `Box.depledge()`. Consider the scenario where `DAI` is a whitelisted collateral token for the funding module and the `WETH/XYZ` market is registered legitimately. Moreover, assume that the Morpho funding module had pledged collateral into the market, e.g. `100 WETH`. An attacker can send `100 DAI` to the funding module and call `Box.depledge()` with facility data for the `WETH/XYZ` market, `DAI` as `collateralToken`, and `100e18` as `collateralAmount`. The module receives `100 WETH` while the box receives `100 DAI`.

During standard mode, results are less severe as a malicious allocator would have to invoke `Box.depledge()` to execute the same exploit. An analogous exploit is possible in `Box.borrow()`. Note that borrowing cannot be executed by non-privileged users in either mode.

Code corrected:

Steakhouse implemented sanity checks on all relevant functions for collateral and debt tokens in `FundingMorpho`.

6.8 Box Decimals Fixed to 18

Correctness Medium Version 1 Code Corrected

CS-STKHSBX-009

The function `Box.decimals()` is never overridden and always returns 18. This is wrong if the underlying asset has a different decimals value (e.g., USDC has 6 decimals), as the shares are minted 1:1.

Code corrected:

In Version 2, Box normalizes its share token decimals to a minimum of 18. For USDC, this means that shares are minted $1 : 10^{12}$ at the beginning.

6.9 Inconsistency in FundingAave facilities()

Design | Low | Version 3 | Code Corrected

CS-STKHSBX-027

FundingAave.facilities() returns empty bytes even if no facility was added before. Moreover, it returns empty bytes for any input index. Considering that FundingAave only accommodates one single facility, and FundingMorpho.facilities() would revert on invalid indices, this is not consistent.

Code corrected:

The function has been updated to return empty bytes only if a facility was added before and the input index is 0.

6.10 ERC-4626 Violations

Design | Low | Version 1 | Code Corrected

CS-STKHSBX-010

Box is described as an EIP-4626 compliant 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. maxWithdraw() and maxRedeem() do not take into account liquidity. According to the standard, they MUST return the maximum amount of assets or shares that could be transferred through withdraw or redeem and not cause a revert. In Box, the implementation effectively treats a user's full pro-rata claim on totalAssets() as immediately withdrawable, even when the vault's asset balance is lower because value is held in other tokens or locked in funding modules.
    2. The standard specifies that totalAssets() "MUST *NOT* REVERT". However, if queried during a flash operation, the function will revert. As a consequence convertToAssets() and convertToShares() will revert during a flash operation as well. Standard allows these functions to revert only if an integer overflow is caused by an unreasonably large input.

- Violations depending on interpretation

    1. The functions previewMint() and previewDeposit() ignore the isFeeder mapping and may return 0 for addresses that are not whitelisted. Depending on interpretation, isFeeder can be interpreted as user limits but could also be interpreted as other reasons of reverts.

Code corrected:

Steakhouse resolved the violations described above.

However, no comments were made regarding the violations that depend on interpretation. Note that `previewMint()` and `previewDeposit()` might return a non-zero amount and that actual deposit/mint might fail.

6.11 Floating Pragma

Correctness **Low** **Version 1** **Code Corrected**

CS-STKHSBX-012

Box uses a floating pragma solidity ^0.8.28. Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively (https://github.com/SmartContractSecurity/SWC-registry/blob/b408709/entries/SWC-103.md).

Code corrected:

All compiled code under the audit is now using 0.8.28.

6.12 Inaccurate Naming, Comments and NatSpec

Correctness **Low** **Version 1** **Code Corrected**

CS-STKHSBX-017

1. The NatSpec of `_winddownSlippageTolerance()` specifies that the slippage should go from 0% to 100%, but the implementation enforces a range from 0% to 1%.
2. The functions `allocate()/deallocate()` of `BoxAdapter` and `BoxAdapterCached` have the following comment: *Safe casts because Box bounds the total supply of the underlying token*. This comment is incorrect as no such bound exists in the `Box`.
3. The parameter's name of the function `Box._findFundingIndex()` and its NatSpec are `fundingData`, even though they refer to a funding module.
4. There is a typo in the main NatSpec of the `Box`: *The Box uses forApprove* instead of *The Box uses forceApprove*.
5. Some elements of the `BoxAdapterCachedFactory` such as function, variables, and event names are misleading as they refer to a `BoxAdapter` instead of a `BoxAdapterCached`.

Version 2

1. There is a typo in the NatSpec of the function `Box.abdicateTimelock()`: *Does not impact previsously queued changes* instead of *Does not impact previously queued changes*.

Code partially corrected:

Version 1

Steakhouse corrected the comments, namings and NatSpecs described in 1-4.

Version 2

Not corrected.

6.13 Inconsistent and Missing Input Sanitization

Design | Low | Version 1 | Code Corrected

CS-STKHSBX-013

Input sanitization is conducted inconsistently across the codebase, below is a non-exhaustive list:

1. The `_curator` address is not checked for `address(0)` in the constructor of the `Box`.
2. As opposed to `setSkimRecipient()`:
    1. `transferOwnership()` does not check that `newOwner != owner`.
    2. `setCurator()` does not check for `address(0)` or that `newCurator != curator`.
    3. `setGuardian()` does not check for `address(0)` or that `newGuardian != guardian`.
    4. `setIsAllocator()` does not check for `address(0)` or that the target `account` is [not] already in the mapping.
    5. `setIsFeeder()` does not check that the target `account` is [not] already in the mapping.
    6. `setMaxSlippage()` does not check that `newMaxSlippage != oldMaxSlippage`.
    7. `changeTokenOracle()` does not check `oracles[token] != oracle`.

    As a result, all of these calls emit spurious events, e.g. `emit EventsLib.FeederUpdated(account, newIsFeeder);`.
3. `addToken()` does not check that the added token is not the underlying asset. If the underlying asset is added, it would be counted twice in the NAV.
4. `addFunding()` does not check that the `Box` is the owner of the `fundingModule`. If it is not the owner, the module will be unusable considering it is either `FundingMorpho` or `FundingAave`.

Code corrected:

Steakhouse implemented the missing input sanitization.

6.14 Rounding Direction

Design | Low | Version 1 | Code Corrected

CS-STKHSBX-014

In a few places in the code, rounding up would help protect the system against insolvency:

1. The value for `minTokens` in the swapping functions can be rounded up to ensure the slippage is always at most `slippageTolerance`.
2. The value passed as parameter in `_increaseSlippage()` can be rounded up to make sure enough slippage is recorded.

Code corrected:

Steakhouse modified `Box` to round up the `minTokens` value and the value passed as parameter of `_increaseSlippage()`.

6.15 Unreachable Code

Design | Low | Version 1 | Code Corrected

CS-STKHSBX-015

The condition `repayAmount == type(uint256).max` in `FundingMorpho.repay()` will (almost) never return true, as the same check is done in `Box.repay()`. The only time it will be triggered is if the `debtAmount` computed in `Box.repay()` returns `type(uint256).max`. If this happens, `debtAmount` in `FundingMorpho.repay()` will also be `type(uint256).max`, making the value assignment redundant.

Code corrected:

Steakhouse removed the unreachable code.

6.16 Unused Event

Design | Low | Version 1 | Code Corrected

CS-STKHSBX-016

The event `Unbox` is declared in `EventsLib` but is never emitted.

Code corrected:

Steakhouse removed the event declaration.

6.17 Wrong LTV on Total Loss

Correctness | Low | Version 1 | Specification Changed

CS-STKHSBX-018

The functions in both `FundingMorpho.ltv()` and `FundingAave.ltv()` returns 0 if the position's collateral value is zero, regardless of whether there is outstanding debt. An LTV of 0 implies a perfectly healthy position. However, a position with debt and no collateral is critically undercollateralized and has an effectively infinite LTV. Returning 0 severely misrepresents this risk. This data integrity vulnerability provides dangerously misleading information to any user or protocol relying on this function for risk assessment, potentially leading to financial losses for users who act on this incorrect data.

Specification changed:

Steakhouse added the following comment before both `ltv()` functions:

```
@dev returns 0 if there is no collateral
```

6.18 Missing Constructor Parameter in Event

Informational | Version 1 | Code Corrected

CS-STKHSBX-022

The constructor of `Box` emits the event `BoxCreated` which contains all the constructor parameters but seems to be missing `shutdownWarmup`.

Code corrected:

Steakhouse updated the event to include the `shutdownWarmup` parameter.

6.19 Unchecked Block for Loop Iterator

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

CS-STKHSBX-023

Since Solidity 0.8.22 (https://www.soliditylang.org/blog/2023/10/25/solidity-0.8.22-release-announcement/), the compiler automatically generates the `unchecked` block for the loop increments in constructs like `for(uint256 i; i < ...; i++)`. Avoiding an explicit `unchecked { i++; }` makes the code more readable. Below is a non-exhaustive list of functions using explicit `unchecked` blocks:

- `Box.removeToken()`
- `Box._nav()`

Code corrected:

The unchecked increments were removed from the code and the default increment counter is used instead.

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 Aave Can Suffer Liquidity Crises

Informational Version 1 Acknowledged

CS-STKHSBX-019

As a multicollateralized lending platform, AaveV3 can suffer a liquidity crisis in a given market if all the tokens are borrowed. If this happens while the `Box` has collateral in that market, the `Box` is forced to keep the position until enough free liquidity returns.

Acknowledged:

Client acknowledged the potential liquidity crises.

7.2 Aave Isolation Mode

Informational Version 1 Acknowledged

CS-STKHSBX-026

The activation of the isolation mode can be front-run by sending some other non-isolated `aToken` to the `FundingModule`, which prevents the isolation mode to be activated.

Acknowledged:

Steakhouse acknowledged the issue and stated that no token available through isolation mode is of interest.

7.3 Abdicated `decreaseTimelock()` Can Be Called Again

Informational Version 1 Acknowledged

CS-STKHSBX-020

The function `decreaseTimelock()` only enforces that the function it targets is not abdicated, but it does not check whether it is abdicated itself. Therefore, `decreaseTimelock()` can still be used after being abdicated.

Acknowledged:

Steakhouse is aware of this corner case behaviour.

7.4 Gas Optimizations

Informational | Version 1 | Code Partially Corrected

CS-STKHSBX-021

1. In almost all the places where a `for` loop is used, the length of an array can be cached to avoid an SLOAD on every iteration.
2. In `Box.constructor()`, `skimRecipient` is explicitly set to `address(0)`. This is redundant because uninitialized state variables are assigned their default value, which is `address(0)` in this case.
3. `Box.setSkimRecipient()` and `Box.transferOwnership()` load the current value from storage (SLOAD) twice. Caching this value will save gas.
4. The `Box` storage variables `_isInFlash` and `_cachedNavForFlash` can be `transient`.
5. The check `newDuration <= TIMELOCK_CAP` is a tautology in `Box.decreaseTimelock()`. When not abdicated, the maximum duration is `TIMELOCK_CAP`, enforced by `Box.increaseTimelock()`. When combined with the third check (`newDuration < timelock[selector]`), it becomes `newDuration < TIMELOCK_CAP`, which is more strict than `newDuration <= TIMELOCK_CAP`.
6. The check `skimRecipient != address(0)` in `Box.skim()` will never revert, as `skimRecipient` is required to be `msg.sender` as well.
7. Part of the logic of `Box.deposit()` and `Box.mint()` is the same, this duplicated logic can be refactored into an internal function that is then called from `Box.deposit()` and `Box.mint()` to reduce the size of the bytecode.
8. Part of the logic of `Box.withdraw()` and `Box.redeem()` is the same, this duplicated logic can be refactored into an internal function that is then called from `Box.withdraw()` and `Box.redeem()` to reduce the size of the bytecode.
9. When adding and removing a funding module to/from the `Box`, checking that `fundingModule.facilitiesLength() == 0` should be enough to ensure the two other requirements by transitivity.
10. The `FundingAave` contract can generally be optimized based on the fact that there will be at most one facility.
11. In `FundingAave`, `rateMode` is immutable and set to `2` as it is the only mode AaveV3 supports. However, `FundingAave._debtBalance()` performs `address aDebtToken = rateMode == 2 ? variableDebtToken : stableDebtToken;`, where `rateMode == 2` will always be true. This adds unnecessary complexity and is gas inefficient.
12. In the function `FundingAave.nav()`, in the return statement, the expression will yield `0` when `totalCollateralValue == totalDebtValue`, so a strict inequality will be more gas efficient.
13. The function `FundingAave.depledge()` could send the token directly to the `owner` via the `to` parameter of `Pool.withdraw()` to save a token transfer.
14. The functions `FundingMorpho.depledge()` and `FundingMorpho.borrow()` could send the token directly to the `owner` via the `receiver` parameter of `Morpho.withdrawCollateral()` and `Morpho.borrow()` to save a token transfer.
15. The function `FundingMorpho.isFacility()`, `FundingMorpho._findFacilityIndex()` recompute `keccak256(facilityData)` on each iteration. Caching the hash will save gas when more than one facility is whitelisted.

Version 2

1. The function `Box.changeTokenOracle()` could use the helper function `_requireNotEqualAddress()` in place of the check `oracles[token] != oracle`.

2. The function `FundingMorpho.skim()` does not need the NAV check as Morpho Blue does not tokenize the positions.

Version 3

1. `BoxAdapterCached._updateTotalAssets()` computes the new value for `totalAssets`, writes it to the storage, and reads from the storage to return it. Storing the computed value in stack and returning it would prevent an `SLOAD`.
2. `FundingMorpho.nav()` calls `oraclesProvider.asset()` twice.

Code partially corrected:

Version 1

Steakhouse applied the gas optimizations described in 1–8, and 11–14.

In **Version 3**, facilities are stored in a set. Thus, the iteration mentioned in 15 is explictly removed and a single hash computation is done.

Version 2

1. Steakhouse applied the described gas optimization.
2. Steakhouse stated:

```
We want to always check for NAV invariance when we skim regardless. The gas cost is worth it
for us to make sure there is no edge case we didn't consider.
```

Version 3

Both gas optimizations have been implemented.

7.5 Unhealthy Positions Valuation

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

CS-STKHSBX-025

The current implementations of the `nav()` does not take the health of a position into account. If a position is liquidatable, the collateral is added in full, without applying the liquidation discount/fee.

Acknowledged:

Steakhouse acknowledged the issues and stated:

```
As there is always the ability for the allocator to unwind a liquidable position,
it seems better to keep the nav as is. Usually liquidation happens instantly so
that shouldn't be an issue.
```

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 LTV of `FundingMorpho`

Note Version 1

The function `FundingMorpho.ltv()` can take arbitrary markets as parameter. Users must call the function only with markets that have been whitelisted in `FundingMorpho`, as arbitrary LP positions can be opened on behalf of `FundingMorpho`.

8.2 Limitations on Aave Withdrawals

Note Version 1

Users must be aware that, under certain conditions, withdrawing from AaveV3 can be temporarily impossible and they should take this into account when evaluating the risks of the protocol. Such conditions are liquidity crises (Aave can suffer liquidity crises) and paused markets.

8.3 Liquidity of Investment Tokens

Note Version 1

It is important that the investment tokens that are whitelisted are liquid enough to avoid significant price manipulation. In particular, Pendle PT tokens that are considered for investment must be carefully assessed. The default TWAP window of the Pendle PT oracle provides some protection against significant price manipulation in liquid markets.

Therefore, Steakhouse is expected to create strategies based on tokens that are liquid enough, and users must check and monitor the set of whitelisted tokens before and after entering the vault.

8.4 Market Movements During Shutdown Warmup

Note Version 1

If a `Box` needs to be shut down, one can assume that the allocators might also not be available. In that scenario, open positions might not be rebalanced in time, leaving shareholders fully exposed to all the market movements. In the worst case, debt positions can be liquidated without anyone being able to prevent it.

Even though unlikely, users must be aware that LPing in `VaultV2` that use the `Box` strategy exposes them to such risks.

8.5 Oracle Price After Winddown

Note Version 1

Users and integrators must be aware of the possibility of share price manipulation after `shutdownSlippageDuration` has elapsed.

Once the `shutdownSlippageDuration` has elapsed, the `guardian` has the power to update the price oracles. Since the `Box` is winding down, one can also assume that the liquidity in connected `VaultV2` will decrease. This would allow an attacker to become the majority shareholder and thus control the `guardian` and update the price oracles. This allows the attacker to set arbitrary prices and manipulate the `Box` and `VaultV2` share price, which can lead to attacks on other protocols as well. Even if fully unwound, such an attacker can send a small amount of whitelisted token to the `Box` and manipulate its price.

In summary, it is in the best interest of users and protocol integrating with the `Box` or `VaultV2` with a `Box` to fully unwind and exit / remove support for the `Box` before `shutdownSlippageDuration` has passed.

8.6 Reputation of Morpho Markets

Note **Version 1**

It is important that the Morpho markets that are used by `FundingMorpho` are carefully reviewed. In particular, using a market where the oracle can be manipulated can result in a loss for the vault. Thus it is critical to use markets that already have a good reputation or that Steakhouse deems trustworthy.

Therefore, Steakhouse is expected to create strategies based on Morpho markets that are trustworthy, and users must check and monitor the set of whitelisted markets before and after entering the vault.

8.7 Timelock Delays and Guardian Delay

Note **Version 1**

Users must be aware that if the timelock of a function is smaller than the voting and execution time of the `guardian`, it is possible for the `curator` to submit actions that cannot be challenged by the `guardian`.

In order to keep control over the actions of the `curator`, users must check and compare the different timelocks with the time a proposal might take to be executed from the `guardian`.

8.8 Underwater Positions Will Not Be Closed

Note **Version 1**

In the case where `collateralAmount < borrowAmount` happens, the adapters will return a NAV of 0. If this happens, Steakhouse considers the underlying protocol and positions as unsalvageable and thus the positions are not expected to be closed.

Steakhouse specified:

```
Lending protocols are non recourse so a negative NAV is actually 0.
It is understood that the Box curator will stop using the FundingModule and just create a new one.
```

8.9 Box Cannot Always Ensure a Fully Liquid VaultV2 Through forceDeallocate()

Note Version 1

The idea behind the function `VaultV2.forceDeallocate()` is to give a guarantee to users that they will always be able to exit the vault by doing in-kind redemptions.

Note that if Box is used by a `VaultV2`, this guarantee disappears as it is not possible for users to swap their position in the `VaultV2` for shares of the Box or portions of its underlying positions during normal operations.

However, the Box can eventually be made fully liquid after some time thanks to the shutdown process.