Omniscia
SMART CONTRACT AUDITS

Smart Contract Security Assessment

09-3-2022

Prepared for
Morpho

Online Report
morpho-labs-tokenized-vaults

Tokenized Vaults Security Audit

Audit Overview

We were tasked with performing an audit of the Morpho codebase and in particular their tokenized vault implementations integrating with the Aave V3 and Compound ecosystem.

Over the course of the audit, we identified certain unsafe casting operations as well as the lack of a slippage system for the EIP-4626 standard that can significantly affect the operation of the vaults.

We advise the Morpho Labs team to closely evaluate all minor-and-above findings identified in the report and promptly remediate them as well as consider all optimizational exhibits identified in the report.

Post-Audit Conclusion

The Morpho Labs team provided a new commit hash meant to address the exhibits outlined in the audit report and additionally advised us on certain exhibits they deem as acknowledge-able.

All exhibits have been adequately dealt with and the Morpho Labs team provided supplemental material to nullify one of the exhibits related to Compound integration as well as promised future actions for two exhibits related to slippage issues based on a secondary protocol's audit conclusion.

Contracts Assessed

Files in Scope	Repository	Commit(s)
ERC4626UpgradeableSafe.sol (ERC)	morpho-tokenized-vaults	fdb7bcbd92, eae33160ec, 9748c619ac
SupplyVault.sol (SVT)	morpho-tokenized-vaults	fdb7bcbd92, eae33160ec, 9748c619ac
SupplyVault.sol (SVL)	morpho-tokenized-vaults	fdb7bcbd92, eae33160ec, 9748c619ac
SupplyHarvestVault.sol (SHV)	morpho-tokenized-vaults	fdb7bcbd92, eae33160ec, 9748c619ac
SupplyHarvestVault.sol (SRC)	morpho-tokenized-vaults	fdb7bcbd92, eae33160ec, 9748c619ac
SupplyVaultUpgradeable.sol (SVU)	morpho-tokenized-vaults	fdb7bcbd92, eae33160ec, 9748c619ac
SupplyVaultUpgradeable.sol (SRO)	morpho-tokenized-vaults	fdb7bcbd92, eae33160ec, 9748c619ac
UniswapV2Swapper.sol (UVS)	morpho-tokenized-vaults	fdb7bcbd92, eae33160ec, 9748c619ac
UniswapV3Swapper.sol (SRU)	morpho-tokenized-vaults	fdb7bcbd92, eae33160ec, 9748c619ac

Audit Synopsis

Severity	Identified	Alleviated	Partially Alleviated	Acknowledged
Unknown	6	6	0	0
Informational	7	7	0	0
Minor	6	6	0	0
Medium	4	2	0	2
Major	1	1	0	0

During the audit, we filtered and validated a total of **9 findings utilizing static analysis** tools as well as identified a total of **15 findings during the manual review** of the codebase. We strongly recommend that any minor severity or higher findings are dealt with promptly prior to the project's launch as they introduce potential misbehaviours of the system as well as exploits.

The list below covers each segment of the audit in depth and links to the respective chapter of the report:

Compilation

The project utilizes `foundry` as its development pipeline tool, containing an array of tests and scripts coded in Solidity.

To compile the project, the `build` command needs to be issued via the `forge` CLI tool:

```bash
forge build
```

The `forge` tool automatically selects a Solidity version as no explicit version is defined in the `foundry.toml` file.

The project contains discrepancies with regards to the Solidity version used as the `pragma` statements of the contracts are open-ended (`^0.8.0`).

We advise them to be locked to `0.8.11` (`=0.8.11`), the same version utilized for our static analysis as well as optimizational review of the codebase.

During compilation with the `foundry` pipeline, no errors were identified that relate to the syntax or bytecode size of the contracts.

To conduct our static analysis round, we had to port the codebase to a `hardhat` configuration and manually re-adjust all remappings performed by `foundry` as the static analysis tools were incompatible with them.

As such, the static analysis findings that an individual may identify on their own can be discrepant with ours due to the adjustments performed in the code for the new compilation structure to work.

Static Analysis

The execution of our static analysis toolkit identified **122 potential issues** within the codebase of which **107 were ruled out to be false positives** or negligible findings.

The remaining **15 issues** were validated and grouped and formalized into the **9 exhibits** that follow:

ID	Severity	Addressed	Title
SHV-01S	Informational	Yes	Illegible Numeric Value Representation
SRC-01S	Informational	Yes	Illegible Numeric Value Representations
SRC-02S	Minor	Nullified	Inexistent Sanitization of Input Address
SHV-02S	Minor	Yes	Inexistent Sanitization of Input Addresses
SVU-01S	Minor	Yes	Inexistent Sanitization of Input Addresses
SRO-01S	Minor	Yes	Inexistent Sanitization of Input Addresses
UVS-01S	Minor	Yes	Inexistent Sanitization of Input Addresses
SRU-01S	Informational	Yes	Illegible Numeric Value Representation
SRU-02S	Minor	Yes	Inexistent Sanitization of Input Address

Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in the tokenized vaults of Morpho Labs.

As the project at hand implements EIP-4626 based tokenized vaults, intricate care was put into ensuring that the **flow of funds within the system conforms to the specifications and restrictions** laid forth within the protocol's specification.

We validated that **all state transitions of the system occur within sane criteria** and that all rudimentary formulas within the system execute as expected. We **pinpointed certain casting and operational vulnerabilities** within the system which could have had **moderate ramifications** to its overall operation, however, they were conveyed ahead of time to the Morpho Labs team to be **promptly remediated**.

Additionally, the system was investigated for any other commonly present attack vectors such as re-entrancy attacks, mathematical truncations, logical flaws and **ERC / EIP** standard inconsistencies. The documentation of the project was satisfactory to the extent it need be.

A total of **15 findings** were identified over the course of the manual review of which **11 findings** concerned the behaviour and security of the system. The non-security related findings, such as optimizations, are included in the separate **Code Style** chapter.

The finding table below enumerates all these security / behavioural findings:

ID	Severity	Addressed	Title
SHV-01M	Unknown	Yes	Potential Points of Concern
SRC-01M	Unknown	Yes	Potential Points of Concern
SVL-01M	Unknown	Yes	Potential Point of Concern
SVT-01M	Unknown	Yes	Potential Points of Concern
SVT-02M	Medium	Yes	Unsafe Casting Operations
SVL-02M	Medium	Yes	Unsafe Casting Operations

ID	Severity	Addressed	Title
SVU-01M	Unknown	Yes	Potential Points of Concern
SRO-01M	Unknown	Yes	Potential Points of Concern
SVU-02M	Medium	Acknowledged	Inexistent Slippage Protection
SRO-02M	Medium	Acknowledged	Inexistent Slippage Protection
SRO-03M	Major	Nullified	Normalized Mantissa Inexistent

SupplyHarvestVault Static Analysis Findings

SHV-01S: Illegible Numeric Value Representation

Type	Severity	Location
Code Style	Informational	SupplyHarvestVault.sol:L49

Description:

The linked representation of a numeric literal is sub-optimally represented decreasing the legibility of the codebase.

Example:

src/aave-v3/SupplyHarvestVault.sol

SOL

```
49  uint16 public constant MAX_BASIS_POINTS = 10_000; // 100% in basis points.
```

Recommendation:

To properly illustrate the value's purpose, we advise the following guidelines to be followed. For values meant to depict fractions with a base of `1e18`, we advise fractions to be utilized directly (i.e. `1e17` becomes `0.1e18`) as they are supported. For values meant to represent a percentage base, we advise each value to utilize the underscore (`_`) separator to discern the percentage decimal (i.e. `10000` becomes `100_00`, `300` becomes `3_00` and so on). Finally, for large numeric values we simply advise the underscore character to be utilized again to represent them (i.e. `1000000` becomes `1_000_000`).

Alleviation:

The underscore character has been relocated according to the specification we shared with the Morpho Labs team thus alleviating this exhibit.

SHV-02S: Inexistent Sanitization of Input Addresses

Type	Severity	Location
Input Sanitization	Minor	SupplyHarvestVault.sol:L70, L91

Description:

The linked function(s) accept `address` arguments yet do not properly sanitize them.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

src/aave-v3/SupplyHarvestVault.sol

SOL

```sol
function initialize(
    address _morpho,
    address _poolToken,
    string calldata _name,
    string calldata _symbol,
    uint256 _initialDeposit,
    uint16 _harvestingFee,
    address _swapper
) external initializer {
    __SupplyVaultUpgradeable_init(_morpho, _poolToken, _name, _symbol, _ini

    harvestingFee = _harvestingFee;
    swapper = ISwapper(_swapper);
}
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that each `address` specified is non-zero.

Alleviation:

A zero-address check has been properly introduced for the relevant input argument as advised.

SupplyHarvestVault Static Analysis Findings

SRC-01S: Illegible Numeric Value Representations

Type	Severity	Location
Code Style	Informational	SupplyHarvestVault.sol:L60, L61

Description:

The linked representations of numeric literals are sub-optimally represented decreasing the legibility of the codebase.

Example:

src/compound/SupplyHarvestVault.sol

SOL

```sol
60  uint16 public constant MAX_BASIS_POINTS = 10_000; // 100% in basis points.
```

Recommendation:

To properly illustrate each value's purpose, we advise the following guidelines to be followed. For values meant to depict fractions with a base of `1e18`, we advise fractions to be utilized directly (i.e. `1e17` becomes `0.1e18`) as they are supported. For values meant to represent a percentage base, we advise each value to utilize the underscore (`_`) separator to discern the percentage decimal (i.e. `10000` becomes `100_00`, `300` becomes `3_00` and so on). Finally, for large numeric values we simply advise the underscore character to be utilized again to represent them (i.e. `1000000` becomes `1_000_000`).

Alleviation:

The underscore characters have been relocated according to the specification we shared with the Morpho Labs team thus alleviating this exhibit.

SRC-02S: Inexistent Sanitization of Input Address

Type	Severity	Location
Input Sanitization	Minor	SupplyHarvestVault.sol:L86

Description:

The linked function accepts an `address` argument yet does not properly sanitize it.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

src/compound/SupplyHarvestVault.sol

SOL

```sol
79  function initialize(
80      address _morpho,
81      address _poolToken,
82      string calldata _name,
83      string calldata _symbol,
84      uint256 _initialDeposit,
85      HarvestConfig calldata _harvestConfig,
86      address _cComp
87  ) external initializer {
88      (isEth, wEth) = __SupplyVaultUpgradeable_init(
89          _morpho,
90          _poolToken,
91          _name,
92          _symbol,
93          _initialDeposit
94      );
95
96      harvestConfig = _harvestConfig;
97
98      cComp = _cComp;
99
100     comp.safeApprove(address(SWAP_ROUTER), type(uint256).max);
101 }
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that the `address` specified is non-zero.

Alleviation:

The `cComp` member along with its corresponding input argument is no longer present in the codebase as part of a style exhibit thus rendering this exhibit no longer relevant.

SupplyVaultUpgradeable Static Analysis Findings

SVU-01S: Inexistent Sanitization of Input Addresses

Type	Severity	Location
Input Sanitization	Minor	SupplyVaultUpgradeable.sol:L46, L47, L80

Description:

The linked function(s) accept `address` arguments yet do not properly sanitize them.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

src/aave-v3/SupplyVaultUpgradeable.sol

SOL

```sol
80  function setRewardsController(address _rewardsController) external onlyOwne
81      rewardsController = IRewardsController(_rewardsController);
82      emit RewardsControllerSet(_rewardsController);
83  }
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that each `address` specified is non-zero.

Alleviation:

The last referenced input argument is no longer present in the codebase, however, the first two linked arguments are now properly sanitized thus addressing this exhibit in full.

SupplyVaultUpgradeable Static Analysis Findings

SRO-01S: Inexistent Sanitization of Input Addresses

Type	Severity	Location
Input Sanitization	Minor	SupplyVaultUpgradeable.sol:L38, L39

Description:

The linked function(s) accept `address` arguments yet do not properly sanitize them.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

src/compound/SupplyVaultUpgradeable.sol

SOL

```sol
function __SupplyVaultUpgradeable_init(
    address _morpho,
    address _poolToken,
    string calldata _name,
    string calldata _symbol,
    uint256 _initialDeposit
) internal onlyInitializing returns (bool isEth, address wEth) {
    ERC20 underlyingToken;
    (isEth, wEth, underlyingToken) = __SupplyVaultUpgradeable_init_unchaine
        _morpho,
        _poolToken
    );

    __Ownable_init();
    __ERC20_init(_name, _symbol);
    __ERC4626UpgradeableSafe_init(ERC20Upgradeable(address(underlyingToken)
}
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that each `address` specified is non-zero.

Alleviation:

The two input arguments are now adequately sanitized via a corresponding `if-revert` clause.

UniswapV2Swapper Static Analysis Findings

UVS-01S: Inexistent Sanitization of Input Addresses

Type	Severity	Location
Input Sanitization	Minor	UniswapV2Swapper.sol:L26-L29

Description:

The linked function(s) accept `address` arguments yet do not properly sanitize them.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

src/UniswapV2Swapper.sol

SOL

```sol
26  constructor(address _swapRouter, address _wrappedNativeToken) {
27      swapRouter = IUniswapV2Router02(_swapRouter);
28      wrappedNativeToken = _wrappedNativeToken;
29  }
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that each `address` specified is non-zero.

Alleviation:

Zero-address checks have been properly introduced for the relevant input arguments as advised.

UniswapV3Swapper Static Analysis Findings

SRU-01S: Illegible Numeric Value Representation

Type	Severity	Location
Code Style	Informational	UniswapV3Swapper.sol:L34

Description:

The linked representation of a numeric literal is sub-optimally represented decreasing the legibility of the codebase.

Example:

src/UniswapV3Swapper.sol

SOL

```
34  uint24 public constant MAX_UNISWAP_FEE = 1_000_000; // 100% in UniswapV3 fe
```

Recommendation:

To properly illustrate the value's purpose, we advise the following guidelines to be followed. For values meant to depict fractions with a base of `1e18`, we advise fractions to be utilized directly (i.e. `1e17` becomes `0.1e18`) as they are supported. For values meant to represent a percentage base, we advise each value to utilize the underscore (`_`) separator to discern the percentage decimal (i.e. `10000` becomes `100_00`, `300` becomes `3_00` and so on). Finally, for large numeric values we simply advise the underscore character to be utilized again to represent them (i.e. `1000000` becomes `1_000_000`).

Alleviation:

The underscore character has been relocated according to the specification we shared with the Morpho Labs team thus alleviating this exhibit.

SRU-02S: Inexistent Sanitization of Input Address

Type	Severity	Location
Input Sanitization	Minor	UniswapV3Swapper.sol:L44-L46

Description:

The linked function accepts an `address` argument yet does not properly sanitize it.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

src/UniswapV3Swapper.sol

SOL

```sol
44  constructor(address _wrappedNativeToken) {
45      wrappedNativeToken = _wrappedNativeToken;
46  }
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that the `address` specified is non-zero.

Alleviation:

A zero-address check has been properly introduced for the relevant input argument as advised.

SupplyHarvestVault Manual Review Findings

SHV-01M: Potential Points of Concern

Type	Severity	Location
Standard Conformity	Unknown	SupplyHarvestVault.sol:L115, L162

Description:

The referenced lines indicate integration with the `IMorpho` contract which is an internal project contract that is not in scope of the audit and thus interactions with it cannot be properly validated.

Example:

src/aave-v3/SupplyHarvestVault.sol

SOL

```sol
15  contract SupplyHarvestVault is SupplyVaultUpgradeable {
```

Recommendation:

As general security recommendations, we advise the Morpho team to ascertain that the rewards yielded by `claimRewards` are properly defined, the `false` flag is correct for the purposes of the `harvest` system, and that the asset supplied in the last call properly update the relevant entries for the **EIP-4626** asset evaluation mechanism. This does not constitute an audit of the Morpho contract and simply indicates best practices and security considerations that should be followed.

Alleviation:

The Morpho team considered our advice and has validated the `IMorpho` integration on their end. This exhibit will remain in the audit report for the sake of prosperity, marked as "addressed" based on Morpho's validation of the integration.

SupplyHarvestVault Manual Review Findings

SRC-01M: Potential Points of Concern

Type	Severity	Location
Standard Conformity	Unknown	SupplyHarvestVault.sol:L162, L166, L173

Description:

The referenced lines indicate integration with the `IMorpho` contract which is an internal project contract that is not in scope of the audit and thus interactions with it cannot be properly validated.

Example:

src/compound/SupplyHarvestVault.sol

SOL

```
14  contract SupplyHarvestVault is SupplyVaultUpgradeable {
```

Recommendation:

As general security recommendations, we advise the Morpho team to ascertain that the reward yielded by `claimRewards` is a single value variable, the `false` flag is correct for the purposes of the `harvest` system, and that the asset supplied in the last call properly update the relevant entries for the **EIP-4626** asset evaluation mechanism. This does not constitute an audit of the Morpho contract and simply indicates best practices and security considerations that should be followed.

Alleviation:

The Morpho team considered our advice and has validated the `IMorpho` integration on their end. This exhibit will remain in the audit report for the sake of prosperity, marked as "addressed" based on Morpho's validation of the integration.

SupplyVault Manual Review Findings

SVT-01M: Potential Points of Concern

Type	Severity	Location
Standard Conformity	Unknown	SupplyVault.sol:L69, L84, L123-L126, L163-L167, L207-L210

Description:

The referenced lines indicate integration with the `IMorpho` and `IRewardsManager` contracts which represent internal project contracts that are not in scope of the audit and thus interactions with them cannot be properly validated.

Example:

src/aave-v3/SupplyVault.sol

SOL

```sol
12  contract SupplyVault is SupplyVaultUpgradeable {
```

Recommendation:

As general security recommendations, we advise the Morpho team to ascertain that the overall reward system exposed by the `IRewardsManager` is sound and is not susceptible to manipulation by external calls. This does not constitute an audit of the Morpho contract and simply indicates best practices and security considerations that should be followed.

Alleviation:

The Morpho team considered our advice and has validated the `IMorpho` & `IRewardsManager` integrations on their end. This exhibit will remain in the audit report for the sake of prosperity, marked as "addressed" based on Morpho's validation of the integration.

SVT-02M: Unsafe Casting Operations

Type	Severity	Location
Mathematical Operations	Medium	SupplyVault.sol:L219, L224-L226

Description:

The relevant casting operations from `uint256` to `uint128` are performed unsafely which can significantly compromise the integrity of the reward system especially when dealing with large offsets such as the `SCALE` value of `1e36`.

Impact:

The overall reward system of the `SupplyVault` can be significantly compromised if a casting underflow is achieved in the `_accrueUnclaimedRewards` function.

Example:

src/aave-v3/SupplyVault.sol

SOL

```sol
218 if (supply > 0 && claimedAmount > 0)
219     rewardsIndex[rewardToken] += uint128(claimedAmount.mulDivDown(SCALE, su
```

Recommendation:

We advise the casting operations to be performed safely via the usage of a relevant library such as `SafeCast` from OpenZeppelin. We should note that the built-in safe arithmetic introduced in post-`0.8.X` Solidity versions **does not cover casting operations**.

Alleviation:

Both operations referenced now make use of the `SafeCastLib` by `@solmate` thus addressing this exhibit in full and performing casting safely.

SupplyVault Manual Review Findings

SVL-01M: Potential Point of Concern

Type	Severity	Location
Standard Conformity	Unknown	SupplyVault.sol:L105

Description:

The referenced line indicates integration with the `IMorpho` contract which is an internal project contract that is not in scope of the audit and thus interactions with it cannot be properly validated.

Example:

src/compound/SupplyVault.sol

SOL

```sol
12  contract SupplyVault is SupplyVaultUpgradeable {
```

Recommendation:

As general security recommendations, we advise the Morpho team to ascertain that the rewards yielded by `claimRewards` are properly defined, and that the `false` flag is correct for the purposes of the `_accrueUnclaimedRewards` system. This does not constitute an audit of the Morpho contract and simply indicates best practices and security considerations that should be followed.

Alleviation:

The Morpho team considered our advice and has validated the `IMorpho` integration on their end. This exhibit will remain in the audit report for the sake of prosperity, marked as "addressed" based on Morpho's validation of the integration.

SVL-02M: Unsafe Casting Operations

Type	Severity	Location
Mathematical Operations	Medium	SupplyVault.sol:L115, L116, L119

Description:

The relevant casting operations from `uint256` to `uint128` are performed unsafely which can significantly compromise the integrity of the reward system especially when dealing with large offsets such as the `WAD` value of `1e18`.

Impact:

The overall reward system of the `SupplyVault` can be significantly compromised if a casting underflow is achieved in the `_accrueUnclaimedRewards` function.

Example:

src/compound/SupplyVault.sol

SOL

```sol
112 if (rewardsIndexDiff > 0) {
113     unclaimed =
114         userRewards[_user].unclaimed +
115         uint128(balanceOf(_user).mulWadDown(rewardsIndexDiff));
116     userRewards[_user].unclaimed = uint128(unclaimed);
117 }
118
119 userRewards[_user].index = uint128(rewardsIndexMem);
```

Recommendation:

We advise the casting operations to be performed safely via the usage of a relevant library such as `SafeCast` from OpenZeppelin. We should note that the built-in safe arithmetic introduced in post-`0.8.X` Solidity versions **does not cover casting operations**.

Alleviation:

All three operations referenced now make use of the `SafeCastLib` by `@solmate` thus addressing this exhibit in full and performing casting safely.

SupplyVaultUpgradeable Manual Review Findings

SVU-01M: Potential Points of Concern

Type	Severity	Location
Standard Conformity	Unknown	SupplyVaultUpgradeable.sol:L69, L70, L89-L92, L96, L108, L118

Description:

The referenced lines indicate integration with the `IMorpho` contract which is an internal project contract that is not in scope of the audit and thus interactions with it cannot be properly validated.

Example:

src/aave-v3/SupplyVaultUpgradeable.sol

SOL

```
20  abstract contract SupplyVaultUpgradeable is ERC4626UpgradeableSafe, Ownable
```

Recommendation:

As general security recommendations, we advise the Morpho team to ascertain that no flash-loan based attacks affect the `p2pSupplyIndex` evaluation and that the `supply` and `withdraw` workflows properly transfer underlying assets out and in of the system. Additionally, all value entries retrieved from the `IMorpho` contract should be validated as non-changeable as otherwise configuration should be dynamic for the contract in scope. This does not constitute an audit of the Morpho contract and simply indicates best practices and security considerations that should be followed.

Alleviation:

The Morpho team considered our advice and has validated the `IMorpho` integration on their end. This exhibit will remain in the audit report for the sake of prosperity, marked as "addressed" based on Morpho's validation of the integration.

SVU-02M: Inexistent Slippage Protection

Type	Severity	Location
Logical Fault	Medium	SupplyVaultUpgradeable.sol:L101, L111

Description:

The **EIP-4626** standard dependency by OpenZeppelin is not meant to be used standalone as highlighted in the documentation of the contract as well given that there may be natural slippage incurred when depositing and withdrawing from the vault which should be accounted for by a router similar to how DEX operations are performed.

Impact:

Inexistent slippage checks will cause arbitrage opportunities to present themselves to potential attackers, hurting the end-users of the vaults.

Example:

src/aave-v3/SupplyVaultUpgradeable.sol

SOL

```sol
20  abstract contract SupplyVaultUpgradeable is ERC4626UpgradeableSafe, Ownable
```

Recommendation:

We advise a router implementation to be introduced to the codebase that interacts with the vaults as otherwise any deposits and withdrawals will be significantly vulnerable to sandwich and MEV attacks.

Alleviation:

The Morpho Labs team stated that they are awaiting a router implementation by the Fei Protocol to become production-ready as it is currently undergoing a security audit and as such will utilize that module once it is ready for alleviating this exhibit in the future.

SupplyVaultUpgradeable Manual Review Findings

SRO-01M: Potential Points of Concern

Type	Severity	Location
Standard Conformity	Unknown	SupplyVaultUpgradeable.sol:L69, L72, L73, L85-L88, L92, L104, L114

Description:

The referenced lines indicate integration with the `IMorpho` contract which is an internal project contract that is not in scope of the audit and thus interactions with it cannot be properly validated.

Example:

src/compound/SupplyVaultUpgradeable.sol

SOL

```
18  abstract contract SupplyVaultUpgradeable is ERC4626UpgradeableSafe, Ownable
```

Recommendation:

As general security recommendations, we advise the Morpho team to ascertain that no flash-loan based attacks affect the `p2pSupplyIndex` evaluation and that the `supply` and `withdraw` workflows properly transfer underlying assets out and in of the system as well as unwrap them for the `CEth` token deposit workflow. Additionally, all value entries retrieved from the `IMorpho` contract should be validated as non-changeable as otherwise configuration should be dynamic for the contract in scope. This does not constitute an audit of the Morpho contract and simply indicates best practices and security considerations that should be followed.

Alleviation:

The Morpho team considered our advice and has validated the `IMorpho` integration on their end. This exhibit will remain in the audit report for the sake of prosperity, marked as "addressed" based on Morpho's validation of the integration.

SRO-02M: Inexistent Slippage Protection

Type	Severity	Location
Logical Fault	Medium	SupplyVaultUpgradeable.sol:L97, L107

Description:

The **EIP-4626** standard dependency by OpenZeppelin is not meant to be used standalone as highlighted in the documentation of the contract as well given that there may be natural slippage incurred when depositing and withdrawing from the vault which should be accounted for by a router similar to how DEX operations are performed.

Impact:

Inexistent slippage checks will cause arbitrage opportunities to present themselves to potential attackers, hurting the end-users of the vaults.

Example:

src/compound/SupplyVaultUpgradeable.sol

SOL

```sol
18  abstract contract SupplyVaultUpgradeable is ERC4626UpgradeableSafe, Ownable
```

Recommendation:

We advise a router implementation to be introduced to the codebase that interacts with the vaults as otherwise any deposits and withdrawals will be significantly vulnerable to sandwich and MEV attacks.

Alleviation:

The Morpho Labs team stated that they are awaiting a router implementation by the Fei Protocol to become production-ready as it is currently undergoing a security audit and as such will utilize that module once it is ready for alleviating this exhibit in the future.

SRO-03M: Normalized Mantissa Inexistent

Type	Severity	Location
Mathematical Operations	Major	SupplyVaultUpgradeable.sol:L91

Description:

The Compound system calculations for assessing the underlying balance of a particular C-Token balance involve the division with a normalizer-mantissa that is calculated dynamically based on the decimals of the underlying asset as well as the decimals of the C-Token, which are fixed. This does not appear to be taken into account here.

Impact:

Currently, over-evaluations and under-evaluations of assets can occur as the Compound pool balance is incorrectly evaluated in underlying units.

Example:

src/compound/SupplyVaultUpgradeable.sol

SOL

```sol
function totalAssets() public view override returns (uint256) {
    IMorpho morphoMem = morpho;
    address poolTokenMem = poolToken;

    Types.SupplyBalance memory supplyBalance = morphoMem.supplyBalanceInOf(
        poolTokenMem,
        address(this)
    );

    return
        supplyBalance.onPool.mul(ICToken(poolTokenMem).exchangeRateStored())
        supplyBalance.inP2P.mul(morphoMem.p2pSupplyIndex(poolTokenMem));
}
```

Recommendation:

We advise the calculation to be re-assessed and the mantissa to be properly calculated depending on the underlying token and C-Token decimals as otherwise the calculation is prone to failure and can significantly compromise the integrity of the vault.

Alleviation:

The Morpho Labs team stated that the calculations are being executed as they are expected to by the overall Morpho Labs protocol given that multiple multi-decimal assets have already been deployed. As a result, we consider this exhibit nullified given that the current calculations fit Morpho Labs' purposes.

SupplyHarvestVault Code Style Findings

SHV-01C: Ineffectual Usage of Safe Arithmetics

Type	Severity	Location
Language Specific	Informational	SupplyHarvestVault.sol:L147

Description:

The linked mathematical operation is guaranteed to be performed safely by surrounding conditionals evaluated in either `require` checks or `if-else` constructs.

Example:

src/aave-v3/SupplyHarvestVault.sol

SOL

```
145 rewardsFee = rewardsAmount.percentMul(harvestingFeeMem);
146 rewardsFees[i] = rewardsFee;
147 rewardsAmount -= rewardsFee;
```

Recommendation:

Given that safe arithmetics are toggled on by default in `pragma` versions of `0.8.X`, we advise the linked statement to be wrapped in an `unchecked` code block thereby optimizing its execution cost.

Alleviation:

The Morpho team has wrapped the referenced statement as well as the accumulation of fees in an `unchecked` code block. We assume that the Morpho team has validated the addition accumulation cannot overflow and as such mark this exhibit as addressed.

SHV-02C: Inefficient Reward Fee Supply Workflow

Type	Severity	Location
Gas Optimization	Informational	SupplyHarvestVault.sol:L148

Description:

The reward fee is transferred on each iteration for the exact same asset in the current implementation which is inefficient.

Example:

src/aave-v3/SupplyHarvestVault.sol

SOL

```sol
102 function harvest()
103     external
104     returns (
105         address[] memory rewardTokens,
106         uint256[] memory rewardsAmounts,
107         uint256[] memory rewardsFees
108     )
109 {
110     address poolTokenMem = poolToken;
111
112     {
113         address[] memory poolTokens = new address[](1);
114         poolTokens[0] = poolTokenMem;
115         (rewardTokens, rewardsAmounts) = morpho.claimRewards(poolTokens, fa
116     }
117
118     address assetMem = asset();
119     ISwapper swapperMem = swapper;
120     uint16 harvestingFeeMem = harvestingFee;
121     uint256 nbRewardTokens = rewardTokens.length;
122     uint256 toSupply;
123     rewardsFees = new uint256[](nbRewardTokens);
124
125     for (uint256 i; i < nbRewardTokens; ) {
126         uint256 rewardsAmount = rewardsAmounts[i];
127
128         if (rewardsAmount > 0) {
129             ERC20 rewardToken = ERC20(rewardTokens[i]);
130
131             // Note: Uniswap pairs are considered to have enough market dep
```

```
132             // The amount swapped is considered low enough to avoid relying
133             if (assetMem != address(rewardToken)) {
134                 rewardToken.safeTransfer(address(swapperMem), rewardsAmount
135                 rewardsAmount = swapperMem.executeSwap(
136                     address(rewardToken),
137                     rewardsAmount,
138                     assetMem,
139                     address(this)
140                 );
141             }
142
143             uint256 rewardsFee;
144             if (harvestingFeeMem > 0) {
145                 rewardsFee = rewardsAmount.percentMul(harvestingFeeMem);
146                 rewardsFees[i] = rewardsFee;
147                 rewardsAmount -= rewardsFee;
148                 ERC20(assetMem).safeTransfer(msg.sender, rewardsFee);
149             }
150
151             rewardsAmounts[i] = rewardsAmount;
152             toSupply += rewardsAmount;
153
154             emit Harvested(msg.sender, address(rewardToken), rewardsAmount,
155         }
156
157         unchecked {
158             ++i;
159         }
160     }
161
162     morpho.supply(poolTokenMem, address(this), toSupply);
163 }
```

Recommendation:

We advise the total reward fee to be accumulated similarly to the `toSupply` variable and a single transfer to be performed at the end of the function's execution.

Alleviation:

The fees are now accumulated instead and transferred in a single action at the end of the function's execution thereby optimizing the gas cost of the function significantly.

SupplyHarvestVault Code Style Findings

SRC-01C: Ineffectual Usage of Safe Arithmetics

Type	Severity	Location
Language Specific	Informational	SupplyHarvestVault.sol:L170

Description:

The linked mathematical operation is guaranteed to be performed safely by surrounding conditionals evaluated in either `require` checks or `if-else` constructs.

Example:

src/compound/SupplyHarvestVault.sol

SOL

```
169 rewardsFee = rewardsAmount.percentMul(harvestConfigMem.harvestingFee);
170 rewardsAmount -= rewardsFee;
```

Recommendation:

Given that safe arithmetics are toggled on by default in `pragma` versions of `0.8.X`, we advise the linked statement to be wrapped in an `unchecked` code block thereby optimizing its execution cost.

Alleviation:

The Morpho team has wrapped the referenced statement in an `unchecked` code block as advised. While the exhibit has been dealt with, the `rewardsFee` assignment within the `unchecked` code block is redundant as `percentMul` will apply safe arithmetics internally.

SRC-02C: Potentially Redundant Contract Member

Type	Severity	Location
Gas Optimization	Informational	SupplyHarvestVault.sol:L67

Description:

The linked contract member is not utilized within the codebase and thus may be redundant.

Example:

src/compound/SupplyHarvestVault.sol

SOL

```sol
67  address public cComp; // The address of cCOMP token.
```

Recommendation:

We advise it to be evaluated and potentially omitted from the codebase to reduce its initialization gas cost and deployment size.

Alleviation:

The `cComp` member has been removed as it was deemed redundant.

Finding Types

A description of each finding type included in the report can be found below and is linked by each respective finding. A full list of finding types Omniscia has defined will be viewable at the central audit methodology we will publish soon.

External Call Validation

Many contracts that interact with DeFi contain a set of complex external call executions that need to happen in a particular sequence and whose execution is usually taken for granted whereby it is not always the case. External calls should always be validated, either in the form of `require` checks imposed at the contract-level or via more intricate mechanisms such as invoking an external getter-variable and ensuring that it has been properly updated.

Input Sanitization

As there are no inherent guarantees to the inputs a function accepts, a set of guards should always be in place to sanitize the values passed in to a particular function.

Indeterminate Code

These types of issues arise when a linked code segment may not behave as expected, either due to mistyped code, convoluted `if` blocks, overlapping functions / variable names and other ambiguous statements.

Language Specific

Language specific issues arise from certain peculiarities that the Solidity language boasts that discerns it from other conventional programming languages. For example, the EVM is a 256-bit machine meaning that operations on less-than-256-bit types are more costly for the EVM in terms of gas costs, meaning that loops utilizing a `uint8` variable because their limit will never exceed the 8-bit range actually cost more than redundantly using a `uint256` variable.

Code Style

An official Solidity style guide exists that is constantly under development and is adjusted on each new Solidity release, designating how the overall look and feel of a codebase should be. In these types of findings, we identify whether a project conforms to a particular naming convention and whether that convention is consistent within the codebase and legible. In case of inconsistencies, we point them out under this category. Additionally, variable shadowing falls under this category as well which is identified when a local-level variable contains the same name as a contract-level variable that is present in the inheritance chain of the local execution level's context.

Gas Optimization

Gas optimization findings relate to ways the codebase can be optimized to reduce the gas cost involved with interacting with it to various degrees. These types of findings are completely optional and are pointed out for the benefit of the project's developers.

Standard Conformity

These types of findings relate to incompatibility between a particular standard's implementation and the project's implementation, oftentimes causing significant issues in the usability of the contracts.

Mathematical Operations

In Solidity, math generally behaves differently than other programming languages due to the constraints of the EVM. A prime example of this difference is the truncation of values during a division which in turn leads to loss of precision and can cause systems to behave incorrectly when dealing with percentages and proportion calculations.

Logical Fault

This category is a bit broad and is meant to cover implementations that contain flaws in the way they are implemented, either due to unimplemented functionality, unaccounted-for edge cases or similar extraordinary scenarios.

Centralization Concern

This category covers all findings that relate to a significant degree of centralization present in the project and as such the potential of a Single-Point-of-Failure (SPoF) for the project that we urge them to re-consider and potentially omit.

Reentrant Call

This category relates to findings that arise from re-entrant external calls (such as EIP-721 minting operations) and revolve around the inapplicacy of the Checks-Effects-Interactions (CEI) pattern, a pattern that dictates checks (`require` statements etc.) should occur before effects (local storage updates) and interactions (external calls) should be performed last.

Disclaimer

The following disclaimer applies to all versions of the audit report produced (preliminary / public / private) and is in effect for all past, current, and future audit reports that are produced and hosted under Omniscia:

IMPORTANT TERMS & CONDITIONS REGARDING OUR SECURITY AUDITS/REVIEWS/REPORTS AND ALL PUBLIC/PRIVATE CONTENT/DELIVERABLES

Omniscia ("Omniscia") has conducted an independent security review to verify the integrity of and highlight any vulnerabilities, bugs or errors, intentional or unintentional, that may be present in the codebase that were provided for the scope of this Engagement.

Blockchain technology and the cryptographic assets it supports are nascent technologies. This makes them extremely volatile assets. Any assessment report obtained on such volatile and nascent assets may include unpredictable results which may lead to positive or negative outcomes.

In some cases, services provided may be reliant on a variety of third parties. This security review does not constitute endorsement, agreement or acceptance for the Project and technology that was reviewed. Users relying on this security review should not consider this as having any merit for financial advice or technological due diligence in any shape, form or nature.

The veracity and accuracy of the findings presented in this report relate solely to the proficiency, competence, aptitude and discretion of our auditors. Omniscia and its employees make no guarantees, nor assurance that the contracts are free of exploits, bugs, vulnerabilities, deprecation of technologies or any system / economical / mathematical malfunction.

This audit report shall not be printed, saved, disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Omniscia.

All the information/opinions/suggestions provided in this report does not constitute financial or investment advice, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report.

Information in this report is provided 'as is'. Omniscia is under no covenant to the completeness, accuracy or solidity of the contracts reviewed. Omniscia's goal is to help reduce the attack vectors/surface and the high level of variance associated with utilizing new and consistently changing technologies.

Omniscia in no way claims any guarantee, warranty or assurance of security or functionality of the technology that was in scope for this security review.

In no event will Omniscia, its partners, employees, agents or any parties related to the design/creation of this security review be ever liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this security review.

Cryptocurrencies and all other technologies directly or indirectly related to cryptocurrencies are not standardized, highly prone to malfunction and extremely speculative by nature. No due diligence and/or safeguards may be insufficient and users should exercise maximum caution when participating and/or investing in this nascent industry.

The preparation of this security review has made all reasonable attempts to provide clear and actionable recommendations to the Project team (the “client”) with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts in scope for this engagement.

All services, the security reports, discussions, work product, attack vectors description or any other materials, products or results of this security review engagement is provided "as is" and "as available" and with all faults, uncertainty and defects without warranty or guarantee of any kind.

Omniscia will assume no liability or responsibility for delays, errors, mistakes, or any inaccuracies of content, suggestions, materials or for any loss, delay, damage of any kind which arose as a result of this engagement/security review.

Omniscia will assume no liability or responsibility for any personal injury, property damage, of any kind whatsoever that resulted in this engagement and the customer having access to or use of the products, engineers, services, security report, or any other other materials.

For avoidance of doubt, this report, its content, access, and/or usage thereof, including any associated services or materials, shall not be considered or relied upon as any form of financial, investment, tax, legal, regulatory, or any other type of advice.