// Smart Contract Security Assessment 11.11.2024 - 11.22.2024

Morpho V1 Lender Integration

Concrete

HALBORN

Morpho V1 Lender Integration - Concrete

Prepared by: HALBORN

Last Updated 12/06/2024

Date of Engagement by: November 11th, 2024 - November 22nd, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
14	0	0	3	3	8

TABLE OF CONTENTS

8. Automated Testing

1. Introduction

Concrete engaged Halborn to conduct a security assessment of their Morpho V1 Lender Integration project beginning on November 11th and ending on November 22nd. The security assessment was scoped to the smart contracts provided in the repositories sc_spokes-v1 and sc_hub-and-spokes-libraries. Commit hash and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided 1 week for the engagement and assigned one full-time security engineer to review the security of the smart contract in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the Concrete team. The main ones are as follows:

- Improve fallback handling during collateral transfers.
- Mitigate or document potential incompatibilities with fee-on-transfer tokens.
- Prevent precision loss in mathematical conversions.
- Enforce complete NatSpec documentation.
- Ensure division by zero protection.
- Remove redundant and unused code.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (slither).
- Testnet deployment (Foundry).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: sc_spokes-v1

(b) Assessed Commit ID: cb9cfb7

(c) Items in scope:

- src/userBlueprints/MorphoV1UserImplO1.sol
- src/userBlueprints/interfaces/IMorphoV1.sol
- src/userBase/utils/ProtectionHandler.sol

Out-of-Scope: Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: sc_hub-and-spokes-libraries

(b) Assessed Commit ID: d268ac1

(c) Items in scope:

- src/libraries/OracleLibV1.sol
- src/libraries/AddressLib.sol
- src/libraries/ProtectionViewLibV1.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 5b30a9e
- 933402b
- e722bb8
- 35da7ff
- 9e39863
- f90b68b
- b4dad07
- 5e25658
- f88e8aa
- f79fd35
- 8afe14d

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	3	3	8

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
IMPROPER FALLBACK HANDLING IN COLLATERAL TRANSFER DURING FORECLOSURE	MEDIUM	SOLVED - 11/27/2024
POTENTIAL INCOMPATIBILITIES WITH FEE-ON-TRANSFER TOKENS	MEDIUM	RISK ACCEPTED - 12/05/2024
PRECISION LOSS IN CHANGEDENOMINATIONOFPRICE AND RELATED FUNCTIONS	MEDIUM	RISK ACCEPTED - 12/05/2024
POTENTIAL MISUSE OF NON-STANDARD DENOMINATIONS	LOW	SOLVED - 11/27/2024
DIVISION BY ZERO NOT PREVENTED	LOW	SOLVED - 11/27/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
UNSAFE DOWNCASTING	LOW	RISK ACCEPTED - 12/05/2024
INCOMPLETE NATSPEC DOCUMENTATION	INFORMATIONAL	SOLVED - 12/02/2024
MULTIPLE TYPOS	INFORMATIONAL	SOLVED - 12/03/2024
COMMENTED-OUT CODE	INFORMATIONAL	SOLVED - 12/03/2024
STYLE GUIDE OPTIMIZATIONS	INFORMATIONAL	SOLVED - 12/03/2024
UNUSED IMPORTS, VARIABLE, AND INTERFACE DECLARATION	INFORMATIONAL	SOLVED - 12/03/2024
REDUNDANT INHERITANCE IN MORPHOV1USERIMPL01	INFORMATIONAL	SOLVED - 12/03/2024
LACK OF EVENT EMISSION	INFORMATIONAL	SOLVED - 12/03/2024
MAGIC NUMBERS IN USE	INFORMATIONAL	SOLVED - 12/03/2024

7. FINDINGS & TECH DETAILS

7.1 IMPROPER FALLBACK HANDLING IN COLLATERAL TRANSFER DURING FORECLOSURE

// MEDIUM

Description

The `_executeForeclosure()` function in `ProtectionHandler` is a critical internal function called by the external `executeForecloseByFundsRequester()` in `UserBaseV1`. This function attempts to transfer any remaining collateral to the `owner`. If the `transfer()` call fails, the function attempts to fallback to the "pull" pattern by increasing the allowance for the owner. However, this fallback mechanism does not handle scenarios where `transfer()` fails and does **not revert but returns** `false`. In such cases, the `catch` block will not execute, leaving the collateral funds trapped in the contract and inaccessible to the owner.

See the vulnerable code:

```
try IERC20(params.collateralToken).transfer(owner, params.collateralAmountL
catch {
  IERC20(params.collateralToken).safeIncreaseAllowance(owner, params.collat
}
```

If `IERC20.transfer()` does not revert but instead returns `false`, the `catch` block will not execute. As a result, the `safeIncreaseAllowance()` fallback will not run, and the `params.collateralAmountLeft` will remain locked in the contract. This creates a significant issue, especially when interacting with non-standard ERC20 tokens that do not conform to the expected behavior of `transfer()`.
The failure to handle such edge cases could lead to a situation where the owner cannot retrieve their collateral, effectively resulting in a loss of funds. Given that this function is part of the foreclosure process, it is crucial to ensure that all remaining collateral is properly returned to the owner.

BVSS

AO:A/AC:L/AX:L/C:N/I:H/A:H/D:H/Y:H/R:P/S:U (6.6)

Recommendation

To address this issue, the fallback logic should be revised as follows:

1. **Use** `safeTransfer()` from OpenZeppelin's `SafeERC20` library instead of `IERC20.transfer()` for the initial attempt to transfer the collateral. This guarantees proper execution and reverts on failure, ensuring the fallback mechanism is triggered as expected.
2. **Implement robust fallback logic** by retaining the `try–catch` mechanism and using `safeTransfer()` within it. However, in the fallback, explicitly check for any scenarios where `safeTransfer()` may fail to

execute properly and handle accordingly.
3. **Add an event to track fallback allowance usage** to enhance transparency and assist in debugging. Emit an event whenever the fallback `safeIncreaseAllowance()` is executed, recording the owner and the amount allowed.

By following these steps, the process of transferring collateral during foreclosure will be more robust, reliable, and secure, ensuring that funds are either returned directly or made accessible to the owner.

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `5b30a9e` by implementing their own version of the `safeTransfer()` function. Notice this new behavior removed the previous intention to give allowance to the `owner` by reverting if the attempt to refund fails.

Remediation Hash

https://github.com/Blueprint-Finance/sc_spokes-v1/commit/5b30a9e4b900af8253f123056449a071d849990c

7.2 POTENTIAL INCOMPATIBILITIES WITH FEE-ON-TRANSFER TOKENS

// MEDIUM

Description

The `_supply()` and `_withdraw()` functions in the `MorphoV1UserImpl01` contract do not account for fee-on-transfer tokens. These tokens deduct a fee during transfers, meaning the amount received or sent may differ from the user-specified `amount` parameter. The contract currently assumes that the specified `amount` equals the transferred amount, leading to potential inconsistencies and vulnerabilities.

For example, the `_supply()` function contains the following code:

```
function _supply(address token, uint256 amount, uint256 modality) internal
    ...

    if (mode != SendFundsModality.ONLY_SECOND_STEP) {
        // either we send through, or we only do the first step (supply col
        IERC20(token).safeTransferFrom(_owner(), address(this), amount);
    }
    if (mode != SendFundsModality.ONLY_FIRST_STEP) {
        // either we send through, or we only do the second step (supply co
        MORPHO.supplyCollateral(_getMarketParams(), amount, address(this),
    }
}
```

This function transfers the specified `amount` from the user and assumes the entire amount is successfully received, without verifying the actual balance increase.

See also the `_withdraw()` function:

```
function _withdraw(address token, uint256 amount, uint256 modality) interna
    // retrieve the mode of sending funds
    // 0 = protocol -> user, 1 = protocol -> userProxy, 2 = userProxy -> us
    SendFundsModality mode = SendFundsModality(modality.decodeMode());

    amount = (amount == type(uint256).max) ? _getLenderSupplied() : amount;

    if (mode != SendFundsModality.ONLY_SECOND_STEP) {
        // either we send through, or we only do the first step (withdraw c
        MORPHO.withdrawCollateral(
            _getMarketParams(),
            amount,
            address(this),
            (mode == SendFundsModality.SEND_THROUGH) ? _owner() : address(t
```

```
            );
        } else {
            // Transfer the amount from the contract to the user
            IERC20(token).safeTransfer(_owner(), amount);
        }
    }
```

This function withdraws the specified `amount` and assumes the full amount is successfully transferred to the user, which may not hold true for fee-on-transfer tokens.

BVSS

AO:A/AC:L/AX:L/C:N/I:H/A:H/D:H/Y:H/R:P/S:U (6.6)

Recommendation

To mitigate this issue, implement checks to validate the actual amount transferred during `safeTransfer()` and `safeTransferFrom()` operations. A balance-based approach can help ensure accurate accounting:

1. **Record the Contract Balance:**
   - Before performing the transfer operation, store the current balance of the contract for the relevant token.
2. **Execute the Transfer:**
   - Perform the `safeTransfer()` or `safeTransferFrom()` operation.
3. **Calculate the Actual Amount:**
   - Subtract the pre-transfer balance from the post-transfer balance to determine the actual amount received or sent.

Or, alternatively, clearly document that fee-on-transfer tokens are not supported by the protocol, ensuring users are aware of this limitation.

Remediation

RISK ACCEPTED: The **Concrete team** indicated that they will not update the code according to the recommendation, indicating:
If we ever need to integrate with a lender that accepts tokens with fee-on-transfer features, we will need to create a new user blueprint.

7.3 PRECISION LOSS IN CHANGEDENOMINATIONOFPRICE AND RELATED FUNCTIONS

// MEDIUM

Description

The `changeDenominationOfPrice()` function in `OracleLibV1`, along with related functions such as `convertStandardUnitsToSmallestUnitsPriceQuote()` and `convertSmallestUnitsToStandardUnitsPriceQuote()`, can experience precision loss during conversions, particularly when converting between systems with differing decimal scales. Solidity's integer arithmetic truncates fractional components, resulting in underrepresented values. While expected in some cases, this behavior can lead to discrepancies in financial calculations, especially when conversions are frequent or large.

Notably, the examples in the `NatSpec` documentation and test files only consider the case where the `oldDenomination` is less than the `newDenomination`. No instances of this function were found in use across the two codebases within the scope of this security assessment, indicating it may be unused.

```
///@notice Converts a price quote from one denomination to another
///@param price The price in the old denomination
///@param oldDenomination The old denomination of the price
///@param newDenomination The new denomination of the price
///@return priceNewDenomination The price in the new denomination
///@dev For example, say we quote the price of WETH in terms of USDC at an
///@dev Now, converting to another denomination (i.e. accuracy) of 10^18, w
function changeDenominationOfPrice(uint256 price, uint256 oldDenomination,
  internal
  pure
  returns (uint256)
{
    return price.mulDiv(newDenomination, oldDenomination);
}
```

Consider the input:

- **Price**: `123456789012345678` (representing `0.123456789012345678` tokens in an 18-decimal system)
- **Old Denomination**: `10^18` (Wei for an 18-decimal token)
- **New Denomination**: `10^6` (micro-units for a 6-decimal token)

Expected Result: The result should ideally be approximately `123456.789012` when scaled down, preserving the fractional component.
Actual Behavior: Due to Solidity's integer division truncation, the output is `123456`, causing a loss of the fractional part (`0.789012`).

While this precision loss may seem minor, in financial or trading applications involving large or frequent transactions, accumulated truncation errors can result in misrepresented values and potential discrepancies in calculations.

Proof of Concept

The following test case illustrates the problem:

```
function test_changeDenominationOfPrice_precisionLoss() public {
    // Set up a number in scale 18
    uint256 price = 1234567890123456789; // A valid number representing toke
    uint256 oldDenomination = 10 ** 18; // Old denomination (e.g., Wei for
    uint256 newDenomination = 10 ** 6; // New denomination (e.g., micro-uni

    uint256 newPrice = OracleLibV1.changeDenominationOfPrice(price, oldDeno

    // Log the actual result for visual confirmation during testing
    console.log("Actual result after scaling down:", newPrice);
}
```

Run the test using:

```
forge test --mp test/OracleLibV1.t.sol --mt test_changeDenominationOfPrice_
```

Observe that the new price is truncated, demonstrating the precision loss.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:L/Y:L/R:P/S:C (4.7)

Recommendation

To mitigate this potential issue, you might consider the following options:

- **Document the Precision Loss**: Clearly indicate in the documentation that precision may be lost when converting between different decimal systems.
- **Implement a Rounding Mechanism**: Implement a consistent rounding strategy (e.g., rounding up or down) to manage fractional results more predictably.
- **Revert on Significant Loss**: Include logic that reverts the function if precision loss surpasses a defined threshold.
- **Remove the Function**: Since the function is currently unused, consider removing it from the codebase to simplify the library and avoid potential future issues.

Remediation

RISK ACCEPTED: The **Concrete team** indicated that they will not update the code according to the recommendation.

7.4 POTENTIAL MISUSE OF NON-STANDARD DENOMINATIONS

// LOW

Description

In the `OracleLibV1` library, the `getDecimalsFromDenomination()` function iteratively divides a given `denomination` by 10 until it becomes less than 10, counting the number of divisions as the "decimal places." However, this logic does not verify if the input `denomination` is a valid power of 10. If a non-power-of-ten input, such as `999999`, is provided, the function will return an incorrect result rather than reverting.
For example:

- Input: `999999`
- Expected Behavior: Revert or return `0` for invalid denominations.
- Actual Behavior: Returns a non-zero value, incorrectly treating `999999` as a valid denomination.

See the affected code:

```
function getDecimalsFromDenomination(uint256 denomination) internal pure re
    uint8 decimals = 0;
    while (denomination >= 10) {
        denomination /= 10;
        decimals++;
    }
    return decimals;
}
```

Without validation, this function may lead to incorrect assumptions about the validity of the input denomination, potentially introducing inconsistencies in dependent calculations.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:M/Y:M/R:P/S:U (4.4)

Recommendation

Add validation to ensure the input is a valid power of 10. If not, the function should revert with a clear error message. For example:

```
function getDecimalsFromDenomination(uint256 denomination) internal pure re
    if(denomination == 0 || denomination % 10 != 0) revert Errors.InvalidDe

    uint8 decimals = 0;
    while (denomination >= 10) {
        denomination /= 10;
```

```
        decimals++;
    }
    return decimals;
}
```

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `933402b` by adding the recommended validations.

Remediation Hash

https://github.com/Blueprint-Finance/sc_hub-and-spokes-libraries/commit/933402bd06273adaecc787370ef6ce14e1783b1e

7.5 DIVISION BY ZERO NOT PREVENTED

// LOW

Description

In `ProtectionViewLibV1`, the functions `surpassCriticalLtvAfterWithdraw()` and `surpassCriticalLtvAfterBorrow()` fail to account for division by zero scenarios, which can cause the contract to revert unexpectedly. This introduces a risk of denial of service in edge cases, leading to contract failures during critical calculations.

In `surpassCriticalLtvAfterWithdraw()`:

- The `currentLtv` argument is directly used in a division operation:

```
uint256 criticalOverCurrentInWad = criticalLtvInWad.mulDiv(WAD, currentLtv)
```

- The `priceQuoteDenomination` argument is used in a denominator:

```
withdrawAmountInTokens.mulDiv(WAD, currentDebtInBorrowTokens).mulDiv(
  priceOfCollateralTokenInBorrowToken, priceQuoteDenomination
);
```

In `surpassCriticalLtvAfterBorrow()`:

- The `currentSupplyInCollateral` argument is directly used in a denominator:

```
borrowAmountInTokens.mulDiv(WAD, currentSupplyInCollateral);
```

- The `priceOfCollateralTokenInBorrowToken` argument is also used as a denominator:

```
mulDiv(priceQuoteDenomination, priceOfCollateralTokenInBorrowToken);
```

If any of these arguments is 0, this will result in a division by zero, reverting the contract.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:N/S:C (3.9)

Recommendation

Introduce validation checks for critical inputs to ensure they are non-zero before performing any division operations.

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `e722bb8` by Introducing validation checks as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_hub-and-spokes-libraries/commit/e722bb8ebf9285b9679f46c0edbea0ac31e3e4a9

7.6 UNSAFE DOWNCASTING

// LOW

Description

In the `AddressLib` library in scope, direct downcasting is used mainly to convert `uint256` values to `uint8` or `uint160`. Unsafe type casting vulnerabilities can occur when the converted value does not fit within the bounds of the target type, potentially leading to unexpected results or overflow.

The following lines of code show direct castings found in `src/libraries/AddressLib.sol`:

```
tokenData = bytes32((uint256(denomination) << 160) | uint256(uint160(token))
...
return uint8(addressDecimalsData >> 160);
...
return uint8(addressDecimalsData >> 248);
...
return bytes32((uint256(eid) << 160) | uint256(uint160(addr)));
...
return bytes32((number << 160) | uint256(uint160(addr)));
...
return address(uint160(encoded));
...
return address(uint160(encoded));
...
return uint32(loanId >> 160);
...
return uint64(loanId >> 192);
```

BVSS

AO:A/AC:L/AX:M/C:N/I:L/A:L/D:N/Y:N/R:N/S:U (2.1)

Recommendation

Consider using OpenZeppelin's `SafeCast` library to safely downcast integers.

Remediation

RISK ACCEPTED: The **Concrete team** indicated that they will not update the code according to the recommendation.

7.7 INCOMPLETE NATSPEC DOCUMENTATION

// INFORMATIONAL

Description

The `MorphoV1UserImpl01.sol` and `IMorphoV1.sol` lack comprehensive NatSpec comments for functions, state variables, and contracts. NatSpec is a widely adopted standard for documenting Solidity contracts, providing clear and structured explanations for developers, auditors, and end-users.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:F/S:U (0.6)

Recommendation

Adopt and implement NatSpec comments across all contracts, functions, and state variables. Use the following structure as a guideline:

1. **Contract-Level Documentation:** Include a brief overview of the contract's purpose, scope, and high-level details. For example:
2. **Function-Level Documentation:** Document each function using NatSpec annotations, detailing:

- `@param` descriptions for function parameters.
- `@return` descriptions for return values.
- `@dev` notes for developers about implementation details or caveats.
- `@notice` for user-facing descriptions.

3. **State Variable Documentation:** Document each variable with a concise explanation of its purpose and usage.
4. **Global Guidelines:**

- Use tools like `solhint` to enforce NatSpec standards.
- Regularly review documentation to ensure alignment with code updates.

By adding NatSpec documentation, the project can improve code clarity, facilitate audits, and enhance developer and user trust in the protocol.

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `35da7ff` by improving NatSpec comments as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_spokes-v1/commit/35da7fff07acd9d3af38180f7147a05a1a9f212b

7.8 MULTIPLE TYPOS

// INFORMATIONAL

Description

Several typographical errors were found in the code and comments throughout the project:

- In `src/userBlueprints/MorphoV1UserImpl01.sol`:

```
REWARD_QUOTE_GRACE_PRERIOD,
```

`REWARD_QUOTE_GRACE_PRERIOD` should be `REWARD_QUOTE_GRACE_PERIOD` instead.

- In `src/userBase/utils/ProtectionHandler.sol`:

```
RepayConcretDebtStruct,
...
RepayConcretDebtStruct memory r;
```

`RepayConcretDebtStruct` should be `RepayConcreteDebtStruct` instead.

```
// and subsequently concrete withdraws its credit injections out fo the len
```

`out fo the` should be `out for the` instead.

```
// bookkeeping the withdrawl
```

`withdrawl` should be `withdrawal` instead.

```
/// @dev It the repay mode is FromUserWithApproval, the fee is taken from t
```

`sceanrio` should be `scenario` instead.

- In `src/libraries/AddressLib.sol`:

```
if (number > type(uint96).max) revert Errors.ExceedsUint96MaxPrecission();
```

`ExceedsUint96MaxPrecission` should be `ExceedsUint96MaxPrecision` instead.

- In `src/libraries/ProtectionViewLibV1.sol`:

```
// number of claims is the slot which also keeps the boolen flag
```

`boolen` should be `boolen` instead.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

To maintain clarity and trustworthiness, it is essential to rectify any typographical errors present within the contracts. Correcting such errors minimizes the likelihood of confusion and reinforces confidence in the accuracy and integrity of the documentation.

Remediation

SOLVED: The **Concrete team** fixed this finding in commits bd53f65 and 9e39863.

Remediation Hash

https://github.com/Blueprint-Finance/sc_hub-and-spokes-libraries/commit/9e398634bb3b900cdf3532295b9b00d4864cf8a3

7.9 COMMENTED-OUT CODE

// INFORMATIONAL

Description

During the audit, multiple instances of commented-out code were observed throughout the codebase. While these elements do not inherently introduce security vulnerabilities, they pose certain risks that should be considered:

1. **Codebase Noise and Confusion:** Commented-out code can obscure the active logic, making the code harder to read and maintain. Developers may misinterpret the purpose of such code or assume it is still in use.
2. **Unintentional Execution:** If uncommented during future updates, the commented code may introduce errors or vulnerabilities, especially if it is outdated or incompatible with the current system.
3. **Disclosure of Deprecated or Experimental Features:** Attackers analyzing the code may infer potential functionality or weaknesses based on commented-out segments, especially if they reveal experimental or untested logic.

Examples of commented-out code include:

```
function _lenderSpecificAssetValidation(address collateralAsset_, address b
    internal
    view
    override(TCustomHooks)
{
    if (COLLATERAL_TOKEN_INFO.getAddress() != collateralAsset_) revert Erro
    if (borrowedAsset_ != address(0)) {
        if (LOAN_TOKEN_INFO.getAddress() != borrowedAsset_) revert Errors.A
    }
    // MorphoV1Helper.assetValidation(COLLATERAL_TOKEN_INFO, collateralAsse
}
```

and:

```
function _viewTotalClaimableRewardAmountInBase()
    internal
    view
    virtual
    override(TViewUserPosition)
    returns (uint256 amount, uint256 denomination)
{
    // return (0, BASE_DENOMINATION);
}
```

and:

```
function _isOracleQuoteInStandardUnits() internal view virtual override(TOr
    // AddressLib.getFlag(ORACLE_INFO) == 1;
    return false;
}
```

and:

```
function _getLenderDebt() internal view override(TViewUserPosition) returns
    amount = MorphoV1Helper.getLenderDebt(MORPHO, _getMarketParams());
    // amount = MorphoBalancesLib.expectedBorrowAssets(MORPHO, _getMarketPa
}
```

and:

```
function _repayConcreteDebt(
    uint256 debt,
    uint256 amountInToken,
    DebtRepayMode repayMode,
    uint256 feeInToken,
    address beneficiary
) internal {
    RepayConcretDebtStruct memory r;

    // if (amountInToken > debt) {
    //     // includes amountInToken == type(uint256).max
    //     amountInToken = debt;
    // }
...
```

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Consider the following recommendations:

1. **Eliminate Commented-Out Code:** Remove commented-out code that is no longer needed. This reduces clutter and ensures that the active logic is clear and maintainable.
2. **Include Documentation for Retained Comments:** If certain sections are retained for reference, include clear documentation explaining their purpose and whether they are part of plans or deprecated features.

3. **Resolve ToDo Comments:** Ensure all ToDo comments are resolved and removed from the codebase before deployment. Leaving unfinished work in comments can give attackers insights into incomplete or untested functionality.
4. **Final Review Before Deployment:** Conduct a thorough review of the code before deployment to confirm that all ToDos and commented-out segments are addressed, and the code reflects its intended deployment state. This helps maintain a clean, readable, and secure codebase.
By addressing these recommendations, the codebase can be kept clean, maintainable, and less prone to unintended errors or disclosures.

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `f90b68b` by removing the commented-out code as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_spokes-v1/commit/f90b68b4d54525f43ee968c39f5afd5382c4db46

7.10 STYLE GUIDE OPTIMIZATIONS

// INFORMATIONAL

Description

In Solidity development, adhering to the official style guide is best practice to ensure code consistency, readability, and maintainability. Throughout the contracts, there are several instances where the code does not follow these guidelines. Some examples include:

- The `internal` variable `encodedTotalRewardInBaseAndTimestamp` in `MorphoV1UserImpl01` does not begin with an underscore:

```
uint256 internal encodedTotalRewardInBaseAndTimestamp;
```

- The `TreasuryAndRevenueSplit` variable in `ProtectionHandler` is not in `mixedCase`:

```
(address claimRouter, bytes32 TreasuryAndRevenueSplit) =
  REMOTE_REGISTRY.getClaimRouterAndEncodedTreasuryAndRevenueSplitInWAD(fals
```

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Apply the following style guide improvements throughout the codebase:

- `Internal` and `private` variables and functions names should begin with an underscore:

```
uint256 internal _encodedTotalRewardInBaseAndTimestamp;
```

- Variable names should be in `mixedCase`:

```
(address claimRouter, bytes32 treasuryAndRevenueSplit) =
  REMOTE_REGISTRY.getClaimRouterAndEncodedTreasuryAndRevenueSplitInWAD(fals
```

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `b4dad07` by applying the recommended style improvements.

Remediation Hash

https://github.com/Blueprint-Finance/sc_spokes-v1/commit/b4dad077638af4e78ce7368424f5219db3966dd5

7.11 UNUSED IMPORTS, VARIABLE, AND INTERFACE DECLARATION

// INFORMATIONAL

Description

During the security assessment of the smart contracts, several instances of unused imports, variables, and interface declarations were identified. These unnecessary components can clutter the codebase, reduce readability, and potentially lead to confusion during development or auditing. Additionally, unused imports may slightly increase the compiled contract's bytecode size, affecting deployment and execution costs.

Unused Imports:

- In `MorphoV1UserImpl01.sol`:

```
import {IOracle} from "@morpho-org/morpho-blue/src/interfaces/IOracle.sol";
...
import {MorphoStorageLib} from "@morpho-org/morpho-blue/src/libraries/periph
...
import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";
...
import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/extensions
...
import {IRewardsMetaHandlerV1} from "../rewardHandler/interfaces/IRewardsMet
...
import {
    WAD,
    BP,
    BASE_DENOMINATION,
    MORPHO_ORACLE_DECIMALS,
    REWARD_QUOTE_GRACE_PRERIOD,
    PRICE_QUOTE_STANDARD_UNIT_DENOM,
    PRICE_QUOTE_SMALLEST_UNIT_DENOM
} from "../helpers/Constants.sol";
...
import {TokenType, SendFundsModality} from "../helpers/DataTypes.sol";
```

The following imports are not utilized: `IOracle`, `MorphoStorageLib`, `SafeCast`, `IERC20Metadata`, `IRewardsMetaHandlerV1`, `WAD`, `BP`, `REWARD_QUOTE_GRACE_PRERIOD`, and `TokenType`.

- In `ProtectionHandler.sol`:

```
import {WAD, BASE_DENOMINATION, FALLBACK_LTV_BUFFER_IN_WAD, MAX_SLIPPAGE_IN
```

The following constants are not utilized: `BASE_DENOMINATION`, `FALLBACK_LTV_BUFFER_IN_WAD`, and `MAX_SLIPPAGE_IN_WAD`.

Unused Variable:

- In `MorphoV1UserImpl01.sol`:

```
uint256 internal encodedTotalRewardInBaseAndTimestamp;
```

Unused Interface:

- In `IMorphoV1.sol`:

```
interface IClaimMorphoRewards {
```

This interface is defined but not referenced anywhere in the codebase.

Unused code increases the attack surface, adds maintenance overhead, and risks being integrated in the future without proper testing, potentially introducing vulnerabilities or confusion.

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

To maintain a clean, readable, and secure codebase, consider the following actions:

- **Remove Unused Imports:** Eliminate the unused imports in `MorphoV1UserImpl01.sol` and `ProtectionHandler.sol`. This reduces the codebase's complexity and ensures only relevant dependencies are present.
- **Remove Unused Variables:** Remove the unused state variable `encodedTotalRewardInBaseAndTimestamp` from `MorphoV1UserImpl01.sol`.
- **Remove Unused Interface:** If `IClaimMorphoRewards` is not intended for future use, remove it from the codebase. If it is part of upcoming development, ensure proper documentation of its purpose and test coverage during implementation.

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `5e25658` by removing the unused code as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_spokes-v1/commit/5e25658241d65093977a925ee478c01156b6edce

7.12 REDUNDANT INHERITANCE IN MORPHOV1USERIMPL01

// INFORMATIONAL

Description

The `MorphoV1UserImpl01` contract includes redundant inheritance, which unnecessarily increases the complexity of the contract. Specifically, it inherits from `Initializable`, which is already inherited by `UserBaseV1`.

```
contract MorphoV1UserImpl01 is IInitializeUserBlueprint, Initializable, Use
```

The `UserBaseV1` contract already inherits from `Initializable`:

```
abstract contract UserBaseV1 is
    Initializable,
    IProtocolIntervention,
    IUserIntervention,
    IViewUserState,
    IReclaimRepayCancel,
    TManagePosition,
    TViewUserPosition,
    TOracleConnector,
    RemoteRegistryConnector,
    FundsRequesterConnector,
    SwapperConnector,
    ExecutorConnector,
    TokenInfo,
    ProtectionHandler
{
```

This creates a redundancy that can complicate contract structure and increase the risk of potential issues related to Solidity's C3 linearization for resolving multiple inheritance.

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

To simplify the contract and eliminate potential linearization issues, consider removing the redundant `Initializable` inheritance from `MorphoV1UserImpl01`:

```
contract MorphoV1UserImpl01 is IInitializeUserBlueprint, UserBaseV1 {
```

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `f88e8aa` by removing the redundant inheritance as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_spokes-v1/commit/f88e8aa8e8a406f97608329f4fe697fe2e36347d

7.13 LACK OF EVENT EMISSION

// INFORMATIONAL

Description

It has been observed that some functionalities are missing event emissions. Events are essential for notifying external observers about critical state changes within a smart contract. They allow transaction initiators and external monitoring tools to track actions, enhancing transparency and usability.

Failing to emit events can lead to:

- Reduced transparency of critical state changes.
- Challenges for users and developers in debugging and auditing.
- Missed opportunities for off-chain systems to react to state changes effectively.

Examples:

- In `src/userBlueprints/MorphoV1UserImpl01.sol`:
  - `initialize()`
  - `supply()`
  - `withdraw()`
  - `borrow()`
  - `repay()`
- In `src/userBase/utils/ProtectionHandler.sol`:
  - `_openProtectionPolicy()`
  - `_executeForeclosure()`
  - `_updateProtectionState()`

This list is not exhaustive, and a thorough review of the entire codebase is recommended to identify additional instances where event emissions can improve transparency and traceability.

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

All functions updating important parameters should emit events.

Remediation

SOLVED: The **Concrete team** fixed this finding in commits `f79fd35` and `b25bb2d` by adding events as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_spokes-v1/commit/f79fd3578b4e21b02fb3ad04715f8421d2274aca

7.14 MAGIC NUMBERS IN USE

// INFORMATIONAL

Description

In programming, `magic numbers` refers to the use of unexplained numerical or string values directly in code, without any clear indication of their purpose or origin. The use of magic numbers can lead to confusion and make your code more difficult to understand, maintain, and update.

To improve the readability and maintainability of your smart contracts, it is recommended to avoid using magic numbers and instead use named constants or variables to represent these values. By doing so, you provide clear context for the values, making it easier for developers to understand their purpose and significance.

Several examples of magic numbers were found in `src/libraries/AddressLib.sol`:

```
tokenData = bytes32((uint256(denomination) << 160) | uint256(uint160(token)
...
return uint256(tokenData >> 160);
...
return (uint256(flag) << 248) | (uint256(decimals) << 160) | uint256(uint16
...
return uint8(addressDecimalsData >> 160);
...
return uint8(addressDecimalsData >> 248);
...
return bytes32((uint256(eid) << 160) | uint256(uint160(addr)));
...
return uint32(uint256(addressEidData) >> 160);
...
return bytes32((number << 160) | uint256(uint160(addr)));
...
return uint256(addressNumberData >> 160);
...
loanId = (uint256(index) << 192) | (uint256(chainId) << 160) | uint256(uint
...
return uint32(loanId >> 160);
...
return uint64(loanId >> 192);
```

This list is not exhaustive, and it is recommended to review the entire codebase to identify additional instances where magic numbers are used.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

To improve code maintainability, readability, and reduce the risk of potential errors, it is recommended to replace magic numbers with well-defined constants. By using constants, developers can provide clear and descriptive names for specific values, making the code easier to understand and maintain. Additionally, updating the values becomes more straightforward, as changes can be made in a single location, reducing the risk of errors and inconsistencies. For large numbers, consider using scientific notation (e.g., `1e4`).

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `8afe14d` by replacing the magic numbers with well-defined constants.

Remediation Hash

https://github.com/Blueprint-Finance/sc_hub-and-spokes-libraries/commit/8afe14d35539e8e7bcb432ef159b357b83edf750

STATIC ANALYSIS REPORT

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.

Output

```
INFO:Detectors:
ProtectionHandler._executeForeclosure(uint256,uint256,uint256,address) (src/userBase/utils/ProtectionHandler.sol#321-415) ignores return value by IERC20(params.collateralToken).transfer(o
r,params.collateralAmountLeft) (src/userBase/utils/ProtectionHandler.sol#407-410)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
ProtectionHandler._executeForeclosure(uint256,uint256,uint256,address) (src/userBase/utils/ProtectionHandler.sol#321-415) uses a dangerous strict equality:
        - params.collateralAmountLeft == 0 (src/userBase/utils/ProtectionHandler.sol#402)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
ProtectionHandler._repayConcreteDebt(uint256,uint256,DebtRepayMode,uint256,address).r (src/userBase/utils/ProtectionHandler.sol#432) is a local variable never initialized
ProtectionHandler._claimProtection(uint256,uint256,uint256,address).params (src/userBase/utils/ProtectionHandler.sol#217) is a local variable never initialized
MorphoV1UserImpl01._repay(address,uint256,uint256).shares (src/userBlueprints/MorphoV1UserImpl01.sol#177) is a local variable never initialized
UserBaseV1.cancelPolicy(uint256,uint256).debt (src/userBase/UserBaseV1.sol#305) is a local variable never initialized
ProtectionHandler._executeForeclosure(uint256,uint256,uint256,address).params (src/userBase/utils/ProtectionHandler.sol#329) is a local variable never initialized
UserBaseV1.reclaimDebtOrForeclose(uint256,address).collateralTokenInfo (src/userBase/UserBaseV1.sol#370) is a local variable never initialized
UserBaseV1.reclaimDebtOrForeclose(uint256,address).borrowTokenInfo (src/userBase/UserBaseV1.sol#369) is a local variable never initialized
ProtectionHandler._openProtectionPolicy(uint256,uint256,uint256,uint256,uint256,bytes32,bytes32).tempAddresses (src/userBase/utils/ProtectionHandler.sol#112) is a local variable never ini
lized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
ProtectionHandler._repayConcreteDebt(uint256,uint256,DebtRepayMode,uint256,address) (src/userBase/utils/ProtectionHandler.sol#425-582) ignores return value by (None,r.creditInToken) = r.c
itInfo.decodeCreditAmount() (src/userBase/utils/ProtectionHandler.sol#533)
```

```
INFO:Detectors:
ProtectionViewLibV1.checkAmountsFromInfo(uint256,uint256,uint256,AmountType,uint256).expectedAmount (src/libraries/ProtectionViewLibV1.sol#184) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
ProtectionViewLibV1.getProtectionType(uint256,uint256,uint256) (src/libraries/ProtectionViewLibV1.sol#28-62) ignores return value by (None,debtInToken) = creditInfo.decodeCreditAmount()
src/libraries/ProtectionViewLibV1.sol#50)
ProtectionViewLibV1.getCurrentInterventionLtv(uint256,uint256,uint256,uint256) (src/libraries/ProtectionViewLibV1.sol#72-112) ignores return value by (None,debtInToken) = creditInfo.deco
eCreditAmount() (src/libraries/ProtectionViewLibV1.sol#98)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.