// Security Assessment

08.12.2024 - 09.10.2024

Spokes V1

Blueprint Finance

HALBORN

Spokes V1 - Blueprint Finance

Prepared by: HALBORN

Last Updated Unknown date

Date of Engagement: August 12th, 2024 - September 10th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
24	1	3	8	4	8

TABLE OF CONTENTS

1. Introduction

`Concrete` engaged `Halborn` to conduct a security assessment on their smart contracts beginning on 2024-08-12 and ending on 2024-09-10. The security assessment was scoped to the smart contracts provided in directly be the team.

Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

`Halborn` was provided four weeks for the engagement and assigned one full-time security engineer to check the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, `Halborn` identified several security concerns that were mostly addressed by the `Concrete team.`

3. Test Approach And Methodology

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

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Static Analysis of security for scoped contract, and imported functions. (`Slither`).
- Local or public testnet deployment (`Foundry` , `Remix IDE`).

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:

EXPLOITABILITY 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

EXPLOITABILITY 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 (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C: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

REPOSITORY

(a) Repository: sc_spokes-v1

(b) Assessed Commit ID: 984a7f0

(c) Items in scope:

- src/libraries/ProtectionViewLib.sol
- src/userBlueprints/UserRadiantV2.sol
- src/userBlueprints/UserAaveV3.sol
- src/userBlueprints/UserSiloV1.sol
- src/userBlueprints/UserCompoundV3.sol
- src/inspection/Inspector.sol
- src/oracleProvider/traits/TPriceProvider.sol
- src/oracleProvider/implementations/ChainlinkPriceProvider.sol
- src/oracleProvider/PriceProviderBase.sol
- src/executors/RemotePublicExecutor.sol
- src/executors/RemoteProtocolProxy.sol
- src/executors/RemoteExecutorsBookkeeper.sol
- src/addresses/EthereumMainnet.sol
- src/swapper/traits/TSwap.sol
- src/swapper/SwapperBase.sol
- src/swapper/implementations/SwapperBEX.sol
- src/swapper/implementations/SwapperUniV3.sol
- src/remoteRegistry/Readme.md
- src/remoteRegistry/CloneFactory.sol
- src/remoteRegistry/RemoteRegistryEvents.sol
- src/remoteRegistry/RemoteRegistry.sol
- src/userBase/UserBase.sol
- src/userBase/traits/TManagePosition.sol
- src/userBase/traits/TCustomHooks.sol
- src/userBase/traits/TViewUserPosition.sol
- src/userBase/traits/TUserBlueprintSettings.sol
- src/userBase/traits/TOracleConnector.sol
- src/userBase/utils/TokenInfo.sol
- src/userBase/utils/SwapperConnector.sol
- src/userBase/utils/ExecutorConnector.sol
- src/userBase/utils/FundsRequesterConnector.sol
- src/userBase/utils/ProtectionHandler.sol
- src/userBase/utils/RemoteRegistryConnector.sol
- src/fundsRequester/FundsRequester.sol
- src/portfolioProxy/PortfolioProxy.sol
- src/portfolioProxy/PortfolioProxyEvents.sol

- src/helpers/Errors.sol
- src/helpers/DataTypes.sol
- src/helpers/Constants.sol
- src/userBlueprints/interfaces/ICompoundV3.sol
- src/userBlueprints/interfaces/IUserBlueprint.sol
- src/userBlueprints/interfaces/IRadiantV2.sol
- src/userBlueprints/interfaces/IUserRadiantV2.sol
- src/userBlueprints/interfaces/IAaveV3.sol
- src/userBlueprints/interfaces/ISiloV1.sol
- src/userBlueprints/interfaces/IUserAaveV3.sol
- src/portfolioProxy/interfaces/IPortfolioProxy.sol
- src/userBase/interfaces/IInitializeUserBlueprint.sol
- src/userBase/interfaces/IProtocolIntervention.sol
- src/userBase/interfaces/IReclaimRepayCancel.sol
- src/userBase/interfaces/IProtocolResponse.sol
- src/userBase/interfaces/IViewUserState.sol
- src/userBase/interfaces/IViewUserStateExtended.sol
- src/userBase/interfaces/IUserIntervention.sol
- src/executors/interfaces/IRemotePublicExecutor.sol
- src/executors/interfaces/IRemoteProtocolProxy.sol
- src/executors/interfaces/IRemoteExecutorsBookkeeper.sol
- src/fundsRequester/interfaces/IFundsRequester.sol
- src/fundsRequester/interfaces/IClaimRouter.sol
- src/oracleProvider/implementations/ChainlinkPriceProvider.sol
- src/oracleProvider/interfaces/IPriceProvider.sol
- src/inspection/interfaces/IInspectorV1.sol

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
1	3	8	4

INFORMATIONAL
8

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT FUNCTION CALL LEADS TO LOSS OF CLAIM ROUTER ASSETS	CRITICAL	SOLVED - 09/22/2024
LACK OF PROTECTION POLICY VALIDATION	HIGH	SOLVED - 09/22/2024
INCORRECT PROTECTION PERIOD CHECKS MAY DENY VALID CLAIMS	HIGH	SOLVED - 09/22/2024
PROTECTION CLAIM AFTER LENDER-SIDE LIQUIDATION CAUSES FUNDS TO BE LOCKED	HIGH	SOLVED - 09/22/2024
INCORRECT BORROWING MODALITY LEADS TO BROKEN FUNCTIONALITY	MEDIUM	SOLVED - 09/22/2024
ON-CHAIN SLIPPAGE CALCULATION CAN BE MANIPULATED	MEDIUM	RISK ACCEPTED - 09/22/2024
INCORRECT MODALITY USAGE RESULTS IN TRANSACTION REVERTS	MEDIUM	SOLVED - 09/22/2024
REMOTE PROTOCOL PROXIES UNABLE TO EXECUTE CORE FUNCTIONS	MEDIUM	SOLVED - 09/22/2024
LACK OF CHAINLINK'S STALE PRICE CHECKS	MEDIUM	SOLVED - 09/22/2024
BROKEN FUNCTIONALITY OF REMOTEPROTOCOLPROXY DUE TO ACCESS CONTROL	MEDIUM	SOLVED - 09/22/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT CALCULATION ORDER IN CREDIT UPDATE	MEDIUM	SOLVED - 09/23/2024
TOKEN TRANSFER REVERTS LEADING TO BLOCKED FORECLOSURE	MEDIUM	SOLVED - 09/23/2024
UNSAFE USE OF TRANSFER()/TRANSFERFROM() WITH IERC20	LOW	SOLVED - 09/25/2024
INSOLVENT POSITION CANNOT BE LIQUIDATED	LOW	SOLVED - 09/23/2024
USAGE OF DIRECT APPROVE CALLS	LOW	SOLVED - 09/23/2024
USAGE OF SINGLE-STEP ROLE TRANSFER	LOW	SOLVED - 09/23/2024
UNLOCKED PRAGMA COMPILERS	INFORMATIONAL	SOLVED - 09/25/2024
CONSIDER USING NAMED MAPPINGS	INFORMATIONAL	SOLVED - 09/23/2024
UNUSED IMPORTS, ERRORS AND EVENTS	INFORMATIONAL	SOLVED - 09/25/2024
COMMENTED OUT CODE	INFORMATIONAL	SOLVED - 09/23/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
EXCESS GAS CONSUMPTION DUE TO REDUNDANT TOKEN TRANSFERS	INFORMATIONAL	ACKNOWLEDGED
LACK OF EVENT EMISSION	INFORMATIONAL	SOLVED - 09/23/2024
INCONSISTENT NAMING CONVENTION IN PROTECTIONLIBV1 FOR VALUE UNITS	INFORMATIONAL	SOLVED - 09/24/2024
UNUSED PARAMETER IN CLAIMPROTECTION FUNCTION	INFORMATIONAL	ACKNOWLEDGED

7. FINDINGS & TECH DETAILS

7.1 INCORRECT FUNCTION CALL LEADS TO LOSS OF CLAIM ROUTER ASSETS

// CRITICAL

Description

The `FundsRequesterConnector` contract is designed to manage interactions with the funds' requester. It includes the `_repayTokenToClaimRouter` functions used for handing foreclosure and debt repayment processes.

```
40      function _repayTokenToClaimRouter(address asset, uint256 amount) internal {
41          address claimRouter = REMOTE_REGISTRY.getEarnClaimRouter();
42          _increaseAllowanceIfNecessary(claimRouter, asset, amount);
43          FUNDS_REQUESTER.requestTokenFromClaimRouter(asset, amount);
44      }
45
46      function _repayTokenToClaimRouter(address claimRouter, address asset, uint256 amount) inte
47          _increaseAllowanceIfNecessary(claimRouter, asset, amount);
48          FUNDS_REQUESTER.requestTokenFromClaimRouter(asset, amount);
49      }
```

However, instead of correctly invoking the `repayTokenToClaimRouter` , the contract mistakenly calls `requestTokenFromClaimRouter` in its logic. It leads to a situation (presented in PoC) where users can exploit the protocol by repaying their debt and, instead of decreasing their balance, they receive additional tokens. This allows them to siphon off more assets than they initially deposited, resulting in significant financial losses for the system.

Proof of Concept

```
    function test_incorrectFunctionCall() external {
        uint256 supplyAmount = 1000 ether;
        uint256 borrowAmount = 1_500_000 * 10 ** 6;

        address userBlueprint = _createAaveV3UserBlueprint();
        weth.mint(alice, supplyAmount + 30 ether);
        vm.prank(alice);
        weth.approve(userBlueprint, supplyAmount + 30 ether);

        (uint256 encodedProtectionInfo, uint256 promisedAmountInCollateral) = _createConcreteProtecti

        vm.startPrank(address(cccm));
            //User supplies WETH and borrows USDC
            IUserIntervention(userBlueprint).supply(supplyAmount, 0);
            IUserIntervention(userBlueprint).setBorrowToken(address(usdc));
            IUserIntervention(userBlueprint).borrow(borrowAmount, 0);

            //User opens a protection policy
            IProtocolIntervention(userBlueprint).openProtection(
                encodedProtectionInfo, concreteForeclosureFeeDebtFractionInWad, promisedAmountInColla
            );
            uint256 protectionAmount =
            promisedAmountInCollateral.mulDiv(conreteClaimOneAmountAsPromisedAmountFractionInWad, WAD
            assertEq(protectionAmount, 30 ether);

            //User claims protection, which increases their debt by the protection amount
            IProtocolIntervention(userBlueprint).claimProtection(protectionAmount, 0, address(0));
            uint userConcreteDebt = IViewUserState(userBlueprint).viewConcreteDebtOfUser();
            assertEq(userConcreteDebt, 30 ether);
```

```
            uint256 supplyBeforeRepay = IViewUserState(userBlueprint).getLenderSuppliedWithInterest()

            //User repays their concrete debt of 30 WETH
            IReclaimRepayCancel(userBlueprint).repayConcreteDebt(userConcreteDebt, false, Uint8CodecL

            // User supplies the outstanding 60 WETH balance to the lender
            assertEq(weth.balanceOf(address(userBlueprint)), 60 ether);
            IUserIntervention(userBlueprint).supply(60 ether, 2);
        vm.stopPrank();

        //Concrete debt is now 0 and the supply has increased by 60 WETH
        assertEq(IViewUserState(userBlueprint).viewConcreteDebtOfUser(), 0);
        uint256 supplyAfterRepay = IViewUserState(userBlueprint).getLenderSuppliedWithInterest();

        assertEq(supplyAfterRepay, supplyBeforeRepay + 60 ether);
    }
```

```
Ran 1 test for test/stories/ProtectionFlow.sol:ProtectionFlow
[PASS] test_incorrectFunctionCall() (gas: 1244531)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.47ms (2.07ms CPU time)

Ran 1 test suite in 141.53ms (10.47ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (10.0)

Recommendation

Ensure that the `FundsRequesterConnector` contract correctly invokes the `repayTokenToClaimRouter` method in the mentioned functions.

Remediation Comment

SOLVED: The **Concrete team** fixed the issue as recommended.

References

Blueprint-Finance/sc_spokes-v1/src/userBase/utils/FundsRequesterConnector.sol#L40-L49

7.2 LACK OF PROTECTION POLICY VALIDATION

// HIGH

Description

When the CCCM begins the process of opening a protection policy, the `claimProtection` function requires the encoded protection information.
The protection policy data is initialized by encoding these parameters using the `encodeProtectionDataFromAbsoluteValues` method provided in the ProtectionLibV1 library. This data, along with the promised protection amount is then used to open the protection policy for a given user.

```
        ProtectionDataAbsolute memory protectionDataAbsolute = ProtectionDataAbsolute({
            endTime: uint40(block.timestamp + DURATION),
            numberOfTranches: 3,
            protocolRights: 0,
            remoteProxyRights: 0,
            remoteConcreteerRights: 0,
            remotePublicRights: 0,
            ltvProtectForClaimsInBP: concreteClaimBufferInBP,
            ltvProtectForForeclosureInBP: concreteForeclosureBufferInBP,
            openingFeeInBase: promisedAmountInCollateral.mulDiv(concreteOpeningFeePromisedAmoiuntFrac
            cancellationFeeInBase: promisedAmountInCollateral.mulDiv(
                concreteCancellationFeePromisedAmountFractionInWad, WAD
            ),
            trancheAmountInBase: trancheAmountInCollateral,
            trancheFeeInBase: trancheFeeInCollateral
        });

        encodedProtectionData =
            ProtectionLibV1.encodeProtectionDataFromAbsoluteValues(promisedAmountInCollateral, protec
    }
```

However, when the policy is claimed, no validation is performed to ensure that the claim adheres to the encoded protection data. Additionally, the protection system is designed to ensure that claims are made within a specified buffer from the current LTV position, but this check is also not enforced. This can lead to financial losses due to unauthorized claims, protection being claimed outside the intended LTV buffer, protection claims being too large or executed outside the intended timeframe, and a decrease in protocol integrity.

Proof of Concept

```
    function test_ProtectionDataNotChecked() external {
        uint256 supplyAmount = 1000 ether;
        uint256 borrowAmount = 1_500_000 * 10 ** 6;

        address userBlueprint = _createAaveV3UserBlueprint();
        weth.mint(alice, supplyAmount);
        vm.startPrank(alice);
            weth.approve(userBlueprint, supplyAmount);
            usdc.approve(userBlueprint, type(uint256).max);
        vm.stopPrank();

        (uint256 encodedProtectionInfo, uint256 promisedAmountInCollateral) = _createConcreteProtecti

        vm.startPrank(address(cccm));
            //User supply WETH and borrow USDC
            IUserIntervention(userBlueprint).supply(supplyAmount, 0);
            IUserIntervention(userBlueprint).setBorrowToken(address(usdc));
            IUserIntervention(userBlueprint).borrow(borrowAmount, 0);
```

```
            //User opens protection policy
            IProtocolIntervention(userBlueprint).openProtection(
                encodedProtectionInfo, concreteForeclosureFeeDebtFractionInWad, promisedAmountInColla
            );
            uint256 protectionAmount =
            promisedAmountInCollateral.mulDiv(conreteClaimOneAmountAsPromisedAmountFractionInWad, WAD
            uint256 protectionFee = promisedAmountInCollateral.mulDiv(conreteClaimOneFeeAsPromisedAmo
            assertEq(protectionAmount, 30 ether);

            //Duration is set to 30 days
            vm.warp(ProtectionLibV1.getEndTime(encodedProtectionInfo) + 100 days);

            //Number of tranches is set to 3
            for(uint i=0; i < 15; i++){
                IProtocolIntervention(userBlueprint).claimProtection(999 ether, 555 ether, address(0)
        }
        vm.stopPrank();
    }
```

```
Ran 1 test for test/stories/ProtectionFlow.sol:ProtectionFlow
[PASS] test_ProtectionDataNotChecked() (gas: 2311075)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 13.36ms (5.66ms CPU time)

Ran 1 test suite in 144.35ms (13.36ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:H/I:N/A:N/D:N/Y:N/R:N/S:U (7.5)

Recommendation

Implement comprehensive validation checks against the protection policy data to ensure that all claims adhere to the encoded parameters, including verification of the LTV buffer, the number of tranches, and the claim amounts and timeframe.

Remediation Comment

SOLVED: The **Concrete team** solved the issue. Additional checks were added to the protocol.

- `repayTotalDebtAndCancel` : Uses a policy cancellation fee if the issuer is 0, otherwise checks if the cancellation fee is within bounds using `ProtectionViewLibV1.checkAmountsFromInfo` .
- `cancelPolicy` : Follows the same approach as `repayTotalDebtAndCancel` .
- `reclaimDebtOrForeclose` : A new function for hub contracts that checks `ProtectionViewLibV1.surpassCriticalLtvAfterWithdraw` and either calls `reclaimDebt` or `foreclose` , ensuring foreclosure fees are within bounds via `ProtectionViewLibV1.checkForeclosureFeeAmount` .
- `claimProtection` : Verifies with `QueryInterventionV1.canClaimProtection` and ensures claim amounts and fees are within bounds using ProtectionViewLibV1. `checkAmountsFromInfo` .
- `foreclose` : First checks `QueryInterventionV1.isLiteForeclosableFromInfo` - if not, checks if it is seizable by claims using `QueryInterventionV1.isForeclosableByClaims` , or by expiration via `reclaimDebtOrForeclose` , ensuring fees are checked with `ProtectionViewLibV1.checkForeclosureFeeAmount` .
- `openProtection` : Ensures the borrow token is set, validates protection data consistency using `ProtectionLibV1.validateEncodedProtection` , and checks if the promised amount exceeds limits using a formula from `PolicyInsightsLibV1.getLowerBoundForMaxPromisedAmountFromInfo` .

7.3 INCORRECT PROTECTION PERIOD CHECKS MAY DENY VALID CLAIMS

// HIGH

Description

The `canClaimProtection` function from `UserBase` contract is designed to determine whether a user can claim protection under a specific Blueprint. This function evaluates the eligibility based on several parameters, including the end time of the protection, the number of claims already made, the number of available tranches and LTV ratio.

```
537     function canClaimProtection() external view override(IViewUserState) returns (bool isClaim
538         uint256 protectionInfo = _getProtectionInfo();
539         if (protectionInfo == 0) return false;
540         uint256 liqLtvInWad = _getLiquidationLtvInWad();
541         isClaimable = QueryInterventionV1.isClaimable(
542             protectionInfo.getEndTime() <= block.timestamp,
543             protectionInfo.getNumberOfClaims(),
544             protectionInfo.getNumberOfTranches(),
545             _getCurrentLtvInWad(),
546             liqLtvInWad - protectionInfo.getLtvProtectBufferForClaimsInWad().mulDiv(liqLtvInWa
547         );
548     }
```

However, the condition `protectionInfo.getEndTime() <= block.timestamp`, used to check whether the protection period has expired, is inverted. It checks if the protection end time is less than or equal to the current timestamp and returns true if the protection period has already expired. This is in contrary to the intended purpose of verifying whether protection is still active.

```
11      function isClaimable(
12          bool isProtected,
13          uint8 numberOfClaimsUsed,
14          uint8 availableClaims,
15          uint256 currentLtvInWad,
16          uint256 foreclosureThresholdInWad
17      ) internal pure returns (bool) {
18          return _isClaimablePrimitive(
19              isProtected,
20              numberOfClaimsUsed >= availableClaims, // not used up all claims
21              currentLtvInWad >= foreclosureThresholdInWad
22          ); //
23      }
```

```
161     function _isClaimablePrimitive(bool isProtected, bool hasUsedUpAllClaims, bool crossLtvPro
162         private
163         pure
164         returns (bool)
165     {
166         return isProtected && !hasUsedUpAllClaims && crossLtvProtect;
167     }
```

The same logic is used in the `claimProtection` function defined in the `UserBaseFull` contract. As the `canClaimProtection` function will return false when the position is in the actual protection period, this can result in legitimate claims being incorrectly denied, undermining the functionality of the protection mechanism.

Proof of Concept

```
    function test_canClaimProtectionFails() external {
        address userBlueprint = _createAaveV3UserBlueprint();
        // how much the user supplies and borrows
        uint256 supplyAmount = 1000 ether;
        uint256 borrowAmount = 1_500_000 * 10 ** 6;
        (uint256 encodedProtectionInfo, uint256 promisedAmountInCollateral) =
            _createConcreteProtectionEncoding(supplyAmount, borrowAmount);

        weth.mint(alice, supplyAmount);
        vm.prank(alice);
        weth.approve(userBlueprint, supplyAmount);

        vm.startPrank(address(cccm));
            // Supply WETH as a collateral and borrow USDC tokens
            IUserIntervention(userBlueprint).supply(supplyAmount, 0);
            IUserIntervention(userBlueprint).setBorrowToken(address(usdc));
            IUserIntervention(userBlueprint).borrow(borrowAmount, 0);

            //Open a protection policy
            IProtocolIntervention(userBlueprint).openProtection(
                encodedProtectionInfo, concreteForeclosureFeeDebtFractionInWad, promisedAmountInColla
            );
            //Simulate a price drop of 10%
            uint256 newEthPrice = initialEthPriceInBaseUnits.mulDiv(90, 100);
            _updatePrice(aaveSetup.priceOracle, address(weth), newEthPrice);

            //The current LTV is greater than or equal to the liquidation LTV minus a buffer
            uint256 currentLTV = IViewUserState(userBlueprint).getCurrentLtvInWad();
            uint256 liquidationLTV = IViewUserState(userBlueprint).getLiquidationLtvInWad();
            assertGe(currentLTV, liquidationLTV - (liteForeclosureThresholdBufferInWad * liquidationL

            //The canClaimProtection indicate that user still cannot claim protection.
            assertEq(IViewUserState(userBlueprint).canClaimProtection(), false);

        vm.stopPrank();
    }
```

```
Ran 1 test for test/stories/ProtectionFlow.sol:ProtectionFlow
[PASS] test_canClaimProtectionFails() (gas: 1176451)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.67ms (657.75µs CPU time)

Ran 1 test suite in 135.66ms (2.67ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (7.5)

Recommendation

Correct the logic in the `canClaimProtection` function by replacing `protectionInfo.getEndTime() <= block.timestamp` with `protectionInfo.getEndTime() >= block.timestamp` to ensure valid claims during the protection period are not mistakenly denied.

Remediation Comment

SOLVED: The **Concrete team** fixed the issue as recommended. The direction of the inequality has been changed from `<=` to `>=`. The `canClaimProtection` function is now a library function inside the `QueryInterventionV1` library.

References

Blueprint-Finance/sc_spokes-v1/src/userBase/UserBase.sol#L542

7.4 PROTECTION CLAIM AFTER LENDER-SIDE LIQUIDATION CAUSES FUNDS TO BE LOCKED

// HIGH

Description

The protocol allows users to open a protection policy for their positions to safeguard against risks that leads to position liquidation.
If the user anticipates issues with their position (such as market volatility), they can claim protection funds by calling the `claimProtection` function, provided the protection policy was opened beforehand. This function deducts fees, updates the user's credit position, and supplies the remaining credit to the lender. Once the protection is claimed, the protocol records the updated credit information to reflect the newly injected funds.
If a position falls into default, the protocol provides a foreclosure mechanism, allowing the system to liquidate the user's position and settle any outstanding debts. The foreclosure logic depends on the credit and collateral amounts in the user's position.
However, when a user's position is liquidated on the lender side the protocol still allows the user to claim protection, even though the position has already been liquidated. This creates a situation where the injected protection funds become trapped within the contract. Since the foreclosure process cannot handle this scenario, the position becomes unrecoverable, leading to assets being stuck in the protocol.

Proof of Concept

```
    function test_positionCannotBeForeclosed() public {
        address userBlueprint = _createAaveV3UserBlueprint();

        uint256 supplyAmount = 1000 ether;
        uint256 borrowAmount = 1_500_000 * 10 ** 6;
        uint256 promisedAmountInCollateral;
        uint256 encodedProtectionInfo;
        (encodedProtectionInfo, promisedAmountInCollateral) = _createConcreteProtectionEncoding(suppl
        assertEq(promisedAmountInCollateral, 100 ether);

        weth.mint(alice, supplyAmount);
        vm.prank(alice);
        weth.approve(userBlueprint, supplyAmount);

        vm.startPrank(address(cccm));
            //User supplies WETH collateral and borrow USDC
            IUserIntervention(userBlueprint).supply(supplyAmount, 0);
            IUserIntervention(userBlueprint).setBorrowToken(address(usdc));
            IUserIntervention(userBlueprint).borrow(borrowAmount, 0);

            //Protection policy is opened
            IProtocolIntervention(userBlueprint).openProtection(
            encodedProtectionInfo, concreteForeclosureFeeDebtFractionInWad, promisedAmountInCollatera

            //User claims protection but their position gets liquidated in the lender right before
            IPool(aaveSetup.pool).setUserCollateralBalance(address(userBlueprint), 0);
            assertEq(IViewUserState(userBlueprint).getLenderSuppliedWithInterest(), 0);

            //User claims protection
            IProtocolIntervention(userBlueprint).claimProtection(20 ether, 0, address(0));
            assertEq(IViewUserState(userBlueprint).getLenderSuppliedWithInterest(),20 ether);

            //Position cannot be liquidated
            vm.expectRevert();
            IProtocolIntervention(userBlueprint).foreclose(0, address(0));
```

```
        vm.stopPrank();
    }
```

```
Ran 1 test for test/stories/ProtectionFlow.sol:ProtectionFlow
[PASS] test_positionCannotBeForeclosed() (gas: 1414265)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.93ms (2.12ms CPU time)

Ran 1 test suite in 146.53ms (9.93ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (7.5)

Recommendation

- Update the foreclosure logic to properly handle cases where protection claims are made after a position has been liquidated on the lender side.
- Prevent users from claiming protection if their position has already been liquidated.
- Introduce a mechanism to allow recovery of protection funds injected into liquidated positions.

Remediation Comment

SOLVED: The **Concrete team** solved the issue. The function `reclaimDebtOrForeclose` was added to the User Base. It has two branches. Either the policy is expired, in which case the `foreclosureFeeFraction` is passed in the arguments and used after a check that it is within the bounds of the policy. Alternatively, the policy is not expired. This branch will be executed when funds are locked because of a lender side liquidation.

7.5 INCORRECT BORROWING MODALITY LEADS TO BROKEN FUNCTIONALITY

// MEDIUM

Description

The `supplyBorrowAndProtect` function facilitates supplying liquidity, borrowing, and opening a protection policy within a single transaction. It internally calls the `_openProtectionPolicy` function, passing the provided parameters.

```
406     function supplyBorrowAndProtect(
407         uint256 supplyAmount_,
408         uint256 borrowAmount_,
409         address borrowToken_,
410         uint256 protectionInfo_,
411         uint256 protectionForeclosureFeeFractionInWad_,
412         uint256 promisedAmountInToken_
413     ) external virtual override(IProtocolIntervention) {
414         bytes32 previousBorrowTokenInfo = _borrowTokenInfo;
415         bytes32 collateralTokenInfo = _collateralTokenInfo;
416         bool borrowTokenNeedsToBeSet = previousBorrowTokenInfo.getAddress() == address(0);
417         if (borrowTokenNeedsToBeSet) {
418             // set Borrow token
419             _lenderSpecificAssetValidation(collateralTokenInfo.getAddress(), borrowToken_);
420             _setBorrowTokenInfo(borrowToken_);
421         } else if (previousBorrowTokenInfo.getAddress() != borrowToken_) {
422             revert Errors.AssetDivergence();
423         }
424 
425         bool callerIsProtocol = _callerIsEndpoint();
426         bool mayOverwriteExistingProtection = _callerIsRemoteProtocolProxy() || callerIsProtoc
427         if (_getProtectionInfo() != 0 && !mayOverwriteExistingProtection) {
428             revert Errors.ProtectionAlreadyOpened();
429         }
430 
431         if (callerIsProtocol) {
432             if (borrowTokenNeedsToBeSet) {
433                 previousBorrowTokenInfo = _borrowTokenInfo;
434             }
435             _openProtectionPolicy(
436                 protectionInfo_,
437                 protectionForeclosureFeeFractionInWad_,
438                 promisedAmountInToken_,
439                 supplyAmount_,
440                 borrowAmount_,
441                 collateralTokenInfo,
442                 previousBorrowTokenInfo
443             );
444         }
```

As the `borrowAmountInToken` was set, the following branch will be executed:

```
223         if (borrowAmountInToken > 0 && borrowTokenInfo.getAddress() != address(0)) {
224             newCreditInfo = _updateUserFractions(
225                 newCreditInfo, _getLenderDebt(false), borrowAmountInToken, PositionInteraction
226             );
227 
228             _borrow(
229                 borrowTokenInfo.getAddress(),
230                 borrowAmountInToken,
231                 Uint8CodecLib.encodeOnlyMode(uint8(SendFundsModality.ONLY_SECOND_STEP))
232             );
233         }
```

However, since the `ONLY_SECOND_STEP` modality in the `_borrow` function only transfers the provided amount of tokens to the owner, the function will revert because the funds were not sent to the Blueprint beforehand. This will cause the transaction to fail unexpectedly, preventing the operation from executing successfully.

Proof of Concept

```
    function test_incorrectModalityBorrow() public {
        address userBlueprint = _createAaveV3UserBlueprint();

        uint256 supplyAmount = 1000 ether;
        uint256 borrowAmount = 1_500_000 * 10 ** 6;

        weth.mint(alice, supplyAmount + 0.5 ether);
        vm.startPrank(alice);
        weth.approve(userBlueprint, supplyAmount + 0.5 ether);
        vm.stopPrank();

        vm.startPrank(address(cccm));

        (uint256 encodedProtectionInfo, uint256 promisedAmountInCollateral) =
            _createConcreteProtectionEncoding(1 ether, 1500e6);

        vm.startPrank(address(cccm));

        //User supplies 1 WETH, and intends to borrow 1500 USDC
        //Transaction reverts due to insufficient funds
        vm.expectRevert();
        IProtocolIntervention(userBlueprint).supplyBorrowAndProtect(1 ether, 1500e6, address(usdc),
            encodedProtectionInfo, concreteForeclosureFeeDebtFractionInWad, promisedAmountInCollatera
        );
    }
```

```
Ran 1 test for test/stories/ProtectionFlow.sol:ProtectionFlow
[PASS] test_incorrectModalityBorrow() (gas: 1016855)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.09ms (1.51ms CPU time)

Ran 1 test suite in 154.73ms (9.09ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation

Instead of using the `ONLY_SECOND_STEP` modality, consider using the `ONLY_FIRST_STEP` modality to borrow assets against the previously supplied collateral.

Remediation Comment

SOLVED: The **Concrete team** solved the issue. The functions `supplyAndProtect` and `supplyBorrowAndProtect` were removed from the User Blueprint Base contract.

References

Blueprint-Finance/sc_spokes-v1/src/userBase/utils/ProtectionHandler.sol#L231

7.6 ON-CHAIN SLIPPAGE CALCULATION CAN BE MANIPULATED

// MEDIUM

Description

The `_swap` function defined in the SwapperUniV3 contract is designed to swap tokens for a given amount and ensure that the swap does not exceed a specified slippage. It is used during the foreclosure process, to swap the collateral to the borrowed asset.
The function transfers the input tokens from the sender, approves them for the swap router, and determines the best possible path and fees for the swap using the `_getPath` function.

```
41      function _swap(address fromToken, address toToken, uint256 amount, uint256 amountOutMin)
42          internal
43          override
44          returns (uint256 swappedAmount)
45      {
46          IERC20(fromToken).transferFrom(msg.sender, address(this), amount);
47          IERC20(fromToken).approve(address(SWAP_ROUTER), amount);
48          // TransferHelper.safeTransferFrom(fromToken, msg.sender, address(this), amount);
49          // TransferHelper.safeApprove(fromToken, address(SWAP_ROUTER), amount);
50
51          (bytes memory path, uint24 bestFee, uint256 bestAmountOut) = _getPath(fromToken, toTok
52          if (amountOutMin == 0) {
53              amountOutMin = bestAmountOut.mulDiv(WAD - MAX_SLIPPAGE_IN_WAD, WAD); // consider 0
54          }
```

The `amountOutMin` value is determined by fetching the best possible output amount via the `_getBestFee` method. The slippage is then calculated based on this value.

```
41      function _getPath(address fromToken, address toToken, uint256 amount)
42          internal
43          returns (bytes memory, uint24, uint256)
44      {
45          (uint24 bestFee0, uint256 bestAmountOut0) = _getBestFee(fromToken, toToken, amount);
46          if (bestAmountOut0 > 0) {
47              return ("", bestFee0, bestAmountOut0);
48          }
49          // If no direct swap is possible, try to find a path with WETH
50          (uint24 bestFee1, uint256 bestAmountOut1) = _getBestFee(fromToken, WETH, amount);
51          (uint24 bestFee2, uint256 bestAmountOut2) = _getBestFee(WETH, toToken, bestAmountOut1)
52          if (bestAmountOut1 == 0 || bestAmountOut2 == 0) {
53              revert Errors.NoPathFound();
54          }
55          bytes memory path = abi.encodePacked(fromToken, bestFee1, WETH, bestFee2, toToken);
56          return (path, bestFee2, bestAmountOut2);
57      }
```

This code attempts to perform on-chain slippage calculations using `Quoter.quoteExactInputSingle`. However, this approach is prone to manipulations, because `quoteExactInputSingle` simulates a swap on-chain, making it vulnerable to sandwich attacks. Attackers can manipulate the price by front-running the transaction, leading to inaccurate slippage calculations. Additionally, the Quoter natspec states that these functions are not gas efficient and should not be called on-chain, which makes the current implementation not only insecure but also inefficient.

```
132     function _getBestFee(address tokenA, address tokenB, uint256 amount)
133         internal
134         returns (uint24 bestFee, uint256 bestAmountOut)
135
```

```
    {
        for (uint8 i; i < feeTiers.length;) {
            if (hasPool(tokenA, tokenB, feeTiers[i])) {
                uint256 amountOut = QUOTER.quoteExactInputSingle(tokenA, tokenB, feeTiers[i],
                if (amountOut > bestAmountOut) {
                    bestFee = feeTiers[i];
                    bestAmountOut = amountOut;
                }
            }
            unchecked {
                ++i;
            }
        }
    }
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:N/R:N/S:U (5.0)

Recommendation

Consider implementing an off-chain price feed for slippage checks to minimize the risk of manipulation during swaps.

Remediation Comment

RISK ACCEPTED: The **Concrete team** accepted the risk.

References

Blueprint-Finance/sc_spokes-v1/src/swapper/implementations/SwapperUniV3.sol#L53
Blueprint-Finance/sc_spokes-v1/src/swapper/implementations/SwapperUniV3.sol#L114-L130
Blueprint-Finance/sc_spokes-v1/src/swapper/implementations/SwapperUniV3.sol#L132-L148

7.7 INCORRECT MODALITY USAGE RESULTS IN TRANSACTION REVERTS

// MEDIUM

Description

The `supplyAndProtect` function allows for supplying liquidity and opening the protection policy in a single transaction. It calls the internal `_openProtectionPolicy` function, passing the provided data. The `_openProtectionPolicy` calculates the opening fee based on the data passed in `protectionInfo` and checks if it's greater than the supplied amount.

```
142         // calculate the openingFee in Token
143         uint256 openingFeeInToken = protectionInfo.getOpeningFeeFractionInWad().mulDiv(promise
144
145         // get treatury and split
146         (address claimRouter, bytes32 TreasuryAndRevenueSplit) =
147             REMOTE_REGISTRY.getClaimRouterAndEncodedTreasuryAndRevenueSplitInWAD();
148
149         TempAddresses memory tempAddresses;
150         tempAddresses.claimRouter = claimRouter;
151         tempAddresses.treasury = TreasuryAndRevenueSplit.getAddress();
152
153         // take from the lender supplied collateral
154         bool withdrawOpeningFeeFromLender = supplyAmountInToken < openingFeeInToken;
```

If it's greater, the following branch will be executed:

```
187             } else {
188                 // fee will be deducted in the next step, but book-keeping will be done here
189                 newCreditInfo = _updateUserFractions(
190                     newCreditInfo, totalSuppliedBefore, supplyAmountInToken, PositionInteracti
191                 );
192
193                 // supply the collateral
194                 _supply(
195                     collateralToken,
196                     supplyAmountInToken,
197                     Uint8CodecLib.encodeOnlyMode(uint8(SendFundsModality.ONLY_SECOND_STEP))
198                 );
199             }
```

However, since the `ONLY_SECOND_STEP` modality in the `_supply` function only approves the provided amount and supplies it to the lender, this function will revert because the funds were not sent beforehand. This will result in the transaction failing unexpectedly, preventing the operation from being successfully executed.

Proof of Concept

```
    function test_incorrectModality() public {
        address userBlueprint = _createAaveV3UserBlueprint();

        uint256 supplyAmount = 1000 ether;
        uint256 borrowAmount = 1_500_000 * 10 ** 6;

        weth.mint(alice, supplyAmount + 0.5 ether);
        vm.startPrank(alice);
        weth.approve(userBlueprint, supplyAmount + 0.5 ether);
        vm.stopPrank();
```

```
        //User supplies WETH collateral and borrow USDC
        vm.startPrank(address(cccm));
            IUserIntervention(userBlueprint).supply(supplyAmount, 0);
            IUserIntervention(userBlueprint).setBorrowToken(address(usdc));
            IUserIntervention(userBlueprint).borrow(borrowAmount, 0);
        vm.stopPrank();

            (uint256 encodedProtectionInfo, uint256 promisedAmountInCollateral) =
                _createConcreteProtectionEncoding(supplyAmount, borrowAmount);

        vm.startPrank(address(cccm));
            //User supplies only 0.5 WETH, which is less than the required fee
            //Transaction reverts due to insufficient funds
            vm.expectRevert();
            IProtocolIntervention(userBlueprint).supplyAndProtect(0.5 ether,
                encodedProtectionInfo, concreteForeclosureFeeDebtFractionInWad, promisedAmountInColla
            );
    }
```

```
Ran 1 test for test/stories/ProtectionFlow.sol:ProtectionFlow
[PASS] test_incorrectModality() (gas: 1054642)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.65ms (622.29µs CPU time)

Ran 1 test suite in 143.22ms (2.65ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation

Instead of using the `ONLY_SECOND_STEP` modality, consider using the `SEND_THROUGH` modality for the described case. If `withdrawOpeningFeeFromLender` is true, the function will withdraw the opening fee using the updated balance.

Remediation Comment

SOLVED: The **Concrete team** solved the issue. The function `supplyAndProtect` was removed from the User Blueprint Base contract.

References

Blueprint-Finance/sc_spokes-v1/src/userBase/utils/ProtectionHandler.sol#L194-L198

7.8 REMOTE PROTOCOL PROXIES UNABLE TO EXECUTE CORE FUNCTIONS

// MEDIUM

Description

The `openProtection`, `SupplyAndProtect`, and `SupplyBorrowAndProtect` functions from the UserBase contract utilize similar logic to validate whether the caller is authorized to perform the intended operation. The logic begins by checking if the caller is the protocol endpoint using `_callerIsEndpoint` method. If this check is true, it sets the `callerIsProtocol` flag. Additionally, the logic checks if the caller is a remote protocol proxy using `callerIsRemoteProtocolProxy` and combines these two checks to determine if the existing protection can be overwritten.

```
        bool callerIsProtocol = _callerIsEndpoint();
        bool mayOverwriteExistingProtection = _callerIsRemoteProtocolProxy() || callerIsProtocol;
        if (_getProtectionInfo() != 0 && !mayOverwriteExistingProtection) {
            revert Errors.ProtectionAlreadyOpened();
        }
        if (callerIsProtocol) {
            LOGIC HERE
            );
        } else {
            revert Errors.NotProtocolAndNotIntervenable();
        }
```

However, the logic incorrectly restricts the execution of the function's core logic to only when the caller is the protocol endpoint. If the caller is a valid remote proxy, the function will incorrectly prevent the remote proxy from executing its intended functionality, causing disruptions or unintended denials of service.

Proof of Concept

```
    function test_remoteProtocolProxyReverts() external {
        address userBlueprint = _createAaveV3UserBlueprint();
        vm.prank(address(cccm));
            IUserIntervention(userBlueprint).setBorrowToken(address(usdc));
        uint256 supplyAmount = 1000 ether;
        uint256 borrowAmount = 1_500_000 * 10 ** 6;
        (uint256 encodedProtectionInfo, uint256 promisedAmountInCollateral) =
            _createConcreteProtectionEncoding(supplyAmount, borrowAmount);

        //Expect a revert if the remote proxy is the caller
        vm.prank(address(remoteProtocolProxy));
            vm.expectRevert(Errors.NotProtocolAndNotIntervenable.selector);
            IProtocolIntervention(userBlueprint).openProtection(
                encodedProtectionInfo, concreteForeclosureFeeDebtFractionInWad, promisedAmountInColla

        vm.prank(address(remoteProtocolProxy));
            vm.expectRevert(Errors.NotProtocolAndNotIntervenable.selector);
            IProtocolIntervention(userBlueprint).supplyAndProtect(
                supplyAmount, encodedProtectionInfo, liteForeclosureFeeFractionInWad, promisedAmountI

        vm.prank(address(remoteProtocolProxy));
            vm.expectRevert(Errors.NotProtocolAndNotIntervenable.selector);
            IProtocolIntervention(userBlueprint).supplyBorrowAndProtect(
                supplyAmount, borrowAmount, address(usdc), encodedProtectionInfo, liteForeclosureFeeF
    }
```

```
Ran 1 test for test/stories/ProtectionFlow.sol:ProtectionFlow
[PASS] test_remoteProtocolProxyReverts() (gas: 591043)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.71ms (1.04ms CPU time)

Ran 1 test suite in 150.84ms (8.71ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation

Modify the access control logic to ensure that both the protocol endpoint and the remote proxy are allowed to execute the functions. Implement a function that allows the Remote Proxy to claim rewards as intended by the logic of the `claimRewards` function.

Remediation Comment

SOLVED: The **Concrete team** solved the issue. The mentioned logic was removed from the functions.

References

Blueprint-Finance/sc_spokes-v1/src/userBase/UserBase.sol#L389-L408
Blueprint-Finance/sc_spokes-v1/src/userBase/UserBase.sol#L419-L437
Blueprint-Finance/sc_spokes-v1/src/userBase/UserBase.sol#L459-L480

7.9 LACK OF CHAINLINK'S STALE PRICE CHECKS

// MEDIUM

Description

The `ChainlinkPriceProvider` contract uses Chainlink feed as their price oracle. When requesting the price via the `_getPrice()` function, the contracts query Chainlink for the underlying token price using the `latestRoundData()` function. This function returns `uint80 roundId`, `int256 answer`, `uint256 startedAt`, `uint256 updatedAt` and `uint80 answeredInRound`.

- `roundId` denotes the identifier of the most recent update round,
- `answer` is the price of the asset,
- `startedAt` is the timestamp at which the round started.
- `updatedAt` is the timestamp at which the feed was updated,
- `answeredInRound` is already deprecated by Chainlink.

```
330     function _getPrice(address priceFeedAddress) internal view override returns (uint256) {
331         (, int256 price,,,) = AggregatorV3Interface(priceFeedAddress).latestRoundData();
332         return uint256(price);
333     }
```

The `getPrice()` function does not check if the feed was updated at the most recent round nor does it verify that `startedAt` (timestamp at which the round started) is not 0, and this can result in accepting stale data which may threaten the stability of the protocol in a volatile market.

BVSS

AO:A/AC:L/AX:M/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (5.0)

Recommendation

Modify the function to utilize Chainlink's `latestRoundData`. This approach provides detailed information about the most recent price round, ensuring that retrieved the price remains up-to-date. Also, consider saving all used the Chainlink aggregators into a mapping and using different grace period for different heartbeats.
Example implementation:

```
uint256 private constant GRACE_PERIOD_TIME = 3600; // duration until the price is considered stale

function getChainlinkPrice(AggregatorV2V3Interface feed) internal {
    (uint80 roundId, int256 price,, uint updatedAt,) = feed.latestRoundData();
    require(price > 0, "Invalid price");
    require(block.timestamp <= updatedAt + GRACE_PERIOD_TIME, "Stale price");
    return price;
}
```

Remediation Comment

SOLVED: The **Concrete team** fixed the issue. The check for stale prices was added. The 1-hour grace period was used. For L2 chains, the team have added sequencer check, where it checks if the sequencer is up and running as well as `updatedAt` timestamp falls within the grace period.

References

Blueprint-Finance/sc_spokes-v1/src/oracleProvider/implementations/ChainlinkPriceProvider.sol#L10-L13

7.10 BROKEN FUNCTIONALITY OF REMOTEPROTOCOLPROXY DUE TO ACCESS CONTROL

// MEDIUM

Description

The `RemoteProtocolProxy` contract is intended to allow foreclosure and claim protection for Blueprint owners with Concreteer privileges. However, both the `foreclose` and `claimProtection` functions implemented in the Blueprint do not use the `_callerIsRemoteProtocolProxy` to allow the Remote Proxy to call, and are callable only by the endpoint, making this functionality broken.

```
330 function foreclose(uint256 foreclosureFeeFractionInWad, address beneficiary)
331         external
332         virtual
333         override(IProtocolIntervention)
334     {
335         // _foreclose();
336         uint256 protectionInfo = _getProtectionInfo();
337         if (_callerIsEndpoint() && protectionInfo.getProtocolEnabledForClosureAndForeclosure()
338             _requestFlashLoan(
339                 _borrowTokenInfo.getAddress(), _getLenderDebt(false), foreclosureFeeFractionIn
340             );
341             return;
342         }
343 
344         revert Errors.UnauthorizedTriggerOfForeclosure();
345     }
```

```
313     function claimProtection(uint256 amount, uint256 protectionFeeInToken, address externalBen
314         external
315         virtual
316         override(IProtocolIntervention)
317     {
318         uint256 protectionInfo = _getProtectionInfo();
319         // if sender is endpoint then claim protection
320         if (_callerIsEndpoint() && protectionInfo.getProtocolEnabledForClaimsAndReclaims()) {
321             // external Beneficiary is disregarded.
322             protectionInfo = _claimProtectionDefault(protectionInfo, amount, protectionFeeInTo
323             _setProtectionInfo(protectionInfo);
324             return;
325         }
326         externalBeneficiary;
327         revert Errors.UnauthorizedTriggerOfClaimProtection();
328     }
```

Proof of Concept

```
function test_remoteProtocolProxyReverts() external {
    address userBlueprint = _createAaveV3UserBlueprint();
    vm.prank(address(cccm));
        IUserIntervention(userBlueprint).setBorrowToken(address(usdc));
    uint256 supplyAmount = 1000 ether;
    uint256 borrowAmount = 1_500_000 * 10 ** 6;
    (uint256 encodedProtectionInfo, uint256 promisedAmountInCollateral) =
        _createConcreteProtectionEncoding(supplyAmount, borrowAmount);

    //Expect a revert if the remote proxy is the caller
    vm.prank(address(remoteProtocolProxy));
        vm.expectRevert(Errors.NotProtocolAndNotIntervenable.selector);
        IProtocolIntervention(userBlueprint).openProtection(
            encodedProtectionInfo, concreteForeclosureFeeDebtFractionInWad, promisedAmountInColla

    vm.prank(address(remoteProtocolProxy));
        vm.expectRevert(Errors.NotProtocolAndNotIntervenable.selector);
```

```
            IProtocolIntervention(userBlueprint).supplyAndProtect(
                supplyAmount, encodedProtectionInfo, liteForeclosureFeeFractionInWad, promisedAmountI

        vm.prank(address(remoteProtocolProxy));
            vm.expectRevert(Errors.NotProtocolAndNotIntervenable.selector);
            IProtocolIntervention(userBlueprint).supplyBorrowAndProtect(
                supplyAmount, borrowAmount, address(usdc), encodedProtectionInfo, liteForeclosureFeeF
    }
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation

Adjust the access of the `RemoteProxy` to the intended functionalities.

Remediation Comment

SOLVED: The **Concrete team** fixed the issue. The mentioned invocation was removed from the contract. The `RemoteProtocolProxy` connectors and `RemotePublicExecutor` variables are retained in UserBaseV1 for potential future use.

References

Blueprint-Finance/sc_spokes-v1/src/userBase/UserBase.sol#L330-L345

Blueprint-Finance/sc_spokes-v1/src/userBase/UserBase.sol#L313-L328

7.11 INCORRECT CALCULATION ORDER IN CREDIT UPDATE

// MEDIUM

Description

When a new protection tranche is claimed, the `claimProtection` method from ProtectionHandler contract, is responsible for updating the credit information. This process involves recalculating the `totalCreditInBase` and `totalCreditInToken` values. The method currently attempts to update `totalCreditInBase` by adding the amount of tokens multiplied by quote denomination (10e8) and then dividing the result by the price returned from the oracle.

```
268             params.creditInfo = _creditInfo;
269             (params.totalCreditInBase, params.totalCreditInToken) = params.creditInfo.decodeCr
270             params.newCreditInfo = params.creditInfo.updateCredit(
271                 params.totalCreditInBase
272                     + params.amountInToken.mulDiv(
273                         PRICE_FEED_QUOTE_DENOMINATION, _getAssetPrice(collateralTokenInfo.getA
274                     ).mulDiv(BASE_DENOMINATION, collateralTokenInfo.getDenomination()),
275                 params.totalCreditInToken + params.amountInToken
276             );
```

However, the calculation is performed in the wrong order. The amount of tokens should first be multiplied by the price and then normalized to the intended `10e8` denomination.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation

Adjust the calculation to first multiply the `amountInToken` by the asset price to ensure accurate conversion.

Remediation Comment

SOLVED: The **Concrete team** solved the issue. The order of calculations was switched.

References

Blueprint-Finance/sc_spokes-v1/src/userBase/utils/ProtectionHandler.sol#L270-L276

7.12 TOKEN TRANSFER REVERTS LEADING TO BLOCKED FORECLOSURE

// MEDIUM

Description

The `_executeForeclosure` function is invoked during the foreclosure flow to execute the logic required to repay unsecured debt. Using the assets withdrawn from the lender, it repays the borrowed amount, subtracts the fee, and sends the remaining collateral back to the position owner.

```
403         // ------ repay to the claimRouter/fundRequester(flashloan) ------
404         params.amountInBorrowTokenLeft -= params.totalRepayAmountInToken;
405         if (isEarnFlashloan) {
406             _increaseAllowanceIfNecessary(params.claimRouter, borrowToken, params.totalRepayAm
407         } else {
408             IERC20(borrowToken).transfer(params.fundRequester, params.totalRepayAmountInToken)
409         }
410         // ------ deduct foreclosure fee ------
411         uint256 foreclosureFeeInBorrow = repayAmount.mulDiv(foreclosureFeeFractionInWad, WAD);
412         if (foreclosureFeeInBorrow > params.amountInBorrowTokenLeft) {
413             foreclosureFeeInBorrow = params.amountInBorrowTokenLeft;
414         }
415         params.amountInBorrowTokenLeft -= foreclosureFeeInBorrow;
416         _deductFee(
417             TempAddresses(params.claimRouter, TreasuryAndRevenueSplit.getAddress(), beneficiar
418             params.borrowTokenInfo,
419             foreclosureFeeInBorrow,
420             TreasuryAndRevenueSplit.getMax96BitNumber()
421         );
422
423         // ------ send back the remaining amount to the owner ------
424         IERC20(borrowToken).transfer(_owner(), params.amountInBorrowTokenLeft);
```

Some tokens implement a blacklist functionality (e.g., USDC), forbidding transfers to or from blocked addresses. Moreover, some tokens revert when attempting to transfer 0 tokens. If the transfer fails for any reason, the entire foreclosure process will revert, preventing the liquidation of the unsecured position.

BVSS

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (5.0)

Recommendation

Consider using the pull pattern. Instead of transferring tokens to the owner's address directly, increase the internal balance of pending withdrawals and introduce a claim functionality that allows the owner to manually withdraw the remaining funds.

Remediation Comment

```
        if (params.amountInBorrowTokenLeft > 0) {
            try IERC20(borrowToken).transfer(_owner(), params.amountInBorrowTokenLeft) {}
            catch {
                IERC20(borrowToken).safeIncreaseAllowance(_owner(), params.amountInBorrowTokenLeft);
            }
        }
```

References

Blueprint-Finance/sc_spokes-v1/src/userBase/utils/ProtectionHandler.sol#L424

7.13 UNSAFE USE OF TRANSFER()/TRANSFERFROM() WITH IERC20

// LOW

Description

Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. For example, Tether (USDT)'s `transfer()` and `transferFrom()` functions on L1 do not return booleans as the specification requires, and instead have no return value. When these tokens are cast to `IERC20`, their function signatures do not match, causing calls to revert.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:L/Y:L/R:N/S:U (3.1)

Recommendation

Use OpenZeppelin's `SafeERC20` library, which provides `safeTransfer()` and `safeTransferFrom()` functions. These functions handle non-standard ERC20 implementations safely, ensuring compatibility with a wider range of tokens.

Remediation Comment

SOLVED: The **Concrete team** solved the issue as recommended.

7.14 INSOLVENT POSITION CANNOT BE LIQUIDATED

// LOW

Description

The `foreclose` function implemented in the `UserBase` contract is designed to handle the liquidation of a position. It interacts with a flash loan mechanism to acquire the necessary funds for foreclosure, either via a primary method or a fallback, depending on availability.
The function invokes the `_requestFlashLoan` method on the funds' requester contract.

```
330     function foreclose(uint256 foreclosureFeeFractionInWad, address beneficiary)
331         external
332         virtual
333         override(IProtocolIntervention)
334     {
335         // _foreclose();
336         uint256 protectionInfo = _getProtectionInfo();
337         if (_callerIsEndpoint() && protectionInfo.getProtocolEnabledForClosureAndForeclosure()
338             _requestFlashLoan(
339                 _borrowTokenInfo.getAddress(), _getLenderDebt(false), foreclosureFeeFractionIn
340             );
341             return;
342         }
343 
344         revert Errors.UnauthorizedTriggerOfForeclosure();
345     }
```

The `requestFlashLoan` function from the FundsRequester attempts to obtain the funds necessary to repay the position on the lending platform, through a claim router. If successful, it proceeds with the foreclosure process and then repays the claim router. If the attempt with the claim router fails, it falls back to using Aave to obtain a flash loan for the required funds.

```
59      function requestFlashLoan(address token, uint256 amountInBorrowToken, uint256 feeFractionI
60          external
61          onlyUserBlueprint
62      {
63          // try and except handling ... first try the earn claim router, then try Aave
64          address userBlueprint = msg.sender;
65          IClaimRouter claimRouter = IClaimRouter(REMOTE_REGISTRY.getEarnClaimRouter());
66          try claimRouter.requestToken(VaultFlags(0), token, amountInBorrowToken, payable(userBl
67              // now execute the flashloan
68              IProtocolIntervention(userBlueprint).executeForecloseByFundsRequester(
69                  amountInBorrowToken, 0, feeFractionInWad, beneficiary
70              );
71              // now pull the funds from the user and repay the claim router
72              claimRouter.repay(token, amountInBorrowToken, payable(userBlueprint));
73          } catch {
74              // fallback to Aave
75              if (address(POOL) == address(0)) revert Errors.AavePoolNotSet();
76 
77              POOL.flashLoanSimple(
78                  address(this), token, amountInBorrowToken, abi.encode(userBlueprint, feeFracti
79              );
80          }
81      }
```

During the flash loan process, the `_executeForeclosure` function in the ProtectionHandler contract is invoked. It repays the borrowed credit using the flash loaned amount, withdraws the position's collateral, swaps it into the borrowed token, and then transfers the swapped amount to the fund requester to repay the flash loan.

```
360         params.totalRepayAmountInToken = repayAmount + premium;
361         params.collateralAmountLeft = _getLenderSupplied(false);
```

```
362
363         // ------ repay lender ------
364         _repay(borrowToken, repayAmount, Uint8CodecLib.encodeOnlyMode(uint8(SendFundsModality.
365         // ------ withdraw collateral from lender ------
366         _withdraw(
367             params.collateralToken,
368             params.collateralAmountLeft,
369             Uint8CodecLib.encodeOnlyMode(uint8(SendFundsModality.ONLY_FIRST_STEP))
370         );
371
372         // User has 3 ETH in AAVE, he has 5000 USDC debtToken in AAVE. OF THE 3 ETH, 0.5 ETH a
373         // 1. We request 5000 USDC from the fundsRequester
374         // 2. Either we get it from claimRouter or from AaveFlashloan
375         // 3. _executeForeclosure is called with uint256 repayAmount, uint256 premium, uint256
376         // 4. We use this money to repay the 5000 USDC debt in AAVE
377         // 5. We get the 3 ETH from the AAVE
378         // 6. We repay to the claimRouter the 0.5 ETH of users Concrete Debt.
379         // 7. We swap as much as possible into USDC of the remaining 2.5 ETH
380         // 8. We need to make sure that enough money (repayAmount + premium) is available in t
381         // 9. We deduct the 50 USDC fee from the 5000 USDC (in borrow token)
382         // 10 The rest of the ETH and USDC is send back to the user.
383
384         // ------ repay the credit injections ------
385         params.concreteDebtInCollateral = _getTotalConcreteDebtOfUser();
386
387         if (params.concreteDebtInCollateral > 0) {
388             if (params.concreteDebtInCollateral > params.collateralAmountLeft) {
389                 params.concreteDebtInCollateral = params.collateralAmountLeft;
390             }
391             _repayTokenToClaimRouter(params.claimRouter, params.collateralToken, params.concre
392             params.collateralAmountLeft -= params.concreteDebtInCollateral;
393         }
394
395         // ------ swap collateral to borrow token ------
396         if (params.collateralAmountLeft > 0) {
397             params.amountInBorrowTokenLeft = _swap(params.collateralToken, borrowToken, params
398         }
399         if (params.amountInBorrowTokenLeft < params.totalRepayAmountInToken) {
400             params.totalRepayAmountInToken = params.amountInBorrowTokenLeft;
401         }
402
403         // ------ repay to the claimRouter/fundRequester(flashloan) ------
404         params.amountInBorrowTokenLeft -= params.totalRepayAmountInToken;
405         if (isEarnFlashloan) {
406             _increaseAllowanceIfNecessary(params.claimRouter, borrowToken, params.totalRepayAm
407         } else {
408             IERC20(borrowToken).transfer(params.fundRequester, params.totalRepayAmountInToken)
409         }
```

However, when a position becomes insolvent, the collateral swapped into the borrowed token is insufficient to repay the flash loan. This situation prevents the foreclosure process from successfully liquidating the position, leading to the position being stuck in an insolvent state. When integrating with high-liquidity and high-performance lending protocols like Aave or Compound, this will not be an issue, as positions are liquidated before reaching insolvency. In the future though, it may be necessary to ensure that the Blueprint position can be always liquidated by the protocol.

Proof of Concept

```
    function test_CannotLiquidateInsolventPosition() public {
        address userBlueprint = _createAaveV3UserBlueprint();

        uint256 supplyAmount = 1000 ether;
        uint256 borrowAmount = 1_500_000 * 10 ** 6;
        uint256 promisedAmountInCollateral;
        uint256 encodedProtectionInfo;
        (encodedProtectionInfo, promisedAmountInCollateral) = _createConcreteProtectionEncoding(suppl
        assertEq(promisedAmountInCollateral, 100 ether);

        weth.mint(alice, supplyAmount);
        vm.prank(alice);
        weth.approve(userBlueprint, supplyAmount);
```

```
        vm.startPrank(address(cccm));
            //The user supplies 1000 WETH and borrow 1500000 USDC
            IUserIntervention(userBlueprint).supply(supplyAmount, 0);
            IUserIntervention(userBlueprint).setBorrowToken(address(usdc));
            IUserIntervention(userBlueprint).borrow(borrowAmount, 0);
        vm.stopPrank();

        //Simulate a price drop of 25%
        uint256 newEthPrice = initialEthPriceInBaseUnits.mulDiv(75, 1000);
        _updatePrice(aaveSetup.priceOracle, address(weth), newEthPrice);

        //The claim router fails to provide funds
        claimRouter.setRevertOnRequest(true);
        vm.prank(address(cccm));
            //Foreclosure reverts because the position is insolvent
            vm.expectRevert();
            IProtocolIntervention(userBlueprint).foreclose(liteForeclosureFeeFractionInWad, address(0
    }
```

```
Ran 1 test for test/stories/ProtectionFlow.sol:ProtectionFlow
[PASS] test_CannotLiquidateInsolventPosition() (gas: 1236828)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.40ms (1.51ms CPU time)

Ran 1 test suite in 142.06ms (9.40ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:H/C:N/I:N/A:H/D:H/Y:N/R:N/S:U (3.1)

Recommendation

In cases where the collateral is insufficient to cover the full repayment of the flash loan, consider implementing a mechanism to obtain the remaining required tokens using a separate liquidity designated specifically for such situations.

Remediation Comment

SOLVED: The **Concrete team** solved the issue. In cases where the flashloan provider cannot be repaid, the missing amount is pulled from the user blueprint. The user blueprint can hold funds, either deposited by its owner or by Concrete's treasury. If there are still insufficient funds (or none) in the user blueprint, the missing amount is pulled from the treasury. If this is not possible, the transaction reverts. This setup allows either the addition of funds to the user blueprint to rescue the position or the approval of the blueprint to pull the remaining funds from the treasury.

References

Blueprint-Finance/sc_spokes-v1/src/userBase/utils/ProtectionHandler.sol#L342-L428

7.15 USAGE OF DIRECT APPROVE CALLS

// LOW

Description

In order to allow for the transfer of tokens from one address, the protocol calls the `approve` function using the IERC20 interface in several places. This approach might be problematic for a few reasons:

- Some tokens, to protect against approval race conditions, do not allow approving an amount M > 0 when an existing amount N > 0 is already approved.
- The approve call does not return a boolean.
- The function reverts if the approval value is larger than uint96.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:L/Y:N/R:N/S:U (3.1)

Recommendation

It is recommended to use `safeIncreaseAllowance` and `safeDecreaseAllowance` from the SafeERC20 library across the entire protocol, instead of calling `approve` directly.

Remediation Comment

SOLVED: The **Concrete team** fixed the issue as recommended.

7.16 USAGE OF SINGLE-STEP ROLE TRANSFER

// LOW

Description

The `RemoteRegistry` contract implements setter functions that allow the modification of critical roles and components within the system. The transfer of these critical roles is done through a single-step process. The address to which ownership is transferred should be verified to ensure it is active and willing to act as the owner. This is especially important for the `setEndpointAddress` and `setNewRegistryAddress` functions.

BVSS

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:U (2.0)

Recommendation

Consider implementing a two-step transfer pattern. In this approach, the privileged account nominates a new address, and the nominated account must then accept the role for the ownership transfer to be fully completed. By separating the process, this ensures that the new account is both active and willing to assume the associated permissions.

Remediation Comment

SOLVED: The **Concrete team** fixed the issue. The `proposeEndpointAddress`, `confirmNewEndpointAddress` and `revokeEndpointProposal` function were added. The `setNewRegistry` function was deleted.

7.17 UNLOCKED PRAGMA COMPILERS

// INFORMATIONAL

Description

The files in scope currently use floating pragma version `^0.8.24`, which means that the code can be compiled by any compiler version that is greater than or equal to `0.8.0`, and less than `0.9.0`. It is recommended that contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.
Additionally, using a newer compiler version that introduces default optimizations, including unchecked overflow for gas efficiency, presents an opportunity for further optimization.

BVSS

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

Recommendation

Lock the pragma version to the same version used during development and testing.

Remediation Comment

SOLVED: The **Concrete team** fixed the issue as recommended.

7.18 CONSIDER USING NAMED MAPPINGS

// INFORMATIONAL

Description

The project is using Solidity version greater than 0.8.18, which supports named mappings. Using named mappings can improve the readability and maintainability of the code by making the purpose of each mapping clearer. This practice will enhance code readability and make the purpose of each mapping more explicit.

BVSS

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

Recommendation

Consider refactoring the mappings to use named arguments.
For example, on `PortfolioProxy`, instead of declaring:

```
mapping(address => uint256) public _portfolioId;
```

The mapping could be declared as:

```
mapping(address userBlueprint => uint256 id) public _portfolioId;
```

Remediation Comment

SOLVED: The **Concrete team** fixed the issue as recommended.

7.19 UNUSED IMPORTS, ERRORS AND EVENTS

// INFORMATIONAL

Description

Throughout the code in scope, there are several instances where the imports, errors, and events are declared but never used.

In `PriceFeedManagerEvents.sol` :

- `event PriceProviderRemoved(address indexed token);`

In `RemoteRegistryEvent.sol`:

- `event RemoteChainwideInterventionTypeSetTo(InterventionConstraintType type_);`
- `event RemoteChainwideAllowUserDirectHandlingSetTo(bool enable);`
- `event OwnerTransferred(address user, address newOwner);`

In `Errors.sol`:

- `error ProtectionMetadataUnequalArrayLengths();`
- `error ProtectionMetadataArrayLengthExceedsLimit();`
- `error NotProtocolAndUserCannotControl();`
- `error PositionAlreadyProtected();`
- `error PositionNotProtected();`
- `error WithdrawalExceedsUserFractionOfFunds();`
- `error OwnerAddressNotSet();`
- `error ClaimRouterAddressZero();`
- `error AssetMustBeCollateral();`
- `error AssetMustBeBorrowToken();`
- `error EndpointCannotBeZeroAddress();`
- `error RemoteProtocolProxyCannotBeZeroAddress();`
- `error PublicExecutorCannotBeZeroAddress();`
- `error UnsupportedAsset(address token);`

In `RemoteExecutorsBookkeeper.sol`:

- `import {IProtocolIntervention} from "../userBase/interfaces/IProtocolIntervention.sol";`

In `FundsRequester.sol`:

- `import {IUserBlueprint} from "../userBlueprints/interfaces/IUserBlueprint.sol";`

In `Inspector.sol`:

- `import {IProtocolIntervention} from "../userBase/interfaces/IProtocolIntervention.sol";`
- `import {IUserIntervention} from "../userBase/interfaces/IUserIntervention.sol";`

In `ProtectionViewLib.sol`:

- `import {Errors} from "../helpers/Errors.sol";`

In `PortfolioProxyRemoteTransfer.sol`:

- import {IConcreteCrossChainMessaging as ICCCM} from "@cccm-evm/src/interfaces/IConcreteCrossChainMessaging.sol";
- import {IRemoteRegistry} from "../remoteRegistry/interfaces/IRemoteRegistry.sol";
- import {IResponseHandler, IProtocol} from "../userBase/interfaces/IProtocolResponse.sol";

In CloneFactory.sol:

- import {EndpointAddressGetterStandalone} from "@cccm-evm/src/integration/Getters.sol";
- import {OnlyEndpointStandalone} from "@cccm-evm/src/integration/OnlyEndpoint.sol";
- import {IRemoteRegistry} from "./interfaces/IRemoteRegistry.sol"; import {Errors} from "../helpers/Errors.sol";

In RemoteRegistry.sol:

- import {OnlyEndpointStandalone} from "@cccm-evm/src/integration/OnlyEndpoint.sol";
- import {InterventionConstraintType} from "../helpers/DataTypes.sol";

In SwapperUniV3.sol:

- import {IRemoteRegistry} from "../../remoteRegistry/interfaces/IRemoteRegistry.sol";

In SwapperBase.sol:

- import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

In UserBase.sol:

- import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";
- import {IERC20, IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";
- import {IResponseHandler, IProtocol} from "./interfaces/IProtocolResponse.sol";

In ExecutorConnector.sol:

- import {Errors} from "../../helpers/Errors.sol";
- import {Uint8CodecLib} from "@hub-and-spokes-libs/src/libraries/Uint8CodecLib.sol";

In ProtectionHandler.sol:

- import {IClaimRouter} from "../../fundsRequester/interfaces/IClaimRouter.sol";
- import {InterventionConstraintType} from "../../helpers/DataTypes.sol";

In RemoteRegistryConnector.sol:

- import {InterventionConstraintType} from "../../helpers/DataTypes.sol";

In UserAaveV3.sol:

- `import {EthereumMainnetAddresses as ADDR} from "../addresses/EthereumMainnet.sol";`
- `import {TokenType, PositionInteractionType, SendFundsModality} from "../helpers/DataTypes.sol";`

In `UserCompoundV3.sol`:

- `import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";`
- `import {EthereumMainnetAddresses as ADDR} from "../../src/addresses/EthereumMainnet.sol";`
- `import {TokenType, PositionInteractionType, SendFundsModality} from "../../src/helpers/DataTypes.sol";`

In `UserRadiantV2.sol`:

- `import {EthereumMainnetAddresses as ADDR} from "../addresses/EthereumMainnet.sol";`
- `import {TokenType, PositionInteractionType, SendFundsModality} from "../helpers/DataTypes.sol";`

In `UserSiloV1.sol`:

- `import {IPriceOracleGetter} from "@aave-v3-core/contracts/interfaces/IPriceOracleGetter.sol";`
- `import {BASE_DENOMINATION, WAD, BP, PRICE_FEED_QUOTE_DENOMINATION} from "../helpers/Constants.sol";`
- `import {EthereumMainnetAddresses as ADDR} from "../addresses/EthereumMainnet.sol";`
- `import {TokenType, PositionInteractionType, SendFundsModality} from "../helpers/DataTypes.sol";`

BVSS

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

Recommendation

Consider removing all unused imports, errors and events.

Remediation Comment

SOLVED: The **Concrete team** fixed the issue as recommended.

7.20 COMMENTED OUT CODE

// INFORMATIONAL

Description

The `SwapperBEX` contract is entirely commented out, including both the constructor and the swap function. While the contract structure is present, no functional code is active and the swap function contains only a placeholder comment for future implementation.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IRemoteRegistry} from "../../remoteRegistry/interfaces/IRemoteRegistry.sol";
import {Errors} from "../../helpers/Errors.sol";
import {SwapperBase} from "../SwapperBase.sol";

// contract SwapperBEX is SwapperBase {
//     constructor(address remoteRegistry_) SwapperBase(remoteRegistry_) {}

//     function _swap(address fromToken, address toToken, uint256 amount)
//         internal
//         override
//         returns (uint256 swappedAmount)
//     {
//         // @Atul, please implement the swap logic here.
//         // If you need some addresses, put them into the constructor.
//     }
// }
```

BVSS

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

Recommendation

Either complete the implementation of the swap logic or remove the unused `SwapperBEX` contract to reduce unnecessary complexity.

Remediation Comment

SOLVED: The **Concrete team** fixed the issue as recommended. The `SwapperBEX` contract was removed.

References

Blueprint-Finance/sc_spokes-v1/src/swapper/implementations/SwapperBEX.sol

7.21 EXCESS GAS CONSUMPTION DUE TO REDUNDANT TOKEN TRANSFERS

// INFORMATIONAL

Description

In logic of function implemented on `SEND_THROUGH` modality on some user blueprints, redundant token transfers are made by first withdrawing tokens to the contract and then transferring them to the user. This leads to unnecessary gas costs due to additional transfer steps. By utilizing `withdrawTo`, `claimTo` (for Compound V3), `withdrawFor` and `borrowFor` (for SiloV1) functions, tokens can be transferred directly to the user, reducing the number of operations and thus gas costs.

BVSS

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

Recommendation

Replace the two-step process of withdrawing and transferring tokens with methods that allow direct sending of funds to the user's address whenever possible, to minimize gas usage.

Remediation Comment

SOLVED: The **Concrete team** acknowledged the issue.

7.22 LACK OF EVENT EMISSION

// INFORMATIONAL

Description

It has been observed that Blueprint and Remote Registry functionalities are missing emitting events. Events are a method of informing the transaction initiator about the actions taken by the called function. It logs its emitted parameters in a specific log history, which can be accessed outside of the contract using some filter parameters. Events help non-contract tools to track changes, and events prevent users from being surprised by changes.

BVSS

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 Comment

SOLVED: The **Concrete team** fixed the issue as recommended.

7.23 INCONSISTENT NAMING CONVENTION IN PROTECTIONLIBV1 FOR VALUE UNITS

// INFORMATIONAL

Description

The ProtectionLibV1 library incorrectly uses `InBase` in variable names and function parameters, even though the values should represent collateral decimals. This mismatch in naming convention can cause confusion in the future maintenance.

BVSS

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

Recommendation

Update variable names and parameters throughout the ProtectionLibV1 library to reflect that the values are in collateral decimals, ensuring clarity and consistency in the codebase.

Remediation Comment

SOLVED: The **Concrete team** fixed the issue as recommended.

7.24 UNUSED PARAMETER IN CLAIMPROTECTION FUNCTION

// INFORMATIONAL

Description

The `externalBeneficiary` parameter in the `claimProtection` function of `UserBase` contract is unused and does not impact the function's logic. This parameter can be safely removed without affecting functionality, as it is disregarded when claims are triggered.

```
313     function claimProtection(uint256 amount, uint256 protectionFeeInToken, address externalBen
314         external
315         virtual
316         override(IProtocolIntervention)
317     {
318         uint256 protectionInfo = _getProtectionInfo();
319         // if sender is endpoint then claim protection
320         if (_callerIsEndpoint() && protectionInfo.getProtocolEnabledForClaimsAndReclaims()) {
321             // external Beneficiary is disregarded.
322             protectionInfo = _claimProtectionDefault(protectionInfo, amount, protectionFeeInTo
323             _setProtectionInfo(protectionInfo);
324             return;
325         }
326         externalBeneficiary;
327         revert Errors.UnauthorizedTriggerOfClaimProtection();
328     }
```

BVSS

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

Recommendation

Remove the `externalBeneficiary` parameter from the `claimProtection` function to simplify the code and improve clarity.

Remediation Comment

ACKNOWLEDGED: The **Concrete team** keep the function signature with the external beneficiary parameter as it will be used in the future versions.

References

Blueprint-Finance/sc_spokes-v1/src/userBase/UserBase.sol#L326

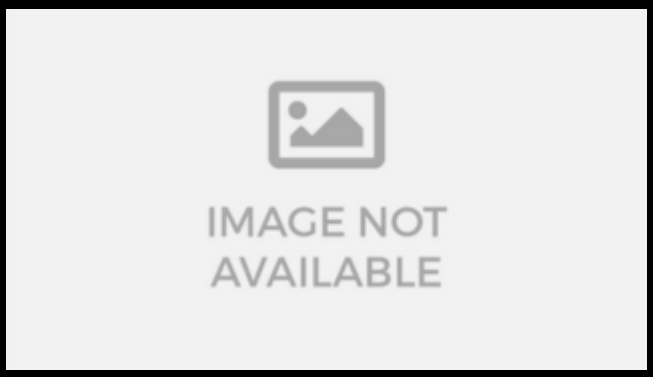
8. AUTOMATED TESTING

Static Analysis Report

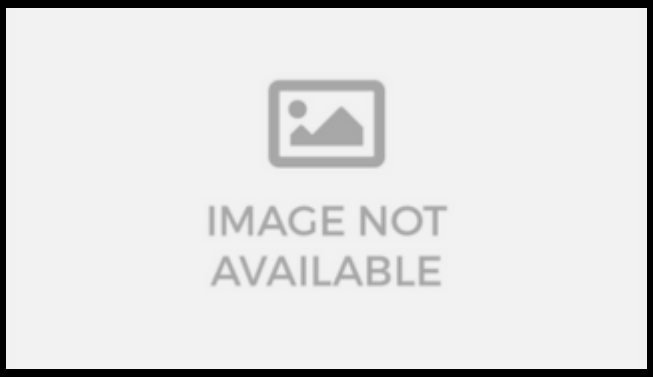
Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the scoped contracts. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their ABI and binary formats, Slither was run on the all-scoped 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.

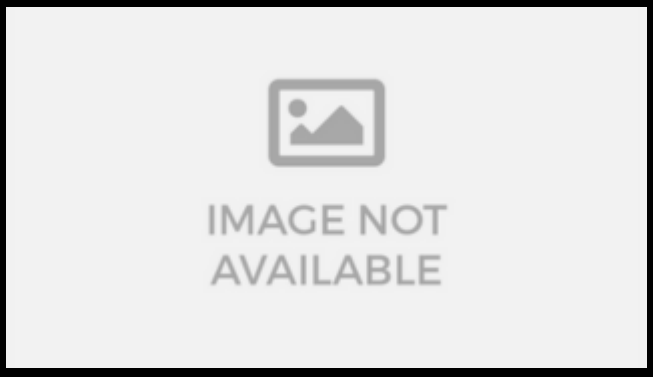
FundsRequester.sol



ProtectionViewLib.sol

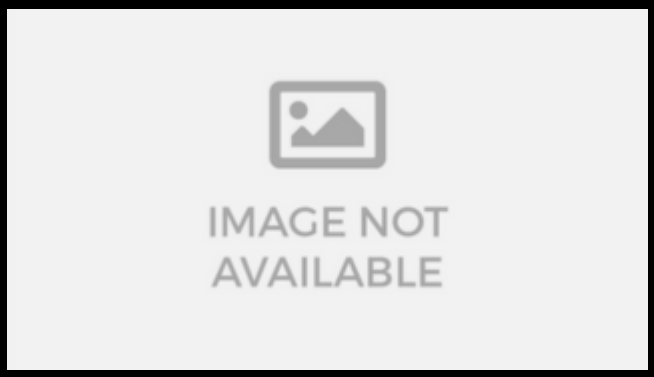


PortfolioProxy.sol

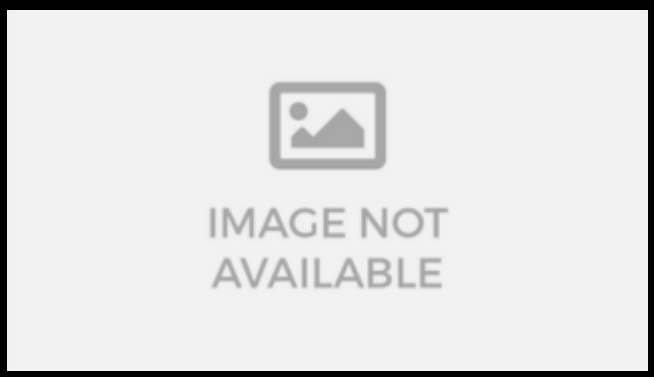


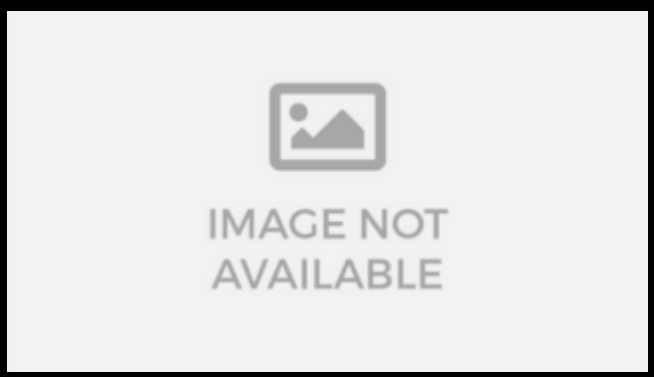
RemoteRegistry.sol

SwapperUniV3.sol

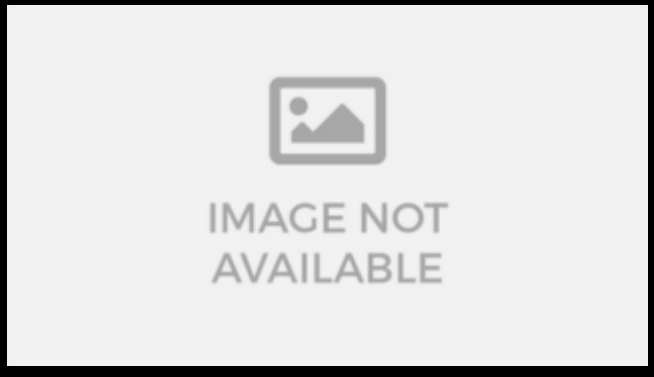


UserBase.sol

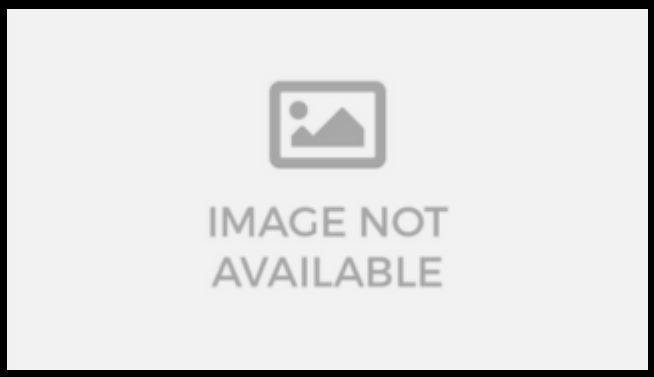


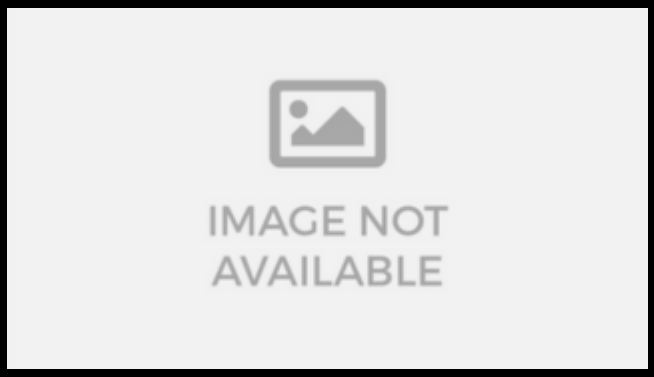


FundsRequesterConnector.sol

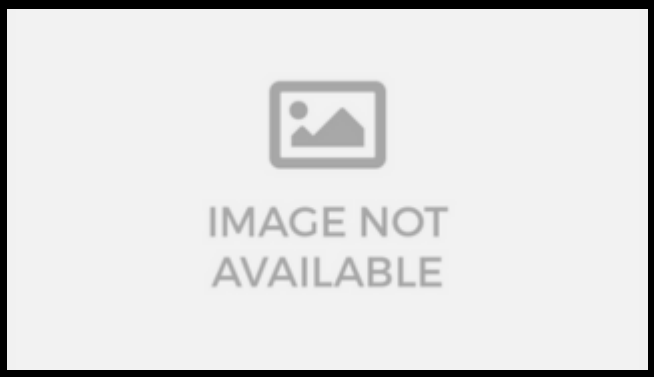


ProtectionHandler.sol

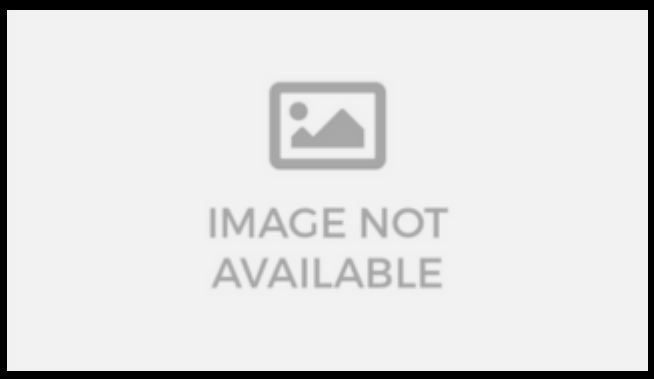


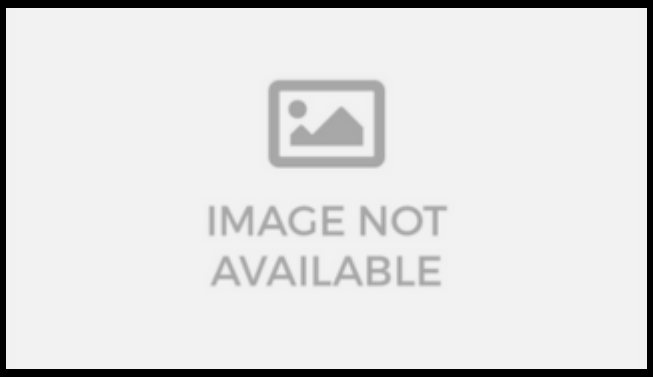


TokenInfo.sol

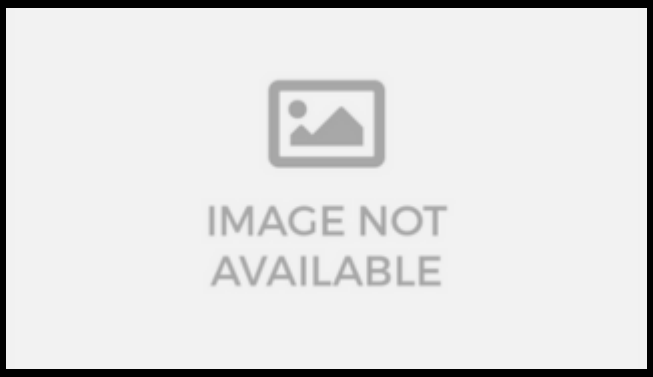


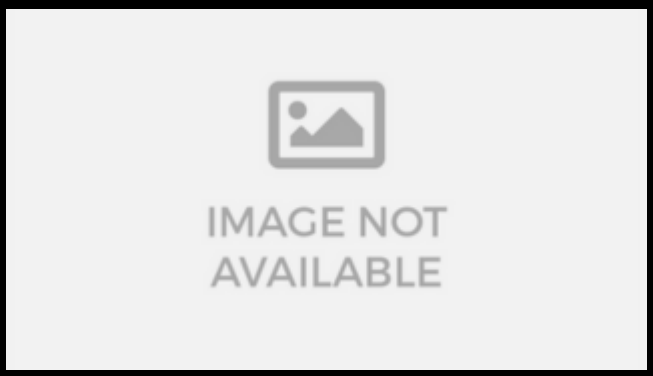
UserAaveV3.sol



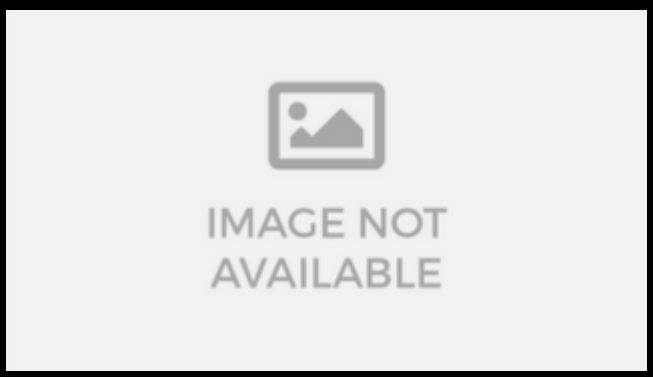


UserCompoundV3.sol

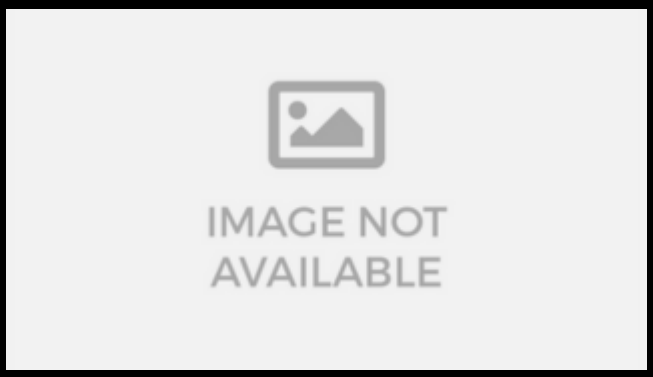


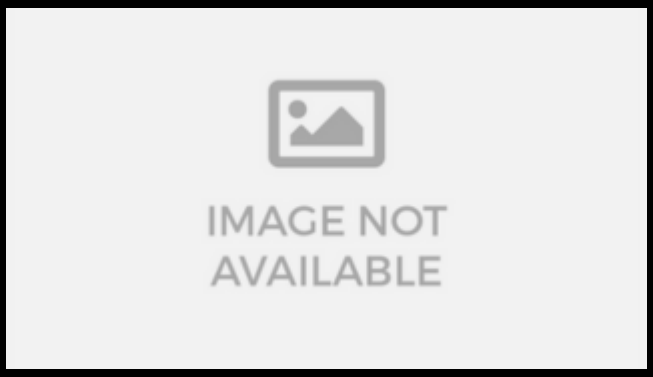


UserRadiantV2.sol



UserSiloV1.sol





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.