// Private Smart Contract Security Assessment 02.26.2024 - 04.03.2024

Core Contracts

GoldLink

HALBORN

Core Contracts - GoldLink

Prepared by: HALBORN

Last Updated 05/19/2024

Date of Engagement by: February 26th, 2024 - April 3rd, 2024

Summary

88% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
8	0	0	1	5	2

TABLE OF CONTENTS

1. Introduction

GoldLink engaged Halborn to conduct a security assessment on their smart contracts beginning on February 26th, 2024 and ending on April 3rd, 2024. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn assigned a full-time security engineer to verify the security of the smart contracts. 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 some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the GoldLink team. The main ones were the following:

- Only execute the transfer logic if the amount of assets to be transferred is different to zero.
- Ensure a minimum executor premium for liquidators or make use of liquidation bots.
- Set upper / lower boundaries for the health score parameters.
- Verify that the premium rates and the optimal utilization parameter are less than 100%.
- Verify if the attempt to fetch the asset decimals is successful or not before further processing.

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

EXPLOITABILIY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
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
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

SEVERITY	SCORE VALUE RANGE
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: goldlink-contracts

(b) Assessed Commit ID: 92aab7e

(c) Items in scope:

- contracts/core/ControllerHelpers.sol
- contracts/core/InterestRateModel.sol
- contracts/core/StrategyAccount.sol
- contracts/core/StrategyBank.sol
- contracts/core/StrategyController.sol
- contracts/core/StrategyReserve.sol

Out-of-Scope: Third party dependencies., Economic attacks., Attack vectors that could arise from the strategies' implementation.

REMEDIATION COMMIT ID:

- 6a643126a64312
- d1ee02fd1ee02f
- b64fea3b64fea3
- 62d294c62d294c
- 0ec51c80ec51c8
- 48fb48048fb480
- b26fcddb26fcdd

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	5	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - IMPROPER HANDLING OF ZERO TRANSFERS FOR SOME ERC20 TOKENS	Medium	SOLVED - 05/14/2024
HAL-02 - LACK OF INCENTIVES TO LIQUIDATE UNPROFITABLE STRATEGY ACCOUNTS	Low	FUTURE RELEASE - 03/12/2024
HAL-03 - HEALTH SCORE PARAMETERS DO NOT HAVE BOUNDARIES	Low	SOLVED - 04/22/2024
HAL-04 - UNCHECKED PREMIUMS	Low	SOLVED - 04/19/2024
HAL-05 - UNCHECKED OPTIMAL UTILIZATION	Low	SOLVED - 05/17/2024
HAL-06 - IMPROPER HANDLING OF FAILED ATTEMPTS TO FETCH ASSET DECIMALS	Low	SOLVED - 04/23/2024
HAL-07 - REPAYING LOANS COULD REVERT WITHOUT AN ACCURATE ERROR MESSAGE	Informational	SOLVED - 05/14/2024
HAL-08 - LACK OF ZERO ADDRESS CHECK	Informational	SOLVED - 04/19/2024

7. FINDINGS & TECH DETAILS

7.1 (HAL-01) IMPROPER HANDLING OF ZERO TRANSFERS FOR SOME ERC20 TOKENS

// MEDIUM

Description

Some functions don't verify if the amount of assets to be transferred are different from zero. Because there are some ERC20 tokens that reverts when trying to transfer zero tokens (e.g. LEND), it could have **three different and independent consequences** in the protocol:

1. `repay`**:** The vulnerability could revert a liquidation if `returnedLoan` is zero (e.g.: in case of a loss) and as a consequence, the **liquidation will get stuck indefinitely**.

2. `syncAndAccrue`**:** The vulnerability affects all the operations in the **StrategyReserve** contract that relies on the `syncAndAccrue` modifier and makes them unusable every time that the interest accrued is zero: `updateModel`, `borrowAssets`, `repay`, `settleInterest`, `deposit`, `mint`, `withdraw` and `redeem`. Although the unavailability in this case is temporal, it could affect multiple users and multiple transactions every time the interest accrued is zero, which happens in two different scenarios:

- Two or more transactions are executed by a user in the same block.
- A function calls other internal functions and the synchronization is executed more than once in the same transaction.

3. Finally, for `StrategyReserve.borrowAssets` and `StrategyAccount.executeWithdrawErc20Assets`, the impact is minor because the vulnerability will impact the users who try to borrow / withdraw zero tokens.

Code Location

The `syncAndAccrue` modifier in the **StrategyReserve** contract doesn't verify if the amount of assets to be transferred is different from zero:

```
84  modifier syncAndAccrue() {
85    // Get the used and total asset amounts.
86    uint256 used = utilizedAssets_;
87    uint256 total = used + reserveBalance_; // equal to totalAssets()
88
89    // Settle interest that has accrued since the last settlement,
90    // and get the new amount owed on the utilized asset balance.
91    uint256 interestOwed = _accrueInterest(used, total);
92
93    // Take interest from `StrategyBank`.
94    //
95    // Note that in rare cases it is possible for the bank to underpay,
96
```

```
    // if it has insufficient collateral available to satisfy the
payment.
    // In this case, the reserve will simply receive less interest than
    // expected.
    uint256 interestToPay = STRATEGY_BANK.getInterestAndTakeInsurance(
      interestOwed
    );

    // Update reserve balance with interest to be paid.
    reserveBalance_ += interestToPay;

    // Transfer interest from the strategy bank. We use the amount
    // returned by getInterestAndTakeInsurance() which is guaranteed to
    // be less than or equal to the bank's ERC-20 asset balance.
    STRATEGY_ASSET.safeTransferFrom(
      address(STRATEGY_BANK),
      address(this),
      interestToPay
    );

    // Run the function.
    _;
  }
```

The `borrowAssets` function in the **StrategyReserve** contract doesn't verify if the amount of assets to be transferred is different from zero:

```
  function borrowAssets(
    address borrower,
    uint256 borrowAmount
  ) external override onlyStrategyBank syncAndAccrue {
    // Verify that the amount is available to be borrowed.
    require(
      availableToBorrow() >= borrowAmount,
      Errors.STRATEGY_RESERVE_INSUFFICIENT_AVAILABLE_TO_BORROW
    );

    // Increase utilized assets and decrease reserve balance.
    utilizedAssets_ += borrowAmount;
    reserveBalance_ -= borrowAmount;

    // Transfer borrowed assets to the borrower.
    STRATEGY_ASSET.safeTransfer(borrower, borrowAmount);

```

```
202
203     emit BorrowAssets(borrowAmount);
    }
```

The `repay` function in the **StrategyReserve** contract doesn't verify if the amount of assets to be transferred is different from zero:

```
214 function repay(
215   uint256 initialLoan,
216   uint256 returnedLoan
217 ) external onlyStrategyBank syncAndAccrue {
218   // Reduce utilized assets by assets no longer borrowed and increase
219   // reserve balance by the amount being returned, net of loan loss.
220   utilizedAssets_ -= initialLoan;
221   reserveBalance_ += returnedLoan;
222
223   // Effectuate the transfer of the returned amount.
224   STRATEGY_ASSET.safeTransferFrom(
225     address(STRATEGY_BANK),
226     address(this),
227     returnedLoan
228   );
229
230   emit Repay(initialLoan, returnedLoan);
231 }
```

The `executeWithdrawErc20Assets` function in the **StrategyAccount** contract doesn't verify if the amount of assets to be transferred is different from zero:

```
414 function executeWithdrawErc20Assets(
415   address receiever,
416   IERC20[] calldata tokens,
417   uint256[] calldata amounts
418 ) external onlyOwner strategyNonReentrant whenNotLiquidating
419 noActiveLoan {
420   uint256 n = tokens.length;
421
422   require(
423     amounts.length == n,
424     Errors.STRATEGY_ACCOUNT_PARAMETERS_LENGTH_MISMATCH
425   );
426
427   for (uint256 i; i < n; ++i) {
428
```

```
429         tokens[i].safeTransfer(receiever, amounts[i]);
430         emit WithdrawErc20Asset(receiever, tokens[i], amounts[i]);
        }
    }
```

Proof of Concept

Foundry test that shows 2 different scenarios:

1. Depositing 500 DAI tokens in the **StrategyReserve** contract will be a successful operation, as expected.
2. Depositing 500 LEND tokens in the **StrategyReserve** contract won't be a successful operation because this kind of token reverts when transferring 0 tokens at some point of the whole transaction.

```
function testDepositingTokens() public {

  IERC20 dai = IERC20(0x6B175474E89094C44Da98b954EedeAC495271d0F);
  IERC20 lend = IERC20(0x80fB784B7eD66730e8b1DBd9820aFD29931aab03);

  IStrategyReserve.ReserveParameters memory reserveParameters =
TestConstants.defaultReserveParameters();
  StrategyAccountDeployerMock strategyAccountDeployerMock = new
StrategyAccountDeployerMock();
  IStrategyBank.BankParameters memory bankParameters =
TestUtilities.defaultBankParameters(
    IStrategyAccountDeployer(address(strategyAccountDeployerMock))
  );

  // Operations with DAI token
  StrategyController strControllerDai = new StrategyController(msg.sender, dai,
reserveParameters, bankParameters);
  IStrategyReserve strReserveDai = strControllerDai.STRATEGY_RESERVE();

  deal(address(dai), msg.sender, 5*1e6*1e18, true);

  vm.prank(msg.sender);
  dai.approve(address(strReserveDai), 500);
  vm.prank(msg.sender);
  strReserveDai.deposit(500, msg.sender); // Depositing 500 DAI

  // Operations with LEND token
  StrategyController strControllerLend = new StrategyController(msg.sender, lend,
reserveParameters, bankParameters);
  IStrategyReserve strReserveLend = strControllerLend.STRATEGY_RESERVE();
```

```
  deal(address(lend), msg.sender, 5*1e6*1e18, true);

  vm.prank(msg.sender);
  lend.approve(address(strReserveLend), 500);
  vm.expectRevert();
  vm.prank(msg.sender);
  strReserveLend.deposit(500, msg.sender); // Depositing 500 LEND
}
```

The result of the test is the following:

```
❯ forge test --fork-url $FORK_URL --match-path tests/core/StrategyReserve.t.sol --match-test testDepositingTokens -vvv
[⠒] Compiling...
[⠢] Compiling 2 files with 0.8.20
[⠆] Solc 0.8.20 finished in 4.18s
Compiler run successful!

Running 1 test for tests/core/StrategyReserve.t.sol:StrategyReserveTest
[PASS] testDepositingTokens() (gas: 12823259)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 12.07s

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

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

Recommendation

Verify the amount of assets to be transferred and only execute the transfer logic if this amount is different from zero.

Remediation Plan

SOLVED: The **GoldLink team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/GoldLink-Protocol/goldlink-contracts/commit/6a6431261c8856ca9ab002060ab2c2a46d0674bb

7.2 (HAL-02) LACK OF INCENTIVES TO LIQUIDATE UNPROFITABLE STRATEGY ACCOUNTS

// LOW

Description

The `_getPremiums` function in the **StrategyBank** contract calculates the executor premiums for the liquidators. In case the `collateral` is 0 (e.g.: after adjusting the value for the loan payment), the `executorPremium` will be 0 and liquidators won't have incentives to liquidate unprofitable strategy accounts. As a consequence, the **StrategyBank** contract could have some temporary liquidity issues until those accounts are liquidated.

Code Location

The `_getPremiums` function doesn't ensure a minimum `executorPremium` for liquidators:

```
837  function _getPremiums(
838    uint256 collateral,
839    uint256 availableAssets
840  )
841    internal
842    view
843    returns (uint256 updatedCollateral, uint256 executorPremium)
844  {
845    // Apply the executor premium: the fee earned by the liquidator,
846    // calculated as a portion of the liquidated account value.
847    //
848    // Note that the premiums cannot exceed the collateral that is
849    // available in the account.
850    executorPremium = Math.min(
851      availableAssets.percentToFraction(EXECUTOR_PREMIUM),
852      collateral
853    );
854    updatedCollateral = collateral - executorPremium;
855
856    // Apply the insurance premium: the fee accrued to the insurance
857  fund,
858    // calculated as a portion of the liquidated account value.
859    updatedCollateral -= Math.min(
860      availableAssets.percentToFraction(LIQUIDATION_INSURANCE_PREMIUM),
861      updatedCollateral
862    );
863
```

```
864     return (updatedCollateral, executorPremium);
    }
```

BVSS

AO:A/AC:L/AX:M/R:P/S:U/C:N/A:H/I:M/D:N/Y:N (2.9)

Recommendation

It is recommended to ensure a minimum executor premium for liquidators or make use of liquidation bots, especially for accounts that are liquidatable and cannot repay completely their loans.

Remediation Plan

PENDING: The **GoldLink team** mentioned that they are working on a solution for this issue and stated the following:
"We are currently in the process of building an open sourced liquidation bot, and we as a company have pledged to liquidate any account regardless of whether or not it is profitable. Luckily, we are deploying on Arbitrum, so gas will be cheap in the worst case."

7.3 (HAL-03) HEALTH SCORE PARAMETERS DO NOT HAVE BOUNDARIES

// LOW

Description

The health score parameters (`LIQUIDATABLE_HEALTH_SCORE` and `minimumOpenHealthScore_`) do not have upper / lower boundaries. As a consequence, if any of them is mistakenly set, it could generate unexpected situations for the borrowing and lending process, e.g.: setting `minimumOpenHealthScore_` with a too low value could allow that loans' collateralizations are much less than expected, which would increase the risk of non-payment. The affected functions are the following:

- `constructor`
- `updateMinimumOpenHealthScore`

Code Location

The `constructor` doesn't have upper / lower boundaries for the `LIQUIDATABLE_HEALTH_SCORE` and `minimumOpenHealthScore_` parameters:

```
    constructor(
      address strategyOwner,
      IERC20 strategyAsset,
      IStrategyController strategyController,
      IStrategyReserve strategyReserve,
      BankParameters memory parameters
    ) Ownable(strategyOwner) ControllerHelpers(strategyController) {
      // Cannot set `minimumOpenHealthScore` at or below
    `liquidatableHealthScore`.
      require(
        parameters.minimumOpenHealthScore >
          parameters.liquidatableHealthScore,
        Errors

    .STRATEGY_BANK_MINIMUM_OPEN_HEALTH_SCORE_CANNOT_BE_AT_OR_BELOW_LIQUIDATAB
      );

      // Set immutable parameters.
      STRATEGY_ASSET = strategyAsset;
      STRATEGY_RESERVE = strategyReserve;
      INSURANCE_PREMIUM = parameters.insurancePremium;
      LIQUIDATION_INSURANCE_PREMIUM =
    parameters.liquidationInsurancePremium;
```

```
    EXECUTOR_PREMIUM = parameters.executorPremium;
    LIQUIDATABLE_HEALTH_SCORE = parameters.liquidatableHealthScore;
    MINIMUM_COLLATERAL_BALANCE = parameters.minimumCollateralBalance;
    STRATEGY_ACCOUNT_DEPLOYER = parameters.strategyAccountDeployer;

    // Set mutable parameters.
    minimumOpenHealthScore_ = parameters.minimumOpenHealthScore;

    // Set allowance for the `STRATEGY_RESERVE` to take allowed assets
  when repaying
    // loans.
    STRATEGY_ASSET.approve(address(STRATEGY_RESERVE), type(uint256).max);
  }
```

The `updateMinimumOpenHealthScore` function doesn't have upper / lower boundaries for the `minimumOpenHealthScore_` parameter:

```
function updateMinimumOpenHealthScore(
  uint256 newMinimumOpenHealthScore
) external onlyOwner {
  // Cannot set `minimumOpenHealthScore_` at or below the liquidation
threshold.
  require(
    newMinimumOpenHealthScore > LIQUIDATABLE_HEALTH_SCORE,
    Errors

.STRATEGY_BANK_MINIMUM_OPEN_HEALTH_SCORE_CANNOT_BE_AT_OR_BELOW_LIQUIDATAB
  );

  // Set new minimum open health score for this strategy bank.
  minimumOpenHealthScore_ = newMinimumOpenHealthScore;

  emit UpdateMinimumOpenHealthScore(newMinimumOpenHealthScore);
}
```

BVSS

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

Recommendation

It is recommended to set upper / lower boundaries for the health score parameters.

Remediation Plan

SOLVED: The **GoldLink team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/GoldLink-Protocol/goldlink-contracts/commit/d1ee02f28cd3db63c3fca6565d4f92911202a673

7.4 (HAL-04) UNCHECKED PREMIUMS

// LOW

Description

The `constructor` in the **StrategyBank** contract does not verify if `INSURANCE_PREMIUM`, `LIQUIDATION_INSURANCE_PREMIUM` and `EXECUTOR_PREMIUM` are less than 100%. As a consequence, if any of them is mistakenly set, it could generate that the protocol stops working, e.g.: setting `INSURANCE_PREMIUM` with a value greater than 100% would make that the transactions revert every time that the balance and accrue interest are synchronized.

Code Location

The `constructor` does not verify if the premium rates are less than 100%:

```
    constructor(
      address strategyOwner,
      IERC20 strategyAsset,
      IStrategyController strategyController,
      IStrategyReserve strategyReserve,
      BankParameters memory parameters
    ) Ownable(strategyOwner) ControllerHelpers(strategyController) {
      // Cannot set `minimumOpenHealthScore` at or below
    `liquidatableHealthScore`.
      require(
        parameters.minimumOpenHealthScore >
          parameters.liquidatableHealthScore,
        Errors

    .STRATEGY_BANK_MINIMUM_OPEN_HEALTH_SCORE_CANNOT_BE_AT_OR_BELOW_LIQUIDATAB
      );

      // Set immutable parameters.
      STRATEGY_ASSET = strategyAsset;
      STRATEGY_RESERVE = strategyReserve;
      INSURANCE_PREMIUM = parameters.insurancePremium;
      LIQUIDATION_INSURANCE_PREMIUM =
    parameters.liquidationInsurancePremium;
      EXECUTOR_PREMIUM = parameters.executorPremium;
      LIQUIDATABLE_HEALTH_SCORE = parameters.liquidatableHealthScore;
      MINIMUM_COLLATERAL_BALANCE = parameters.minimumCollateralBalance;
      STRATEGY_ACCOUNT_DEPLOYER = parameters.strategyAccountDeployer;
```

```
152     // Set mutable parameters.
153     minimumOpenHealthScore_ = parameters.minimumOpenHealthScore;
154
155     // Set allowance for the `STRATEGY_RESERVE` to take allowed assets
    when repaying
      // loans.
      STRATEGY_ASSET.approve(address(STRATEGY_RESERVE), type(uint256).max);
    }
```

BVSS

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

Recommendation

It is recommended to verify that the premium rates are less than 100% before further processing.

Remediation Plan

SOLVED: The **GoldLink team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/GoldLink-Protocol/goldlink-contracts/commit/b64fea352c7572c606048e966f25e8959fbb5eb3

7.5 (HAL-05) UNCHECKED OPTIMAL UTILIZATION

// LOW

Description

The `_updateModel` function in the **InterestRateModel** contract does not verify if `optimalUtilization` is less than 100%. As a consequence, if this value is mistakenly set too high, the `_getInterestRate` function will calculate an incorrect interest rate without notifying about the mistaken value in `optimalUtilization`.

Code Location

The `_updateModel` function does not verify if `optimalUtilization` is less than 100%:

```
62  function _updateModel(InterestRateModelParameters memory model)
63  internal {
64    model_ = model;
65
66    emit ModelUpdated(
67        model.optimalUtilization,
68        model.baseInterestRate,
69        model.rateSlope1,
70        model.rateSlope2
71    );
    }
```

BVSS

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

Recommendation

It is recommended to verify that the optimal utilization parameter is less than 100% before further processing.

Remediation Plan

SOLVED: The **GoldLink team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/GoldLink-Protocol/goldlink-contracts/commit/62d294c580ce76f665e3856f383fd3c9af88abfb

7.6 (HAL-06) IMPROPER HANDLING OF FAILED ATTEMPTS TO FETCH ASSET DECIMALS

// LOW

Description

The `constructor` in the **StrategyReserve** contract does not verify if the attempt to fetch the asset decimals is successful or not, i.e.: boolean return value in the `_tryGetAssetDecimals` function inherited from the ERC4626 contract.
As a consequence, a failed attempt (e.g., using an asset that has not been created yet) will set the decimals as 18, which could not match the final value for the asset decimals and invalidate all the operations that rely on the decimals value. Furthermore, a failed attempt could constitute a warning about the use of an incorrect asset address.

Code Location

The `constructor` does not verify if the attempt to fetch the asset decimals is successful or not before further processing:

```
121 constructor(
122   address strategyOwner,
123   IERC20 strategyAsset,
124   IStrategyController strategyController,
125   IStrategyReserve.ReserveParameters memory reserveParameters,
126   IStrategyBank.BankParameters memory bankParameters
127 )
128   Ownable(strategyOwner)
129   ERC20(reserveParameters.erc20Name, reserveParameters.erc20Symbol)
130   ERC4626(strategyAsset)
131   ControllerHelpers(strategyController)
132   InterestRateModel(reserveParameters.interestRateModel)
133 {
134   STRATEGY_ASSET = strategyAsset;
135
136   // Create the strategy bank.
137   STRATEGY_BANK = new StrategyBank(
138     strategyOwner,
139     strategyAsset,
140     strategyController,
141     this,
142     bankParameters
143   );
144
```

```
145     // Set TVL cap for this reserve.
146     tvlCap_ = reserveParameters.totalValueLockedCap;
147   }
```

BVSS

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

Recommendation

It is recommended to verify if the attempt to fetch the asset decimals is successful or not before further processing.

Remediation Plan

SOLVED: The **GoldLink team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/GoldLink-Protocol/goldlink-contracts/commit/0ec51c8474722979bb0127c3cdb942bd00ffb073

7.7 (HAL-07) REPAYING LOANS COULD REVERT WITHOUT AN ACCURATE ERROR MESSAGE

// INFORMATIONAL

Description

The `repayLoan` function in the **StrategyBank** contract tries to repay part of the loan using the assets in the strategy account. However, it could happen that when trying to repay, there is no enough liquidity in the account (e.g.: assets could have been used to open positions in GMX), so the transaction will revert without showing an accurate description of the root cause of the error.

Code Location

The `repayLoan` function will revert if there is no enough liquidity in the account without showing the root cause of the error:

```
473   // Reduce loan in holdings by total `repayAmount` as the portion of the
474   strategy account
475   // and potentially a portion of collateral are being transferred to the
476   strategy reserve.
477   // Since the account is not liquidatable, there is no concern that the
478   `repayAmount`
479   // will not be fully paid.
480   holdings.loan -= repayAmount;
481
482   // Repay loan portion (`repayAmount`) to strategy reserve.
483   _repayAssets(
484     repayAmount,
485     repayAmount,
486     strategyAccount,
487     collateralRepayment
      );

      emit RepayLoan(strategyAccount, repayAmount, collateralRepayment);
```

Score

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

Recommendation

It is recommended to show accurate descriptions of the root cause of errors, so users can understand them more easily.

Remediation Plan

SOLVED: The **GoldLink team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/GoldLink-Protocol/goldlink-contracts/commit/48fb48025fb1d72b6152a6757e462d2d08b6fa85

7.8 (HAL-08) LACK OF ZERO ADDRESS CHECK

// INFORMATIONAL

Description

Some functions in the codebase do not include a **zero address check** for their parameters. If one of those parameters is mistakenly set to zero, it could affect the correct operation of the protocol. The affected contracts are the following:

- `ControllerHelpers`
- `StrategyReserve`

Code Location

The `constructor` in the **ControllerHelpers** contract does not have a zero address check for the `STRATEGY_CONTROLLER` parameter:

```
42  constructor(IStrategyController controller) {
43    STRATEGY_CONTROLLER = controller;
44  }
```

The `constructor` in the **StrategyReserve** contract does not have a zero address check for the `STRATEGY_ASSET` parameter:

```
121 constructor(
122   address strategyOwner,
123   IERC20 strategyAsset,
124   IStrategyController strategyController,
125   IStrategyReserve.ReserveParameters memory reserveParameters,
126   IStrategyBank.BankParameters memory bankParameters
127 )
128   Ownable(strategyOwner)
129   ERC20(reserveParameters.erc20Name, reserveParameters.erc20Symbol)
130   ERC4626(strategyAsset)
131   ControllerHelpers(strategyController)
132   InterestRateModel(reserveParameters.interestRateModel)
133 {
134   STRATEGY_ASSET = strategyAsset;
135
136   // Create the strategy bank.
137   STRATEGY_BANK = new StrategyBank(
138     strategyOwner,
139     strategyAsset,
140
```

```
141       strategyController,
142       this,
143       bankParameters
144     );
145
146     // Set TVL cap for this reserve.
147     tvlCap_ = reserveParameters.totalValueLockedCap;
    }
```

Score

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

Recommendation

It is recommended to add a zero address check in the functions mentioned above.

Remediation Plan

SOLVED: The **GoldLink team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/GoldLink-Protocol/goldlink-contracts/commit/b26fcdd49e2da07542bcb885a51272a640a07c4e

---