SMART CONTRACT AUDIT REPORT

for

Stella

Prepared By: Xiaomi Huang

PeckShield

June 3, 2023

Document Properties

Client	Stella
Title	Smart Contract Audit Report
Target	Stella
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 3, 2023	Xuxian Jiang	Final Release
1.0-rc	May 20, 2023	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Stella` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Stella

`Stella` is a leveraged strategies protocol with 0% cost to borrow, utilizing the `pay-as-you-earn (PAYE)` model. Lenders can lend supported assets to earn shared profits from leveraged users. On the other hand, leveraged users can borrow lent assets at 0% cost to gain higher leverage of up to $10x$ and earn more profits. Supported strategies include yield farming, liquidity providing, staking, and integrations with many other protocols. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Stella

Item	Description
Name	Stella
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 3, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/AlphaFinanceLab/stella-arbitrum-private-contract.git (e62f8d5)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/AlphaFinanceLab/stella-arbitrum-private-contract.git (dcdd1f5)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Stella` implementations. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	3	■ ■ ■
Informational	2	■ ■
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Revisited getUSDPriceE36() Logic in BandAdapterOracle	Coding Practices	Resolved
PVE-002	Informational	Repeated Reads Avoidance in RewardVault	Coding Practices	Resolved
PVE-003	Medium	Incorrect Position Redemption Logic in UniswapV3Strategy	Business Logic	Resolved
PVE-004	Medium	Trust Issue on Admin Keys	Security Features	Mitigated
PVE-005	Low	Improved Validation in RiskFramework	Coding Practices	Resolved
PVE-006	Low	Missing receive() Function in NativeLendingPool	Coding Practices	Resolved
PVE-007	High	Incorrect PendingFee Calculation in UniswapV3PositionViewer	Business Logic	Resolved
PVE-008	Low	Incorrect Slippage Control in SwapHelper	Coding Practices	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Revisited getUSDPriceE36() Logic in BandAdapterOracle

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BandAdapterOracle`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

`Stella` has a price oracle module that can accept `Band` and `Chainlink` price feeds. While examining the price feed from `Band`, we notice its `getUSDPriceE36()` can be improved.

In the following, we show the code snippet of the `getUSDPriceE36()` routine. This routine has a rather straightforward logic in retrieving the price value of the given input token per unit. For consistency, all price values have the 36 decimals. Notice that the final result is returned as `(data.rate * ONE_E36)/ 10 ** (18 + decimals)`, which may be simplified as `(data.rate * ONE_E18)/ 10 ** decimals`.

```
function getUSDPriceE36(address _token) external view returns (uint) {
  // 0. load states + sanity check
  string memory sym = symbols[_token];
  uint maxDelayTime = maxDelayTimes[_token];
  if (bytes(sym).length == 0) {
    revert NoSymbolRegistered();
  }
  if (maxDelayTime == 0) {
    revert MaxDelayTimeNotSet();
  }
  // 1. get price
  uint decimals = uint(IBandDetailedERC20(_token).decimals());
  IStdReference.ReferenceData memory data = IStdReference(ref).getReferenceData(sym,
      USD);
  if (
    data.lastUpdatedBase < block.timestamp - maxDelayTime
```

```
      data.lastUpdatedQuote < block.timestamp - maxDelayTime
    ) {
      revert DataDelayed();
    }

    return (data.rate * ONE_E36) / 10 ** (18 + decimals);
  }
```

Listing 3.1: BandAdapterOracle::getUSDPriceE36()

Recommendation Simplify the above `getUSDPriceE36()` logic as suggested.

Status The issue has been resolved in the following commit: `48d8134`.

3.2 Repeated Reads Avoidance in RewardVault

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `RewardVault`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `Stella` protocol has a `RewardVault` contract that is programmed to distribute requested tokens to the lending pools. In the process of analyzing the current distribution logic, we notice the implementation has repeated reads on the same storage, which can be avoided.

To elaborate, we show below the related code snippets from the `distributeReward()` routine. While it achieves the intended goal, we notice inside the `for`-loop, the code will repeatedly read the storage states `dailyRewardDistributionRateE18` and `maxRewardDistributionFactorE18`, which can be optimized to read only once right before entering the `for`-loop.

```
  function distributeReward(
    address[] calldata _tokens
  ) external override onlyAuthorized(keccak256('rewardVaultWorker')) {
    for (uint i; i < _tokens.length; ) {
      uint balance = IERC20(_tokens[i]).balanceOf(address(this));
      if (balance == 0) {
        revert ZeroAmount();
      }

      // gas saving
      uint _lastDistributedTimestamp = lastDistributedTimestamp[_tokens[i]];
      uint _dailyRewardDistributionRateE18 = dailyRewardDistributionRateE18;
      uint _maxRewardDistributionFactorE18 = maxRewardDistributionFactorE18;
```

```
      // calculate factor to distribute (the first distribution = the daily rate)
      uint distributionFactorE18 = _lastDistributedTimestamp == 0
        ? _dailyRewardDistributionRateE18
        : (_dailyRewardDistributionRateE18 * (block.timestamp -
            _lastDistributedTimestamp)) /
          1 days;

      // cap distribute factor
      distributionFactorE18 = distributionFactorE18 < _maxRewardDistributionFactorE18
        ? distributionFactorE18
        : _maxRewardDistributionFactorE18;

      // calculate amount to distribute
      uint distributeAmount = (balance * distributionFactorE18) / ONE_E18;

      // update distribute timestamp
      lastDistributedTimestamp[_tokens[i]] = block.timestamp;

      // get lending pool address
      address lendingPool = ILendingProxy(lendingProxy).lendingPools(_tokens[i]);

      // transfer to lending pool address.
      IERC20(_tokens[i]).safeTransfer(lendingPool, distributeAmount);

      emit DistributeReward(lendingPool, distributeAmount);
      unchecked {
        ++i;
      }
    }
  }
```

Listing 3.2: RewardVault::distributeReward()

Recommendation Revisit the above logic to avoid repeated storage reads.

Status The issue has been resolved in the following commit: `48d8134`.

3.3 Incorrect Position Redemption Logic in UniswapV3Strategy

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `UniswapV3Strategy`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

Each strategy in `Stella` is paired with the respective position manager and positive viewer. With close coordination of these contracts, the protocol can efficiently manage each user position. While examining the logic to redeem a user position, we notice the implementation needs to be revisited.

To elaborate, we show below the related code snippet from the `_redeemPosition()` routine. As the name indicates, this routine is invoked when the user position is closed. As part of the position-closing logic, the third step here – implemented in `takeAllCollTokens()` subroutine – is to take `LP` as well as extra collateral tokens from the lending proxy. The `takeAllCollTokens()` subroutine accidentally deletes the `pos.collAmt` (line 67), which makes the subsequent `decreaseLiquidity()` call (line 176) with an incorrect liquidity amount argument `IUniswapV3PositionManager(_positionManager).getPositionCollAmt(_user, _posId)` (line 179). With the incorrect liquidity amount, the redemption logic is then considered flawed and needs to be fixed.

```
163   function _redeemPosition(
164     address _user,
165     uint _posId
166   ) internal override returns (address[] memory rewardTokens, uint[] memory rewardAmts)
        {
167     address _positionManager = positionManager;
168     // 3. take lp & extra coll tokens from lending proxy
169     _takeAllCollTokens(_positionManager, _user, _posId, address(this));

171     UniV3ExtraPosInfo memory extraPosInfo = IUniswapV3PositionManager(_positionManager)
172       .getPositionExtraInfo(_user, _posId);

174     address _uniswapV3NPM = uniswapV3NPM; // gas saving
175     // 4. remove underlying tokens from lp (internal remove in NPM)
176     IUniswapV3NPM(_uniswapV3NPM).decreaseLiquidity(
177       IUniswapV3NPM.DecreaseLiquidityParams({
178         tokenId: extraPosInfo.uniV3PositionId,
179         liquidity: IUniswapV3PositionManager(_positionManager).getPositionCollAmt(_user,
              _posId),
180         amount0Min: 0,
181         amount1Min: 0,
182         deadline: block.timestamp
183       })
184     );
```

```
    // 5. collect liquidity & fees
    IUniswapV3NPM(_uniswapV3NPM).collect(
      IUniswapV3NPM.CollectParams({
        tokenId: extraPosInfo.uniV3PositionId,
        recipient: address(this),
        amount0Max: type(uint128).max,
        amount1Max: type(uint128).max
      })
    );

    // 6. burn LP position
    IUniswapV3NPM(_uniswapV3NPM).burn(extraPosInfo.uniV3PositionId);
  }
}
```

Listing 3.3: `UniswapV3Strategy::_redeemPosition()`

```
  function takeAllCollTokens(address _user, uint _posId, address _to) external onlyStrategy {
    // 0. load states
    Position storage pos = __positions[_user][_posId];
    UniV3ExtraPosInfo memory extraPosInfo = _getPositionExtraInfo(pos.extraPosInfo);
    address[] memory underlyingTokens = __underlyingTokens; // gas saving
    uint128[] memory extraCollAmts = pos.extraCollAmts; // gas saving

    // 1. update position info before token transfer
    delete pos.collAmt;
    delete pos.extraCollAmts;

    // 2. transfer extra collateral tokens
    for (uint i; i < extraCollAmts.length; ) {
      if (extraCollAmts[i] > 0) {
        IERC20(underlyingTokens[i]).safeTransfer(_to, extraCollAmts[i]);
      }

      unchecked {
        ++i;
      }
    }

    // 3. transfer uni v3 lp
    IERC721(uniV3NPM).safeTransferFrom(address(this), _to, extraPosInfo.uniV3PositionId);
  }
```

Listing 3.4: `UniswapV3PositionManager::takeAllCollTokens()`

Recommendation Revise the above redemption logic to make use of the right liquidity amount for redemption.

Status The issue has been resolved in the following commit: `48d8134`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Stella protocol, there is a privileged account (with the exec role) that plays a critical role in governing and regulating the contract-wide operations (e.g., configuring various system parameters and assigning other roles). In the following, we show the representative functions potentially affected by the privilege of the account.

```
function setGateway(address _gateway) external onlyAuthorized(keccak256('dev')) {
  gateway = _gateway;
  emit SetGateway(_gateway);
}

/// @dev exec sets new reward vault (address to receive lending reward).
/// @param _vault vault address.
function setRewardVault(address _vault) external onlyAuthorized(keccak256('exec')) {
  rewardVault = _vault;
  emit SetRewardVault(_vault);
}

/// @dev exec sets new treasury (address to receive protocol fee).
/// @param _treasury treasury address.
function setTreasury(address _treasury) external onlyAuthorized(keccak256('exec')) {
  treasury = _treasury;
  emit SetTreasury(_treasury);
}

/// @dev exec sets new risk framework.
/// @param  _riskframework risk framework address.
function setRiskFramework(address _riskframework) external onlyAuthorized(keccak256('exec')) {
  riskFramework = _riskframework;
  emit SetRiskFramework(_riskframework);
}

/// @dev exec sets protocol's fee (when profit sharing).
/// @param _token token address.
/// @param _feeBPS fee bps max @ 100% (10000).
function setFeeBPS(address _token, uint _feeBPS) external onlyAuthorized(keccak256('exec')) {
  if (_feeBPS > BPS) {
    revert FeeTooHigh();
```

```
    }
    feeBPS[_token] = _feeBPS;
    emit SetFeeBPS(_token, _feeBPS);
  }
```

Listing 3.5: Example Privileged Operations in `LendingProxy`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed and mitigated with a multisig account.

3.5 Improved Validation in RiskFramework

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `RiskFramework`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `Stella` protocol has a dedicated `RiskFramework` contract to configure the allowed exposure from the leveraged strategies. Accordingly, it provides a number of setters to configure the exposure from different perspectives. While reviewing the function to set token exposure limit amounts, the current setter can be improved with additional validation.

To elaborate, we show below the implementation of the related `setTokenExposureLimitAmounts()` routine. As the name indicates, this routine is used to set the token exposure with the given limit amount. Naturally, we will assume the exposure token needs not to be the given collateral token to account exposure against. And this assumption can be explicitly enforced and this enforcement is currently missing.

```
  function setTokenExposureLimitAmounts(
    address[] calldata _tokens,
    address[][] calldata _exposedTokens,
    uint120[][] calldata _amounts
```

```
) external onlyAuthorized(keccak256('riskFrameworkWorker')) {
  for (uint i; i < _tokens.length; ) {
    if (_exposedTokens[i].length != _amounts[i].length) {
      revert InvalidLength();
    }
    for (uint j; j < _exposedTokens[i].length; ) {
      tokenExposureInfos[_tokens[i]][_exposedTokens[i][j]].exposureLimitAmount =
          _amounts[i][j];
      emit SetTokenExposureLimitAmount(_tokens[i], _exposedTokens[i][j], _amounts[i][j
          ]);

      unchecked {
        ++j;
      }
    }

    unchecked {
      ++i;
    }
  }
}
```

Listing 3.6: `RiskFramework::setTokenExposureLimitAmounts()`

Recommendation Apply an additional requirement to the above routine to ensure the given `_exposedTokens[i]` is not equal to `_collateralTokens[i][j]`.

Status The issue has been resolved in the following PR: 59.

3.6 Missing receive() Function in NativeLendingPool

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `NativeLendingPool`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `Stella` protocol supports the native coin as the borrow token in `NativeLendingPool`. To facilitate the user interaction, the contract provides the support of `Ether` wrapping and unwrapping. While examining the current logic, we notice that the implementation does not support the `receive()` method.

To elaborate, we show below again the `NativeLendingPool` contract. Note the contract inherits from the common `BaseLendingPool` logic, which has not defined the fallback routine. With that, we suggest the need of adding a `receive()` method. Note this `receive` keyword is introduced since `Solidity` 0.6.x in order to make contracts more explicit when their fallback functions are called. In particular, the `receive()` method is used as a fallback function in a contract and is called when the native coin is sent to a contract with no calldata. If the `receive()` method does not exist, it will use the default fallback function.

```
contract NativeLendingPool is BaseLendingPool {
  // ================== Constructor & Initializer =====================

  constructor(address _lendingProxy) BaseLendingPool(_lendingProxy) {}

  // ====================== external functions ======================

  /// @dev transmit amount of base token from lending proxy. (only lending proxy)
  /// @param _receiver receiver address.
  /// @param _amount amount to transfer.
  /// @return mintShares shares amount to mint to receiver
  function mint(
    address _receiver,
    uint _amount
  ) external payable override onlyLendingProxy returns (uint mintShares) {...
  }

  /// @dev burn msg.sender's shares to withdraw token. (only lending proxy)
  /// @param _shares shares to burn.
  /// @return redeemAmount underlying redeem amount to send to receiver
  function redeem(
    address _receiver,
    uint _shares
  ) external override onlyLendingProxy returns (uint redeemAmount) {...
  }
}
```

Listing 3.7: The `NativeLendingPool` contract

Recommendation Add the `receive()` function as there is no default fallback routine in the above `NativeLendingPool` contract.

Status This issue has been resolves as the team confirms that the `NativeLendingPool` contract will be upgradeable with the `TransparentUpgradeableProxyReceiveETH` contract, which has the `receive()` support.

3.7 Incorrect PendingFee Calculation in UniswapV3PositionViewer

- ID: PVE-007
- Severity: High
- Likelihood: High
- Impact: High
- Target: `UniswapV3PositionViewer`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

As mentioned earlier, each strategy in `Stella` is paired with the respective position manager and positive viewer. With close coordination of these contracts, the protocol can efficiently manage each user position. While examining the current logic to compute the accrued position fees, we notice the implementation needs to be revised.

To elaborate, we show below the current implementation of the `_getPendingFeeAmts()` routine. This routine has a rather straightforward logic in returning a position's pending fees. We notice the fee collection should be based on the position's liquidity, not the total liquidity managed by the `uniV3NPM` on the given tick range `[tickLower, tickUpper]`. Moreover, the `tokensOwed0` and `tokensOwed1` pairs should be un-collected but entitled token amount for the user position, not all positions managed by `uniV3NPM`.

```
function _getPendingFeeAmts(
  address _positionManager,
  bytes memory _extraPosInfo
) internal view returns (uint[] memory rewards) {
  UniV3ExtraPosInfo memory extraPosInfo = _getPositionExtraPosInfo(_extraPosInfo);
  address collToken = IBasePositionManager(_positionManager).collToken(); // gas saving
  int24 tickLower;
  int24 tickUpper;
  uint[] memory feeGrowthInsideLastX128s = new uint[](2);
  (
    ,
    ,
    ,
    ,
    ,
    tickLower,
    tickUpper,
    ,
    feeGrowthInsideLastX128s[0],
    feeGrowthInsideLastX128s[1],
    ,
```

```
    ) = IUniswapV3NPM(uniV3NPM).positions(extraPosInfo.uniV3PositionId);

    (, int24 curTick, , , , , ) = IUniswapV3Pool(collToken).slot0();

    (uint128 liquidity, , , uint128 tokensOwed0, uint128 tokensOwed1) = IUniswapV3Pool(
        collToken)
      .positions(keccak256(abi.encodePacked(uniV3NPM, tickLower, tickUpper)));

    rewards = _computePendingFeesToBeEarned(
      collToken,
      feeGrowthInsideLastX128s,
      curTick,
      tickLower,
      tickUpper,
      liquidity
    );

    rewards[0] += tokensOwed0;
    rewards[1] += tokensOwed1;
  }
```

Listing 3.8: `UniswapV3PositionViewer::_getPendingFeeAmts()`

Recommendation Properly compute the pending fees in the above routine for collection.

Status The issue has been resolved in the following PR: `60`.

3.8 Incorrect Slippage Control in SwapHelper

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SwapHelper`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `Stella` protocol has a `SwapHelper` contract to facilitate the need of swapping from one token to another. While examining the current token-swapping logic, we notice the use of slippage control needs to be improved.

To elaborate, we show below the implementation of the `getCalldata()` routine, which is proposed to return the swap calldata. The routine supports two types of operations: `SwapOperation.EXACT_IN` and `SwapOperation.EXACT_OUT`. The former assumes the input amount and computes the expected output amount while the latter assumes the output amount and computes the expected input amount. The current latter logic accidentally sets the `UNISWAP_V2`'s `amountInMax` argument to `swapTokensForExactTokens()` and the `UNISWAP_V3`'s `amountInMaximum` argument to `exactOutput()` to 0, which will revert all possible `SwapOperation.EXACT_OUT` operations.

```
function getCalldata(
  address _tokenIn,
  address _tokenOut,
  SwapOperation _operation,
  address _router,
  uint _amount
) external view returns (bytes memory) {
  bytes memory _swapInfo = swapInfos[_tokenIn][_tokenOut][_router];
  RouterType _routerType = routerTypes[_router];
  if (_operation == SwapOperation.EXACT_IN) {
    if (_routerType == RouterType.UNISWAP_V2) {
      UniswapV2SwapInfo memory _decodedSwapInfo = abi.decode(_swapInfo, (
          UniswapV2SwapInfo));
      return
        abi.encodeWithSelector(
          ISwapRouter.swapExactTokensForTokens.selector,
          _amount,
          0,
          _decodedSwapInfo.path,
          msg.sender,
          block.timestamp
        );
    } else if (_routerType == RouterType.UNISWAP_V3) {
      UniswapV3SwapInfo memory _decodedSwapInfo = abi.decode(_swapInfo, (
          UniswapV3SwapInfo));

      return
        abi.encodeWithSelector(
          ISwapRouter.exactInput.selector,
          ExactInputParams({
            path: _decodedSwapInfo.path,
            recipient: msg.sender,
            deadline: block.timestamp,
            amountIn: _amount,
            amountOutMinimum: 0
          })
        );
    } else if (_routerType == RouterType.CURVE) {
      CurveSwapInfo memory _decodedSwapInfo = abi.decode(_swapInfo, (CurveSwapInfo));
      return
        abi.encodeWithSelector(
          ISwapRouter.exchange.selector,
          _decodedSwapInfo.tokenInIndex,
          _decodedSwapInfo.tokenOutIndex,
          _amount,
          0
        );
    } else {
      revert UnsetRouterType();
```

```
      }
    } else if (_operation == SwapOperation.EXACT_OUT) {
      if (_routerType == RouterType.UNISWAP_V2) {
        UniswapV2SwapInfo memory _decodedSwapInfo = abi.decode(_swapInfo, (
            UniswapV2SwapInfo));
        return
          abi.encodeWithSelector(
            ISwapRouter.swapTokensForExactTokens.selector,
            _amount,
            0,
            _decodedSwapInfo.path,
            msg.sender,
            block.timestamp
          );
      } else if (_routerType == RouterType.UNISWAP_V3) {
        UniswapV3SwapInfo memory _decodedSwapInfo = abi.decode(_swapInfo, (
            UniswapV3SwapInfo));
        return
          abi.encodeWithSelector(
            ISwapRouter.exactOutput.selector,
            ExactOutputParams({
              path: _reversePath(_decodedSwapInfo.path),
              recipient: msg.sender,
              deadline: block.timestamp,
              amountOut: _amount,
              amountInMaximum: 0
            })
          );
      } else if (_routerType == RouterType.CURVE) {
        revert UnsupportedOperation(); // NOTE: curve has no exact out function
      } else {
        revert UnsetRouterType();
      }
    } else {
      revert UnsupportedOperation();
    }
  }
```

Listing 3.9: `SwapHelper::getCalldata()`

Recommendation Make use of the right `amountInMax/amountInMaximum` in the above routine to avoid unnecessary swap reverts.

Status The issue has been resolved in the following PR: 62.

4 Conclusion

In this audit, we have analyzed the `Stella` design and implementation. `Stella` is a leveraged strategies protocol with 0% cost to borrow, utilizing the `pay-as-you-earn (PAYE)` model. Lenders can lend supported assets to earn shared profits from leveraged users. On the other hand, leveraged users can borrow lent assets at 0% cost to gain higher leverage of up to $10x$ and earn more profits. Supported strategies include yield farming, liquidity providing, staking, and integrations with many other protocols. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.