SMART CONTRACTS REVIEW

zokyo

July 23rd 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING COPRA

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 1 Critical issue: 1 acknowledged = - 10 points deducted
- 2 High issues: 1 resolved and 1 acknowledged = - 4 points deducted
- 3 Medium issues: 2 resolved and 1 acknowledged = - 2 points deducted
- 7 Low issues: 3 resolved and 4 acknowledged = - 1 points deducted
- 0 Informational issues: 0 points deducted

Thus, 100 - 10 - 4 - 2 - 1 = 83

TECHNICAL SUMMARY

This document outlines the overall security of the Copra smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Copra smart contracts codebase for quality, security, and correctness.

Contract Status

There was 1 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Copra team put in place a bug bounty program to encourage further active analysis of the smart contracts.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Copra repository:
Repo:https://github.com/copra-finance/copra-v3-audit-1/blob/main/src

PR with the last fixes: https://github.com/copra-finance/copra-v3-audit-1/pull/61

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- CurveLiquidityWarehouse.sol
- GammaSwapLiquidityWarehouse.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Copra smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contracts by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team has performed a security audit of the provided codebase. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

The `CurveLiquidityWarehouse.sol` is a Copra smart contract that contains two functions. The `_getAssetIdxInCurvePool` function returns the index position of a certain asset in the Curve pool's array. The `_getDeployedAssetValue` function returns the amount of tokens resulting from a swap from x pool LP tokens. The main function, `_withdrawFromTarget`, calculates if the amount returned from a withdrawal from a Curve pool is greater than or equal to the amount expected by the user minus the maximum loss.

The `GammaLiquidityWarehouse.sol` is a Copra smart contract that inherits LiquidityWarehouse contract where the lenders can deposit their tokens for a fixed yield and the borrows can borrow it for a leveraged yield. The `_withdrawFromTarget` function is a hook for the Liquidity Warehouse that facilitates withdrawal from the target. The `_executeWithdrawal` function called in the `_withdrawFromTarget` function interacts with the PositionManager to withdraw reserves based on shares.The `_calculateAmountsMin` function calculates the minimum amount based on slippage via invariants and uses `_getAssetOraclePrice` to get the latest prices from a Chainlink Oracle. The `_swapToBaseAsset` allows swapping of assets using exactInput() function of the SwapRouter. The `_getBaseAssetValueInGammaSwapPool` function gets the total asset value in USD using the oracles based on the total pool invariants.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Copra team and the Copra team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	SharePrice can get manipulated	Critical	Acknowledged
2	Function's return value is incorrect	High	Resolved
3	Insufficient slippage protection can lead to displaced transactions	High	Acknowledged
4	Lack of Fallback Mechanism for Stale Price Feeds in _getAssetOraclePrice Function	Medium	Resolved
5	Hardcoded or Fixed Slippage can lead to Loss of User funds	Medium	Acknowledged
6	Incorrect Staleness Threshold for Chainlink Price Feeds	Medium	Resolved
7	Usage of balanceOf(address(this)) can lead to manipulation	Low	Acknowledged
8	Unused Constant DELTASWAP_PROTOCOL_ID in GammaLiquidityWarehouse Contract	Low	Resolved
9	Return value ignored and unhandled	Low	Acknowledged
10	For loop over Dynamic array	Low	Acknowledged
11	_withdrawFromTarget Would Revert If asset Decimals > Withdraw Target Decimals	Low	Resolved
12	Inefficient handling of function parameters.	Low	Acknowledged
13	Unhandled Chainlink Oracle Access Revocation Risk	Low	Resolved

SharePrice can get manipulated

The `_withdrawFromTarget` function in the `CurveLiquidityWarehouse.sol` contract calculates the theoric price for a LP share relying on the spot price of the curve pool.

```
uint256 sharePrice = curveStableSwapPool.calc_withdraw_one_coin(10 **
(withdrawTargetDecimals), curvePoolCoinIdx);
```

The `calc_withdraw_one_coin` function from the Curve's StableSwap pool calculates the amount received when withdrawing a single coin. The amount returned from this function is calculated on chain; it works by retrieving the spot price from a pool which can lead to a price manipulation by an attacker. Once the transaction is pending in the mempool an attacker can front-run this legit transaction and submit a transaction to the curve pool that manipulates the price that is going to be returned by this function leading to getting an inflated or uninflated price for `sharePrice`.

Recommendation:

`sharePrice` is used to calculate `burnAmt` which represents the amount of LP tokens that needs to get burnt in order to withdraw `withdrawAmount` from the asset. Instead of calculating `burnAmt` on chain add a parameter representing this amount calculated of-chain to avoid price manipulation.

Zokyo Auditor's Comment: Added a partial, but not complete, mitigation fix that involves retrieving the pool price from two different oracles, one of them using EMA, and checking if its deviation is within a dynamic threshold compared with the spot price, needing a larger amount of funds to manipulate. However, the risk of vulnerabilities has not been 100% mitigated, as it should not rely on on-chain manipulable calculations, but instead use off-chain mechanisms or rely on External oracles like Chainlink, pyth etc.

Client comment: we aren't using an off-chain oracle i.e none exists for the underlying assets of the Curve LP tokens we are dealing with

HIGH-1 | RESOLVED

Function's return value is incorrect

From the docs it can be observed that the `_withdrawFromTarget` function should `return a `boolean` that evaluates to true if the full `withdrawalAmount` could be withdrawn fro the `withdrawTarget.` ` However the current implementation of the function is returning true is the receivedAmount is less or equal to the expected withdrawnAmount - maxLossAmount.

Recommendation:
The return statement should be `return withdrawAmount - maxLossAmount <= uint256(receivedAmount);`

HIGH-2 | ACKNOWLEDGED

Insufficient slippage protection can lead to displaced transactions

The `_withdrawFromTarget` function in the `CurveLiquidityWarehouse.sol` contract does not receive a deadline argument representing the last moment at which the transaction must be accepted. This can lead to a scenario where the transaction is displaced and executed in later blocks obtaining a worse price.

Recommendation:
Add a `deadline` parameter and check when the transaction is executed, if block.timestamp is greater than deadline, revert.

Lack of Fallback Mechanism for Stale Price Feeds in `_getAssetOraclePrice` Function

Location: GammaSwapLiquidityWarehouse.sol

Description

The `_getAssetOraclePrice` function in the `GammaLiquidityWarehouse` contract checks for stale price feeds but does not handle the case where the price feed is stale. If the price feed is stale, the function reverts, which can disrupt the contract's operations.

Issue: If the price feed is stale, the function reverts, but there is no fallback mechanism to handle this scenario. This can lead to disruptions in the contract's operations, especially if the price feed remains stale for an extended period.

Impact

- **Operational Disruption:** The contract's operations can be disrupted if the price feed is stale, leading to potential loss of functionality.
- **User Experience:** Users may experience failed transactions due to the lack of a fallback mechanism, leading to frustration and loss of trust in the contract.

Recommendation:

Implement a fallback mechanism to handle stale price feeds. This can be achieved by using a secondary price feed or a predefined fallback value. The fallback mechanism ensures that the contract can continue to operate even if the primary price feed is stale.

Suggested Implementation

Here is a suggested implementation that includes a fallback mechanism using a secondary price feed:

```
function _getAssetOraclePrice(address asset) internal view returns (uint256) {
    AggregatorV3Interface priceFeed = s_assetConfigs[asset].chainlinkPriceFeed;
    AggregatorV3Interface fallbackPriceFeed = s_assetConfigs[asset].fallbackPriceFeed; // Add a secondary price feed
    (, int256 oraclePrice,, uint256 updatedAt,) = priceFeed.latestRoundData();
    if (block.timestamp - updatedAt > STALENESS_THRESHOLD) {
        // Use fallback price feed if the primary price feed is stale
        (, int256 fallbackOraclePrice,, uint256 fallbackUpdatedAt,) = fallbackPriceFeed.latestRoundData();
```

```
        if (block.timestamp - fallbackUpdatedAt > STALENESS_THRESHOLD) {
            revert ChainlinkPriceFeedStale();
        }
        oraclePrice = fallbackOraclePrice;
    }
    (, int256 answer, uint256 startedAt,,) = i_sequencerUptimeFeed.latestRoundData();
    if (answer == 1) {
        revert SequencerDown();
    }
    if (block.timestamp - startedAt <= GRACE_PERIOD_TIME) {
        revert GracePeriodNotOver();
    }
    return oraclePrice.toUint256() * 10 ** (18 - priceFeed.decimals());
}
```

MEDIUM-2 | ACKNOWLEDGED

Hardcoded or Fixed Slippage can lead to Loss of User funds

In the GammaSwapLiquidityWarehouse.sol contract the hardcoded Slippage values is not adivsed as it can lead to loss of user funds when the volatility of token is high.
Line: 48 and 97. The function on **line: 102** only allows the admin to change the slippage and not the users.

In addition to this, the admin can change the slippage to a very high or low value without user's notice which could lead to loss of user funds.

Recommendation:
Consider allowing mechanism for user to override the existing slippage value so that he can override it if he wants. And use a multisig wallet to ensure that the ownership of the contract is sufficiently decnetralized to prevent loss of private keys or any malicious changes to the slippage.

Refer- https://dacian.me/defi-slippage-attacks#heading-hard-coded-slippage-may-freeze-user-funds

Incorrect Staleness Threshold for Chainlink Price Feeds

Location: GammaSwapLiquidityWarehouse.sol

Description
The current implementation uses a single STALENESS_THRESHOLD constant (24 hours) for all Chainlink price feeds. This approach is problematic because different price feeds have vastly different heartbeat intervals. For example, the ETH/USD feed has a heartbeat of 1 hour, while the AMPL/USD feed has a heartbeat of 48 hours.

Using a single threshold for all feeds can lead to two critical issues:

- For feeds with shorter heartbeats (e.g., ETH/USD), the current implementation allows prices to be considered fresh for up to 23 hours longer than intended. This could result in the use of severely outdated prices, potentially causing significant financial losses.
- For feeds with longer heartbeats (e.g., AMPL/USD), the current threshold might incorrectly flag fresh prices as stale, potentially causing unnecessary service interruptions.

A real-world incident highlighting the risks of this approach occurred when the Chainlink ETH/USD price feed experienced a 6-hour delay. In such scenarios, using outdated prices could lead to substantial financial losses or incorrect contract executions. https://cryptobriefing.com/chainlink-experiences-6-hour-delay-eth-price-feed/
The current implementation in the _getAssetOraclePrice function uses a single STALENESS_THRESHOLD:

if (block.timestamp - updatedAt > STALENESS_THRESHOLD) revert ChainlinkPriceFeedStale();

This check does not account for the varying heartbeat intervals of different price feeds.

Recommendation:

• Implement a token-specific staleness threshold system: Replace the single STALENESS_THRESHOLD constant with a mapping that stores individual staleness thresholds for each token:

```
mapping(address ⇒ uint256) private s_stalenessTresholds;
```

- Modify the _getAssetOraclePrice function to use the token-specific threshold:

```
function _getAssetOraclePrice(address asset) internal view returns (uint256) {
    AggregatorV3Interface priceFeed = s_assetConfigs[asset].chainlinkPriceFeed;
    (, int256 oraclePrice,, uint256 updatedAt,) = priceFeed.latestRoundData();
    if (block.timestamp - updatedAt > s_stalenessTresholds[asset]) revert
ChainlinkPriceFeedStale();
    // ... rest of the function
}
```

- Implement a function to set and update staleness thresholds for each asset:

```
function setStalnessThreshold(address asset, uint256 threshold) external
onlyRole(DEFAULT_ADMIN_ROLE) {
    s_stalenessTresholds[asset] = threshold;
}
```

When adding new assets or updating configurations, ensure that appropriate staleness thresholds are set based on the specific Chainlink feed's heartbeat interval. It's recommended to set the threshold slightly higher than the heartbeat to account for minor delays, but not so high as to accept severely outdated prices.

Reference
Price Feed Contract Addresses | Chainlink Documentation - click on Show more details for heartbeats

Usage of balanceOf(address(this)) can lead to manipulation

In the GammaSwapLiquidityWarehouse.sol contract the balanceOf(address(this)) has been used on line: 185 and 284. balanceOf(address(this)) must be avoided as this leads to manipulation of values by directing sending tokens to the contract via Flash loan attack.

An attacker can directly send funds to the contract result in incorrect shareAmt been calculated on line: 131. But this behaviour is a griefing attack and does not result in any profit for the attacker. It results only in his loss of funds. Even sending large amount of GammaPool tokens would not actually affect the ShareAmt as minimum Math is being used on the line: 145.

Recommendation:

As a best security practice it is still advisable to:

1. Use mapping to keep track of balances, which get updated only when the deposit and withdraw is done via the functions in the contract legitimately. This would prevent manipulation of balanceOf() function by directly sending funds to the contract.
2. It is advised to disallow depositing and withdrawing funds in the same block to prevent flash loan attacks. Flash loans allow the borrowing of funds in large number provided they are returned back in the same block. Disallowing depositing and borrowing funds in the protocol in the same block drastically reduces the chances of a Flash Loan attack. Refer LiquidityWarehouse for the same.

Unused Constant DELTASWAP_PROTOCOL_ID in GammaLiquidityWarehouse Contract

Location: GammaSwapLiquidityWarehouse.sol

Description

The GammaLiquidityWarehouse contract defines a constant DELTASWAP_PROTOCOL_ID with a value of 3. However, this constant is never used within the contract. Unused constants can lead to confusion and clutter in the codebase, making it harder to maintain and understand.

The unused constant is defined as follows:

```
/// @notice The deltaswap protocol id
uint16 internal constant DELTASWAP_PROTOCOL_ID = 3;
```

Impact

- Code Clarity: Unused constants can make the code harder to read and understand.
- Maintenance: Future developers may spend unnecessary time trying to understand the purpose of the unused constant.

Potential Bugs: If the constant was intended to be used but was forgotten, it could indicate a potential bug or incomplete implementation.

Recommendation:

Remove the unused DELTASWAP_PROTOCOL_ID constant to improve code clarity and maintainability. If the constant was intended to be used, ensure it is integrated into the relevant parts of the contract.

LOW-3 | ACKNOWLEDGED

Return value ignored and unhandled

In the GammaSwapLiquidityWarehouse.sol contract, in the _swapToBaseAsset() function, the value returned by the exactInput() is being ignored and unhandled.

```
Line: 289 ISwapRouter(s_assetConfigs[asset].swapRouter).exactInput(
            ISwapRouter.ExactInputParams({
               path: s_assetConfigs[asset].swapPath,
               recipient: address(this),
               deadline: block.timestamp + s_assetConfigs[asset].deadline,
               amountIn: assetBalance,
               amountOutMinimum: assetOutExpected * (WAD - slippage) / WAD
            })
         );
```

Recommendation:
It is advised that the returned value is handled properly and returned and emitted as an event.

LOW-4 | ACKNOWLEDGED

For loop over Dynamic array

In the GammaSwapLiquidityWarehouse.sol contract, there is for loop over Dynamic array on **lines 174 and 265**. In addition to this there is nested for loop on **line: 193 and 196**. This is not advised as for loops over dynamic arrays and nested loops can lead of out of gas errors and potentially lead to Denial of Service.

Recommendation:
It is advised that the length of the array is limited to a certain number and checked to prevent out of gas errors.

LOW-5 | RESOLVED

_withdrawFromTarget Would Revert If asset Decimals > Withdraw Target Decimals

The minReceived inside withdrawFromTarget is calculated as follows →

```
uint256 minReceived =
        burnAmt / 10 ** (withdrawTargetDecimals -
IERC20Metadata(address(s_terms.asset)).decimals()) - maxLossAmount;
```

This normalises burnAmt to be in the asset's decimal precision , but it would revert if (s_terms.asset)).decimals() > withdrawTargetDecimals .

Recommendation:
If (s_terms.asset)).decimals() > withdrawTargetDecimals then normalise through burnAmt * 10 ** ((s_terms.asset)).decimals() - withdrawTargetDecimals)

LOW-6 | ACKNOWLEDGED

Inefficient handling of function parameters.

The function `_withdrawFromTarger` in `CurveLiquidityWarehouse.sol` receives a parameter `bytes memory data` which is later decoded to `uint256 maxLossAmount`. However using directly uint256 without needing to decode it will help on the efficiency of the execution..

Recommendation:
Use `uint256 maxLossAmount` directly as function parameter.

Unhandled Chainlink Oracle Access Revocation Risk

Location: GammaSwapLiquidityWarehouse.sol

Description
The _getAssetOraclePrice function in the GammaLiquidityWarehouse contract directly calls the latestRoundData function of Chainlink price feeds without proper error handling. This approach is vulnerable to potential access revocation by multisigs controlling the price feeds, which could lead to unexpected contract failures and denial of service.

Currently, the function makes two direct calls to latestRoundData:

For the asset price feed:

(, int256 oraclePrice,, uint256 updatedAt,) = priceFeed.latestRoundData();

For the sequencer uptime feed:

(, int256 answer, uint256 startedAt,,) = i_sequencerUptimeFeed.latestRoundData();
If access to these price feeds is revoked or blocked, these calls will revert, causing the entire function to fail. This could potentially render critical parts of the contract inoperable, as price information is essential for many DeFi operations.

Recommendation:
Implement a try-catch mechanism for both latestRoundData calls to handle potential access revocations gracefully. This approach allows the contract to maintain control and implement appropriate fallback mechanisms or error handling.
Here's an example of how the function could be refactored:

```
function _getAssetOraclePrice(address asset) internal view returns (uint256) {
    AggregatorV3Interface priceFeed = s_assetConfigs[asset].chainlinkPriceFeed;
    try priceFeed.latestRoundData() returns (
        uint80,
        int256 oraclePrice,
```

```
        uint256,
        uint256 updatedAt,
        uint80
    ) {
        if (block.timestamp - updatedAt > STALENESS_THRESHOLD) revert
ChainlinkPriceFeedStale();

        try i_sequencerUptimeFeed.latestRoundData() returns (
            uint80,
            int256 answer,
            uint256 startedAt,
            uint256,
            uint80
        ) {
            if (answer == 1) {
                revert SequencerDown();
            }
            if (block.timestamp - startedAt <= GRACE_PERIOD_TIME) {
                revert GracePeriodNotOver();
            }
           return oraclePrice.toUint256() * 10 ** (18 - priceFeed.decimals());
        } catch {
            // Handle sequencer uptime feed failure
            // e.g., revert with a custom error, use a fallback mechanism, etc.
            revert SequencerUptimeFeedFailure();
        }
    } catch {
        // Handle price feed failure
        // e.g., revert with a custom error, use a fallback oracle, etc.
        revert PriceFeedAccessRevoked();
    }
}
```

This refactored version wraps both latestRoundData calls in try-catch blocks. If either call fails due to access revocation or other issues, the contract can handle the error gracefully, either by reverting with a specific error message or implementing alternative logic (such as using a fallback oracle).

Reference

Smart Contract Security Guidelines #3: The Dangers of Price Oracles - OpenZeppelin blog

	GammaSwapLiquidityWarehouse.sol CurveLiquidityWarehouse.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Copra team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Copra team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo